Vrije Universiteit, Amsterdam

Faculty of Sciences,
Computer Science Department

Cristian Mihai Bârcă, student no. 2514228

Dynamic Resource Management in Vectorwise on Hadoop

Master Thesis in Parallel and Distributed Computer Systems

Supervisor / First reader:
Prof. Dr. Peter Boncz, Vrije Universiteit, Centrum Wiskunde & Informatica

Second reader:
Prof. Dr. Henri Bal, Vrije Universiteit

Amsterdam, August 2014

Contents

1 Introduction **1**
- 1.1 Background and Motivation . . . 1
- 1.2 Related Work . . . 2
  - 1.2.1 Hadoop-enabled MPP Databases . . . 4
  - 1.2.2 Resource Management in MPP Databases . . . 7
  - 1.2.3 Resource Management in Hadoop Map-Reduce . . . 8
  - 1.2.4 YARN Resource Manager (Hadoop NextGen) . . . 9
- 1.3 Vectorwise . . . 11
  - 1.3.1 Ingres-Vectorwise Architecture . . . 12
  - 1.3.2 Distributed Vectorwise Architecture . . . 13
  - 1.3.3 Hadoop-enabled Architecture . . . 16
- 1.4 Research Questions . . . 19
- 1.5 Goals . . . 20
- 1.6 Basic ideas . . . 20

2 Approach **21**
- 2.1 HDFS Local Reads: DataTransferProtocol vs Direct I/O. . . . 22
- 2.2 Towards Data Locality . . . 24
- 2.3 Dynamic Resource Management . . . 26
- 2.4 YARN Integration . . . 28
- 2.5 Experimentation Platform . . . 31

3 Instrumenting HDFS Block Replication **33**
- 3.1 Preliminaries . . . 33
- 3.2 Custom HDFS Block Placement Implementation . . . 35
  - 3.2.1 Adding (New) Metadata to Database Locations . . . 36
  - 3.2.2 Extending the Default Policy . . . 36

4 Dynamic Resource Management **42**
- 4.1 Worker-set Selection . . . 42
- 4.2 Responsibility Assignment . . . 47
- 4.3 Dynamic Resource Scheduling . . . 53

5 YARN integration **67**
- 5.1 Overview: model resource requests as separate YARN applications . . . 68
- 5.2 DbAgent–Vectorwise communication protocol:
  update worker-set state, increase/decrease resources . . . 72
- 5.3 Workload monitor & resource allocation . . . 73

6 Evaluation & Results **80**
- 6.1 Evaluation Plan . . . 80
- 6.2 Results Discussion . . . 83

List of Figures

1.1 Hadoop transition from v1 to v2 10
1.2 The Ingres-Vectorwise (single-node) High-level Architecture 12
1.3 The Distributed Vectorwise High-level Architecture 14
1.4 The Vectorwise on Hadoop High-level Architecture (the *session-master*, or Vectorwise Master, is involved in query execution). 17

2.1 The Vectorwise on Hadoop dynamic resource management approach in 7 steps. . 22
2.2 Local reads through DataNode (DataTransferProtocol). 23
2.3 Direct local reads (Direct I/O - open(), read() operations). 23
2.4 A flow network example for the min-cost problem, edges are annotated with *(cost, flow / capacity)*. 27
2.5 The flow network (bipartite graph) model used to determine the responsibility assignment. 28
2.6 Out-of-band approach for Vectorwise-YARN integration. 30

3.1 An example of two tables (R and S) loaded on 3 workers / 4 nodes, using 3-way partitioning and replication degree of 2: after the initial-load (upper image) vs. during a system failover (lower image). 34
3.2 Instrumenting the HDFS block replication, in three scenarios, to maintain block (co-)locality to worker nodes. 35
3.3 Baseline (default policy) vs. collocation (custom policy) – HDFS I/O throughput measured in 3 situations (1) healty state, (2) failure state (2 nodes down), (3) recovered state. Queries 1, 4, 6, and 12 on partitioned tables, 32-partitioning, 8 workers / 10 Hadoop "Rocks" nodes, 32 mpl. 40

4.1 Starting the Vectorwise on Hadoop worker-set through *VectorwiseLauncher* and *DbAgent*. All related concepts to this section are emphasized with a bold black color. 43
4.2 The flow network (bipartite graph) model used to match partitions to worker nodes. 46
4.3 Assigning responsibilities to worker nodes. All related concepts to this section are emphasized with bold black and red/blue/green colors (for *single* (partition) responsibilities). 48
4.4 The flow network (bipartite graph) model used to assign responsibilities to workers nodes. 49
4.5 The flow network (bipartite graph) model used to calculate a part of the *resource footprint* (what nodes and table partitions we use for a query, given the worker's available resource and data-locality). Note: f_x is the unit with which we increase the *flow* along the path at step x of the algorithm. 57

4.6 Selecting the (partition responsibilities and worker nodes involved in query execution. Only the first two workers from left are chosen (bold-italic black) for table scans; first one reads from two partitions (red and green), whereas the second reads just from one (blue). The third worker is considered *overloaded* and so, it is excluded from the I/O. Red/blue/green colors are used to express *multiple* (partition) *responsibilities*. 58

5.1 Timeline of different query workloads. 69
5.2 Out-of-band approach for Vectorwise-YARN integration. An example using 2 x Hadoop nodes that equally share 4 x YARN containers. 70
5.3 Timeline of different query workloads (one-time release). 71
5.4 Database session example with one active query: the communication protocol to increase/decrease resources. 72
5.5 The *workload monitor's* and *resource allocation's* workflow for: **a) successful** (resource) **request** and **b) failed** (resource) **request** (VW – Vectorwise, MR – Map-Reduce) 74
5.6 YARN's allocation overhead: TPC-H scale factor 100, 32 partitions, 8 "Stones" nodes, 128 *mpl*. 78

6.1 Running *with* (w/) and *without* (w/o) *short-circuit* when *block colloc.* is enabled. TPC-H benchmark: 16-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl. 83
6.2 Overall runtime performance: *baseline* vs. *collocation* versions, based on Table 6.3 84
6.3 Average throughput per query: *baseline* vs. *collocation* versions, related to Table 6.3. 85
6.4 Overall runtime performance: *baseline* vs. *collocation* versions, based on Table 6.4 86
6.5 The *worker nodes* and (partition) *responsibilities involved* in query execution during **(a)** and **(b)**. We use *red* color to emphasize the latter, plus the amount of threads per worker to satisfy the *32 mpl*. Besides, we also show the state *before the failure*. The *local responsibilities* are represented with *black* color and the *partial* ones, from the two new nodes, with *grey* color. 87
6.6 Overall runtime performance: *collocation* vs. *drm* versions, based on Table 6.5. . 88
6.7 Overall runtime performance: Q1, Q4, Q6, Q9, Q12, Q13, Q19, and Q21, based on Table 6.5 89
6.8 Average throughput per query: *collocation* vs. *drm* versions, related to Table 6.5. 90
6.9 The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.6. We use *red* color to highlight the latter information. The metadata for *partition locations* (see Section 3.2) is shown on the left and the *responsibility assignment* on the right. 92
6.10 The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.7's **half-set** busyness runs. We use *red* color to highlight the latter information. 93
6.11 The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.8's **half-set** busyness runs. We use *red* color to highlight the latter information. 94
6.12 The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.9's **half-set** busyness runs. right. We use *red* color to highlight the latter information. With *blue* color we mark the *remote* responsibilities being assigned at runtime due to overloaded workers. 96
6.13 Average CPU load (not. from *[min* to *Max]*) for *drm*, while running Q1 and Q13 side by side with 16, 32 and 64 Mappers and Reducers, on a **full-set** in left and **half-set** in right. 96

6.14 Average CPU load (not. from *[min* to *Max]*) for *drmy*, while running Q1 and Q13 side by side with 16, 32 and 64 Mappers and Reducers, on a **full-set** in left and **half-set** in right. 97

List of Tables

1.1 A short summary of the related research in Hadoop-enabled MPP databases (including Vectorwise), comprising their architectural advantages and disadvantages. Abbreviations: tuple-at-a-time (tat), just-in-time compilation (jit), vectorized (vz). 7

2.1 Comparison between local reads through DataNode, without short-circuit (w/o), and direct local reads, with short-circuit (w/), using the TestDFS I/O benchmark on 1 and 10 "Rocks" (Section 2.5) cluster nodes; the results are averaged over two benchmark runs. 24

3.1 Baseline (default policy) vs. collocation (custom policy) – execution times in 3 situations (1) healty state, (2) failure state (2 nodes down), (3) recovered state. Queries 1, 4, 6, and 12 on partitioned tables, 32-partitioning, 8 workers / 10 Hadoop "Rocks" nodes, 32 mpl. 39

3.2 Baseline (default policy) vs. collocation (custom policy) – execution times in 3 situations (1) healty state, (2) failure state (2 nodes down), (3) recovered state. Queries 1, 4, 6 and 12 on partitioned tables, 16-partitioning, 8 workers / 10 Hadoop "Rocks" nodes, 32 mpl. 41

4.1 **sin** vs **drm**: TPC-H scale factor 100, with 12-partitioned tables and replication degree 2, on a 4-node Vectorwise on Hadoop cluster (internal nodes, from which 3 nodes were selected as workers. The *mpl* is set to 24. 62

4.2 **sin** vs **drm**: Q6 **cold-runs**, TPC-H scale factor 100, with 12-partitioned tables and replication degree 2, on a 4-node Vectorwise on Hadoop cluster (internal nodes, from which 3 nodes were selected as workers. The *mpl* is set to 24. 65

6.1 Parameters configuration values 81

6.2 Workload runtime and CPU avg./peak load for *TeraSort* benchmark runs on different data-set sizes, number of Mappers/Reducers and node sets. 82

6.3 Query execution times: *baseline* (default policy) vs. *collocation* (custom policy) versions, in 3 situations (1) healthy state, (2) failure state (2 nodes down), (3) recovered state. TPC-H benchmark: 16-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl (W/N = workers/hadoop-nodes). 84

6.4 Query execution times: *baseline* (default policy) vs. *collocation* (custom policy) versions, in 3 situations (1) healthy state, (2) failure state (2 nodes down), (3) recovered state. TPC-H benchmark: 32-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl (W/N = workers/hadoop-nodes). 86

6.5 Query execution times during node failures: colloc. *without* dynamic resource management (not. *collocation*) vs. colloc. *with* dynamic resource management (not. *drm*). TPC-H benchmark: 16-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl, 0.25 core over-allocation, RMax resp. degree 2. Since RMax = 2 we can form two separate worker-subsets. In **(a)** we fail 1 worker from a subset and 1 from the other, in **(b)** we fail 2 workers from just one of the subsets. 88

6.6 Query execution times without Hadoop busyness, no overlapping external workloads: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2. 92

6.7 Query execution times during overlapping external (Hadoop) workloads, 16 Mappers / 16 Reducers: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2. 93

6.8 Query execution times during overlapping external (Hadoop) workloads, 32 Mappers / 32 Reducers: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2. 94

6.9 Query execution times during overlapping external (Hadoop) workloads, 64 Mappers / 64 Reducers: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2. 95

Chapter 1

Introduction

1.1 Background and Motivation

In the recent past of *Big Data Analytics* many companies shifted from a "one size fits all" software stack towards a data-pipeline composed of new generation technologies deployed on commodity clusters, a collaborative approach where workloads were naturally served to the platform that handles them best. Typically, there are two categories of big-data platforms that could be used to build a data-pipeline for storing and processing huge amounts of data, either structured or unstructured. The first category includes MPP[1] analytical databases that are designed to store huge amount of structured data across a cluster of servers and perform fast parallel queries over it. Most of the MPP solutions follow a shared nothing architecture, which means that every node will have a dedicated disk, memory, processor, and a high speed network connection. As these databases are designed to hold structured data, there is a need to extract the structure from raw data using an ETL[2] tool. This is where Hadoop [3], the second category, comes into help. Apache Hadoop is unquestionably the *"de facto"* platform for big-data environments. It is a system for distributing computation among cluster nodes, which, at its core, consists of HDFS [4] and the Map-Reduce framework [5]. The latter is a computational approach [1] that involves breaking large volumes of data down into smaller batches, and processing them separately. A cluster of compute nodes, each one built on commodity hardware, scan multiple batches and aggregate data. Then the nodes' output is shuffled and merged into the final result. We will invariably find Hadoop working as part of a system with an MPP database, because big-data solutions do not fall entirely into either structured or unstructured data categories. Therefore, when combined altogether, this kind of big-data pipeline becomes essentially a connector-based system approach where Hadoop is used to extract structure from data and then load it into the MPP database's own custom format storage in order to enable fast analytical query execution. In practice, this signifies having connectors in place to *ship data back and forth* over network between the Hadoop and MPP database *separate* environments. But, given that both systems handle data at their best, there is a lot more to gain if we perceive them as a single ecosystem. Nowadays, simply no one wishes to begin its big-data journey with two logically separate systems. Connectors between Hadoop and MPP database clusters are quickly going to disappear. Many organizations working on big-data already ought to have an easy to manage data platform that can support complex analytical query workloads, but also batch processing of unstructured data, free-text searches, graph analysis and whatever

[1]MPP: massively parallel processing
[2]ETL: extract transform load
[3]Apache Hadoop: hadoop.apache.org
[4]HDFS: Hadoop Distributed File System, hortonworks.com/hadoop/hdfs
[5]Map-Reduce: wiki.apache.org/hadoop/MapReduce

else may come up in the future. For that purpose, the industry is standardizing on Hadoop as the unifying infrastructure, since MPP clusters are generally non-standardized systems and difficult to configure and manage. There is a tremendous interest in leveraging the Hadoop ecosystem instead, trying to make MPP solutions *converge* around its HDFS and YARN [1] [4] infrastructure, and thus *coexist with a Hadoop cluster*. This shift to modern MPP databases on top of Hadoop opens at least two new research directions: using *HDFS for storage management* and *YARN for resource management*. The first one, brings MPP databases the support to run query workloads directly on top of HDFS stored tables, using on-disk custom file formats. This topic has been receiving a lot of attention lately, from projects such as Parquet [2] and ORC-File [3]. On the other hand, the second direction is less explored (or not at all). Nevertheless, it is as equally important as storage management and they both work in tandem if we think about *data-locality* and *failover* aspects. Resource management, if tackled from the right angle, can give us the advantage of sharing the same cluster resources between different application platforms (e.g. Map-Reduce jobs and query workloads), and most importantly, it will comply with Hadoop's standards and its system administrators' know-how.

Therefore, the main question we address in this thesis is: *How can we efficiently share compute resources within a Hadoop environment?* More specifically, how can we share compute resources between an MPP analytical database and other computing frameworks, like Map-Reduce, in order to achieve good utilization within the same Hadoop cluster? Not for so long, Apache started to address this question in their undergoing YARN [4] project (as part of Hadoop NextGen), which enables resource management for heterogeneous parallel applications running in a Hadoop cluster. In older Hadoop release versions, the scheduler was purely a Map-Reduce job scheduler. If you wanted to run multiple parallel computation frameworks on the same cluster, you would have to statically partition all the resources or hope that the same resources given to a Map-Reduce job would not have also been allocated by another framework's scheduler, causing Operating Systems to thrash. With YARN's new design, the scheduler can now handle compute resources for different applications (mixed-workloads) running on the same cluster, which should allow for more multi-tenancy and a richer, more diverse Hadoop ecosystem. Therefore, our research focus is to *achieve dynamic resource management in Vectorwise on Hadoop* [2].

1.2 Related Work

To offer a better understanding of our research topic, we start this section with a short overview of the few most important key features in large-scale data analytics system, as introduced in [5]. This is just an attempt to cover the design principles and core features of a database system from a wide perspective, filling the gap between the more classical category and the new Hadoop-enabled category (SQL-on-Hadoop). We then describe some of the technological innovations in Hadoop-enabled MPP architectures that have each spawned a distinct (product) system for data analytics. The last part of this section draws the attention towards resource management in MPP databases and in a Hadoop environment. We focus on systems for large-scale data analytics, namely, the field that is called Online Analytical Processing (OLAP) as opposed to Online Transaction Processing (OLTP).

Data Model. A data model provides the definition and the logical structure of the data, and determines in which manner it can be stored, organized, and manipulated by the system. The most popular example of a data model supported by most parallel database systems is the *relational model* (which uses a table-based format), with its row-based or the more advanced columnar storage formats, whereas most systems in the Map-Reduce categories permit data to

[1]YARN: Yet Another Resource Negotiator

[2]Vectorwise on Hadoop (Prject Vortex) presentation: www.slideshare.net/Hadoop_Summit/actian-vector-on-hadoop-first-industrialstrength-dbms-to-truly-leverage-hadoop

be handled in any *arbitrary format*, even flat files. A relational database consists of *relations* (or, *tables*) that, in turn, consist of *tuples*. Every tuple in a table conforms to a *schema* which is defined by a fixed set of attributes. The data model used by each system is closely related to the query interface exposed by the system, which allows users to manage and manipulate the stored data.

Storage Layer. At a high level, a storage layer is simply responsible for persisting the data as well as providing methods for accessing and modifying the data. The design, implementation and features provided by the storage layer used by each of the different system categories vary greatly, especially as we start comparing systems across the different categories. For example, classic parallel databases use integrated and specialized data stores that are tightly coupled with their execution engines, whereas Map-Reduce systems typically use HDFS as an independent distributed file-system for accessing data. Only recently, with the outcome of Hadoop-enabled architectures, MPP database systems started to leverage HDFS as their native storage layer, using columnar data formats [2, 3] that are specifically tailored for its I/O capabilities.

Query Optimization. In general, query optimization is the process a system uses to determine the most efficient way to execute a given query by generating several alternative, yet equivalent, execution plans. The techniques used for query optimization in the systems we consider are very different in terms of: (i) the space of possible execution plans (e.g. relational operators in databases versus configuration parameter settings in Map-Reduce systems), (ii) the type of query optimization (e.g. cost-based versus rule-based), (iii) the type of cost modeling technique (e.g. analytical models versus models learned using machine-learning techniques), and (iv) the maturity of the optimization techniques (e.g. fully automated versus manual tuning).

Query Execution. When a database system receives a query for execution, it will mostly convert it into a physical plan for accessing and processing the query's input data. The execution engine is the entity responsible for actually running a given execution plan in the system and generating the query result. Most query interpreters follow the so-called Volcano iterator-model [6], in which each operator implements an API that consists of *open(), next() and close()* methods. In the original proposal, the *next()* call produces a *tuple-at-a-time*. The query evaluation follows a *pull model*: the *next()* operator is applied recursively on the operator tree from the root downwards, while the resulted tuples being pulled upwards.

However, it has been observed that the *tuple-at-a-time* model leads to interpretation overhead: the situation that much more time is spent in evaluating the query plan than in actually calculating the query result, affecting the recent innovations in modern CPUs [7]. MonetDB [8] reduced this overhead by using bulk processing instead, making each operator fully process its input and only then invoke the next execution stage. This idea has been further improved in the X100 project [9] and later evolved into *vectorized execution*, a form of block-oriented query processing [10] in which the *next()* method produces small (typically 100-10000) single-dimensional arrays of tuples (called *vectors*), rather than a single tuple. This type of tuple representation is easily accessible for modern CPUs and has the consequence that the percentage of instructions spent in interpretation logic is reduced by a factor equal to the vector-size [11]. Another popular method used by today's execution engines in order to eliminate the overhead of interpretation is *just-in-time query compilation*. Upon receiving a query for the first time, the query processor compiles (a part of) the query into a routine that gets subsequently executed. Query compilation removes the interpretation overhead altogether. A compiled code can always be made faster than an interpreted code and it is reusable for queries that are repeated with different parameters.

In the MPP systems that we consider, the execution engine is also responsible for parallelizing the computation across large-scale clusters of machines and setting up inter-machine communication to make efficient use of the network and disk bandwidth. The final execution plan is composed of operators that support both *intra-operator* and *inter-operator* parallelism, as well as mechanisms to transfer data from *producer* operators to *consumer* operators.

Scheduling. Given the distributed nature of most data analytics systems, scheduling the query execution plan makes it an important part of the system. Systems must now take several scheduling decisions, including scheduling where to run each computation, scheduling inter-node data transfers, as well as scheduling rolling updates and maintenance tasks.

Resource Management. Resource management primarily refers to the efficient and effective use of a cluster's resources based on the resource requirements of the queries or applications running in the system. In addition, many systems today offer elastic properties that allow users to dynamically add or remove resources as needed according to workload requirements.

Scheduling and resource management are not only important, but also very challenging. Clusters are exposed to a wide range of applications, that have highly diverse characteristics and performance requirements. For example, Map-Reduce applications are usually characterized by long running jobs. These jobs desire shorter turnaround time - time from submission to completions. Queries in database systems for analytical or transactional workloads, on the other hand, are much shorter-lived and much more interactive. Further, these environments need to support a large number of users simultaneously. As a result, the underlying system needs to respond to these kind of workloads as soon as they arrive, even at the cost of stretching their overall turnaround time slightly. Managing both types of workloads at the same time makes it even more difficult for a cluster resource manager, especially when each workload belongs to a different applications (e.g. running Map-Reduce jobs and OLAP queries). Scheduling and resource management needs to take numerous system parameters, such as CPU speed, memory size, I/O bandwidth, network bandwidth, context switch overhead, etc, into account. These metrics are often inter-related and can affect the choice of an effective scheduling strategy. Depending on the characteristics of the system (on-line vs off-line, closed vs open, etc) and the nature of the jobs/queries (preemptive vs non-preemptive, service times with known or unknown distribution, prioritized vs equal jobs, etc), several static and dynamic scheduling algorithms have been proposed [12]. In parallel database systems, however, the consensus is that dynamic load-balancing is mandatory for effective resource utilization [13, 14]. Various algorithms have been proposed for these kind of systems depending on the different levels of parallelism: inter-transaction, inter-query, inter-operator and intra-operator parallelism [15].

Failover. Machine failures are relatively common in large clusters. Hence, most systems have built-in failover functionalities that would allow them to continue providing services, possibly with performance degradation, in the face of unexpected events like hardware failures, software bugs, and data corruption. Examples of typical failover features include restarting failed tasks (or services) either due to application or hardware failures, recovering data due to machine failure or corruption, and using speculative execution to avoid stragglers.

System Administration. System administration refers to all tasks where additional human effort may be needed to keep the system running while the system serves the needs of multiple users and applications. Common activities under system administration include performance monitoring and tuning, diagnosing the cause of poor performance or failures, capacity planning, and system recovery from temporary and permanent failures (e.g. failed disks).

1.2.1 Hadoop-enabled MPP Databases

In this section we discuss the research in the literature that constitutes the state of the art in Hadoop-enabled MPP databases. A summary of the system's architectural advantages and disadvantages is given in Table 1.1.

Hive. One of the first initiatives (in 2008) to bring the familiar concepts of tables, columns, partitions and a subset of SQL to the unstructured world of Hadoop is Hive [16], an open-source data warehousing solution built entirely on top of Hadoop and HDFS. Hive supports queries expressed in a SQL-like declarative language called HiveQL statements, which are compiled and broken down into individual Map-Reduce jobs that are later executed with Hadoop. With HiveQL users can also plug in custom Map-Reduce scripts into their queries. The language includes a type system with support for tables containing primitive types and as well as more complex collections like arrays, maps and structs. The latter ones can be nested arbitrarily in order to construct more complex compositions. Hive also includes a system catalog called Metastore that is stored on HDFS and contains table schemas and statistics, useful in data exploration, query optimization and query compilation. The underlying IO libraries can be extended to query data in custom formats, thus allowing users to implement their own types and functions. Even so, the HDFS data files do not have a custom storage format, nor a row-based or columnar-based, which makes the query execution very slow and inefficient. Moreover, Hive does not support in-memory buffering/caching mechanisms. It is known to be a one-at-a-time data analysis framework, where the data has to be read from disk and parsed again each time a user performs a query; HDFS tuples cannot be passed as intermediate results. The per-query Map-Reduce overhead prevents the ability of this technology to process queries interactively, placing Hive in the "batch processing" category of computational frameworks. Yet, with the delivery of the Stinger project [1] in June 2014, the last Hive release (Hive 0.13) got enhanced with columnar storage, more intelligent data placement to improve scan performance, and vectorized execution [17]. Besides, on the project's roadmap one of the goals was also to integrate Hive with Tez [2], a modern implementation of Map-Reduce that allows interactive queries (reducing the processing overhead) and in-memory materialized results.

HadoopDB. HadoopDB [18] is another example of an open source Hadoop-enabled database project, initiated at Yale in 2009, whose architecture is of a hybrid system. The basic idea behind HadoopDB is to use Map-Reduce as the communication layer above multiple nodes running single-node DBMS instances. Queries are expressed in SQL, translated into Map-Reduce by extending the existing Hive components, and as much work as possible is pushed into the underlying layer of database; the rest of the query is processed using the more generic Map-Reduce framework. For a Map-Reduce job in order to communicate with the database layer the authors implemented a custom database connector – an interface between independent database instances and TaskTrackers, usually mapped 1-to-1 on any node in the cluster. So, each Map-Reduce job supplies the connector with SQL query and connection parameters such as: which JDBC driver to use, query fetch size and other query tuning parameters. The connector connects to the database, executes the SQL query and returns results as key-value pairs. Therefore, HadoopDB resembles a shared-nothing parallel database with Hadoop providing services necessary for scalability. The creators of HadoopDB have since started a commercial software company (Hadapt) to build a system that unites Hadoop/Map-Reduce and SQL, a Postgres database is placed in nodes of a Hadoop cluster. Altogether, combining Map-Reduce and Postgres has the following drawbacks. First, using Map-Reduce as the communications layer adds a big overhead to query execution and makes it very slow. In [19], for the authors' experimental setup, it takes an average of 35 seconds per query just to get it started. This is a long-standing complaint about Map-Reduce: high-latency, batch-mode response is painful for lots of applications that want to process and analyze data in critical time. Data communication happens via HDFS: latency is introduced as extensive network I/O is involved in moving data around, e.g. in the reduce phase. That aside, since there is a sort and shuffle phase after the map phase all the outputs always have to be sorted and sent to the reducers. So, the execution time depends on the slowest phase among all three map, reduce and shuffle phases. Second, since Postgres needs to store data on local files we have to load all the data from HDFS to the engine's storage system before running any queries. This makes loading data a cumbersome slow process, a complete pre-load and data organization leads to the worst initial response time of about 800 seconds [19]. On the other hand Postgres was not built with replication in mind (is not relying on HDFS for that purpose), which means HadoopDB cannot handle failovers and so, the database's availability has to suffer in case of failures.

[1] Stinger project: hortonworks.com/labs/stinger
[2] Tez: hortonworks.com/hadoop/tez

Hawq. A commercial product that recently entered the MPP Hadoop-like database market is Hawq [20, 21], a mix between EMC's Greenplum [22] MPP database and Hadoop v2, part of a new EMC on Hadoop stack called the Pivotal Hadoop Distribution. In contrast with Hive, Hawq tries to take all of the inherent scalability and replication benefits from HDFS; it does not embrace a Map-Reduce execution style nor the Hive's query engine, or other friendly query-interfaces for Hadoop. The system allows to be queried with any standard SQL syntax or any tool that speaks SQL, using a custom query engine based on Greenplum MPP engine (a mature optimized version of Postgres engine). Also, Hawq's approach is different from HadoopDB's invisible loading [19]. The invisible loading algorithm "invisibly" rearranges data from slow, file-system style data layout to fast, relational data-layout. In Hawq, the same data is pulled from HDFS into Greenplum's execution engine every time, i.e. if data is stored as a flat file in HDFS, then when the query is over, it will still be stored as a flat file in HDFS. However, the announcement out of Hawq Pivotal HD is still a port of a decade-old technology (meaning the Greenplum's customized Postgres database) that is albeit rather slow for OLAP queries, has its own, independent, non-integrated schema metadata layer which cannot share information with the rest of the Hadoop platform (it exists only on the Master node, the workers are stateless), and supports just limited update and delete operations (e.g. single row update/delete).

Impala. One of the main contributors of the Hadoop project, Cloudera, has been developing as well a distributed parallel SQL query engine that blends into Hadoop's stack, turning it into a real-time fault-tolerant distributed query engine. The Impala [23] query engine can run directly on top of HDFS, enabling users to issue low-latency SQL queries on data stored in HDFS using custom formats, such as Parquet columnar storage format [2], or the HBase tabular overlay. All these changes bring Impala closer to the Hadoop ecosystem. With Impala, Cloudera is basically replacing parts of Hive's and HBase's components with custom built ones, while still being compatible with Hive and HBase APIs. The result is that large-scale data processing (via Map-Reduce) and interactive queries can be done on the same system using the same data and metadata, removing the need to migrate data sets into specialized database storage systems and/or proprietary formats simply to perform analytical queries. The underlying SQL engine in Impala is not reliant on Map-Reduce as Hive is, but rather on a single-core (lack of parallelism) distributed database stack that has a new query planifier, a new query coordinator, and a new query execution engine. Furthermore, as an improvement to the last component (the query execution engine), Impala uses just in time query compilation [24]. This significantly improves the CPU efficiency, plus the overall query execution time, and outperforms the traditional interpreted query processing with tuple-at-a-time execution [25]. However, since Imapala is still at the first release versions, it does not have a query optimizer, nor a mature execution engine yet. For instance, the QET [1] currently limits the execution level of subqueries, usually fixed at 2 or 3 levels deep, and won't generate plans that are arbitrarily deep. Also, adding that Hadoop v2 was released in October 2013, it lacks of YARN integration.

Presto. Presto [26] is a distributed SQL query engine developed by Facebook and optimized for ad-hoc analysis at interactive speed. The execution model of Presto is fundamentally different from Hive [27], which translates queries into multiple stages of Map-Reduce tasks that execute one after another and each one reads inputs from disk and writes intermediate output back to it. In contrast, the Presto engine does not use Map-Reduce. It employs a custom query and execution engine with operators designed to support SQL semantics. In addition to improved scheduling, all processing is in memory and pipelined across the network between stages. This avoids unnecessary I/O and associated latency overhead. The pipelined execution model runs multiple stages at once, and streams data from one stage to the next as it becomes available. This significantly reduces end-to-end latency for many types of queries. The Presto system is implemented in Java and certain portions of the query plan are dynamically compiled down to byte code which lets the JVM optimize and generate native machine code. Presto was designed with a simple storage abstraction that makes it easy to provide SQL query capability

[1]QET: query execution tree

against sources from HDFS, HBase and others. Storage plugins (called connectors) only need to provide interfaces for fetching metadata, getting data locations, and accessing the data itself. However, this severely draws down the query processing time since the engine was not design to read optimized custom storage format to avoid unnecessary transformation before executing the physical query plan. Also, other restrictions at this stage are the size limitations on the join tables and cardinality of unique keys/groups. The system also lacks the ability to write output data back to tables (currently query results are streamed to the client).

Table 1.1: A short summary of the related research in Hadoop-enabled MPP databases (including Vectorwise), comprising their architectural advantages and disadvantages. Abbreviations: tuple-at-a-time (tat), just-in-time compilation (jit), vectorized (vz).

Approach	Row/columnar format	HDFS/DBMS storage	Fast/slow execution	Mature optimizer	Updates
Hive	none (columnar in 0.13, ORCFile)	HDFS	slow (tat) (vz in 0.13)	no	no
HadoopDB (Hadapt)	row	DBMS (copy-out)	slow (tat)	no (Postgres)	no
Hawq (PivotalHD)	row	HDFS	slow (tat)	yes (customized Postgres)	limited
Impala	columnar (Parquet)	HDFS	fast (jit)	no	no
Presto	row	HDFS	slow (tat)	no	no
Vectorwise	columnar (custom)	HDFS	fast (vz)	yes	yes

1.2.2 Resource Management in MPP Databases

Resource management in parallel database systems happens at two levels: at the level of *individual nodes* and at the level of the *entire system*. Most parallel database systems are implemented by building on the software of a centralized database system at each node. Thus, many configuration parameters can be set independently at each node, e.g. size of memory buffers, maximum number of concurrent threads (or *mpl* [1]), and the amount of memory. This feature allows the database administrator to tune the performance of each individual node based on the differences in hardware capacities, database partition sizes, and workloads across the nodes in the system. In literature, the resource management that happens at the level of the entire system is divided into two main classes: *external* and *internal*. The general idea of external workload management is to control the number of queries that can be run concurrently in the system, *admission control*, so that every query gets some guaranteed fraction of the total resources during execution [28, 29, 30]. On the other hand, internal workload management systems typically control the available resources, such as CPU or main memory, and assign them to queries. A known technique is *workload differentiation*, the system tries to identify which queries are short-running and which ones are long-running and allocates resources appropriately in order to meet the respective requirements of these query classes [31, 32, 33].

Recent work in DBMS workload management has mainly focused around admission control. Schroeder et al. [28, 29] propose an external queue management system that schedules queries based on defined service-levels per query-class and a number of allowed queried in the database, the so called multiprogramming level. Niu et al. [30] approach manages a mixed workload of OLTP and OLAP queries by controlling the OLAP queries based on the response times of OLTP queries. The work of Dayal, Kuno and Krompass et al. [31, 32, 33] focuses on the management of analytical queries in the area of data warehouses. It can be considered as a mixture of

[1]mpl: maximum parallelism level

external and internal workload management. Besides admission control based on extensive query run-time prediction, they additionally control the execution of queries and propose the possibility to kill long running queries in case higher priority queries are present.

Another important dimension of resource management is how the system can scale up in order to continue to meet performance requirements (quality of service) as the system load increases. The load increase may be in terms of the size of input data stored in the system and processed by queries, the number of concurrent queries that the system needs to run, the number of concurrent users who access the system, or some combination of these factors. The objective of shared-nothing parallel systems is to provide linear scalability. For example, if the input data size increases by 2x, then a linearly-scalable system should be able to maintain the current query performance by adding 2x more nodes. Partitioned parallel processing is the key enabler of linear scalability in parallel database systems. One challenge here is that data may have to be repartitioned in order to make the best use of resources when nodes are added to or removed from the system.

1.2.3 Resource Management in Hadoop Map-Reduce

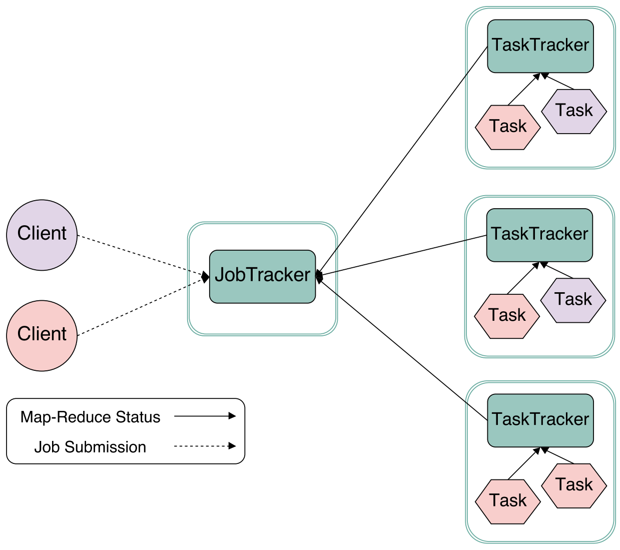
We now briefly review some of the related work about resource management in a Hadoop (Map-Reduce) environment. There have been several efforts that investigate efficient resource sharing while considering fairness constraints. For example, Facebook's fairness scheduler [34] aims to provide fast response time for small jobs and guaranteed service levels for production jobs by maintaining job "pools" each of which is assigned a guaranteed minimum share and by dividing excess capacity among all jobs or pools. Yahoo's capacity scheduler [35] supports multi-tenancy by assigning capacities to different job queues, so each job queue gets a fair share of the cluster resources. Zaharia et al. [36] developed a scheduling algorithm called LATE that performs effective speculative execution to reduce the job running time in Hadoop heterogeneous environments; it uses the estimated remaining execution time of tasks as the guideline to select tasks to speculate and avoids assigning speculative tasks to slow nodes. However, without a data-locality aware placement scheme, previously mentioned approaches have only limited opportunities for optimizations during task placement. To improve data-locality, several enhancements have been proposed. In an environment where most jobs are small, delay scheduling [37] can improve data-locality by delaying the scheduling of tasks that cannot achieve data-locality by a short period of time. This seems a simple technique to provide better locality, but it does not consider global scheduling, and so it loses the opportunity for achieving better overall performance. Purlieus [38] tries to solve the fundamental problem of optimizing data placement so as to obtain local execution of the jobs during scheduling, minimizing the cross-rack traffic during both map and reduce phases. To do so, it categorizes Map-Reduce jobs into three classes: map-input heavy, map-and-reduce-input heavy and reduce-input heavy, and proposes data and virtual machine placement strategies accordingly to minimize the cost of data shuffling between map tasks and reduce tasks. Similar in approach is CAM [39], a topology aware resource manager for Map-Reduce applications in the cloud that also proposes a three level approach to avoid placement anomalies due to inefficient resource allocation: placing data within the cluster that run jobs that most commonly operate on the data, selecting the most appropriate physical nodes to place the set of virtual machines assigned to a job and exposing, otherwise hidden, compute, storage and network topologies to the Map-Reduce job scheduler. CAM uses a minimum-cost flow network based algorithm that is able to reconcile resource allocation with a variety of other constraints, such as storage utilization, changing CPU load and network link capacities. Unlike Purlieus, CAM's minimum-cost flow based approach is able to considers both VM migration as well as delayed scheduling to arrive at a global optimal placement. Quincy [40] is a Dryad resource scheduler for concurrent jobs that tackles the conflict between data-locality and fairness by converting the scheduling problem to a graph that encodes both network structure and waiting tasks and solving it with a minimum-cost flow solver.

Though it uses similar graph techniques, it differs from CAM in terms of problem space and associated flow network construction. Quincy balances between fairness and data-locality, while CAM focuses on optimizing data/VM placement of Map-Reduce applications in the cloud. As a result, the factors encoded in the flow graph (VM closeness, etc.) are quite different from that of Quincy's.

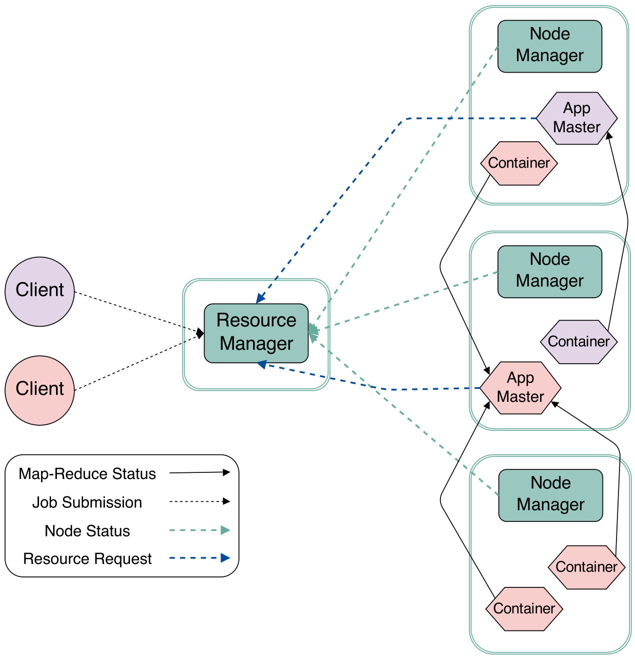
However, not all these ideas (described above) are suitable for a Hadoop-enabled MPP architecture. At least three assumptions considered by some of the authors are not applicable to these architectures: (1) workloads can be delayed, and so the waiting time can be taken into account when scheduling and managing compute resources, (2) only Map-Reduce workloads run in a Hadoop cluster (multi-tenancy is not considered), and (3) job profiling can be done beforehand on a small subset of data in order to categorize the workload from the I/O-, CPU-, network-perspective.

1.2.4 YARN Resource Manager (Hadoop NextGen)

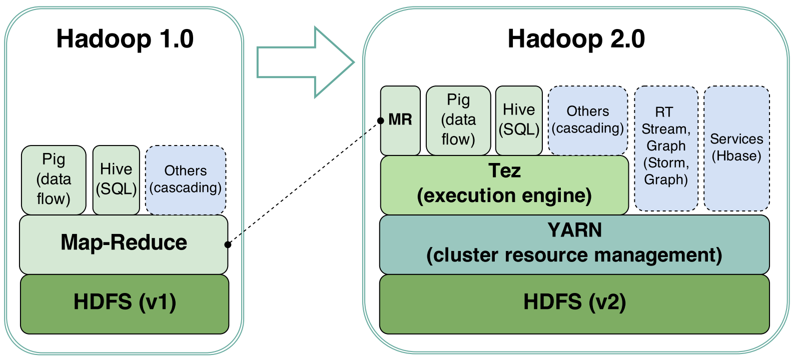
In the previous Hadoop version (v1), figure 1.1a, each node in a cluster was statically assigned a predefined number of map and reduce slots for running map and reduce tasks altogether [?], meaning that the allocation of cluster resources to Map-Reduce jobs was done in the form of these *slots*. This static allocation model had the obvious drawback of lowered cluster utilization, since slot requirements vary during a Map-Reduce job life cycle. There is a high demand for map slots when the job starts, whereas there is a similar demand for reduce slots towards the end. However, the simplicity of this scheme facilitated the resource management performed by the *Job Tracker*, in addition to the scheduling decisions. Hadoop Job Tracker had multiple roles, such as managing the cluster resources as well as scheduling, monitoring, and managing the life-cycle of Map-Reduce jobs. On the other hand, the *Task Tracker* had fewer and simpler responsibilities, such as starting/tearing-down or pinging the job-application to check if it is still alive. Hadoop NextGen (v2), figure 1.1b, also known as YARN [4], separates the above functionalities into: a global *Resource Manager (RM)*, responsible for allocating resources to running applications, a per-application *Application Master (AM)* to manage the application's life-cycle, including its resource requests towards *RM*, and also a per-machine *Node Manager (NM)* to handle the user processes and the available resources of that node. The AM is a framework-specific library that negotiates resources from the RM and works with the NM(s) to execute and monitor an application's set of tasks. To build a custom YARN Application from scratch a user needs to implement the API of the *YARN Client Application*. The transition from Hadoop v1 to NextGen v2 is depicted in figure 1.1. Historically, Map-Reduce applications were the only ones that could have been managed by Hadoop (v1) at that time, due to its limited resource management capabilities. Today, due to YARN's new design, Map-Reduce is just one of Hadoop's application frameworks. Hadoop NextGen permits building and deploying distributed applications using other frameworks as well, see figure 1.1c. The resource allocation scheme in YARN addresses the static allocation issues of the later Hadoop version by introducing the idea of *resource containers*. A container represents a specification of a node compute resources (or attributes), such as CPU, memory, and in future, disk and network bandwidth. In this model, only a minimum and a maximum for each attribute are defined, and AMs can request different container sizes at different times with attribute values as multiples of the minimum. Therefore, YARN provides resource negotiation and management for the entire Hadoop cluster and all the applications running on it. This leaves us the believe that, once integrated with YARN, non-Hadoop distributed applications (e.g. Vectorwise query workloads) can run as first-class citizens on Hadoop clusters by sharing resources side-to-side with Hadoop-native applications.



(a) Job submission in Hadoop v1



(b) Job submission in Hadoop v2



(c) Hadoop stack: v1 to v2

Figure 1.1: Hadoop transition from v1 to v2

1.3 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 performance of modern hardware (e.g. CPU cache, multi-core parallelism, SIMD instructions, out-of-order execution) in conjunction with a scan-optimized I/O buffer manager, fast transactional up dates and a compressed NSM/DSM storage model [7].

The Volcano Model. The query execution engine of Vectorwise is designed according to the widely-used Volcano iterator model [6]. A query execution plan is represented as a tree of *operators*, hence often referred to as a *query execution tree* (QET). Operators are an implementation of constructs responsible for performing a specific step of the overall required query processing. All operators implement a common API, providing three methods: *open()*, *next()* and *close()*. Upon being called on some operator, these methods recursively call their corresponding methods on the QET's children operators, starting from the root operator downwards. The *open()* and *close()* are one-off complementary methods that perform initialization work and resource release, 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, at the root operator the final results being returned. The benefits of using such a model include flexibility and extensibility, as well as a demand-driven data flow that guarantees that the whole query plan is executed within a single process and without unnecessary intermediate materialized data.

Multi-core Parallelism. Vectorwise (single-node SMP version) is 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 (not. Xchg)* [41].

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* [8] 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 features by adapting the QET operators in order to process and return not one, but a fixed number of tuples (from 100 - 10000, default is 1024) whenever the *next()* method is called. This block 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 a processor. The processing of vectors is done by 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 the performance features of modern CPUs (e.g. superscalar CPU architecture, instruction pipelining, SIMD instructions) [7], as well as compiler optimizations (e.g. loop unrolling).

The rest of this section is organized as follows. We first give an overview of the Ingres-Vectorwise design in Section 1.3.1. Next, in Section 1.3.2, we discuss the distributed Vectorwise MPP database version. Last but not least, Section 1.3.3 presents the Hadoop-enabled (Vectorwise on Hadoop) architecture.

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

1.3.1 Ingres-Vectorwise Architecture

Vectorwise is integrated within the Ingres [1] open-source DBMS. A high-level diagram of the whole system's architecture, showing the main architectural components, is depicted in Figure 1.2.

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 perform. Also, Ingres keeps track of the database schemas, metadata and statistics (e.g. cardinality, distribution) and uses this information for computing and providing to Vectorwise an optimal query execution plan. Vectorwise can be seen as the database system's engine, being responsible for the whole query-tree execution, buffer management and storage. Its main components are described below.

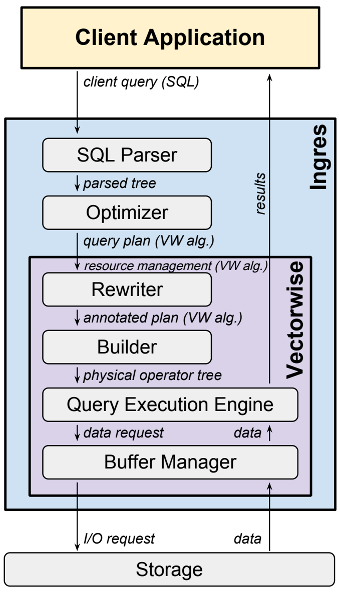


Figure 1.2: The Ingres-Vectorwise (single-node) High-level Architecture

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*. The 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.

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 of the Volcano model. The latter (producing the physical tree) corresponds to calling the *open()* method on all operators in a depth-first, post-order traversal of the query execution tree. Also, this is when most of the memory allocation and initialization 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.

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

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.

Resource Manager. When a query is received in (the SMP single-node version of) Vectorwise, an *equal-share* of resources is allocated (at the beginning) for it. This amount of resources, namely the number of threads, is called the *maximum parallelism level* 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. 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.

The reason behind an *equipartition* strategy for resource management is that the queries processed by the execution engine are all independent, their service time distribution is unknown and differences in workloads may be significant. Moreover, queries have different degrees of resource utilization and of parallelism at different levels of the query plan. It has been proven that it can be an efficient strategy for different class of workloads and service time distributions [42], which is the typical case for Vectorwise queries.

The information about the number of queries running in the system and the number of cores used by each query is stored in the Resource Manager's state. Note that this information is a mere approximation of the CPU utilization. Depending on the stage of execution and/or the number and size of the Xchg operator's buffers, a query can use less cores than the number specified, but it can also use more cores for shorter periods of time.

1.3.2 Distributed Vectorwise Architecture

Most organizations demand from their OLAP capable DBMS systems a small response time to queries on volumes of data that increases at a constant pace. In order to comply with these requirements, the most successful commercial database service providers have developed MPP database solutions meant to run on supercomputers or clusters.

The Vectorwise SMP product, though is successful in achieving high performance, low cost and usability, its ability to handle large amounts of data is limited by the fact that it is designed to work on a single machine. The research developed in the master-thesis project [43] shows how this version can be enriched with distributed query processing capability, such that it can be deployed on a computer cluster. To achieve this, both the query execution engine and the query optimizer had to be extended with new functionality, while maintaining the original qualities.

Figure 1.3 presents the high-level architecture of the distributed Vectorwise MPP database as it was designed and implemented by the authors of [43], currently Actian software engineers. The architecture relies on a *"Virtual" Shared Disk* storage layer and follows a *Master-Worker* pattern, in which the Master node is the single point of access for client applications. The main advantage of a Master-Worker design is that it hides from clients the fact that the DBMS they connect to actually runs on a computer cluster. The Shared Disk (SD) approach was preferable, opposed to Shared Nothing (SN), for ensuring *non-intrusiveness*, *ease of use* and *maintainability*. The main difference between both, SD vs SN, architectures 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. For a more elaborated comparison we recommend reading the second chapter of [43], the author's chosen "Approach" for their research project. 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 maintaining it 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 the distributed MPP version of Vectorwise, it was a too complex task to tackle in a master-thesis project.

Because of the design choice, 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 least. However, as can be seen in the architecture diagram, it has more than one task assigned, 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
- broadcasting them to all worker nodes that are involved

The Vectorwise MPP database 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 before-mentioned processing steps sequentially is acceptable. Moreover, depending on the system load and scheduling policy, the Master may also get involved in query execution to a larger extent than simply producing the distributed plan and gathering the query results.

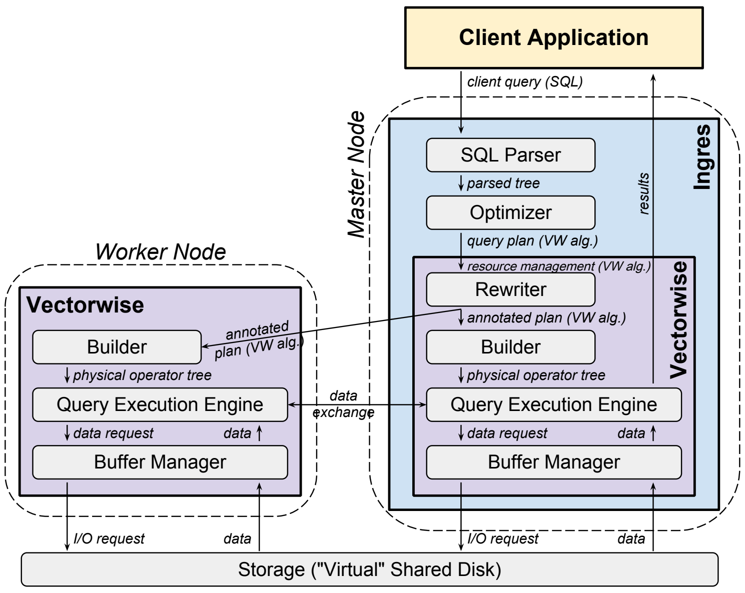


Figure 1.3: The Distributed Vectorwise High-level Architecture

For a distributed Vectorwise MPP solution, the following modules of the single-node version needed to be modified or extended:

Rewriter. A new logical operator (called *Distributed Exchange*) was introduced in query plans, which completely encapsulates parallelism and distribution. The transformations of the parallel rules and the operators' costs were adapted too. The new distributed plan contains both intra-operator and inter-operator parallelism, as well as mechanisms to transfer data from producers to consumers. Also, in order for this module to be cluster-aware, the Rewriter requests information about the current state of the cluster (e.g. the number of cores available on each node, number of running threads, etc) from the Resource Manager.

Builder. Every node is able to build its corresponding parts of a query execution tree and physically instantiate each new operators. Since a single node can run several parts of the query, is required for the Builder to produce a forest of sub-trees, rather than a single rooted tree.

Execution Engine. The *Distributed Exchange* operator was implemented in a way to allow the Master and various Workers to efficiently cooperate (e.g. exchange data, synchronize) when executing a given query. After the execution of a query is finished, this module will release resources in a parallel fashion, similar to the idea employed in the (parallel) Builder, and then inform the Resource Manager about that. For the network-communication layer it was decided to rely on the MPI (Message Passing Interface) library, which is considered to be the *"de-facto"* standard for communication among process that model a parallel program running on a distributed memory system. No other changes were made at this layer.

Resource Manager. Besides the existent functionality of the the single-node Resource Manager, two new policies for query scheduling were introduced for the distributed version of Vectorwise: the *single-node* and *multiple-nodes* distribution policies.

The *single-node distribution policy* (no-distribution) tries to keep the network traffic at a minimum and, at the same time, process linearly more queries than the single-node DBMS version. 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 this policy is to choose *one single node* that is the *least loaded* one in the cluster, calculate the *mpl* relative to the node, apply the transformations to generate a parallel query plan that should run 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 can be sent to the client.

On the other hand, the *multiple-nodes distribution 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: 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. Then, the task is to find the *smallest set of least loaded nodes* that can provide the required parallelism level. 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).

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 *up-to-date* there. To determine on which node to run a received query, the Master node assigns a *load factor* to every node in the cluster:

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

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 (no real usage, e.g. percentage information) 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 $w_t + w_q = 1$ (experimentally determined).

1.3.3 Hadoop-enabled Architecture

Vectorwise has been now extended to bring its data analytics performance to Hadoop. It has a mature RDBMS engine that can perform SQL processing of data stored natively in HDFS, a rich SQL language support, unique update capabilities, and an advanced query optimizer. The Hadoop-enabled architecture aims to scale-out Vectorwise on top of Hadoop/HDFS in order to provide a general-purpose, high-performance, Big Data Analytics product.

High-level overview

The core of the Hadoop-enabled version is the distributed Vectorwise MPP engine, which is based on the research work from [43]. We briefly remind that the authors of [43] have designed and implemented a solution for distributed (multi-core parallel) queries using a *"Virtual" Shared Disk* approach. Although the file system the authors used (GlusterFS [1]) did not performed as expected, they were nevertheless able to prove their core concepts.

The HDFS layer is also perceived as a *"Virtual" Shared Disk*. However, each node in the cluster has its own (physically separated) set of disks and that gives better I/O performance for *local-* data accesses over *remote-* ones. Using partitioned tables with block location affinity in HDFS we can achieve a more flexible shared-nothing and performant system. Therefore, this makes our new (Vectorwise on Hadoop) approach a combination of *a "Virtual" Shared disk with partitioning and affinity*. The Hadoop-enabled architecture, figure 1.4, still follows the traditional Master-Worker pattern. Whenever the database restarts, a new session is initiated and one of the X100 backend workers is elected as the *session-master* (or Vectorwise Master) without any additional actions. This session is kept alive during the whole lifespan of the Vectorwise database to let the Vectorwise Master fulfill its responsibilities, see Section 1.3.2. Within a Hadoop cluster, the NameNode can serve as a node location for Ingres and the Vectorwise Master, if not involved in query processing, and (just a *designated subset* of) the DataNodes can serve, respectively, for the Vectorwise Workers. Otherwise, if the *session-master* is involved in query execution, there would be no difference as such in the node locations because every X100 processes must then have access to a local DataNode (for short-circuit *local reads*). In the hindmost case, the NameNode will share the same location with on of the DataNodes, as illustrated in Figure 1.4. We further refer to the *designated subset of nodes* starting the (session of) X100 backend processes as the *Vectorwise worker-set*. Given that a large-scale Hadoop cluster can have nodes with various hardware configurations, the *worker-set* concept is introduced to help us on selecting (throughout a process that is described in Section 4.1) a set of machines with identical capabilities and thus, achieve a better load-balancing. While it is recommended to have Vectorwise X100 servers installed on many of the DataNodes as possible, in order to chose at will on which nodes we should run the database, is not necessary that all servers should be used at once. The choice of HDFS keeps the system simpler and, though in principle each worker can see the entire system state because of its *global (remote) access*, it enables us to fall back into single-writer algorithms for update operations. It also enables easy implementation of failover functionalities, since in general, if performance is not considered for remote data accesses, no data needs to be moved in case the worker-set changes.

[1]GlusterFS website: www.gluster.org

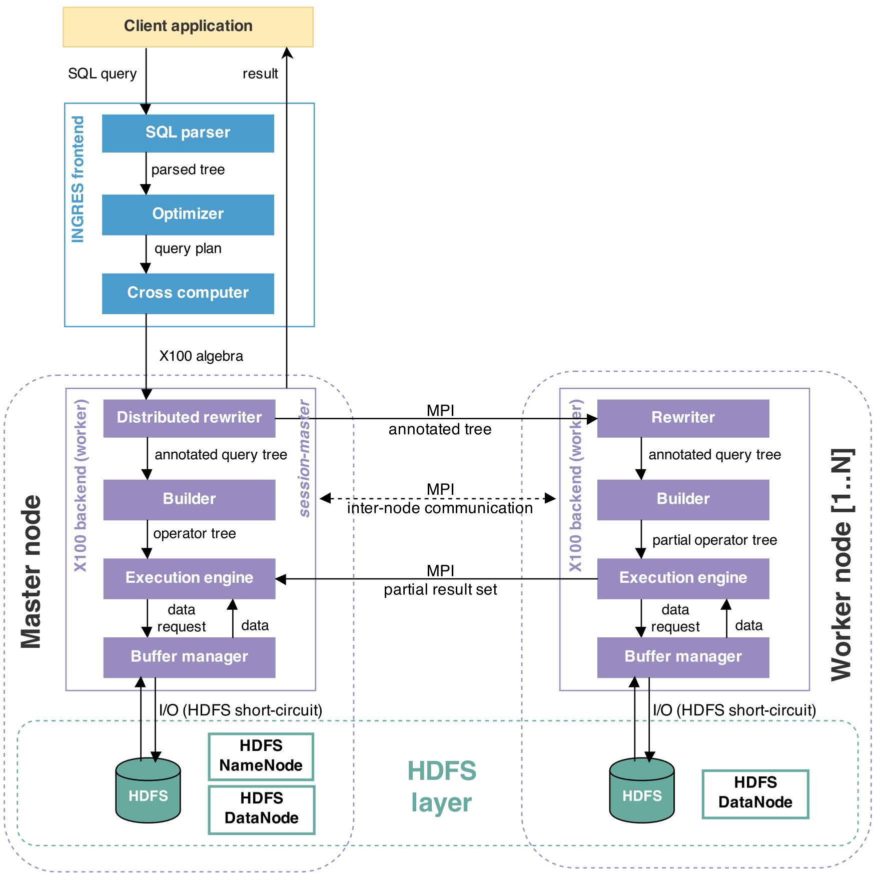


Figure 1.4: The Vectorwise on Hadoop High-level Architecture (the *session-master*, or Vectorwise Master, is involved in query execution).

To offer a better understanding of what is special about Vectorwise on Hadoop, let us review some of the new key concepts introduced in the Hadoop-enabled architecture:

Basic partitioned table support. One thing to take into account, in order to have scalability on large joins, is that all-to-all communication between the Vectorwise worker nodes must be avoided – even Infiniband networks will not support the speed of Vectorwise joins, and in a typical Hadoop scenario we just have a 10Gb Ethernet network. Therefore, some kind of partitioning or clustering is needed that allows to execute joins and aggregation locally on each worker. The idea is to introduce table partitioning that splits a logical table into multiple physical tables on HDFS using hash-partitioning on a column (or a set of columns). Every partition has in principle one responsible worker, and that is the worker which stores (locally) the first replica of that partition. Still, this mapping is not strict, if a node goes down we can make another node responsible for its table partitions. We explain in Chapter 4 more about the previous situation and how it can change if we take as well in account the secondary partition replicas. Table partitioning is introduced during bulk-loading, where each worker gets data determined by a partitioning key. Partitioning is physically implemented by adding union tables to Vectorwise, which is a virtual table (view) made up by the union of physical table partitions. Besides *hash-partitioned (collocated) tables*, Vectorwise on Hadoop supports *global (non-partitioned) tables* as well. The latter of the two, are in-memory (all over the worker-set) cached tables used in general for performing local shared-HashTable hash-joins.

HDFS Database Locations. To store data on HDFS we do need to define a directory structure, a hierarchy of locations to which we can store separately (and isolate) one database's files (logs, tables, metadata, profiles, etc.) from another. Each Vectorwise on Hadoop session starts with a *database location*, an HDFS path where log files, global (non-partitioned) tables, catalog metadata, etc. are stored (e.g. hdfs:///location/name). A database location generally points to *a list of table locations* – of which only one is the *default location* and set to *hdfs:///<database location>/default*, whereas the others are *defined per table* during schema creation. The *list of table locations* is written in a metadata file and stored at the Vectorwise database location. Beside all these, there is also a local (non-HDFS) *work location* that is used for disk– sorting and spilling.

Columnar HDFS storage model. The parallel-rewriter has a rule that transforms a data-loading query having a cluster-load (cluster VWLoad) operator into a distributed query plan, where each node loads into a specified partition of the table. The data is distributed by hashing on key columns. We assume that data is stored in HDFS in multiple files and each file contains data for just a single column. For the cluster VWLoad queries, the rewriter associates files to load operators in a round-robin fashion. However, files may have affinities to particular nodes (and there should be a way of detecting these affinities using the HDFS library). Therefore, in future stages, files will be assigned to operators with affinity. The VWLoad operator used to be (in the single-node version) a utility component in the client-space, a command-line client used to parse and send data through a TCP connection to the local X100 server. Nowadays, in the distributed MPP version, these functionalities have been separated from the utility component and redesigned into a query operator instead. For parsing and loading the data into HDFS, this operator forks a helper (JVM) process that communicates through Unix pipes with the X100 (parent) backend process.

Supporting Updates in HDFS, via Positional Delta Trees (PDTs). Vectorwise implements a fully ACID–compliant transactional database with multi-version read consistency. Any new transaction will see all previously committed transactions, both small incremental transactions and large bulk data loads. Changes are always written persistently to HDFS into a *write ahead transaction log* (WAL) before a commit completes. This will always ensure full transaction recoverability. Nonetheless, one of the biggest challenges with HDFS is that it is not designed for incremental updates. HDFS files are append-only and so, in place updates are not allowed. Vectorwise on Hadoop addresses this challenge with its high-performance in-memory *Positional Delta Trees* [44] (PDTs), which are used to store small incremental changes (inserts that are not appended), as well as updates and deletes. Conceptually, a PDT is an in-memory structure that stores the position and the change (delta) at that position and permit queries to efficiently merge the PDTs changes with the HDFS stored data. Because of the in-memory nature of PDTs, small DML statements can be processed very efficiently. A background process writes the in-memory changes to disk once a memory threshold is exceeded. Important to mention is that only the *session-master* writes to the WAL (PDTs on-commit). The WAL file is *read-only* for the Workers and *read-write* for the Master. At startup, each worker reads the (global accessible) WAL, but skips the PDTs on the partitions it is not responsible for. If an update is triggered, then the worker nodes will ship their local PDTs changes over the network to the *session-master*, who finally writes all these changes to the WAL.

Query plan on logically partitioned tables. For the query-plan search strategy, the existent approach uses the resource scheduler developed in the MSc thesis [43]. For each new query, the algorithm provides us a list of pairs ($\mathrm{n_x}$, $\mathrm{t_y}$) denoting how many threads/cores ($\mathrm{t_y}$) are allocated for its execution on node ($\mathrm{n_x}$). In order to restrict the search strategy from trying all possible combinations of threads/node (which will lead to an exponential complexity), we need to accept the following: (a) assuming that partitions will be created with almost equal sizes, it only makes sense to try to assign an equal number of resources (in our case threads) to each partition, (b) for every partition, there will be *at least one thread assigned* to it; if this restriction is not set, there would be cases in which a thread will be assigned to scan multiple partitions. The search starts with a feasible (initial) solution and tries all its multiples (e.g. $[(1,3),(2,5)]$: $[(1,6),(2,10)]$: $[(1,9),(2,15)]$: $etc\ldots$). It stops at the first combination which needs more threads (on any machine) than allocated by the resource scheduler. The implementation adds two new multiple-nodes distribution policies for queries: *(1) equal-share for non-partitioned* – similar with the multiple-nodes distribution used in the Shared Disk approach it selects the least loaded nodes and takes their amount of CPU resources until it satisfies the *mpl* and *(2) equal-share for partitioned tables* – it assumes the default round-robin partitioning and equally assigns threads to the operators until the *mpl* is achieved. However, ideally would be to balance the assignment with respect to *data-locality* (i.e. local partitions) and each node's *load factor* (i.e. overloaded worker nodes).

Ingres client side mpi_fork and worker-set selection. In Vectorwise on Hadoop, the Ingres client should query YARN Resource Manager and see which nodes are alive in the cluster. Once we have selected the most resourceful N worker nodes from the entire Vectorwise cluster (consisting of M nodes, M can be equal to N) and have reserved the initial amount of resources (CPU and memory), we can then start the X100 worker-set using an mpi_fork command. For the worker-set initialization process we assume (and is important) that the assignment of partitions across the nodes, i.e. the exact node-locations of each partition, is already known. We refer to this assignment in the next paragraphs as *the responsibility assignment*, or *the node's responsibilities*. Knowing the latter information, each of the X100 backend servers can replay the LOG and skip PDTs, MinMax and Join -Indexes for all the partitions that do not correspond to them (or, better said, that is not responsible of). Among all the worker nodes, only one is chosen to be the *Master*. This node is the one to receive the sequential query (algebra) plan and rewrite it into a distributed parallel one. The final (best-cost) plan initiates the distributed execution on top of the Vectorwise worker-set. Also, as we mentioned above, the Master node is the only one that can change/update the WAL file and handle non-partitioned tables. So far, there is no automatic approach to select the nodes participating in the worker-set. To start a Vectorwise on Hadoop cluster, human intervention is currently needed to explicitly specify the worker nodes. Hence, due to the lack of this feature, the existent Vectorwise on Hadoop architecture cannot handle *transparent failovers* and so, it cannot reassign the responsibilities of the failed nodes to the remaining nodes in the worker-set. This process should be handled transparently by querying the HDFS NameNode to find out where the partitions are located and, using this information to determine the missing replicas, have them re-replicated to other (new) nodes.

1.4 Research Questions

We now address the most important questions related to dynamic resource management in Vectorwise on Hadoop MPP database:

- What metric sets (application, cluster, node -specific metrics) from YARN Resource Manager can be used to manage a parallel database system, likewise Vectorwise on Hadoop?
- How can we get our metric sets and inform YARN about a *specific* resource allocation (e.g. nodes' locations, number of cores, amount of memory)?
- What is the granularity of resources that we can claim with YARN?
- How can we use data-locality to influence YARN's resource allocation?
- What algorithms to compute node assignments are appropriate for us, in three situations:
    1. to determine the initial Vectorwise worker-set?
    2. to dynamically reassign responsibilities to worker-set at startup time (e.g. in case of a node failure)?
    3. to determine the optimal subset of workers to run a particular query?

- What are the challenges in dynamic resource management for Hadoop-enabled MPP database technologies?

1.5 Goals

Through the planning and development stages of our project, we have been seeking to create a prototype that:

- is in control of the HDFS replication layer and always keeps the data local to the worker nodes, especially after node failures when missing replicas have to be re-replicated to other (new) nodes
- transparently handles failovers by reassigning the responsibilities of the failed nodes to the remaining nodes (or new ones) in the worker-set, such that the load is spread as evenly as possible with respect to data-locality
- increases the performance during failovers, favoring local-reads over remote-reads
- achieves dynamic resource management, with focus on load-balancing and resource utilization
- is integrated with YARN, aiming towards an elastic approach to increase/decrease the resource requirements

1.6 Basic ideas

The outline below is meant to give the reader an idea of the directions according to which the project/prototype was carried out:

- control data-locality by instrumenting the HDFS block placement policy
- implement an algorithm to decide node responsibilities at startup with respect to data-locality and load-balancing factors
- similarly, decide the subset of nodes and the amount of resources we can use to run a query; we refer to it as the *resource footprint* (or the resource requirements) of a Vectorwise query
- use YARN to acquire/release resources within the Hadoop ecosystem

Implementation constraints

- requires easy and non-intrusive modifications to the current implementation
- requires no additional effort in order to configure and use it
- adds no overhead to the rewriter, builder and query execution

Chapter 2

Approach

This chapter presents a high-level overview of the main components and algorithms we implemented in order to enable dynamic resource management in Vectorwise on Hadoop, and also explains how various design choices were made in accordance with the research questions of this project. The rationale behind the "Basic Ideas" presented in Section 1.6 should become clear by the end of this chapter.

To begin with, for a cost-effective, scalable, yet efficient solution, we focused our attention towards the open-source Apache Hadoop v2.2 (YARN) [1], which we have installed and configured on two independent sets of cluster nodes with different hardware configuration, but all interconnected by a low-latency, high bandwidth network fabric. Section 2.1 elaborates on the choice for HDFS direct local reads (short-circuit reads). In Section 2.2 we argue that instrumenting the block-placement policy in Hadoop is preferable to ensure data-locality, not only after data bulk-loading, but also after data re-replication (i.e. HDFS recovery phase) due to node failures, ease of use (i.e. no additional configurations) and maintainability (i.e. increasing/decreasing the worker-set and automatically moving data during a file-system check). In Section 2.3 we briefly describe the algorithms for dynamic resource management in Vectorwise on Hadoop. In Section 2.4 we present the solution space behind a good integration of long-running applications/jobs with Yarn and the high-level overview of our chosen approach for Vectorwise, mentioning the architectural components and their work-flow to acquire/release resources. Finally, more details about our experimentation platform / cluster's hardware specs are found in Section 2.5.

Below, in Figure 2.1, we illustrate our research approach (and the project's contributions) in *7 steps*, more or less in the order of their development: (1) start Vectorwise database, (2) select the worker-set and assign (partition) responsibilities, (3) generate metadata, (4) store metadata file in HDFS, (5) instrument HDFS block replication, (6) get cluster resource information and (7) acquire/release resources through YARN. These steps are going to be introduced by the remaining of the section and their goal should be clear by the end of reading it. Nevertheless, we do reserve a chapter for each of the most important topics: Chapter 3 for *instrumenting HDFS block replication*, Chapter 4 for *dynamic resource management* (i.e. worker-set selection, responsibility assignment, computing the worker-set resource footprint) and Chapter 5 for *YARN integration*. Last but not least, Chapter 6 highlights some of the results we achieved during this research project.

[1] Hadoop v2.2 (YARN) release: hadoop.apache.org/docs/r2.2.0/

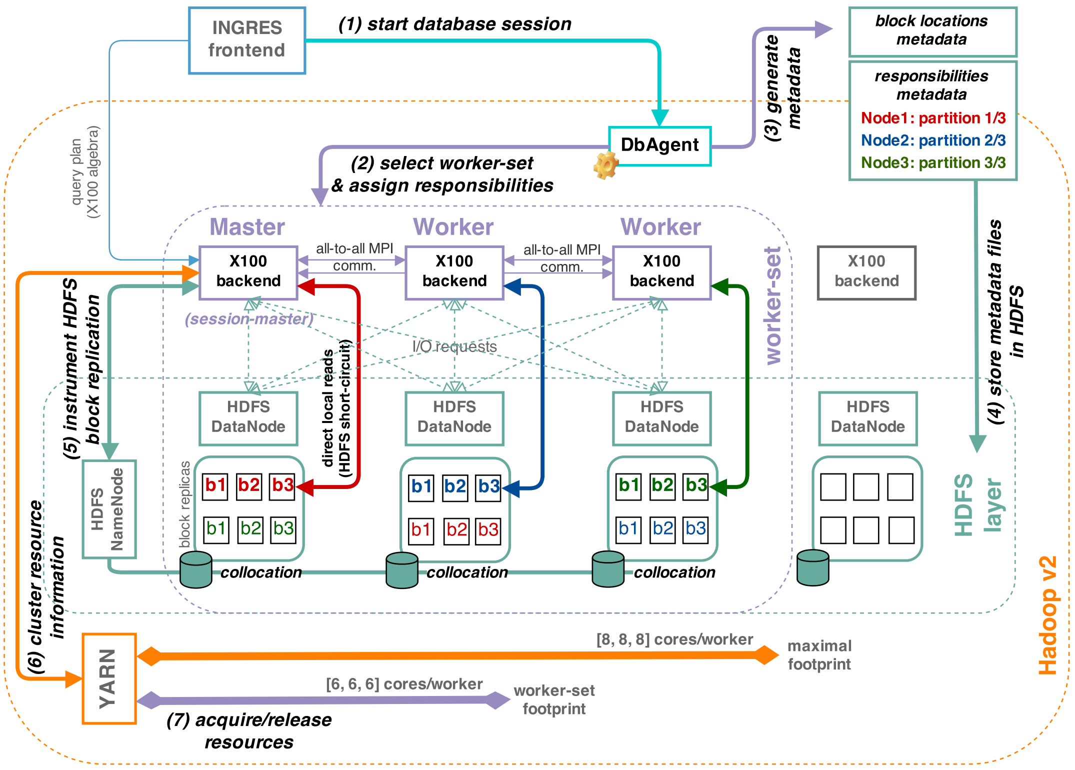


Figure 2.1: The Vectorwise on Hadoop dynamic resource management approach in 7 steps.

2.1 HDFS Local Reads: DataTransferProtocol vs Direct I/O.

One of the key assumption behind Hadoop is that moving computation is cheaper than moving data. A Hadoop-like application will prefer moving the computation to the data whenever possible, rather than the other way around. Thus, in general, HDFS should be able to handle many local reads (i.e. for which the reader is on the same node as the data).

Local reads through DataNode - DataTransferProtocol

Initially, local reads in HDFS were handled the same way as remote reads. Any client, connected to the DataNode via TCP socket, transferred data using the *DataTransferProtocol*. Therefore, *all reads* were passed *through the DataNode* (Figure 2.2): a client asks the DataNode to read a file, the DataNode reads that file off of the disk and then sends it to the client over a TCP socket, whether or not the reader is collocated with the data it needs. This approach was simple, but it had some downsides. For example, the DataNode had to keep an active thread and a TCP socket for each client that was reading a block. Hence, it adds the overhead of the TCP protocol in the kernel, as well as the overhead of DataTransferProtocol itself. The so-called *short-circuit reads* (described next) bypass the DataNode and implicitly the overhead of the protocol, allowing the client to read the file directly.

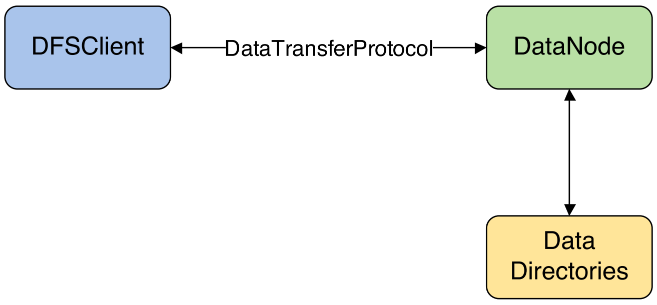


Figure 2.2: Local reads through DataNode (DataTransferProtocol).

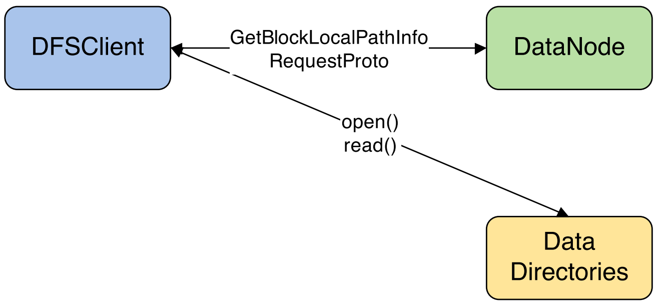


Figure 2.3: Direct local reads (Direct I/O - open(), read() operations).

Direct local reads – Direct I/O

The key idea behind *short-circuit local reads* (Figure 2.3) boils down to the following: if a client runs on the same node with its data, there is no need to involve the DataNode in the reading path. Rather, the client itself can simply read the data from the local disk using direct I/O primitives, *open()* and *read()* operations. Obviously, this is only possible in cases where the client is collocated with data. Short-circuit reads provide a substantial performance boost to many applications and so we have enabled this option in all our performance tests. This performance optimization made it into Hadoop v2.2 (YARN) as well [1]. To configure short-circuit local reads, you will need to enable *libhadoop.so*. Short-circuit reads make use of a UNIX domain sockets. This is a special path in the filesystem that allows the client and the DataNodes to communicate. You will need to set a path to this socket and the DataNode needs to be able to create this path. On the other hand, it should not be possible for any user except the HDFS user or root to create this path, for security concerns. For this reason, paths under */var/run* or */var/lib* are often used.

A short comparison between the two of these read options, reads through DataNode and short-circuit reads, is shown in Table 2.1. We can see from the results that, even though the map-readers access only local data, with *short-circuit reads* (direct local reads) the I/O throughput boosts in performance by 5-15% for both *single file – single map-reader* and also *multiple files – multiple map-readers* tests.

[1]Short-circuit local reads configuration: hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html

Table 2.1: Comparison between local reads through DataNode, without short-circuit (w/o), and direct local reads, with short-circuit (w/), using the TestDFS I/O benchmark on 1 and 10 "Rocks" (Section 2.5) cluster nodes; the results are averaged over two benchmark runs.

TestDFSIO | Reads	w/o	w/o	w/	w/
Number of files (1 per mapper):	1	10	1	10
Total MB processed:	10000	10000	10000	10000
Throughput MB/s:	94.75	57.88	98.78	64.75
Average IO rate MB/s	94.75	63.61	98.78	67.60
IO rate std deviation:	0.0132	19.6334	0.0182	15.3133
Test exec time (s):	135.294	51.403	126.225	41.605

2.2 Towards Data Locality

As mentioned in 1.3.3 each node loads data into specific partitions of the table, based on the query plan (VWLoad operators) that is rewritten by the parallel-rewriter. Tuples are distributed by hashing on key columns and stored in multiple HDFS files, where each file contains data for just a single column of a partition – a Vectorwise (table) partition has as many HDFS files as columns in the logical table and one single file may require more than an HDFS block (default size is 128 MB) for storage. HDFS uses a simple policy to store replicas for a block. If a process that is running on any of the HDFS cluster nodes opens a file for writing a block, the first replica of that block is stored on the same machine which the client is running on (we call it *first-write* data-locality). The second replica is placed on a randomly chosen rack that is different from the rack on which the first replica was written. Then, the third replica is stored on a randomly chosen machine from the same remote rack that was chosen in the first step. This means that a block is present on two unique racks. One point to note is that there is no relationship between replicas of different blocks of the same file as far as their location is concerned. Each block is stored independently. On one hand, the algorithm is very simple and yet good for availability and scalability. On the other hand, there are obvious cases where collocating blocks of the same file on the same set of datanode(s) or rack(s) is beneficial for performance reasons. For instance, from a Map-Reduce application construct, if many blocks of the same file are found on the same datanode, a single mapper instance could process all these blocks using the *CombineFileInputFormat* operator. Similarly, if a dataset contains many small files that are collocated on the same datanode(s) or rack(s), one can use the same operator to process all these file together by using fewer mapper instances. Moreover, if an application always uses one dataset in combination with another dataset (e.g. think of join operations on tables), then collocating these two datasets on the same set of datanodes is beneficial.

HDFS Block Placement Policy. In Vectorwise on Hadoop VWLoad queries, the rewriter associates files to load operators in a round-robin fashion keeping the first-write data-locality in place. However, at scan time, files may have different affinities to particular nodes (and there should be a way of detecting these affinities using the HDFS library) due to a possible system failover. Likewise, in case of a failover scenario (i.e. a node crashes) the default re-replication policy is triggered and so the first-write data-locality will be lost. Hence, our approach towards achieving full data-locality is to control the HDFS block replication by ourselves. To do so we can use the *HDFS-385* [1] feature, which is already part of HDFS since its 0.21 release. This improvement provides a way for the developer to write Java code, within the same HDFS namespace, that can instrument the block placement policy and specify, directly for instance, what target nodes should be used to store one file's blocks and their replicas. The HDFS API is currently incomplete and experimental in nature, but that is expected to change and become stable in the near future. The default behavior of the block placement policy can be

[1]HDFS-385: issues.apache.org/jira/browse/HDFS-385

modified by extending the *BlockPlacementPolicy* abstract class [1] and pointing the new class to the *dfs.block.replicator.classname* property in the HDFS configuration file.

To sum up, we have enhanced the default HDFS replication and implemented a custom block placement policy for Vectorwise partitioned tables. Our custom policy makes sure that all the HDFS blocks of a Vectorwise table partition (implicitly all columns of that table) are collocated on R target nodes (R equal to the replication factor) that we specify in the *global location-map* file. This file is generated at startup by our *Vectorwise-on-Hadoop DbAgent* component (or simply called the *DbAgent*, Section 2.3) and contains the R node locations for all table's partition. The end result is that multiple Vectorwise column format HDFS files are collocated on the same set of nodes and that re-replication (the recovery phase) is now under the control of the global location-map whenever some nodes might fail. Besides, we have also extended the HDFS *file-system-check* primitives to ease-up the system administration routines, i.e. a sysadmin can check if the node locations of Vectorwise HDFS blocks are consistent with the global location-map and if not, those blocks will be automatically moved to their right destination. More details about the implementation can be read in Chapter 3.

Comparison with other column based HDFS formats (Parquet and ORCFile)

Parquet [2] vs ORCFile [3]. Parquet's configurations [47] allows storing columns in different HDFS block sizes (the RowGroup can be up to 1GB [48]), in which respect it is superior to Stinger's [2] (Apache Hive's v0.12 release name) ORCFile. ORCFile uses a relatively small RowGroup (called Stripes), such that even when a subset of columns is scanned, I/O must be performed for the whole file. Hence, Parquet, with its maximum RowGroup of 1GB, can store a RowGroup in 16 x 64MB HDFS blocks, such that there is still hope for certain queries, which only select a few columns, to read less than all 16 blocks.

Parquet Schema. We note that Parquet has a ProtocolBuffer-like data format which allows to store nested data-models, which would be similar to multiple tables with a JoinIndex in-between (similar with TPC-H [3] Orders-Lineitem tables join in Vectorwise). Though this is outside of our project scope, it would be an interesting challenge to re-interpret a Parquet file into a relational schema. Note that, one Parquet file would potentially map to multiple tables connected by a JoinIndex. Vectorwise would need C-code-extensions (in MScan operator) to be able to read the Parquet format, aside of our custom columnar file format.

HDFS Block Locality. The current algorithm that analyzes block placement could be used to start-up Vectorwise server nodes such that they become responsible mostly for local files (i.e in case of re-replication locality is lost on the new nodes). The Map-Reduce based writer will create some locality in Parquet files, in that the blocks belonging to the same RowGroup will have been written by the same Reduce jobs, which means that the default HDFS policy will place the first replica/copy on that same node together (i.e. *first-write data-locality*). However, the other HDFS copies will randomly be scattered. As such, when nodes from the original Map-Reduce-set that wrote a specific file fail, it will be impossible for them to regain full I/O locality. In that sense, the failover behavior of native Vectorwise columnar files will be better than Parquet.

For future work, experiments can be done by loading the Orders and Lineitem tables of TPC-H into ORCFile and Parquet using Map-Reduce writers, showing that: (1) ORCFile cannot avoid unnecessary I/O even when a subset of columns is scanned, (2) Parquet can do so, however not at a very fine-grained level (it will still read unnecessary data), (3) Parquet data-locality is

[1] BlockPlacementPolicy abstract class: grepcode.com/file/repo1.maven.org/maven2/org.apache.hadoop/hadoop-hdfs/0.22.0/org/apache/hadoop/hdfs/server/namenode/BlockPlacementPolicy.java

[2] Stinger initiative's website: hortonworks.com/labs/stinger/

[3] TPC-H benchmark's website: www.tpc.org/tpch/

lost when some of the original writers are down, (4) ORCFile does not have this vulnerability (because blocks are self-contained), but it is inferior on I/O and cannot collocate multiple tables, whereas Parquet can do this because of its nested data model (Lineitem can be seen as a multi-valued child structure of Orders, all stored in the same RowGroup).

2.3 Dynamic Resource Management

This section describes the steps towards enabling dynamic resource management in Vectorwise on Hadoop (and the algorithms' choice). The approach is threefold: (1) Determine the Vectorwise worker-set, (2) Assign responsibilities to Vectorwise worker nodes and (3) Schedule cluster resources for different query workloads at runtime, by computing the optimal *resource footprint* (i.e. what particular nodes and the amount of resource, in terms of CPU and memory, to use). We start by describing the algorithm that is behind all these steps, and then briefly talk about how we made it suitable for each step in particular. For a more detailed description, we recommend reading Chapter 4.

Min-cost flow problem

The min-cost flow problem [45] is a network model that holds a central position among network optimization models, because it encompasses such a broad class of applications. Like the maximum-flow problem, it considers the flow through a network with limited arc capacities. Similar with the shortest-path problem, it considers a cost for the flow through an arc. And, like the transportation or assignment problem, it can consider multiple sources (supply nodes) and multiple destinations (demand nodes) for the flow, again with associated costs. In fact, all these problems are special cases of the original *min-cost flow problem*. The single commodity min-cost flow problem is one of the most fundamental models in flow network theory. This model can be useful in various real world applications, which arises in transportation, logistics, communication networks, resource planning, scheduling and many other domain areas. As mentioned in Section 1.2, there are few research papers [39, 40] explaining how to apply this algorithm, with variations in cost models, for resource management in Hadoop Map-Reduce.

Generalized statement. We are given a network - a directed graph, in which every edge has a certain capacity u and a cost c associated with it, a starting vertex, *the source* (s in Figure 2.4), and an ending vertex, *the destination* (t in the same figure). Also, we associate another value f per edge, satisfying $f \leq c$, such that, for every vertex other than the source and the destination, the sum of the f values associated to the edges that enter the vertex, *the in-edges*, must equal the sum of the f values associated to the edges that leave the vertex, *the out-edges*). In literature, this kind of network is called a *flow network* (or transportation network) and f represents the *flow* along the edge. The *min-cost flow problem* is to maximize the *total flow* in the network, the flow leaving the source (s), and, at the same time, to find the minimum cost path of sending it to the destination (t). In other words: *to find the minimum cost maximum flow from **s** to **t**.*

Algorithm's choice. We have decided to go for the *successive shortest path* [46] algorithm to solve the *min-cost flow* problem. For the implementation we use Edmonds-Karp relabelling, plus Dijkstra (with priority-queues) for the shortest path selection. The overall complexity is $O(min(E^2 * V * logV, E * logV * Flow))$, where E is the number of edges and V the number of vertices composing the network.

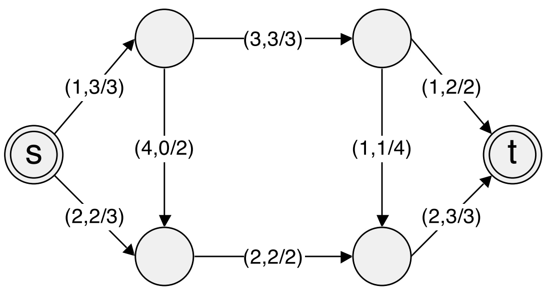


Figure 2.4: A flow network example for the min-cost problem, edges are annotated with *(cost, flow / capacity)*.

Worker-set selection & responsibility assignment

Worker-set selection. The idea is to select the best N nodes out of M with respect to their resource (e.g. CPU, memory, etc) capabilities and local data, Vectorwise column format HDFS local files, if any. This process is handled transparently by one of our components, the *DbAgent*: for the first part – *getting the cluster's capabilities* – it asks YARN for the cluster node reports and for the second part – *finding HDFS data* – it queries the HDFS Namenode and finds out where the Vectorwise files are located, assuming that the data was loaded into the system beforehand. Moreover, based on the data-locality information, we can determine missing replicas (i.e. in case of node failures) and have them re-replicated to other (possibly new) workers. The problem of (re)balancing table partitions to worker nodes, therefore creating the *global location-map*, can be stated as a min-cost matching problem [1] on a bipartite graph with a cost model to reflect the (Vectorwise files') data-locality to worker nodes (Chapter 4). As a side note, for the initial load we use a round-robin (horizontally) data partitioning scheme. However, following the initial load, the global location-map will always be re-generated according to the current locations of the Vectorwise HDFS files (including their replicas) at startup time.

Responsibility assignment. There are multiple ways to decide which worker nodes are responsible for specific partitions, depending on the HDFS replication degree. *One-to-one assignment – maximum one node responsible per partition:* To assign responsibilities to worker nodes, we apply the min-cost flow algorithm on a custom bipartite graph, Figure 2.5. The graph is structured in: *(1) left-side : partitions* and *(2) right-side : workers*. From our source (s) to each partition, we connect edges with cost 0 and capacity *RMax* equal to 1 – Figure 2.5. Then, we connect edges from partitions to workers using a cost model that is explained in Chapter 4 and capacity 1. Finally, we pair each worker to its destination node (t). We assign these edges a cost 0 and a capacity equal to the partition-capacity of the worker we connect with (i.e. how many partitions a node can handle, annotated with *PCap* in the figure). *Many-to-one assignment – two or more nodes responsible per partition:* This approach is similar with the previous one, except one thing: the capacity of an edge connecting s with a partition, *RMax* from Figure 2.5, is set to the maximum number of nodes responsible per partition.

Dynamic resource scheduling

To schedule resources for queries that run on (shared-nothing) partitioned tables, we have enhanced the existent Vectorwise resource scheduler with a min-cost flow algorithm. For this matter of problem, the algorithm uses a more advanced cost model (Chapter 4), though it is based on a similar bipartite graph like the one used to *assign responsibilities*. We briefly remind that, in the existent approach, the algorithm provides us a list of pairs ($\mathrm{n_x}$, $\mathrm{t_y}$) denoting how

[1]Note that the min-cost matching is just a specialization of the min-cost flow problem

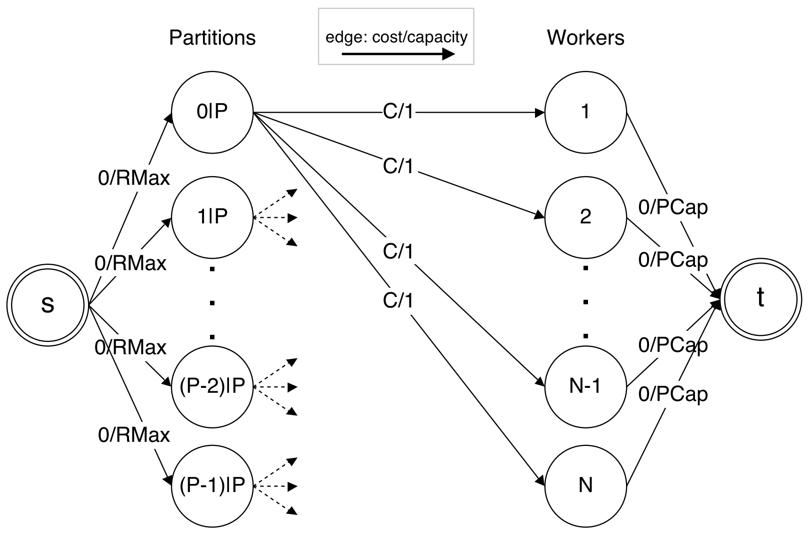


Figure 2.5: The flow network (bipartite graph) model used to determine the responsibility assignment.

many threads/cores (t_y) are allocated for its execution on node (n_x). The search starts with a feasible (initial) solution and tries/verifies all its multiple (e.g. $[(1,3),(2,5)]:[(1,6),(2,10)]:[(1,9),(2,15)]:etc\ldots$). It stops at the first combination which uses more threads (on any machine) than allocated by the resource scheduler. Moreover, the implementation adds two new multiple-nodes distribution policies for queries: *(1) equal-share for non-partitioned tables*, which similar with the multiple-nodes distribution used in the Shared Disk approach it selects the least loaded nodes and takes their amount of CPU resources until it satisfies the MPL, and *(2) equal-share for partitioned tables* that assumes the default round-robin partitioning and equally assigns threads to the operators until the MPL is achieved. In the new approach, we try to combine both (1) and (2) policies with a min-cost flow algorithm, so that we take in account data-locality and load-balancing factors. One advantage is that we can make the most of the local replicas. By increasing the responsibility per partition up to the HDFS replication degree we can add, to some extent, flexibility in computing the optimal *resource footprint* for a query. For instance, the algorithm can choose one half of the worker-set to run a query in case the other half is facing some serious problems, e.g. no local-data due to node failures, overloaded resources, stragglers etc. Of course, for this to happen, the HDFS replication degree should be ≥ 2.

2.4 YARN Integration

Currently, YARN expects to manage the lifecycle of its native applications that are used to run Hadoop jobs. Some frameworks/systems that could benefit from sharing resources with YARN run their work within long-running processes owned by the framework/system itself. These type of processes are not a good fit for the YARN containers. They need to run the work on behalf of many users and that their resources may increase as well over time. Similarly, Vectorwise workers (x100 backend servers) are long-running processes, which ideally are started once and from that point queries run within their context using multiple threads in order to achieve multi-core parallelism. As part of our thesis we also have been working on how to integrate Vectorwise with YARN, and we think our ideas might be applicable to other MPP on Hadoop databases.

There are different ways of integrating long-living application processes within YARN. Using *service X* as an example for the application and *queries* for the computational-unit, the different approaches to integrate service X with YARN can be characterized as follows:

Isolation outside YARN. Carving out part of the cluster nodes for service X and part for YARN (e.g. via Linux CGroups [1] / Linux Container Executor). This is better than just sharing the same cluster and hoping for no resource contention. This approach is simple and effective, for example, assigning 60% to service X and 40% to YARN. However, it lacks flexibility as load conditions may change for both sides. Also, it is not possible to assign a common share to a user or a queue across service X and YARN.

Model the service as a YARN application. Implementing service X as a YARN application could be another solution. With this approach, the YARN's scheduler is responsible for sharing resources among different users and applications. Because service X is a pure YARN application, resources utilization for this service and other YARN applications is dynamically based on the load conditions. A drawback of this approach is that service X will be tightly bound to just one of the scheduler's queues. While it is possible to run multiple service X instances (multiple YARN applications), one per queue, this may not be practical because the different service X instances may have to coordinate among each other and may have to duplicate their resource headroom utilization in advance. In addition, as stated in YARN-896 [2], there is significant work to be done to allow long live services running as YARN applications. Besides, for the moment, is not even possible to increase or decrease resource utilization within YARN containers (YARN-1197 [3]), being impossible "per se" to have some elasticity and to acquire/release resources for service X when its workload increases/decreases.

Model queries as YARN applications. For this approach we consider that service X resides outside YARN, having its own resources isolated from YARN, but each of its running queries translates into individual YARN applications. Optimizations such as pooling and reusing YARN application instances, and the containers within those applications, can help reducing latency significantly. An important benefit of this approach is that the resources utilization and distribution is completely managed by YARN. However, this approach assumes that each query submitted to service X is a set of independent processes from the service X's query dispatcher, but in our case all queries run within the Vectorwise worker-set's processes. Hence, this approach is not an option that could work out and be easily implemented.

Model resource requests as YARN applications. This idea sounds similar with the previous one, except that now we shift our focus towards requests *per workload* instead of *per query* to decrease the application's startup latency overhead (in number of requests) and that service X is part of the integration too. Making service X use a set of unmanaged YARN Application Master (more details in Chapter 5) sounds more likely to us. These applications are in principle spawned by (1) an initial request for the minimum amount of resources – *CPU and memory* – needed to bootstrap Vectorwise and be operable to users (e.g. at least 1 core + enough memory for query execution per node), and (2) extra resource requests to increase the performance of different query workloads. In this case, service X manages its own processes *out-of-band from YARN* and only uses the amount of resources allocated by the scheduler. On the other hand, the distribution of resources is completely managed by YARN. If service X runs queries over a set of processes, common resources (i.e. data-structures) can be easily shared among different runs without duplication. A drawback of this approach is that service X must manage on its own the enforcement of resource utilization per query. Our proposal for the Vectorwise-YARN integration fits this approach. However, as YARN-1197 is being pushed for the next release version, we are going to improve this approach in near future. Hence, instead of creating one YARN application per resource request (per workload), we could multiplex all the requests towards a single YARN application (or a pool of applications distributed over different

[1]Linux CGroups: www.kernel.org/doc/Documentation/cgroups/cgroups.txt
[2]YARN-896: issues.apache.org/jira/browse/YARN-896
[3]YARN-1197: issues.apache.org/jira/browse/YARN-1197

scheduling queues) that dynamically increases/decreases the containers' resource context. In this way we can avoid the application startup overhead entirely, reducing the acquire/release request latency even more (i.e. a YARN application can run in background and listen only for new requests).

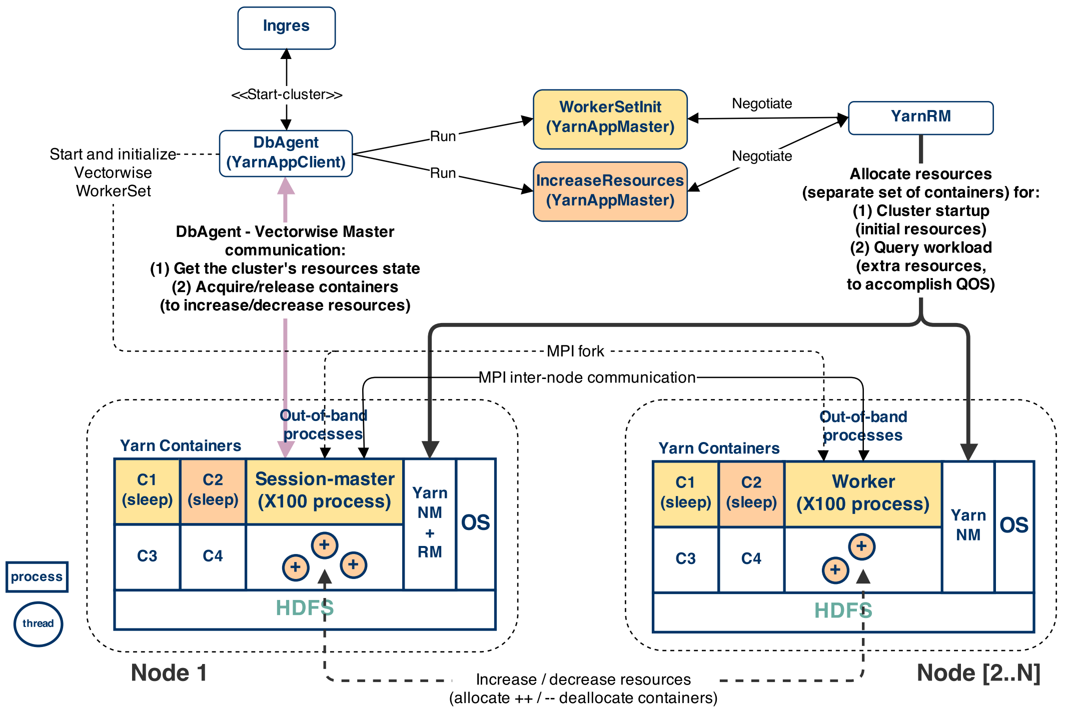

Ingres
<<Start-cluster>>
DbAgent (YarnAppClient)
Start and initialize Vectorwise WorkerSet
Run
Run
WorkerSetInit (YarnAppMaster)
IncreaseResources (YarnAppMaster)
Negotiate
Negotiate
YarnRM
Allocate resources (separate set of containers) for: (1) Cluster startup (initial resources) (2) Query workload (extra resources, to accomplish QOS)
DbAgent - Vectorwise Master communication: (1) Get the cluster's resources state (2) Acquire/release containers (to increase/decrease resources)
MPI fork
MPI inter-node communication
Out-of-band processes
Yarn Containers
C1 (sleep)
C2 (sleep)
Session-master (X100 process)
Yarn NM + RM
OS
C3
C4
HDFS
Node 1
Out-of-band processes
Yarn Containers
C1 (sleep)
C2 (sleep)
Worker (X100 process)
Yarn NM
OS
C3
C4
HDFS
Node [2..N]
process
thread
Increase / decrease resources (allocate ++ / -- deallocate containers)

Figure 2.6: Out-of-band approach for Vectorwise-YARN integration.

Figure 5.2 depicts our approach for the Vectorwise-YARN integration. As a reminder, in this approach we model every resource request (per workload) as a separate YARN application. The integration starts with the Vectorwise worker-set initialization phase. During this phase we allocate through YARN at least the minimum amount of resources for the database to be operable, i.e. run distributed queries w/o any parallelism (e.g. 1 core + enough memory for query execution per node). This phase is managed by the *WorkersSetInit* component (a custom YARN Application Master). From that point on, any needs to increase the worker-set amount of resources, i.e. extra CPU and memory to run a new workload, are translated into message-requests and sent directly to the *DbAgent* component (a custom *YARN Client Application*). Besides what is mentioned in Section 2.3, the *DbAgent* is also responsible for resource allocation within YARN, i.e. *to release or acquire new containers*. What happens under the hood is that, for each request, we spawn a new YARN Application Master (*IncreaseResources* component) that negotiates with the *ResourceManager* the demand for extra containers: it reserves and starts containers with specific resource requirements on a subset of (or all) worker nodes and then it manages their lifecycle. Previously to any resource request, the Vectorwise Master node asks the *DbAgent* for the *cluster's current resource state* in order to compute the *resource footprint* of the current query workload. More details about the Vectorwise-YARN integration are presented in Chapter 5.

2.5 Experimentation Platform

All our Linux based cluster nodes granted by Centrum Wiskunde & Informatica [1] (CWI) were set up using the before-mentioned specifications (e.g. Hadoop v2.2, w/ short-circuit, 128MB HDFS block size, replication degree equal to 3) and served as the *experimentation platform* for our prototype during development. For functionality testing and debugging we used the company's internal cluster of 4 nodes, being easier for us to run corner case examples and investigate issues, if any. A brief description of the CWI's cluster relevant hardware configuration is given below.

CWI SciLens cluster

Hardware ("Rocks" nodes)

Nodes: 144 nodes (Shuttle boxes), granted up to 32 nodes

Sockets/CPUs: 1 x Intel Core i7-2600K processor (4 cores, 3.4 GHz, 8 MB cache) per node; 2 threads per core with Hyper-Threading

Memory: 16 GB DDR3 per node

Storage: 1 x 2 TB HDD disks per node (/scratch 1.8 TB partition used for the HDFS data directory)

Disk I/O Throughput Tests (results averaged over 3 runs and rounded up)

dd if=/dev/zero of=out_file bs=1G count=1 oflag=direct : 131 MB/s

dd if=/dev/zero of=out_file bs=1G count=5 oflag=direct : 132 MB/s

Network: 40 Gbit/s InfiniBand interconnection, 1 Gb/s Ethernet

Hardware ("Stones" nodes)

Nodes: 16 nodes (hot-pluggable systems), granted up to 8 nodes

Sockets/CPUs: 2 x Intel Xeon E5-2650 v2 processors (8 cores, 2.6 GHz, 20 MB cache) per node; 2 threads per core with Hyper-Threading

Memory: 256 GB DDR3 per node

Storage: 3 x 3 TB HDD disks per node w/ software-RAID0 (/scratch 5.4 TB partition used for the HDFS data directory)

Disk I/O Throughput Tests (results averaged over 3 runs and rounded up)

dd if=/dev/zero of=out_file bs=1G count=1 oflag=direct : 473 MB/s

dd if=/dev/zero of=out_file bs=1G count=5 oflag=direct : 534 MB/s

Network: 40 Gbit/s InfiniBand interconnection, 1 Gb/s Ethernet

Actian internal cluster

Hardware

Nodes: 4 nodes (2 $\times$ 2 hot-pluggable nodes in a 2U form-factor)

[1] CWI SciLens cluster: www.monetdb.org/wiki/Main_Page

Sockets/CPUs: 2 x Intel Xeon E5645 processors (6 cores, 2.4 GHz, 12 MB cache) per node; Hyper-Threading disabled

Memory: 48 GB DDR3 per node

Storage: 1 x 2 TB HDD (used for the HDFS data directory) disks per node

Disk I/O Throughput Tests - for the HDD disk (results averaged over 3 runs and rounded up)

dd if=/dev/zero of=out_file bs=1G count=1 oflag=direct : 126 MB/s

dd if=/dev/zero of=out_file bs=1G count=5 oflag=direct : 128 MB/s

Network: 40 Gbit/s InfiniBand interconnection, 1 Gb/s Ethernet

Chapter 3

Instrumenting HDFS Block Replication

This chapter presents in detail the *Custom Block Placement Policy* for HDFS that we implemented in order to control the block (re-)replication and to collocate table partitions to the same group of DataNodes, therefore achieving data-locality to Vectorwise worker nodes during failover situations (e.g. a node fails running our services and it is replaced by a new one).

3.1 Preliminaries

It is sometimes beneficial to have selective overlap among the workers on which the partitions of two or more tables are stored. If the same hash function and number of partitions are chosen for two (or more) tables, then there will be a *one-to-one correspondence* between the partitions of both (or all) tables that will join with one another. If this is the case, then it is possible to collocate the joining partitions of both (or all) tables. That is, any pair of joining partitions will be stored on the same nodes participating in the workers-set. The advantage of collocation is that tables can be joined without the need to move any data from one node to another. However, collocation of joining tables can be nontrivial when there are complex join relationships.

Let us consider an example of two tables R and S loaded on a Vectorwise cluster of 3 workers / 4 nodes, using a 3-way partitioning and replication degree of 2. Also, assume that both tables are hash partitioned on the same attribute *key* and a partition fits in one single HDFS block. An example of a default HDFS block placement, after R and S tables are loaded, is shown in Figure 3.1 - upper image. In this context, consider the following example of a query that joins tables *R* and *S* on attribute *key*:

```sql
Select *
From R, S
Where R.key = S.key
```

If we assume that nothing happens with the cluster/system (i.e. a node crashes), we can then process parts of the join locally on each node. The default HDFS block replication guarantees that it stores the first replica of a block on the node which initiates the write (*first-write locality*). As for the second replica (or the others, if more) that is not the case. However, only for their first replicas, R's and S's partitions are collocated on the same nodes and can be joined locally – R1 can be joined locally with S1 on N1, R2 with S2 on N2 and, respectively, R3 with S3 on N3. Unfortunately, this all changes when a node (or more) fails. All missing block replicas will be discovered and *re-replicated* (recovery phase) randomly to other nodes within the whole HDFS cluster, which can be even larger then the Vectorwise workers-set. If that is the case, we lose our first-write locality and, implicitly, the collocation that is set by the initial bulk load. As a result, this will make the collocation of joining tables less probable. Considering Figure 3.1 - lower image, to join R3 with S3 on N4 (after re-replication) we would have to access S3 remotely from N1/N2 during the query's cold-run, or to move this join computation to N2.

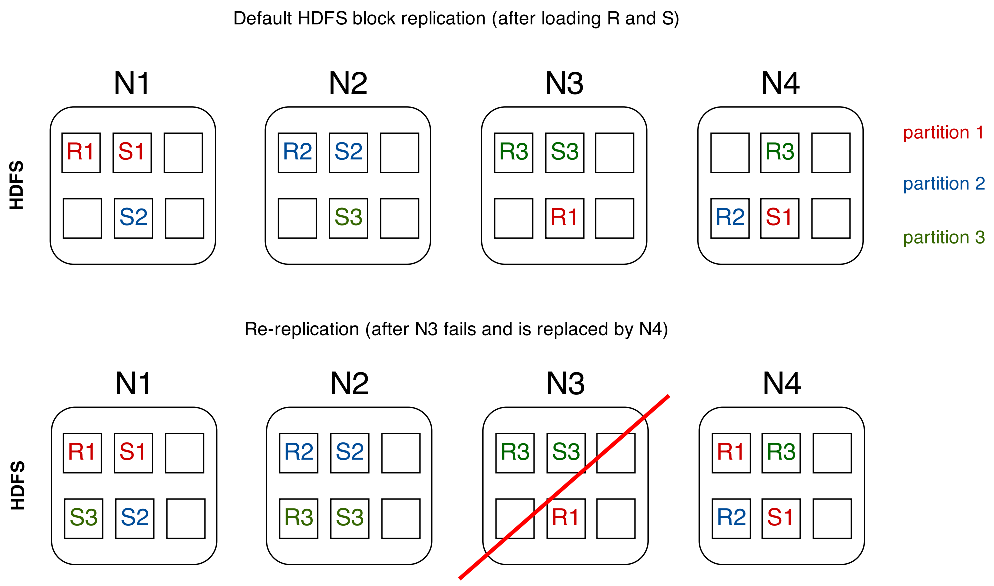


Figure 3.1: An example of two tables (R and S) loaded on 3 workers / 4 nodes, using 3-way partitioning and replication degree of 2: after the initial-load (upper image) vs. during a system failover (lower image).

The previous situation becomes even worse if each partition has multiple blocks and/or its (table) columns are stored in separate HDFS files, as it happens to Vectorwise on Hadoop because of the custom columnar HDFS file format. One way to mitigate this problem and achieve full data-locality, plus collocation of joining tables, is to properly instrument the HDFS replication by implementing a custom block placement policy. If tables are partitioned using the same *partitioning-class* and on the same attribute or set of attributes, then this policy must collocate and replicate their partitions (their columns and so, implicitly, their HDFS files/blocks) on the same subset of worker nodes, Section 3.2. A partitioning-class (or partitioning-scheme) is simply the degree of partitioning, i.e. a 3-way partitioning is a partitioning-class. Non-partitioned tables have their partitioning-class too, 1-way partitioning (i.e. 1 single partition). To sum up, one can think of this policy as replicating and collocating Vectorwise column-format HDFS files instead of logical partitions.

A formal definition for the custom policy can be stated as: given T tables partitioned on the same attribute(s), R (degree of replication), P (a partitioning-class) $\Rightarrow t_i(p_j) \in W_j$, where $i = 1, \ldots, T$, $j = 1, \ldots, P$, W = the workers-set, W_j = a subset of our workers-set, $|W_j| = R$. This definition applies to more than one partitioning-class. We note that two or more replicas of the same HDFS block cannot be stored on the same node.

Going back to our previous example, now R's and S's partitions can be replicated and collocated on the same subset of worker nodes: R1 with S1, R2 with S2 and R3 with S3, on $\{N1, N2\}$, $\{N2, N3\}$ and, respectively, $\{N3, N1\}$. A general illustration of what can we achieve with a 3-way partitioning on 3 workers / 4 nodes is depicted in Figure 3.2.

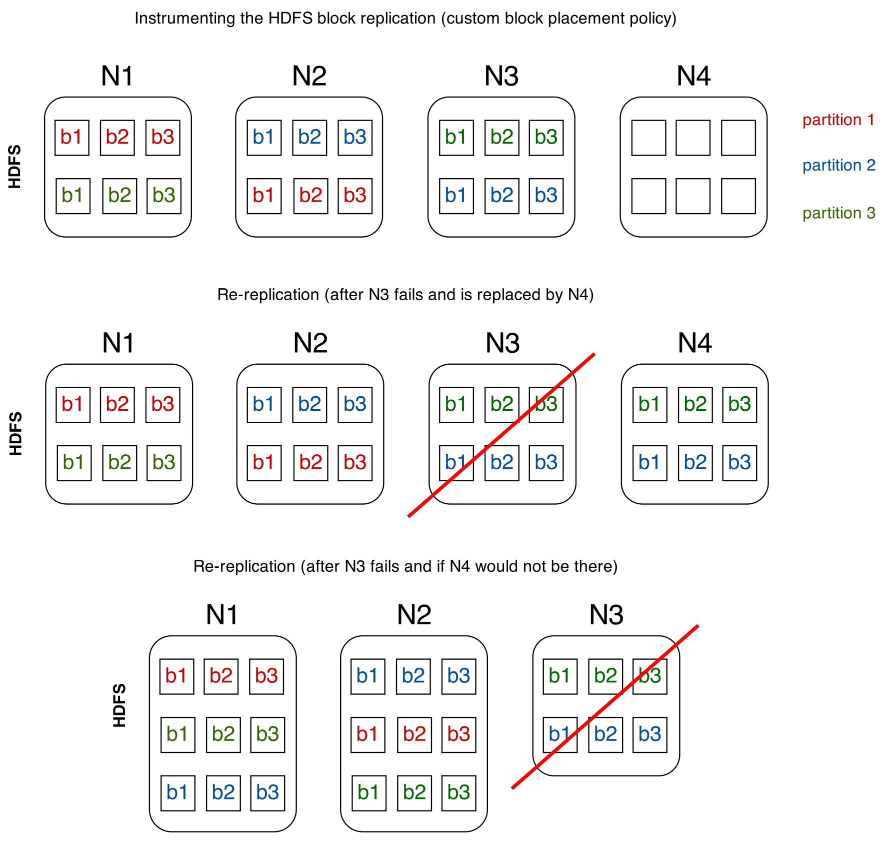

Instrumenting the HDFS block replication (custom block placement policy)
N1
N2
N3
N4
HDFS
b1
b2
b3
partition 1
partition 2
partition 3
Re-replication (after N3 fails and is replaced by N4)
Re-replication (after N3 fails and if N4 would not be there)

Figure 3.2: Instrumenting the HDFS block replication, in three scenarios, to maintain block (co-)locality to worker nodes.

3.2 Custom HDFS Block Placement Implementation

We have enhanced the default HDFS block replication and implemented a custom block placement policy (only) for Vectorwise partitioned tables. Other HDFS (non-Vectorwise) files are not considered by our replication policy, since we filter Vectorwise files by their naming pattern *<table-name>***@***<partition-ID>*, where *<partition-ID>* is *<partition-number>***c***<partitioning-class>*. The custom policy makes sure that the HDFS blocks of all files with the same partition-id (implicitly all columns of a table) and their replicas are collocated on exact *R* target nodes (R equal to the replication degree). We specify the exact R nodes in a *global location-map*. This particular data structure is generated at startup by the *DbAgent* component and then saved at the Vectorwise HDFS *database location* into a metadata file – called *plocations*. The end result is that multiple Vectorwise column-format HDFS files are collocated on the same set of nodes and that (re-)replication (the recovery phase) is now under the control of the global location-map whenever nodes might fail.

An example of a *plocations* file (or *global location map*), in Json format, for 3 nodes, replication degree 2, and 3 (1-, 3-, 6-) partitioning-classes is:

```
{   // plocations metadata file in Json format: <partition-ID>:[<location>,...]
    // we assign a list of locations (worker nodes) to every <partition-ID>
    "0c1":["node1","node2"],"0c3":["node1","node2"],"1c3":["node2","node3"],
    "2c3":["node3","node1"],"0c6":["node1","node2"],"1c6":["node2","node3"],
    "2c6":["node3","node1"],"3c6":["node1","node2"],"4c6":["node2","node3"],
    "5c6":["node3","node1"]   }
```

3.2.1 Adding (New) Metadata to Database Locations

The *plocations* file is stored only once at the *Vectorwise database location*, since we can easily share this path with our DbAgent at startup time. All metadata files that the *DbAgent* generates are stored there. Moreover, whenever a (Vectorwise columnar-) file is being written to HDFS block by block, the only information we are provided by the API is its full source path. So, though we can extract and parse from a Vectorwise file's path its exact *table location*, simply, from the Block Placement Policy class, we cannot determine to which *database location* it belongs to (not without extra communication with the Vectorwise Master node). This means that the *global location-map* is not (directly) visible to other table locations, including the default one. Nevertheless, this information is still needed to select the target (worker-) nodes for block replication. It is clear by now that what we need is a way to refer to the database location from any table location. An easy way to do so is to keep the default database location (the HDFS path) into a metadata file, and, whenever a table location is created (including the default one), have it stored directly within that path.

3.2.2 Extending the Default Policy

To implement our Vectorwise custom block placement policy, we used the *HDFS-385* [1] feature, which is already part of HDFS since the 0.21 release. This improvement provides a way for the developer to write Java code, within the same HDFS namespace, that can instrument the block replication and specify, directly for instance, what target nodes should be used to store one file's blocks including their replicas. The default behavior of the block placement policy can be modified by extending the *BlockPlacementPolicy* abstract class [2].

Another way is to overload some of the *BlockPlacementPolicyDefault's* class [3] methods, like we did in order to maintain/keep the default block replication for the rest of the (non-Vectorwise custom format) HDFS files. More specifically, what we did was to overload the *chooseTargetNodes(...)* method and diverge the default work-flow for Vectorwise data. Whenever an HDFS block of a Vectorwise column-file is being stored and the latter method is called, we use our own *choose target nodes* method instead – named *chooseVWTargetNodes(...)* – to compute (and return) the list of datanode descriptors for block replication.

Algorithm 3.3 contains the pseudo-code for *chooseVWTargetNodes(...)* method, referred previously. First, at *lines 1 to 3* we read the (static) configuration values for *global location-map file's name, table's location metadata file's name and the Vectorwise HDFS file pattern.* In steps 6-8 we extract from the block's source path *its directory path, the name of the file which belongs to, and the table's partition-ID.* Then, at *line 9* we cache the *global location-map* for the current database location. *Line 12* determines the available (alive) target nodes for block replication using the (in-memory cached) global location-map. The target nodes are mapped by the table's partition-ID. From *13 to 22* we check if there are stale entries in cache and, if any, we re-cache all entries by reading again the global location-map metadata from its source file. Finally, *lines 23 to 34* fills out the *results* list with target descriptors and returns the *writer* descriptor. It converts the available target nodes (hostnames) into a datanode descriptors list. Then, it uses this list to merge with *results* and to determine the *writer* descriptor.

Both algorithms, Algorithm 3.1 and Algorithm 3.2, come to help in Algorithm 3.3 when caching is needed for the *global location-map*. By this, we omit multiple metadata file reads that can degrade our performance. Reading once and then saving the metadata in (NameNode's) memory makes it fast enough to retrieve the right target nodes during the HDFS block replication.

[1] HDFS-385: issues.apache.org/jira/browse/HDFS-385

[2] BlockPlacementPolicy abstract class: grepcode.com/file/repo1.maven.org/maven2/org.apache.hadoop/hadoop-hdfs/0.22.0/org/apache/hadoop/hdfs/server/namenode/BlockPlacementPolicy.java

[3] BlockPlacementPolicyDefault class: grepcode.com/file/repo1.maven.org/maven2/org.apache.hadoop/hadoop-hdfs/2.2.0/org/apache/hadoop/hdfs/server/blockmanagement/BlockPlacementPolicyDefault.java

However, when the cache contains stale entries (i.e. due to node failures), we do have to discard the data by (re-)reading again the (possible new) file's content.

Algorithm 3.1 Cache the table's location metadata file:
cacheTableLocMetadata(tableLocation, tableLocFilename)

```
Input: tableLocation: the HDFS path towards the table location, tableLocFilename: the table's
    location metadata filename (default is .metadata)
1: if $\neg dbLocationsMap.containsKey(tableLocation)$ then // synchronized section
2:   $tableLocMetadataPath \leftarrow makePath(tableLocation, tableLocFilename)$
3:   $dbLocation \leftarrow readDbLocMetadata(tableLocMetadataPath)$ // read HDFS path, one liner
4:   if $dbLocation = \emptyset$ then // invalid path or the file is empty
5:     throw $error$
6:   end if
7:   $dbLocationsMap.putEntry(tableLocation, dbLocation)$
8: end if
```

The first algorithm, Algorithm 3.1, uses a *concurrent hash map* to store *table locations* to *database location many-to-1* relations, where the *database location* (HDFS path) is read from the table's location metadata file. The second one, Algorithm 3.2, uses a *concurrent hash map* data structure too, but to store *1:1* relations between a *database location* and its *plocations* metadata file (the *global location-map* generated each time by the *DbAgent* during the Vectorwise cluster startup).

Algorithm 3.2 Cache the global location-map file:
cachePLocations(tableLocation, tableLocFilename, pLocFilename)

```
Input: tableLocation: the path towards the HDFS table location, tableLocFilename: the table's
    location metadata filename (default is .metadata), pLocFilename: the global location-map
    metadata filename (default is plocations), one per Vectorwise database location
1: $cacheTableLocMetadata(tableLocation, tableLocFilename)$ // we make sure to cache the table's
   location metadata beforehand
2: $dbLocation \leftarrow dbLocationsMap.getValue(tableLocation)$
3: if $dbLocation$ does not exist anymore in HDFS then // stale entry
4:   // remove the correspondent metadata entries from dbLocationsMap and pLocationsMap
5:   $dbLocationsMap.removeEntry(tableLocation)$
6:   $pLocationsMap.removeEntry(dbLocation)$
7:   $cacheTableLocMetadata(tableLocation, tableLocFilename)$ // and re-cache
8:   $dbLocation \leftarrow dbLocationsMap.getValue(tableLocation)$
9: end if
10: if $\neg pLocationsMap.containsKey(dbLocation)$ then // synchronized section
11:   $pLocationsPath \leftarrow makePath(dbLocation, pLocFilename)$
12:   $pLocations \leftarrow readPLocMetadata(pLocationsPath)$ // read the global location-map for the
      current database session
13:   if $pLocations = \emptyset$ then // invalid path or the file is empty
14:     throw $error$
15:   end if
16:   $pLocationsMap.putEntry(dbLocation, pLocations)$
17: end if
```

Besides Algorithm 3.3, we have also extended/overloaded the HDFS *file-system-check* primitives (not shown) to ease-up the system administration routines, i.e. a sysadmin can check if the node locations of Vectorwise HDFS blocks are consistent with the global location-map and if not, those blocks will be automatically moved to their right destination.

Algorithm 3.3 Choose the Vectorwise target (worker-) nodes for block replication: *chooseVWTargetNodes(blockSrcPath, numOfReplicas, writer, results)*

Input: *blockSrcPath:* the full source path of the HDFS block being written, *numOfReplicas:* number of replicas needed, *writer:* the datanode's descriptor that initiates the write operation, *results:* the list of chosen target nodes (datanode descriptors) for block replication – for a new block this list is empty, whereas for an old block with missing replicas it contains the rest of the nodes (datanode descriptors) where that block is stored

Output: the datanode descriptor for the *first-write* locality and, indirectly, the target nodes (the descriptors *results* list)

```
 1: $pLocFilename \leftarrow configPropertyGet(ploc\_metadata\_property)$
 2: $tableLocFilename \leftarrow configPropertyGet(tableloc\_metadata\_property)$
 3: $pattern \leftarrow configPropertyGet(vwfile\_pattern\_property)$
 4: $newBlock \leftarrow (results.size = 0)$ // **true** if is empty, **false** otherwise
 5: $totalReplicas \leftarrow numOfReplicas + results.size$
 6: $srcDirname \leftarrow extractDirname(blockSrcPath)$
 7: $srcFilename \leftarrow extractFilename(blockSrcPath)$
 8: $pId \leftarrow extractPartitonId(blockSrcPath, pattern)$
 9: $cachePLocations(srcDirname, tableLocFilename, pLocFilename)$
10: $dbLocation \leftarrow dbLocationsMap.get(srcDirname)$
11: $pLocations \leftarrow pLocationsMap.get(dbLocation)$
12: $targetNodes \leftarrow getAvailableTargetNodes(pLocations, pId)$
13: **if** $targetNodes.size < totalReplicas$ **then** // stale metadata
14:    // remove the correspondent *plocations* entry from *plocationsMap*
15:    $pLocationsMap.removeEntry(dbLocation)$
16:    $cachePLocations(srcDirname, tableLocFilename, pLocFilename)$ // and re-cache
17:    $pLocations \leftarrow pLocationsMap.get(dbLocation)$
18:    $targetNodes \leftarrow getAvailableTargetNodes(pLocations, pId)$
19:    **if** $targetNodes.size < totalReplicas$ **then** // still stale, *DbAgent* was not yet restarted
20:       *fill-out targetNodes with nodes from the remaining Vectorwise cluster (use the least-
          loaded nodes)*
21:    **end if**
22: **end if**
23: $targetDesc \leftarrow toDataNodeDescriptors(targetNodes)$
24: **if** $newBlock =$ **true then**
25:    **if** $writer =$ **null** $\vee \neg targetDesc.contains(writer)$ **then**
26:       $writer \leftarrow targetDesc[0]$
27:    **end if**
28:    $results.add(writer)$ // the *local* descriptor has to be the first
29:    merge $results$ with $targetDesc$, maximum $numOfReplicas$
30:    **return** $writer$
31: **end if**
32: // *an old block*
33: merge $results$ with $targetDesc$, maximum $numOfReplicas$ // fill-out *missing replicas*
34: **return** $writer$
```

Preliminary Results

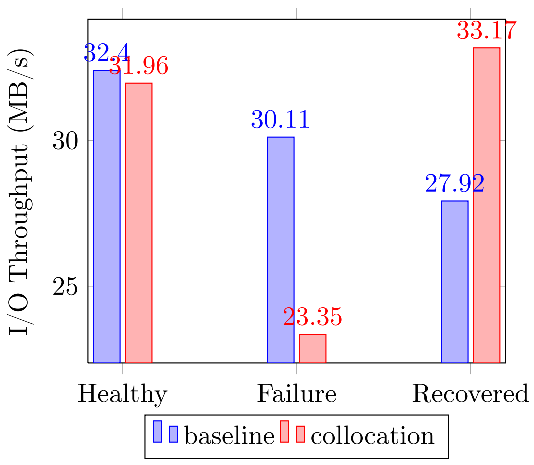
In this part of the section we present some of the experiments we run against both HDFS block replication policies discussed earlier, the default policy (not. default) and the custom one (not. collocation), in order to determine which one is better for Vectorwise on Hadoop, *during* and *after* node *failures*. Hence, we compare these policies in three situations: the cluster is in *(1) Healthy state*, *(2) Failed state* and *(3) Recovered state*. In the third situation, when using the custom policy, we recompute the *global location-map* by restarting the Vectorwise (new) workers-set through our *DbAgent* component. For the experimental workload, we have selected just 4 of the TPCH-H queries (1, 4, 6, and 12, all I/O bounded for cold-runs) together with 32- and a 16- partitioning schemes (for Lineitem and Orders only) on scale-factor 100. We ran these queries individually on 8 workers, out of 10 "Rocks" Hadoop nodes, with a 32 mpl. This means that, during our experiments, 1 partition can be read by maximum 1 thread, respectively, 2 threads. Moreover, with the default policy, we remind that blocks are replicated among all 10 Hadoop nodes; the default block replication is not aware of our workers-set and collocation requirements. To simulate the *Failed state* scenario we simply stopped (and decommissioned) 2 of our (workers-set's) DataNodes, so that we could use the remaining 2 available Hadoop nodes as their replacements / new workers. In addition, to ensure real cold-runs, we always free the Vectorwise Buffer Pool and the Operating System's cache before each query. We have disabled any Hadoop caching mechanism too, to make sure we do not have another caching layer in between Vectorwise and OS that could affect our results.

The end result is quite obvious: whereas the custom block replication overcomes (by a lot) the default behavior after the re-replication process, because we have control over data (co)locality to worker nodes, the default behavior performs better during node failures, since new worker nodes (replacing the failed ones) may already have some local data that queries could read. However, this advantage of the default behavior during failures is effectively lost if we limit the number of Hadoop nodes to the workers-set size, i.e. number of workers equal with the total number of nodes. New worker nodes, chosen to replace the failed ones, will no longer have local data for sure. We show what happens in such case in Chapter 6. Nevertheless, it takes around 8-9 minutes to recover the 3rd (missing) replicas (in total there are approximately 26 GB of data to re-replicate), time that can be easily amortized considering the side effect of controlling data-locality, i.e. reading local data during cold-runs or when is not enough buffer pool to cache all the tables.

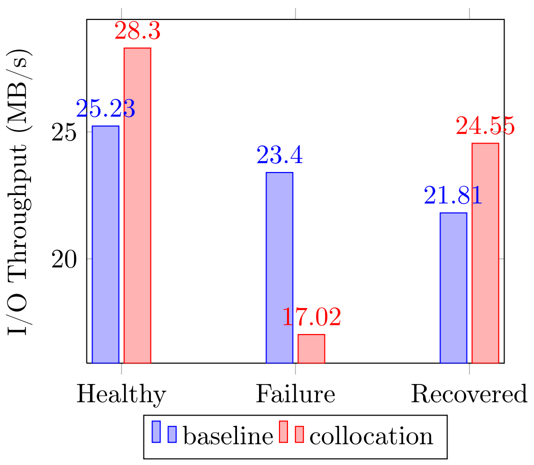
Table 3.1: Baseline (default policy) vs. collocation (custom policy) – execution times in 3 situations (1) healty state, (2) failure state (2 nodes down), (3) recovered state. Queries 1, 4, 6, and 12 on partitioned tables, 32-partitioning, 8 workers / 10 Hadoop "Rocks" nodes, 32 mpl.

Vers.	**baseline**			**collocation**		
State	Healthy	Failure	Recovered	Healthy	Failure	Recovered
$Q1$	18.94 s	21.64 s	27.57 s	19.65 s	23.48 s	18.36 s
$Q4$	2.07 s	2.56 s	3.61 s	2.25 s	3.93 s	2.25 s
$Q6$	3.35 s	4.36 s	5.94 s	3.55 s	5.09 s	3.44 s
$Q12$	4.78 s	6.56 s	8.28 s	4.55 s	7.66 s	4.54 s
Total	29.14 s	*35.12* s	**45.40** s	30.00 s	*40.16* s	**28.59** s

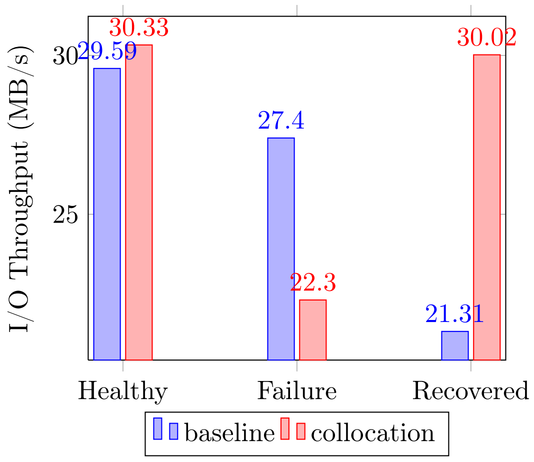
What is important to see is that, both the individual query execution times, Table 3.1, and the HDFS I/O throughput per query, Figure 3.3, do not change too much in the *Recovered* state as compared to the *Healthy* state. By all means, the results are about the same (with insignificant differences) in both states, which implies that we have achieved data (co)locality in Vectorwise on Hadoop for the Recovered state. In Chapter 4 we further explain how we can improve the performance of cold-runs during failovers, favoring local (disk) reads over remote (network) access to the extent of maximum resources available. The same behavior repeats in experiments with a lower partitioning scheme as well, from 32- to 16- partitioning, Table 3.2.



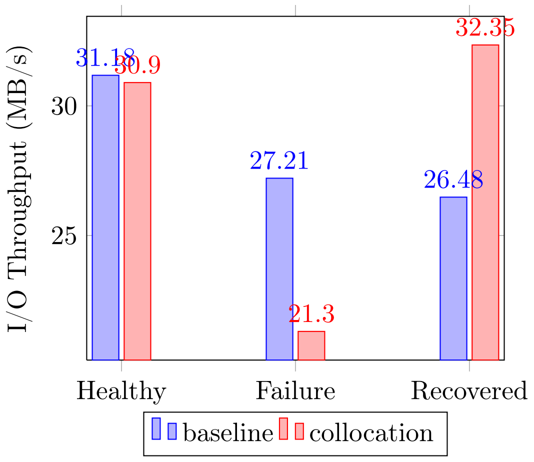
(a) Query-1 (cold-runs)



(b) Query-4 (cold-runs)



(c) Query-6 (cold-runs)



(d) Query-12 (cold-runs)

Figure 3.3: Baseline (default policy) vs. collocation (custom policy) – HDFS I/O throughput measured in 3 situations (1) healty state, (2) failure state (2 nodes down), (3) recovered state. Queries 1, 4, 6, and 12 on partitioned tables, 32-partitioning, 8 workers / 10 Hadoop "Rocks" nodes, 32 mpl.

On the other hand, it is curious that the performance of the baseline Vectorwise on Hadoop version (with default block replication) degrades from *Failure* to *Recovered* state. Our experimental results show that: (a) the query execution time during cold-runs increases, Table 3.1, and (b) the HDFS I/O throughput per query gets even worse, Figure 3.3. This is somehow counter-intuitive, as one would expect to get better I/O performance when data recovers (i.e. the chances of having more local data increases). The 1st and 2nd replicas we believe are still in place, if blocks were not moved around because of load-balancing issues. We checked our experiments and no data seemed to be relocated unnecessarily. So, a possible cause of this peculiar performance loss may be related to the recovery phase: the fact that new (3rd) replicas are scattered randomly within the cluster could affect the I/O reading pattern, local-reads vs. remote-reads, by making remote-read requests change their source of read. For instance, a better look into the DataNode log files revealed ±30-40% fluctuations in the SHORT_CIRCUIT_READS request counts for the 2 new worker nodes and ±10-15%, respectively, for the other nodes. The same behavior repeats with 16- partitioning, Table 3.2, but, interestingly though, not if we restrain the number of Hadoop nodes to the workers-set size (as shown in Chapter 6).

Table 3.2: Baseline (default policy) vs. collocation (custom policy) – execution times in 3 situations (1) healty state, (2) failure state (2 nodes down), (3) recovered state. Queries 1, 4, 6 and 12 on partitioned tables, 16-partitioning, 8 workers / 10 Hadoop "Rocks" nodes, 32 mpl.

Vers.	**baseline**			**collocation**		
State	Healthy	Failure	Recovered	Healthy	Failure	Recovered
$Q1$	17.64 s	19.37 s	27.65 s	17.29 s	29.34 s	17.78 s
$Q4$	1.52 s	2.43 s	3.33 s	1.37 s	2.58 s	1.30 s
$Q6$	3.00 s	4.02 s	6.92 s	3.21 s	5.53 s	3.04 s
$Q12$	4.72 s	6.12 s	7.85 s	3.90 s	7.85 s	3.47 s
Total	26.88 s	*31.94* s	**45.75** s	25.77 s	*45.30* s	**25.59** s

Conclusion

We have presented in detail our *Custom Block Placement Policy* for HDFS, designed to control the HDFS block (re-)replication and to collocate (Vectorwise) table partitions to the same group of DataNodes (specified *via plocations* metadata file). The preliminary results from Tables 3.1 and 3.2 prove that we can still achieve data-locality to Vectorwise worker nodes during failover situations (e.g. a node fails running our services and it is replaced by a new one). However, as Chapter 6 shows, there is yet some room left for improving the latter by favoring *local reads* over *remote* (over-the-network) *reads*.

Chapter 4

Dynamic Resource Management

This chapter describes the steps towards enabling dynamic resource management in Vectorwise on Hadoop. Our approach is threefold:

1. we first have to *determine the Vectorwise worker-set* (Section 4.1),
2. then *assign (partition) responsibilities to these workers* (Section 4.2)
3. and finally, *schedule the cluster resources that a query can use at runtime* by computing an optimal *resource footprint*, what particular nodes and the amount of resources, such as CPU and memory, to use (Section 4.3).

4.1 Worker-set Selection

Currently in Vectorwise on Hadoop, to start a database cluster session, human intervention is needed to explicitly specify the list of worker nodes as an argument to the Vectorwise cluster mpi_run command. An automatic approach for the worker-set selection is yet to be done. One way of doing this automatically in Vectorwise on Hadoop is to enhance the Ingres client with the ability to query YARN (Resource Manager) and discover which nodes are alive in the cluster, plus what are their resource specifications. Once it determines the *best* N nodes from all M *(*$\geq$ N*)* Vectorwise cluster nodes, with respect to their data-locality and resource capabilities, the Ingres client can then start the X100 worker-set through the mpi_fork command. All that a sysadmin has to do is to write down the list of Vectorwise (on Hadoop) cluster nodes, the M hostnames that have the X100 server binaries already deployed, in the configuration file.

However, the Ingres client can be configured and used on any remote host-machine outside the Hadoop cluster. That makes it difficult to manage a Vectorwise (X100) worker-set from the frontend itself. We could probably use YARN's and HDFS's REST APIs to perform the three steps enumerated in the beginning of this chapter, but for more complex management tasks, for instance allocate or release cluster resources within YARN, scan HDFS for data, file system check, etc., we must run within the Hadoop environment and implement its native Java APIs. Due to this constraint, we have decoupled the Vectorwise *start/stop* database functionality from Ingres, implemented with the mpi_fork command, and replaced it with a separate client-server communication. Whereas the server is a new Java component that runs in the Hadoop environment (the *DbAgent* component) and takes care of the worker-set selection, starting/stopping the X100 workers, responsibility assignment and Vectorwise-YARN integration for resource management, the client (the *VectorwiseLauncher* component, written in Java too) runs on the Ingres side and forwards the *start/stop* database commands as (TCP/IP) message requests to the agent, along with the list of parameters needed to initialize the X100 backend processes.

Upon receiving a *start* request, the DbAgent determines the worker nodes and then spawns their X100 processes using the same *mpi_fork* command. The VectorwiseLauncher connection to DbAgent is kept alive during the entire database session and it can be *stopped* from both sides, either *(1) from the (DbAgent) server side*, signifying that there was an issue with running the cluster, or *(2) from the (VectorwiseLauncher) client side* if the user decides to do so. The whole process of starting the worker-set through the DbAgent component is depicted in Figure 4.1. No changes were done to the (Ingres to X100 session-master) communication protocol that provides Vectorwise with an optimal query execution plan and then returns back the final result to the user.

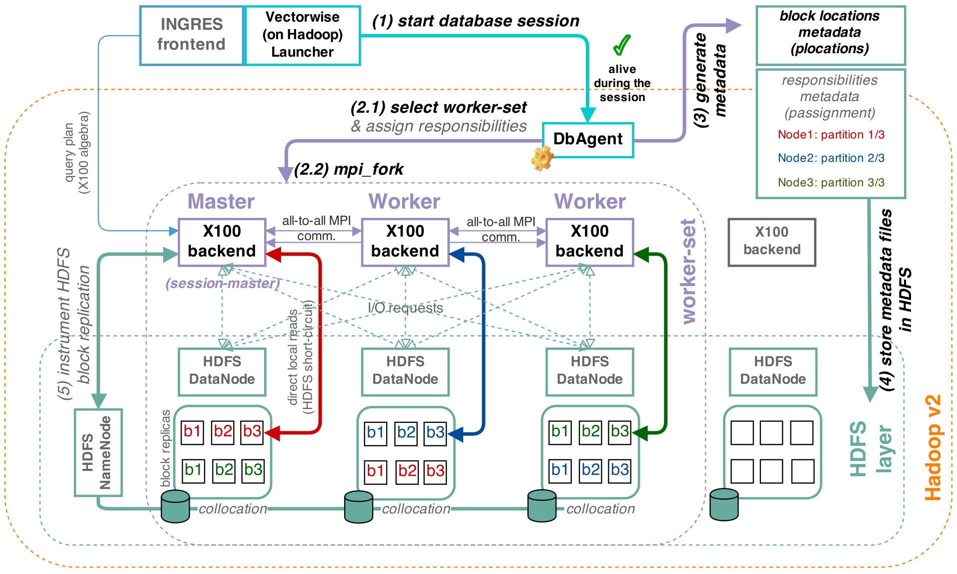

INGRES frontend
Vectorwise (on Hadoop) Launcher
(1) start database session
alive during the session
(3) generate metadata
block locations metadata (plocations)
responsibilities metadata (passignment)
Node1: partition 1/3
Node2: partition 2/3
Node3: partition 3/3
(2.1) select worker-set & assign responsibilities
DbAgent
(2.2) mpi_fork
query plan (X100 algebra)
Master
Worker
Worker
X100 backend
all-to-all MPI comm.
(session-master)
I/O requests
worker-set
(4) store metadata files in HDFS
(5) instrument HDFS block replication
direct local reads (HDFS short-circuit)
HDFS DataNode
HDFS NameNode
block replicas
b1 b2 b3
collocation
HDFS layer
Hadoop v2

Figure 4.1: Starting the Vectorwise on Hadoop worker-set through *VectorwiseLauncher* and *DbAgent*. All related concepts to this section are emphasized with a bold black color.

The general idea for the *worker-set selection* is to first sort out all M Vectorwise (Hadoop) cluster nodes in descending order by their local data size (i.e. existent Vectorwise columnar HDFS local files) and resource (e.g. CPU, memory, etc) capabilities, then choose the top N *(*$\leq M$*)* nodes. This twofold process is handled transparently by the *DbAgent*, using the Hadoop (YARN and HDFS) native Java APIs:

Get the HDFS block locations. To *find out where (on which Hadoop nodes) the Vectorwise data is located*, the DbAgent component queries the HDFS Namenode for the *location-information* of all files stored at any of the Vectorwise table locations (HDFS paths) and of which names comply with the *table-name@partition-ID* pattern (see Section 3.2). The tables' locations are read from the Vectorwise location-map metadata file, found at the *database location* path. For each *location-information* entry we get the *file's size*, the *list of block IDs* composing the file, plus the *node locations* of all its R *block copies*, where R is the HDFS replication degree. Assuming that our custom HDFS block replication is enabled and that Vectorwise data was bulk-loaded into the system beforehand, every block from the list must have the *same* R *node locations*. This implies that we just need the first block from it in order to determine the file's R node locations, making the HDFS data scan faster.

Sort nodes by resource capabilities. In order to *get the resource capabilities of Vectorwise nodes*, the DbAgent first asks YARN (Resource Manager) for the list of all *cluster node reports*. A node report contains an updated summary of runtime information. It includes details such as the ID of the node, HTTP tracking URL, rack name, used resources, total resources, and the number of running containers on the node. An example of a Json-format response for node reports, using an API call with a similar functionality (but from YARN's REST API), is:

```
{ // Cluster Nodes API
  // JSON response from the HTTP Request:
  // GET http://<Resource-Manager-http-address:port>/ws/v1/cluster/nodes
  "nodes":{
    "node":[
      {
        "rack":"default-rack",
        "state":"NEW",
        "id":"node2:1235",
        "nodeHostName":"node2",
        "nodeHTTPAddress":"node2:2",
        "healthStatus":"Healthy",
        "lastHealthUpdate":1324056895432,
        "healthReport":"Healthy",
        "numContainers":0,
        "usedMemoryMB":0
        "totalMemoryMB":8192,
        "usedVirtualCores":0,
        "totalVirtualCores":8
      },
      {
        "rack":"default-rack",
        "state":"RUNNING",
        "id":"node1:1234",
        "nodeHostName":"node1",
        "nodeHTTPAddress":"node1:2",
        "healthStatus":"Healthy",
        "lastHealthUpdate":1324056895092,
        "healthReport":"Healthy",
        "numContainers":0,
        "usedMemoryMB":4096,
        "totalMemoryMB:8129,
        "usedVirtualCores":4
        "totalVirtualCores":8,
      }
    ]
  }
}
```

From the entire list of Hadoop node reports we then filter out those that are not for Vectorwise nodes (based on the list of nodes provided in the configuration file) and their *state* information is not set to *running*. From the report itself we find the *total/used amount of memory* and the *total/used number of virtual-cores* as the only information useful to us, from which we can derive the current *available memory and virtual-cores*. Using these metrics we can build our own internal node reports list and sort it in decreasing order (by "totals") to get the top most resourceful Vectorwise cluster nodes. That aside, an important aspect which matters for the worker-set selection and should be added to the sorting criterias, is the size of Vectorwise local block data. Since our custom block replication policy tries to limit the replication domain just to the worker-set, a node that has more local data than another will have bigger chances to hold more table-partitions. To sum up, this approach would help us to determine the appropriate Vectorwise worker-set which has all of our data and its worker nodes are among the cluster's best nodes.

Based on the data-locality information obtained during the HDFS file system scan, we can *determine* the *missing replicas* (i.e. in case of node failures) from the system and start re-replicating them to (possibly new) other workers. Also, we can automatically increase and decrease the worker-set at startup time and re-arrange the Vectorwise partitions to a new configuration of worker nodes. The problem of (re-)balancing data to worker nodes and creating the *global location-map* can be formulated as a min-cost matching problem on a bipartite graph where the cost model reflects the Vectorwise data-locality to worker nodes. For the initial bulk-load we use a round-robin (horizontally) data partitioning scheme, Algorithm 4.1.

Algorithm 4.1 Initial round-robin replicas placement:
roundRobinPlacement(workerSet, pClass, R)

Input: *workerSet:* the list of worker nodes (after selecting the top N workers from the Vectorwise cluster nodes sorted in descending order by resource capabilities and data size), *pClass:* partitioning-class (to generate the partition-IDs), *R:* HDFS replication factor
Output: the global location-map (control data-locality)

```
 1: $partitionIds \leftarrow computePartitionIds(pClass)$ // generate partition IDs for the current pClass
 2: $workerSet.computePartitionCap(pClass, R)$ // given the worker-set compute resources
 3: $remainingReplicas \leftarrow R$, $i \leftarrow 0$, $currPCap \leftarrow workerSet.getWorker(i).getPCap()$
 4: while $remainingReplicas > 0$ do
 5:   for all $pId : partitionIds$ do
 6:     while $workerSet.getWorker(i).hasLocalPartition(pId) \vee currPCap[i] = 0$ do
 7:       $i \leftarrow (i+1)\%workerSet.size$
 8:     end while
 9:     $workerSet.getWorker(i).addLocalPartition(pId)$
10:     $currPCap[i] \leftarrow currPCap[i] - 1$
11:     $i \leftarrow (i+1)\%workerSet.size$
12:   end for
13:   $remainingReplicas \leftarrow remainingReplicas - 1$
14: end while
15: return $locationMap \leftarrow buildLocationMap(workerSet)$ // we use the worker's local partitions
```

An example of a round-robin placement for the initial bulk-load, generated by Algorithm 4.1, using 1- (non-partitioned) and 12- (partitioned) for *partitioning-classes* (see Section 3.1), on 3 (identical) nodes with 2 replicas, is given below:

```
// Horizontal round-robin placement for table partitions and their replicas
// For a new partitioning-class (blue color) we restart from the first worker
   worker1 <- 0c1, 0c12, 3c12, 6c12,  9c12, 2c12, 5c12, 8c12, 11c12,
   worker2 <- 0c1, 1c12, 4c12, 7c12, 10c12, 0c12, 3c12, 6c12,  9c12,
   worker3 <- ___, 2c12, 5c12, 8c12, 11c12, 1c12, 4c12, 7c12, 10c12
```

Algorithm 4.2 Matching partitions to worker nodes:
matchPartitionsToWorkers(pClasses, workerSet, hdfsPartitionsLoc, R)

Input: *pClasses:* the list of partitioning-classes, *workerSet*, *hdfsPartitionsLoc:* the result of the HDFS scan for block locations, *R*
Output: the global location-map (controls data-locality)

```
 1: if $hdfsPartitionsLoc = \emptyset$ then // empty cluster / nonexistent data
 2:   for all $pClass : pClasses$ do
 3:     // the workers are still sorted by their resource capabilities
 4:     return $pLocations \leftarrow roundRobinPlacement(pClass, workerSet, R)$
 5:   end for
 6: else
 7:   $pLocations \leftarrow \emptyset$
 8:   for all $pClass : pClasses$ do
 9:     $flowNet \leftarrow buildFlowNet(costModel)$
10:     $residualNet \leftarrow minCostFlow(flowNet, balanced \leftarrow true)$
11:     $pLocations.putAll(residualNet.getMatchings())$ // all the edges with $flow = 1$ in the residual graph
12:   end for
13:   $workerSet.updateLocalPartitions(pLocations)$ // update each worker's local partitions based on the pLocations mapping
14:   return $pLocations$
15: end if
```

However, following the initial bulk-load, the global location-map (*plocations* metadata) will always be re-generated according to the current location of the Vectorwise partitions (i.e. the columnar HDFS files including their replicas) at startup time, Algorithm 4.2. Before digging into this algorithm's pseudo-code, we recommend reading the *cost model* description in the next paragraphs.

When both *(1) querying HDFS for Vectorwise block locations* and *(2) sorting Hadoop nodes by resource capabilities* steps are done, we can compute the cost model and generate the flow network for the the matching algorithm, the annotated bipartite graph from Figure 4.2. Finding an optimal matching in a bipartite-graph is a min-cost flow sub-problem which can be solved with the *Successive Shortest Path* algorithm [49]. We explain the algorithm's implementation (and our own changes to it) in the last section of this chapter. As for the current and next section we briefly explain the (bipartite graphs) cost models and the pseudo-code of the main functions, more specifically Algorithm 4.2 and 4.3, from which the min-cost flow algorithm is being called.

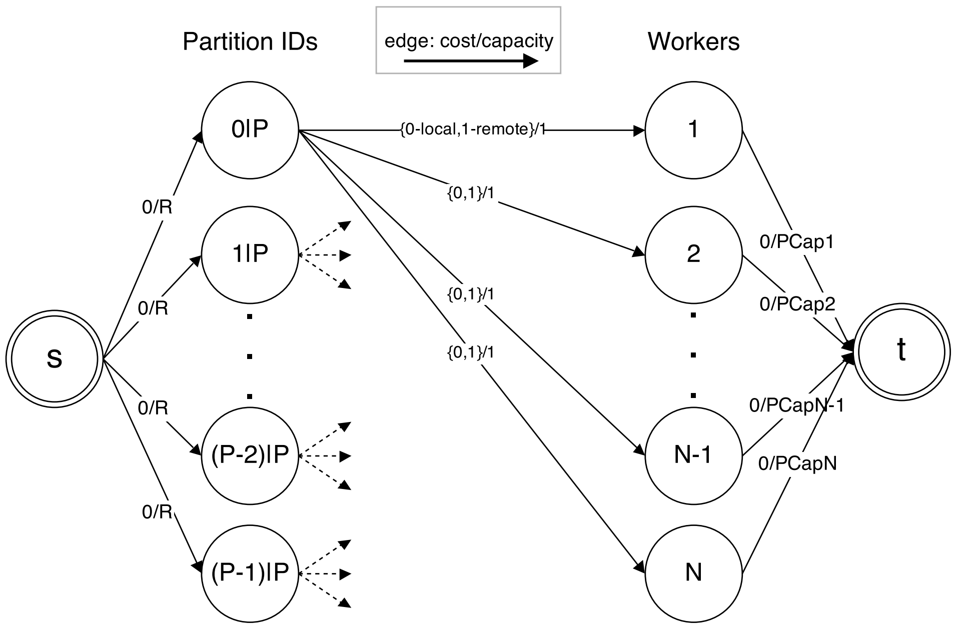


Figure 4.2: The flow network (bipartite graph) model used to match partitions to worker nodes.

Cost model. The bipartite-graph from Figure 4.2 is structured as follows: *(1) left-side : partition-IDs* and *(2) right-side : workers (the whole worker-set)*. What exactly is a *partition-ID* we can remember from Section 3.2. From our *source* (s) to each *partition-ID*, we connect edges with cost 0 and capacity equal to the HDFS replication degree (R). Then, from a *partition-ID* to a *worker* we use the capacity 1 and cost 0 for local partitions, or cost 1 for non-local/remote partitions. To determine whether a partition is physically stored or not on a particular cluster node, we simply use the HDFS results from the previous file system scan. Finally, we link our workers to their destination node (t). We assign these edges a cost 0 and a capacity equal to the partition-capacity ($PCap$) of that worker node. The end result is a load-balanced mapping from partitions to nodes; it actually represents the *global location-map* to which we referred many times in previous paragraphs. Already mentioned, this mapping is used mostly during the *(1) custom HDFS block replication* (Section 3.2) and *(2) responsibility assignment* (Section 4.2), and it is materialized into the *plocations* metadata file that is found at the Vectorwise *database location* path.

With each node's *PCap* (*partition-capacity*) we can restrict the edge capacities in the cost model with respect to heterogeneous cluster configurations, although all recent Hadoop implementations assume all nodes in the cluster to be homogeneous in nature. We know from [50] that Hadoop is lacking performance in an environment with different hardware specs and therefore, in practice, all cluster nodes should roughly have the same computing capacity. If not, this can effect MPP database systems, which usually prefer an *equipartition* strategy to load-balance data for effective resource utilization. The latter has bee proven to be an efficient strategy in general [13, 14, 42]. Nevertheless, Vectorwise on Hadoop uses its own *worker-set* selection strategy (see Section 1.3.3), in which Hadoop nodes with similar resource capabilities are chosen to run the X100 backend processes.

The $PCap_i$ *partition-capacity* (or the maximum partition load) determines how many partitions (plus R replicas) matching with any of the *partition-IDs* (left side) can be assigned to a *worker node* (right side), considering the number of cores and amount of memory, while running (min-cost flow) algorithm. It simply acts as an *arc capacity* in the bipartite-graph. On the one hand, this is quite relevant for computing the cost model in order to *(re-)balance the Vectorwise table partitions* (with partition-IDs in left-side) *to the worker-set* (on right side), but also for the round-robin placement we perform before the initial data bulk-load. The worker's partition-capacity has the following formula:

$$PCap_i = \lceil (C_i * PRatio_i) * \frac{M_i}{MAvg} \rceil,$$
$$PRatio_i = \frac{PTot - PCap_{i-1}}{CTot - C_{i-1}},$$
$$i = 1, \ldots, N,\ PCap_0 = 0 \text{ and } C_0 = 0$$

where:

- N is the worker-set size,
- C_i and M_i are the number of cores, respectively, the memory size of $worker_i$,
- $PRatio$ is the partitions-per-core ratio (computed iteratively),

  $PTot = P * R$ is the total number of partitions (including their R replicas),

  $CTot$ is the worker-set total number of cores (aggregated number),
- $MAvg = \frac{MTot}{N}$ is the worker-set average amount of memory,

  $MTot$ is the the worker-set total amount of memory (aggregated number)

The reason why the above formula refers to one *partitioning-class* (see Section 3.2) is that we compute the cost model and run the matching algorithm iteratively for each class. This gives us a much better load-balancing per node since the matching (implicitly the min-cost flow) algorithm, by itself, cannot distinguish between different partitioning-classes. Otherwise, we might end up with all the partitions from the same partitioning-class stored on one single node. Obviously, such a result is not load-balanced, i.e. different partitioning-classes implies different partition sizes. We note the reader that we will continue referring to one partitioning-class in the following paragraphs for simplicity reasons, though in our implementation all the algorithms and cost models are applied individually to each of these classes.

4.2 Responsibility Assignment

There are multiple ways to assign partition responsibilities to worker nodes. We have introduced the meaning of *partition responsibilities* and *responsibility assignment* in Section 1.3.3. These concepts are emphasized in Figure 4.3. Depending on the HDFS replication degree, the responsibility assignment can vary from *one-to-one assignment* with one node being responsible per partition, to *many-to-one assignments* where two or more nodes may be responsible for the same partition. As we pointed in Section 4.1, the *location-map* metadata returned by the previous Algorithm 4.2 is one of the input data to find the *responsibility assignment*, as will see in Algorithm 4.3. Even though both algorithms use the same approach (i.e. finding a min-cost flow) to solve the *matching problem in bipartite graphs*, the difference is in the cost model. What is important to know is that we have now extended the cost model to take in account local partitions that were assigned last time already and so, they may have been cached by the worker's OS while doing I/O for Vectorwise queries.

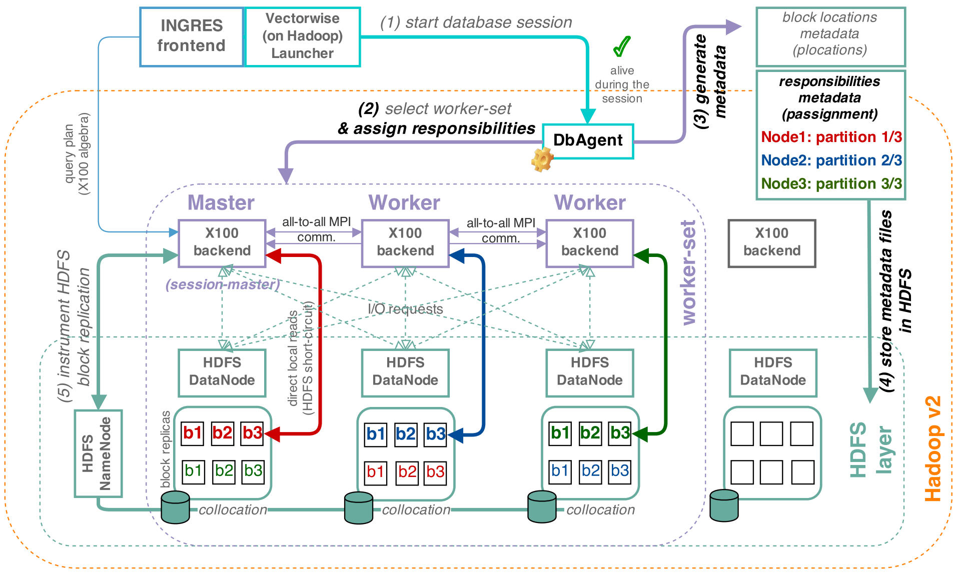

INGRES frontend
Vectorwise (on Hadoop) Launcher
(1) start database session
alive during the session
(3) generate metadata
block locations metadata (plocations)
responsibilities metadata (passignment)
Node1: partition 1/3
Node2: partition 2/3
Node3: partition 3/3
(2) select worker-set & assign responsibilities
DbAgent
query plan (X100 algebra)
Master
Worker
Worker
X100 backend
all-to-all MPI comm.
(session-master)
I/O requests
worker-set
(4) store metadata files in HDFS
(5) instrument HDFS block replication
direct local reads (HDFS short-circuit)
HDFS DataNode
HDFS NameNode
block replicas
b1
b2
b3
collocation
HDFS layer
Hadoop v2

Figure 4.3: Assigning responsibilities to worker nodes. All related concepts to this section are emphasized with bold black and red/blue/green colors (for *single* (partition) responsibilities).

Cost model. To assign responsibilities to worker nodes we run the min-cost flow algorithm for the bipartite graph that is illustrated in Figure 4.4. Likewise the previous bipartite graph, this one is structured in *(1) left-side : partition-IDs* and *(2) right-side : workers* as well. From the source (s) to each *partition* vertex, we connect edges with cost 0 and capacity equal to the *responsibility degree*, annotated in the figure with *RMax*. For a *one-to-one assignment*, with maximum one node being responsible per partition, *RMax* is set to 1. When two or more nodes can be responsible for the same partition, i.e. *many-to-one assignment*, the *RMax* capacity is set to the maximum number of responsibilities per partition. The latter value should be $\leq$ than the HDFS replication degree to benefit from data-locality. Otherwise, we may have *remote responsibilities* (i.e. need remote read access) assigned to workers. Next, the all-to-all edges from partition nodes to worker nodes have capacity 1 and a range of costs that varies according to the *locality* relationship between the two of them. We assign *0 ∼ local+* costs, to local partitions that were used during the last responsibility assignment, if any, *1 ∼ local*, to any other local partitions different from the previous, and *2 ∼ remote partitions*, to partitions that need remote access for I/O. The reason why we favor ex-responsibilities of local partitions over any other options, giving these edges the lowest cost in the model, is to minimize the responsibility changes during a system failover and/or after a restart. The implication is that we could keep the OS cache hot (with data cached from previous reads) and so, reduce the probability of hitting the disk when the Vectorwise database restarts with an empty buffer pool. In the end, we connect the worker nodes with their destination node (t). These edges have the cost equal to 0 and the capacity equal to the partition-capacity (*PCap*) of the worker node we connect with, except that *PCap* is now determined with respect to the *RMax* degree (replacing R in the upper formula). The result is a load-balanced (data-locality aware) $partition : [nodes, \ldots]$ mapping, where $|[nodes, \ldots]| = RMax, RMax \leq R$. The *responsibility assignment* is written in Json format in a metadata file called *passignment*, which is stored at the *database default* *location*. We use the *passignment* metadata for further tasks in our project, for instance to compute the *resource footprint* (i.e the resource requirements in terms of CPU, partitions and node locations) of a query at runtime.

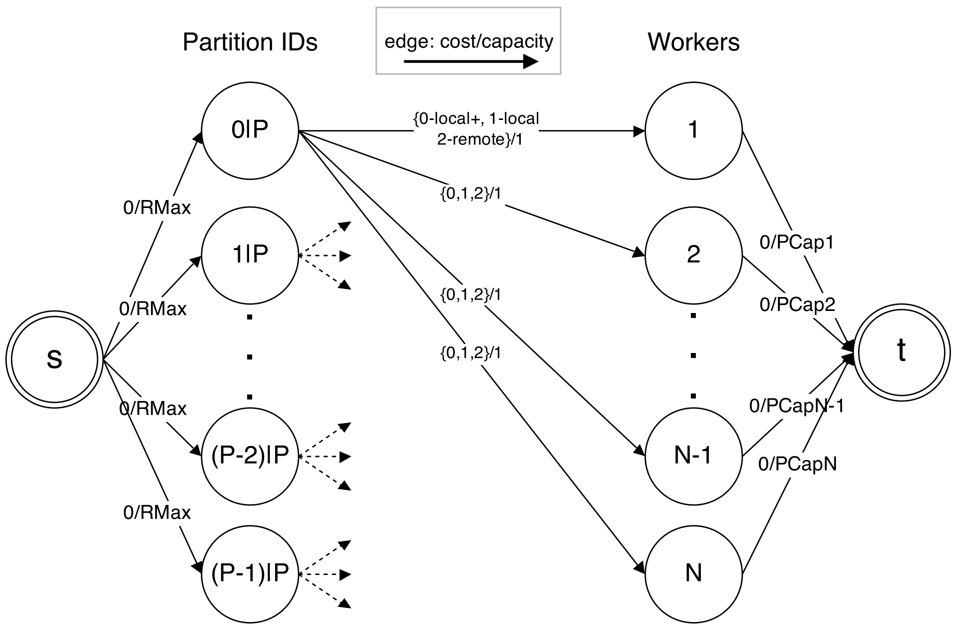


Figure 4.4: The flow network (bipartite graph) model used to assign responsibilities to workers nodes.

Algorithm 4.3 Decide the worker-set responsibilities:
assignResponsibilities(pClasses, pLocations, RMax)

Input: *pClasses:* the list of partitioning-classes, *pLocations:* the global location-map, *RMax:* the responsibility degree per partition (should be $\leq$ than the HDFS replication degree)
Output: the responsibility assignment

```
1: $respAssignment \leftarrow readRespAssignment(database\_location\_path)$ // read the Json format metadata fie located at the database location
2: if $respAssignment \neq \emptyset$ then
3:     $costModel.setPrevRespAssignment(repsAssignment)$
4: end if
5: $respAssignment \leftarrow \emptyset$
6: for all $pClass : pClasses$ do
7:     if $pClass = 1$ then // non-partitioned tables
8:         continue // skip non-partitioned tables (being replicated on each worker node anyways)
9:     end if
10:    $flowNet \leftarrow buildFlowNet(costModel)$
11:    $residualNet \leftarrow minCostFlow(flowNet, balanced \leftarrow true)$
12:    $respAssignment.putAll(residualNet.getMatchings())$ // all the edges with $flow = 1$ in the residual graph
13: end for
14: return $pAssignment \leftarrow respAssignment$
```

Simulating node failures: data (re)replication & responsibility assignment

For a better understanding of how the two previous algorithms (Algorithm 4.2 and Algorithm 4.3) work in practice, in this paragraphs we are going to simulate a failover scenario in a Vectorwise on Hadoop cluster of 4 nodes, of which 3 are chosen as workers, and explain what happens at every step by using the DbAgent's log file. The HDFS replication degree is set to 2 and the block size is 128 MB. We use 1- and 12- partitioning-classes (1 GB per partition roughly) and RMax = 1 (single responsibility degree). We note that this is just a basic example meant only to support what comes in the following section, Section 4.3.

We start the Vectorwise worker-set through the *VectorwiseLauncher* component. The input consists of 4 Hadoop nodes (from Actian's cluster), the worker-set specs (e.g. number of workers, responsibility degree and X100 initialization arguments) and the initial resource requirements, such as the container's (1 per node) amount of memory and number of cores.

```
{
  "message_type":"WSET_INIT_REQ",
  "init_args":[
    "--cluster_nodes","NLAM-CLUSTER01","NLAM-CLUSTER02","NLAM-CLUSTER03","NLAM-CLUSTER04",
    "--num_workers", "3",
    "--max_resp_degree", "1",
    "--container_memory", "10240",
    "--container_vcores", "8",
    "--x100_prefix", "/.../x100/bin",
    "--x100_args", "--dbfarm hdfs:///dblocation --dbname dbname --port 0 --config /.../cl_vectorwise.conf"
  ]
}
```

What we see below is the Vectorwise worker-set selection phase, where the DbAgent performs the twofold approach described in Section 4.1:

```
dbagent: Scan HDFS file system for Vectorwise files...
dbagent: Location added to HDFS scan: hdfs:///dblocation/dbname/default
dbagent: Location added to HDFS scan: hdfs:///custom_table_location
dbagent: The pattern we are looking for: .+@[0-9]+c[1-9]+[0-9]*
dbagent: Number of Vectorwise files: 0
dbagent: Sort Vectorwise cluster nodes by resources...
dbagent: Extended cluster node report (sorted by resources):
dbagent: <hostname>: [local-partitions], block-counts, virtual-cores, memory-size, session-master or not
dbagent: NLAM-CLUSTER02: [empty], blockCount=0, vCores=8, mem=32768, isMaster=1 // the first to select because it is the session-master (see Section 1.3.3)
dbagent: NLAM-CLUSTER01: [empty], blockCount=0, vCores=8, mem=32768, isMaster=0
dbagent: NLAM-CLUSTER03: [empty], blockCount=0, vCores=8, mem=32768, isMaster=0
dbagent: NLAM-CLUSTER04: [empty], blockCount=0, vCores=8, mem=32768, isMaster=0
dbagent: Chosen worker-set: NLAM-CLUSTER02 NLAM-CLUSTER01 NLAM-CLUSTER03
dbagent: Initialize worker-set: mpi_fork
dbagent: Partition IDs: [0c1, 0c12, 1c12, 2c12, 3c12, 4c12, 5c12, 6c12, 7c12, 8c12, 9c12, 10c12, 11c12] // see Section 3.2 for the IDs meaning
```

Two table locations, the default location (hdfs:///dblocation/dbname/default) and a custom location (hdfs:///custom_table_location) defined during schema creation, are added for the HDFS scan process. The *regex file pattern* we are looking for is .+@[0-9]+c[1-9]+[0-9]* and matches any file whose name complies with the *<table-name>***@***<partition-number>***c***<partitioning-class>* convention. After the *worker-set* is chosen, the *mpi_fork* command is used to initialize the workers' backend (X100) processes.

Given that we have not stored data in the system yet, the DbAgent performs a *first hand replica placement* using the horizontal round-robin assignment scheme from Section 4.2, Algorithm 4.1:

```
dbagent: First hand replica placement (round-robin), HDFS replication degree = 2
dbagent: For pClass = 1 (non-partitioned)
dbagent: Node NLAM-CLUSTER02, max-partition-load is 1
dbagent: Node NLAM-CLUSTER01, max-partition-load is 1
dbagent: Node NLAM-CLUSTER03, max-partition-load is 0
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [empty][0c1]
dbagent: NLAM-CLUSTER01: [empty][0c1]
dbagent: NLAM-CLUSTER03: [empty][___]
dbagent: For pClass = 12 (partitioned)
dbagent: Node NLAM-CLUSTER02, max-partition-load is 8
dbagent: Node NLAM-CLUSTER01, max-partition-load is 8
dbagent: Node NLAM-CLUSTER03, max-partition-load is 8
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [empty][0c1, 0c12, 3c12, 6c12,  9c12, 2c12, 5c12, 8c12, 11
    c12]
dbagent: NLAM-CLUSTER01: [empty][0c1, 1c12, 4c12, 7c12, 10c12, 0c12, 3c12, 6c12,  9
    c12]
dbagent: NLAM-CLUSTER03: [empty][___, 2c12, 5c12, 8c12, 11c12, 1c12, 4c12, 7c12, 10
    c12]
dbagent: (Re)Assigning responsibilities to worker nodes, RMax = 1
dbagent: For pClass = 12 (partitioned)
dbagent: [0c12, 1c12, 2c12, 3c12, 4c12, 5c12, 6c12, 7c12, 8c12, 9c12, 10c12, 11c12]
dbagent: Node NLAM-CLUSTER02, max-partition-load is 4
dbagent: Node NLAM-CLUSTER01, max-partition-load is 4
dbagent: Node NLAM-CLUSTER03, max-partition-load is 4
dbagent: Assignment done: flow = 12, cost = 12
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [0c12, 3c12,  5c12,  9c12][0c1, 0c12, 3c12, 6c12,  9c12, 2
    c12, 5c12, 8c12, 11c12]
dbagent: NLAM-CLUSTER01: [1c12, 4c12,  6c12,  7c12][0c1, 1c12, 4c12, 7c12, 10c12, 0
    c12, 3c12, 6c12,  9c12]
dbagent: NLAM-CLUSTER03: [2c12, 8c12, 10c12, 11c12][___, 2c12, 5c12, 8c12, 11c12, 1
    c12, 4c12, 7c12, 10c12]
dbagent: (Over)Writing the worker-set plocations metadata file
dbagent: (Over)Writing the worker-set passignment metadata file
```

We now fast-forward to a situation where we have bulk-loaded data in the Vectorwise database, 1 GB per partition (a non-partitioned table of 1 GB and a partitioned table of 12 GB). If nothing went wrong (i.e. node failure) happens to the system and we restart the Vectorwise on Hadoop database using the same request parameters, we notice that *(1) HDFS finds the Vectorwise files' locations* and that *(2) we can automatically match the table partitions to the worker nodes* with respect to data-locality and load-balancing factors. Since these files are still located on the same workers (nothing is changed compared to the initial round-robin assignment), all 12 partitions are distributed equally among our worker-set with cost 0. The same applies to assigning responsibilities: DbAgent assigns 4 responsibilities per worker node with 0 costs, it uses the last responsibility assignment (historical information) to adjust the cost model for the local "OS cached" partitions, Section 4.2. We can see all these steps in the following section of the log file:

```
dbagent: Scan HDFS file system for Vectorwise files...
dbagent: Location added to HDFS scan: hdfs:///dblocation/dbname/default [1-file]
dbagent: Location added to HDFS scan: hdfs:///custom_table_location [12-files]
dbagent: The pattern we are looking for: .+@[0-9]+c[1-9]+[0-9]*
dbagent: Number of Vectorwise files: 13
dbagent: Sort Vectorwise cluster nodes by resources...
dbagent: Extended cluster node report (sorted by resources):
dbagent: <hostname>: [local-partitions], block-counts, virtual-cores, memory-size,
    session-master or not
dbagent: NLAM-CLUSTER02: [0c1, 0c12, 3c12, 6c12,  9c12, 2c12, 5c12, 8c12, 11c12],
    blockCount=72, vCores=8, mem=32768, isMaster=1
```

```
dbagent: NLAM-CLUSTER01: [0c1, 1c12, 4c12, 7c12, 10c12, 0c12, 3c12, 6c12,  9c12],
    blockCount=72, vCores=8, mem=32768, isMaster=0
dbagent: NLAM-CLUSTER03: [___, 2c12, 5c12, 8c12, 11c12, 1c12, 4c12, 7c12, 10c12],
    blockCount=64, vCores=8, mem=32768, isMaster=0
dbagent: NLAM-CLUSTER04: [], blockCount=0, vCores=8, mem=32768, isMaster=0
dbagent: Chosen worker-set: NLAM-CLUSTER02 NLAM-CLUSTER01 NLAM-CLUSTER03
dbagent: Initialize worker-set: mpi_fork
dbagent: Partition IDs: [0c1, 0c12, 1c12, 2c12, 3c12, 4c12, 5c12, 6c12, 7c12, 8c12, 9
    c12, 10c12, 11c12]
dbagent: (Re)Matching table partitions to worker nodes, HDFS replication degree = 2
dbagent: No missing replicas
.......
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [][0c1, 0c12, 3c12, 6c12,  9c12, 2c12, 5c12, 8c12, 11c12]
dbagent: NLAM-CLUSTER01: [][0c1, 1c12, 4c12, 7c12, 10c12, 0c12, 3c12, 6c12,  9c12]
dbagent: NLAM-CLUSTER03: [][___, 2c12, 5c12, 8c12, 11c12, 1c12, 4c12, 7c12, 10c12]
dbagent: (Re)Assigning responsibilities to worker nodes, RMax = 1
dbagent: For pClass = 12 (partitioned)
dbagent: [0c12, 1c12, 2c12, 3c12, 4c12, 5c12, 6c12, 7c12, 8c12, 9c12, 10c12, 11c12]
dbagent: Passignment metadata file exists: adjusts costs for ex-responsibilities
dbagent: Node NLAM-CLUSTER02, max-partition-load is 4
dbagent: Node NLAM-CLUSTER01, max-partition-load is 4
dbagent: Node NLAM-CLUSTER03, max-partition-load is 4
dbagent: Assignment done: flow = 12, cost = 0
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [0c12, 3c12,  5c12,  9c12][0c1, 0c12, 3c12, 6c12,  9c12, 2
    c12, 5c12, 8c12, 11c12] // ex-resp.
dbagent: NLAM-CLUSTER01: [1c12, 4c12,  6c12,  7c12][0c1, 1c12, 4c12, 7c12, 10c12, 0
    c12, 3c12, 6c12,  9c12] // ex-resp.
dbagent: NLAM-CLUSTER03: [2c12, 8c12, 10c12, 11c12][___, 2c12, 5c12, 8c12, 11c12, 1
    c12, 4c12, 7c12, 10c12] // ex-resp.
dbagent: (Over)Writing the worker-set plocations metadata file
dbagent: (Over)Writing the worker-set passignment metadata file
```

However, when a worker node fails (the third node), the DbAgent removes it from the Vectorwise worker-set and selects a new Hadoop node as a replacement for the failed one. Missing replicas are identified and so, the new node is used for re-replication. In our case, the cost model for matching table partitions to worker nodes keeps the local partitions that were already stored among the 1st and 2nd nodes intact and assigns (re-replicates) those that belonged to the 3rd node to the 4th node instead (its partition capacity is 8). The cost that we get after performing the min-cost flow is 8, equal to the number of partitions to be re-assigned and scheduled for re-replication to node 4. Our 4th (new empty) node did not have any local data beforehand. In the bipartite graph, all its in-edges from the table partitions were labeled with cost *1 ∼ remote access*. Furthermore, the new node was not part of the last Vectorwise worker-set, so it is assigned 4 responsibilities with cost 4 – basically the same responsibilities the 3rd node previously had. But, since these responsibilities did not appear in the last responsibility assignment, the costs were not adjusted to *0 ∼ local+*. The following log section confirms what we have just explained:

```
dbagent: Scan HDFS file system for Vectorwise files...
dbagent: Location added to HDFS scan: hdfs:///dblocation/dbname/default [1-file]
dbagent: Location added to HDFS scan: hdfs:///custom_table_location [12-files]
dbagent: The pattern we are looking for: .+@[0-9]+c[1-9]+[0-9]*
dbagent: Number of Vectorwise files: 13
dbagent: Sort Vectorwise cluster nodes by resources...
dbagent: Extended cluster node report (sorted by resources):
dbagent: <hostname>: [local-partitions], block-counts, virtual-cores, memory-size,
    session-master or not
dbagent: NLAM-CLUSTER02: [0c1, 0c12, 3c12, 6c12,  9c12, 2c12, 5c12, 8c12, 11c12],
    blockCount=72, vCores=8, mem=32768, isMaster=1
dbagent: NLAM-CLUSTER01: [0c1, 1c12, 4c12, 7c12, 10c12, 0c12, 3c12, 6c12,  9c12],
    blockCount=72, vCores=8, mem=32768, isMaster=0
```

```
dbagent: NLAM-CLUSTER04: [], blockCount=0, vCores=8, mem=32768, isMaster=0
dbagent: Chosen worker-set: NLAM-CLUSTER02 NLAM-CLUSTER01 NLAM-CLUSTER04
dbagent: Initialize worker-set: mpi_fork
dbagent: Partition IDs: [0c1, 0c12, 1c12, 2c12, 3c12, 4c12, 5c12, 6c12, 7c12, 8c12, 9
    c12, 10c12, 11c12]
dbagent: (Re)Matching table partitions to worker nodes, HDFS replication degree = 2
dbagent: Missing replicas from [2c12, 5c12, 8c12, 11c12, 1c12, 4c12, 7c12, 10c12]
dbagent: For pClass = 1 (non-partitioned)
dbagent: Node NLAM-CLUSTER02, max-partition-load is 1
dbagent: Node NLAM-CLUSTER01, max-partition-load is 1
dbagent: Node NLAM-CLUSTER04, max-partition-load is 0
dbagent: Assignment done: flow = 2, cost = 0
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [][0c1] // local already
dbagent: NLAM-CLUSTER01: [][0c1] // local already
dbagent: NLAM-CLUSTER04: [][___]
dbagent: For pClass = 12 (partitioned)
dbagent: Node NLAM-CLUSTER02, max-partition-load is 8
dbagent: Node NLAM-CLUSTER01, max-partition-load is 8
dbagent: Node NLAM-CLUSTER04, max-partition-load is 8
dbagent: Assignment done: flow = 24, cost = 8
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [][0c1, 0c12, 3c12, 6c12,  9c12, 2c12, 5c12, 8c12, 11c12] //
     local already
dbagent: NLAM-CLUSTER01: [][0c1, 1c12, 4c12, 7c12, 10c12, 0c12, 3c12, 6c12,  9c12] //
     local already
dbagent: NLAM-CLUSTER04: [][___, 2c12, 5c12, 8c12, 11c12, 1c12, 4c12, 7c12, 10c12] //
     re-replicate
dbagent: (Re)Assigning responsibilities to worker nodes, RMax = 1
dbagent: For pClass = 12 (partitioned)
dbagent: [0c12, 1c12, 2c12, 3c12, 4c12, 5c12, 6c12, 7c12, 8c12, 9c12, 10c12, 11c12]
dbagent: Passignment metadata file exists: adjusts costs for ex-responsibilities
dbagent: Node NLAM-CLUSTER02, max-partition-load is 4
dbagent: Node NLAM-CLUSTER01, max-partition-load is 4
dbagent: Node NLAM-CLUSTER04, max-partition-load is 4
dbagent: Assignment done: flow = 12, cost = 4
dbagent: Assignment stats:
dbagent: <hostname>: [responsibilities][local-partitions]
dbagent: NLAM-CLUSTER02: [0c12, 3c12,  5c12,  9c12][0c1, 0c12, 3c12, 6c12,  9c12, 2
    c12, 5c12, 8c12, 11c12] // ex-resp.
dbagent: NLAM-CLUSTER01: [1c12, 4c12,  6c12,  7c12][0c1, 1c12, 4c12, 7c12, 10c12, 0
    c12, 3c12, 6c12,  9c12] // ex-resp.
dbagent: NLAM-CLUSTER04: [2c12, 8c12, 10c12, 11c12][___, 2c12, 5c12, 8c12, 11c12, 1
    c12, 4c12, 7c12, 10c12] // new-resp.
dbagent: (Over)Writing the worker-set plocations metadata file
dbagent: (Over)Writing the worker-set passignment metadata file
```

4.3 Dynamic Resource Scheduling

A dynamic resource scheduling approach in Vectorwise on Hadoop implies that cluster compute resources should be now scheduled dynamically according to:

- the worker-set resource availability,
- load-balancing constraints,
- and data-locality.

The goal of such an approach is to improve the Vectorwise on Hadoop system's performance in two situations: **(a)** during failovers and **(b)** when two or more internal (analytical queries) and external (Map-Reduce jobs) concurrent workloads overlap on the same worker nodes. The outcome of the resource scheduler is called a *footprint* and contains runtime information about the *worker nodes*, the *responsibilities* (table partitions), and the *number of threads per worker* that are involved in query execution. We refer to the query's footprint in next paragraphs as the *resource footprint*.

The problem to solve is not trivial at all. By all means, a good approach should (1) inform X100 about the resource availability (e.g. number of available cores, free amount of memory), (2) achieve the *maximum parallelism level* (or *mpl*, i.e. the maximum number of threads to use for that query's execution) of a query by just using the workers' available resources proportionally to their partition locality, (3) determine, given the responsibility assignment, what are the worker nodes and partitions to be involved in the computation, and, at the same time, (4) improve data-locality. More details about (1), plus on how we use YARN to claim the *resource footprint*, are presented in Chapter 5.

On the other hand, to tackle (2), (3), and (4), we have enhanced the existent Vectorwise resource scheduler with a min-cost flow algorithm. Though the algorithm uses a slightly more advanced cost model, Figure 4.5, the flow network (bipartite graph) is partitioned exactly as before. In addition, we extend our two previous ideas, *matching partition tables* and *assigning responsibilities* to worker nodes, and use the last algorithm's (Algorithm 4.3) result as input to our new problem. As will see, the min-cost flow algorithm's implementation, the *successive shortest path* [49], is the common nominator of this chapter.

Successive Shortest Path. The successive shortest path algorithm searches for the maximum flow and optimizes the objective cost function simultaneously. It solves the so-called max-flow min-cost problem by using the following idea. Suppose we have a flow network G and we have to find an optimal flow across it. As we described in Section 2.3, the *flow network* is a directed graph defined by a set V of vertexes (nodes) and set E of edges (arcs). For each edge we associate a capacity u_{ij}, denotes the maximum amount of *flow* that can pass through the edge. Each edge also has an associated cost c_{ij} that stands as the cost per unit flow. We associate with each vertex a number b_i. This values represent the *supply* or the *demand* of the vertex. If $b_i > 0$, node i is a *supply node*, else if $b_i < 0$, node i is a *demand node* (its demand is equal to $-b_i$). We call vertex i a *transportation node* if b_i is zero. We further transform the network by adding two vertexes s and t (source and destination) and some edges as follows: for each node i in V (the set of vertexes), we add a source arc (s, i) with capacity $u_{si} = b_i$ and $c_{si} = 0$. For each node i in V with $b_i < 0$, we add a sink edge (i, t) with capacity $u_{it} = -b_i$ and $c_{it} = 0$. Then, instead of searching for the maximum flow as usual, we send flow from s to t along the shortest path (with respect to arc costs). Next we update the residual network, find another shortest path and augment the flow again, and so on. The algorithm terminates when the residual network contains no path from s to t (the flow is maximal). A high-level pseudocode to summarize the before-mentioned steps is as follows:

```
1 // Successive Shortest Path Algorithm
2 Transform network G by adding source (s) and destination (t)
3 Initial flow x is zero
4 while ( G(x) contains a path from s to t ) do
5   Find any shortest path P from s to t
6   Augment current flow x along P
7   Update G(x)
```

Since the flow is maximal, it corresponds to a feasible solution of the original min-cost flow problem. Moreover, this solution will be optimal [49]. The successive shortest path algorithm can be used when G contains no negative cost cycles. Otherwise, we cannot say exactly what the shortest path means.

The *successive shortest path* pseudocode from above was molded to our purposes (for Figure 4.2, 4.4, 4.5 flow networks). The transformation from *line 2* applies to a bipartite graph G instead. In this case $V = \{P, W\}$ is the set of vertexes, where $P = \{partitions, ...\}$ is the *supply nodes* set ($|P| = pClass$), $W = \{workers, ...\}$ is the *demand nodes* set ($|W| = N$) and E represents the set of edges from P to W. The bipartite graph has no *transportation nodes* and s and t serve the same roles. From *line 3 to 6* there are no modifications as such. Moreover, as the cost model suggests, during the *bipartite graph's update* (*update G(x)*) at *line 7* the cost of a (w_i, t) sink edge, from a worker node (w_i) to the destination node (t), is increased each time we find the *successive shortest path*. Our own implementation is shown in Algorithm 4.4. At *line 3* we find the shortest path from the flowNet and from *line 6 to 8* we augment the current flow. From *12 to 21* we update the residual graph, total minimum cost, and total flow.

Algorithm 4.4 Solving the min-cost flow problem (*successive shortest path* algorithm): *minCostFlow(flowNet, balanced)*

Input: *flowNet:* the flow network built using the cost model, *balanced:* whether or not to increase the sink edges' cost each time we find a *min cost path* from s to t (load-balanced approach)

Output: the residual network (a min-cost load-balanced matching in bipartite graph)

```
1:  $C \leftarrow 0$
2:  $F \leftarrow 0$
3:  while $\exists path \leftarrow dijkstra(flowNet)$ do // repeatedly find the min cost path from $s$ to $t$ to which we can increase their capacities with a unit of flow
4:    // (1) get the bottleneck delta capacity: find the minimum residual capacity of the edges belonging to the above path; the residual capacity of an edge $e$ is capacity($e$) - flow($e$)
5:    $delta \leftarrow \infty, s \leftarrow path.s, t \leftarrow path.t$
6:    for $v \leftarrow t, u \leftarrow path.parent(v); v \neq s; u \leftarrow path.parent(v \leftarrow u)$ do
7:      $delta \leftarrow MIN(flowNet[u][v].cap - flowNet[u][v].flow, delta)$
8:    end for
9:    $nodeId \leftarrow path.parent(t).id$
10:   threadCost $\leftarrow balanced \ ? \ \frac{w_i}{availableCores(nodeId)} : 0$
11:   // (2) augment the flow network: again, following parent of $t$, add delta to flow(e) and subtract it from flow(rev-e); the network capacities are not altered but the network costs of workers to $t$ are updated at each step with the threadCost value
12:   for $v \leftarrow t, u \leftarrow path.parent(v); v \neq s; u \leftarrow path.parent(v \leftarrow u)$ do
13:     if $v = t$ then
14:       $flowNet[u][v].cost \leftarrow flowNet[u][v].cost +$ threadCost
15:       $flowNet[v][u].cost \leftarrow flowNet[v][u].cost -$ threadCost
16:     end if
17:     $C \leftarrow C + delta * flowNet[u][v].cost$
18:     $flowNet[u][v].flow \leftarrow flowNet[u][v].flow + delta$
19:     $flowNet[v][u].flow \leftarrow flowNet[v][u].flow - delta$
20:   end for
21:   $F \leftarrow F + delta$
22: end while
23: return $residualNet \leftarrow flowNet$
```

Cost model. The cost model, which decides what workers and table partitions to use for a query workload, is now computed dynamically at query runtime not at (the *DbAgent's*) startup time, in contrast with the two previous models. All graph's edges from source s to *partition-IDs* have cost 0 and a capacity equal to $RMax' \leq RMax$, where $RMax$ is the *maximum responsibility degree* and *Rmax′* is a fixed value that is used to *restrict* the latter *responsibility degree* at query runtime. $RMax$ is defined in DbAgent, whereas $RMax'$ is a X100 configuration parameter. We argue that $RMax'$ should be set to **1**, case in which the min-cost flow algorithm (based on our cost model) returns a one-to-one (responsibility) assignment. The reason behind this is discussed later. Continuing, from the *partition-IDs* to the *workers*, the edge capacities are 1 and the costs are chosen from the $\{resp+, resp-, remote\}$ range set:

- $resp+ = LL_i$, the worker's up-to-date *load factor*

  – this capacity is attributed to (partition) *responsibilities* which are *local* to the worker nodes

  – the *new* LL_i *load factor* is aware of both, the *internal* and *external* workloads (i.e. the number of used cores)

  – it is based on Section 1.3.2's *load factor* early formula, except that T_i (the thread load on worker i) is now redefined as $\frac{internalThreads_i + externalThreads_i}{C_i}$, where C_i is the *total number of cores* on worker i (i.e. the max cores capability)

  – $resp+$ cost should be able to make a difference between the various *load factors*, such that we favor the least loaded nodes for running the current Vectorwise query

- $resp- = LL_i + w_i$, the worker's *load factor* plus a penalty

  – this capacity is attributed to (partition) *responsibilities* which are *not yet local* to the worker nodes (i.e. are still during re-replication), we refer to it with *partial* (partition) responsibilities

  – adding the w_i thread weight (Section 1.3.2) to the upper formula creates a cost gap between the *local* and *partial* (partition) responsibilities; we choose w_i just to be consistent with the surrounding developed code, as well this penalty can be set to any different constant value

  – the cost gap is reduced when almost all *local* (partition) responsibilities have been assigned while running the min-cost flow algorithm, in which case some of the *partial* (partition) responsibilities will start being assigned to the least loaded nodes

  – to determine the *partial* (partition) responsibilities from the *passignment* metadata, an up-to-date snapshot of the Vectorwise block locations is needed; more about this in next Chapter 5

- $remote = LL_i + w_i * penalty$, the worker's *load factor* plus a magnified penalty

  – this capacity is attributed to all the remaining partitions of which the worker nodes are not responsible for

  – a 1.25 value for the *penalty* is determined experimentally to take in account (as its best) *remote* (partition) responsibilities; we multiply w_i by this penalty in order to reduce the likelihood of *remote* (partition) responsibilities being assigned (over *local* or even *partial* ones) to worker nodes while running the min-cost flow algorithm

Finally, each edge from a *worker* to the destination vertex t has the capacity equal to the worker's *partition capacity* ($PCap$, see Section 4.1) and its cost is expressed as a $TCost : \mathbb{N} \rightarrow \mathbb{N}$ function, $TCost(f_x) = f_x * threadCost$ – increases each time we select the *shortest path* during the *while-loop* (step x) of Algorithm 4.4. Note that, we consider $RMax \leq R$ to benefit from data-locality and to make sure that all of a worker's (partition) *responsibilities* are either *local* or, at most, *partial* responsibilities. Moreover, because we do not have any memory requirements for queries as such, the worker's *partition capacity* is simply figured as $PCap_i = availableCores_i * core\_overalloc$, $i = 1, \ldots, N$, where $availableCores_i$ information is always up-to-date and $core\_overalloc$ is a system configuration parameter.

We shortly remind that in Vectorwsie *distributed MPP version* the *resource scheduling* algorithm provides us a list of pairs ($\mathrm{n_x}$, $\mathrm{t_y}$) denoting how many threads/cores ($\mathrm{t_y}$) are allocated for its execution on node ($\mathrm{n_x}$). The search starts with a feasible (initial) solution and tries/verifies all its multiple (e.g. $[(1,3),(2,5)] : [(1,6),(2,10)] : [(1,9),(2,15)] : etc\ldots$). It stops at the first combination which uses more threads (on any machine) than allocated by the resource scheduler. Moreover, from Section 1.3.3, we know that the algorithm uses two multiple-nodes

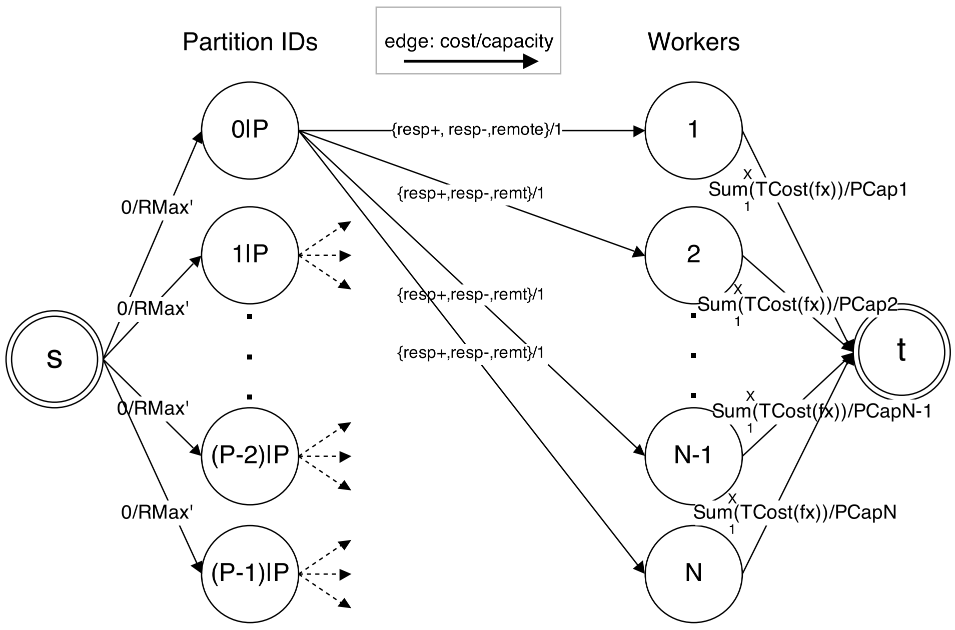


Figure 4.5: The flow network (bipartite graph) model used to calculate a part of the *resource footprint* (what nodes and table partitions we use for a query, given the worker's available resource and data-locality). Note: f_x is the unit with which we increase the *flow* along the path at step x of the algorithm.

distribution policies for queries: *(1) equal share for non-partitioned tables*, which similar with the multiple-nodes distribution used in the Shared Disk approach it selects the least loaded nodes and takes their amount of CPU resources until it satisfies the *mpl* and *(2) equal share for partitioned tables* that assumes the default round-robin partitioning and equally assigns threads to the operators until the *mpl* is achieved.

The new dynamic approach tries to combine these policies within a min-cost flow algorithm. So, ideally, we must *find the smallest subset of workers* that *fulfills* the *maximum parallelism level* of a query with respect to the worker-set current *data-locality* and *load-balancing* constraints imposed by the outside world (i.e. other workloads sharing the same Hadoop infrastructure). In this form, the problem is similar to the *weighted set covering problem* [51, 52]. Given a set of elements $U = \{1, 2, ..., m\}$ (called the *universe*) and a S of n sets whose union equals the universe, the *set covering problem* [51] is to identify the smallest subset of S whose union equals the universe. For example, consider the universe $U = \{1, 2, 3, 4, 5, 6\}$ and the set of sets $S = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{5, 6\}\}$. Clearly the union of S is U. However, we can cover all of the elements with one of the following, smaller number of sets solutions: $\{\{1, 2, 3\}, \{3, 4\}, \{5, 6\}\}$, $\{\{1, 2, 3\}, \{2, 4\}, \{5, 6\}\}$, or $\{\{1, 2, 3\}, \{4, 5\}, \{5, 6\}\}$. More formally, given the universe U and a family S of subsets of U, a cover is a subfamily $C \subseteq S$ of sets whose union is U. In the set covering decision problem, the input is a pair (U, S) and an integer k; the question is whether there is a set covering of size k or less. In the set covering optimization problem, the input is a pair (U, S), and the task is to find a set covering that uses the fewest sets. The decision version of the set covering problem is *NP-complete* [53], and the optimization version is *NP-hard.* If additionally, you want to minimize the cost of the set cover (costs > 1), the problem becomes a *weighted set covering problem.* The latter is by definition an *optimization* variant as well, which makes it *NP-hard* to solve.

Moreover, the set covering problem can be reduced to bipartite graphs: an instance of the set covering can be viewed as an arbitrary *bipartite graph*, with sets represented by vertices on the left, the universe represented by vertices on the right, and edges representing the inclusion of elements in sets. The task is then to find a minimum cardinality subset of left-vertices which covers all of the right-vertices. In the *hitting set problem* citeset-covering-hitting-set, the objective is to cover the left-vertices using a minimum subset of the right vertices. Converting from one problem to the other is therefore achieved by interchanging the two sets of vertices. If the graph's edges has costs then it becomes the *weighted hitting set problem*. Though it sounds similar to a min-cost flow problem, just applying the *successive shortest path* algorithm will not result in finding the perfect (optimal cost minimal subset) solution. Only the representation of the problem changes, but its complexity (NP-hard) remains the same. Any general polynomial-time algorithm, such as the *successive shortest path*, that always outputs the optimal solution to our optimization problem would imply that $P{=}NP$ (which is quite improbable). Instead, what we actually need is an approximation algorithm for the problem, or, more likely, an *heuristic algorithm*.

One thing to do, is to take advantage of the HDFS local replicas. Increasing the *RMax* (the *maximum responsibility degree*, see Section 4.2) value per partition, up to the HDFS replication degree, adds a certain elasticity in computing an optimal *resource footprint* for a query workload. With it, we can vary the available resources from *1 thread per node* to *the maximum threads possible per partition* and, at the same time, choose what worker nodes to involve in the computation. In this way, we can reduce the (resource) contention with other jobs sharing the same cluster. For instance, the algorithm can choose just a few workers from the worker-set to run a Vectorwise query. This may happen in case some workers are overloaded or facing other problems, e.g. no local-data due to a failure. With *RMax' = 1* we can restrict the *maximum responsibility degree* at runtime and instead compute multiple *one-to-one assignment* (optimal cost) solutions (e.g. different workers, different partitions, etc.), of which just some will be of minimum size as well. It all depends on the data-locality, load-factors, etc., for the flow-algorithm to determine a starting feasible solution, a one-to-one assignment that allows us to scale with the number of threads up to the *maximum level of parallelism*. The latter, as will see in Chapter 5, leads to computing the *resource footprint*. Of course, for this to happen, the HDFS replication degree should be ≥ 2 and so the *RMax* as well.

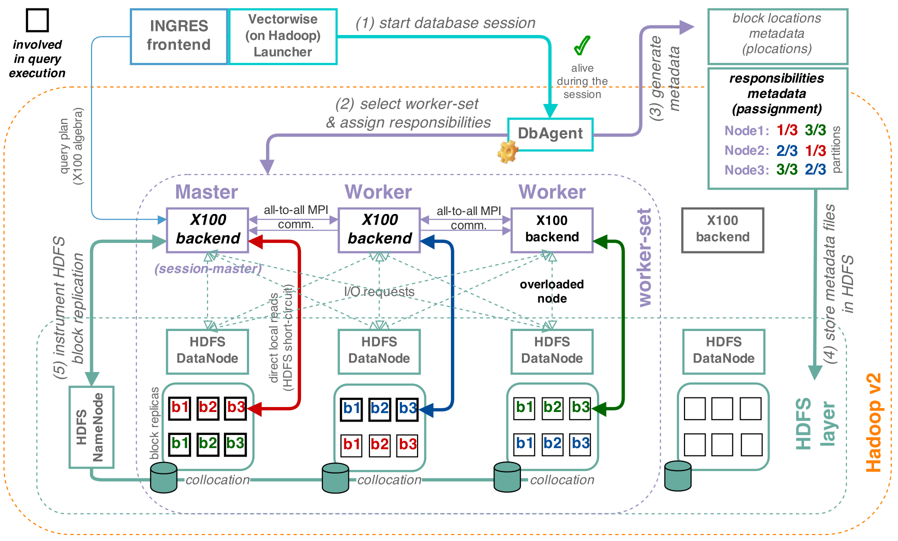

involved in query execution
INGRES frontend
Vectorwise (on Hadoop) Launcher
(1) start database session
alive during the session
(3) generate metadata
block locations metadata (plocations)
responsibilities metadata (passignment)
Node1: 1/3 3/3
Node2: 2/3 1/3
Node3: 3/3 2/3
partitions
(2) select worker-set & assign responsibilities
DbAgent
query plan (X100 algebra)
Master
Worker
Worker
X100 backend
all-to-all MPI comm.
(session-master)
I/O requests
overloaded node
worker-set
(4) store metadata files in HDFS
(5) instrument HDFS block replication
HDFS DataNode
direct local reads (HDFS short-circuit)
block replicas
b1 b2 b3
HDFS NameNode
collocation
HDFS layer
Hadoop v2

Figure 4.6: Selecting the (partition responsibilities and worker nodes involved in query execution. Only the first two workers from left are chosen (bold-italic black) for table scans; first one reads from two partitions (red and green), whereas the second reads just from one (blue). The third worker is considered *overloaded* and so, it is excluded from the I/O. Red/blue/green colors are used to express *multiple* (partition) *responsibilities*.

Figure 4.6 depicts the concepts discussed in this section, exemplifying the latter case when an overloaded worker node is excluded from scanning.

Last but not least, is to rely on the min-cost flow algorithm to compute an optimal cost solution and, for the minimum size condition, to add some greedy heuristics in our implementation. Therefore, we came up with a *shrinking procedure:* an approach to cut (one by one *recursively*) the most overloaded worker nodes with the least available cores capacity, while running the *successive shortest path* algorithm to (maintain and) verify that the cost is still optimal and, more importantly, that the maximum level of parallelism can be achieved.

In Algorithm 4.5, we use both helper methods 4.6 and 4.7 (in this order) to build-up and return the *resource footprint (workerSubset, responsibilities, maximum threads per worker)* tuple. The latter is required to claim the compute resources from YARN and proceed to running the Vectorwise query, Chapter 5.

Algorithm 4.5 Apply the shrinking procedure & balance the *mpl*:
footprint(workerSet, mpl, numQueries)

```
Input: workerSet: the Vectorwise worker-set, mpl: the maximum parallelism level, numQueries: the number of queries running in the system
Output: the resource footprint
1: $costModel.setRMax'(\mathbf{1})$
2: $flowNet \leftarrow buildFlowNet(costModel)$
3: $residualNet \leftarrow minCostFlow(flowNet, balanced \leftarrow true)$
4: $optCost \leftarrow cost(residualNet)$
5: $(maxLoad, leastCores) \leftarrow$ the max load and, on equality, with the least number of cores from the $workerSet$
6: for all $worker : workerSet$ do
7:    if $worker \simeq (maxLoad, leastCores)$ then
8:       // apply shrinkworkerSet(...) recursively
9:       $result \leftarrow shrinkworkerSet(workerSet, mpl, numQueries, worker.getIndex(), optCost)$
10:      // choose the best worker-subset: with the maximum cores available
11:      $workerSubset \leftarrow best(workerSet, result)$
12:   end if
13: end for
14: $costModel.setworkerSet(workerSubset)$
15: $flowNet \leftarrow buildFlowNet(costModel)$
16: $residualNet \leftarrow minCostFlow(flowNet, balanced \leftarrow true)$
17: $pAssignment \leftarrow residualNet.getMatchings()$
18: $maxThreadsPerWorker \leftarrow balanceMpl(workerSubset, pAssignment, mpl)$
19: return $footprint \leftarrow (workerSubset, pAssignment, maxThreadsPerWorker)$
```

Algorithm 4.6 presents the worker-set recursive *shrinking* process, which seeks to minimize the optimal cost solution. When the minimum worker-subset is found and all the above conditions are met, we can start setting the *maximum parallelism level* (*mpl*, or the maximum number of threads allowed to use for a query run) for the current subset. What we mean by setting the *mpl*, is to distribute (or balance) the number of threads (equal to *mpl*) over the worker-set, proportionally to the worker's local data and available CPU (#threads) resources. The pseudocode for the latter is shown in Algorithm 4.7.

Algorithm 4.6 Shrinking procedure:
shrinkWorkerSet(workerSet, mpl, numQueries, remIndex, optCost)

Input: *workerSet:* the full worker-set (to start the shrinking with), *mpl:* the maximum parallelism level, *numQueries:* the number of queries running in the system, *remIndex:* the worker's node index to be removed from the worker-set, *optCost:* the optimal cost to achieve
Output: the minimum worker-subset that has all of our (query) data and tries to achieve the maximum parallelism level, while maintaining the optimal cost

```
1: // remove the worker from the set
2: $workerSubset \leftarrow workerSet.removeWorker(remIndex)$
3: $totalAvailableCores \leftarrow totalAvailableCores(workerSubset)$ // total number of available cores without the 'removed' worker node
4: if $totalAvailableCores * core\_overalloc < mpl$ then
5:   return $workerSet$ // nothing changes, since we cannot achieve the mpl
6: end if
7: $costModel.setworkerSet(workerSubset)$
8: $flowNet \leftarrow buildFlowNet(costModel)$
9: $residualNet \leftarrow minCostFlow(flowNet, balanced \leftarrow (numQueries > 1))$
10: $newCost \leftarrow residualNet.getFlowCost()$
11: if $newCost > optCost$ then
12:   return $workerSet$ // nothing changes, since we cannot maintain the optimal cost
13: end if
14: $(maxLoad, leastCores) \leftarrow$ the max load and, on equality, with the least number of cores from the $workerSubset$
15: for all $worker : workerSubset$ do
16:   if $worker \simeq (maxLoad, leastCores)$ then
17:     // apply shrinkworkerSet(...) recursively
18:     $result \leftarrow shrinkworkerSet(workerSubset, mpl, numQueries, worker.getIndex(), newCost)$
19:     // choose the best worker-subset: minimum size with maximum cores available
20:     $workerSubset \leftarrow best(workerSubset, result)$
21:   end if
22: end for
23: return $workerSubset$
```

Algorithm 4.6's pseudocode describes how to shrink the worker-set and so, to minimize the optimal cost solution. At *line 2* we remove the worker node whose index was chosen in a previous iteration of this recursive algorithm. Afterwards, from *lines 6 to 9* we recompute the cost model and apply the min-cost flow algorithm to get the new cost. The two if-conditions from *lines 4 and 11* verify that the we can still *(1) fulfill the mpl* and *(2) maintain the optimal cost* at the same time. From *lines 15 to 22* we find the next worker to be removed (with max-load and least available cores), apply again the *shrinking* function recursively, and select the best worker-subset among all future solutions. Important to note is that, at *line 9*, we choose to disable the balanced property of the min-cost flow algorithm ($balanced \leftarrow false$) when a single query is active in the system. Otherwise, the algorithm will tend to use more nodes to balance the flow and lower the costs, which contradicts with our goal to actually minimize the worker-set size.

Algorithm 4.7 Balance the *mpl* (#threads) on top of the current workers-(sub)set (∼ proportionally to the worker's local data and available resources):
balanceMpl(workerSet, pAssignment, mpl)

Input: *workerSet:* the workers-(sub)set, *pAssignment:* the responsibility assignment with respect to the new workers-(sub)set *mpl:* the maximum parallelism level
Output: array of maximum number of threads per worker node

```
1: // distribute the mpl bounded number of threads over the worker-set, proportionally to the worker's local
   data (i.e. remaining partitions) and available CPU (#cores) resources
2: $remaining \leftarrow pAssignment.getPartitionsNum()$
3: for all $worker : workerSet \wedge mpl > 0$ do
4:    $current \leftarrow pAssignment.getPartitionsFor(worker).size$
5:    if $current = 0$ then
6:       $cntZero \leftarrow cntZero + 1$
7:       continue
8:    end if
9:    $ratio \leftarrow \frac{mpl}{remaining}$
10:   $threads \leftarrow round(ratio * current)$
11:   $threads \leftarrow MIN(threads, worker.getAvailableCores() * core\_overalloc)$
12:   $worker.setThreads(threads)$
13:   $remaining \leftarrow remaining - current$
14:   $mpl \leftarrow mpl - threads$
15: end for
16: if $cntZero = 0 \vee mpl = 0$ then
17:    return $maxThreadsPerWorker \leftarrow threadsPerWorker(workerSet)$
18: end if
19: // equally redistribute the remaining threads (mpl) to what is left from the workers-(sub)set, to those
    worker nodes that were not assigned with any partitions; if there are no such workers left, then the mpl is
    reduced to the cluster's available CPU (#cores) capacity (with over-allocation)
20: $equalShare \leftarrow mpl/cntZero$
21: $plusOne \leftarrow mpl\%cntZero$
22: $i \leftarrow 0$
23: for all $worker : workerSet \wedge mpl > 0$ do
24:    $current \leftarrow pAssignment.getPartitionsFor(worker).size$
25:    if $current > 0$ then
26:       continue
27:    end if
28:    $threads = i < plusOne\ ?\ (equalShare + 1) : equalShare$
29:    $worker.setThreads(threads)$
30:    $mpl \leftarrow mpl - threads$
31:    $i \leftarrow i + 1$
32: end for
33: return $maxThreadsPerWorker \leftarrow threadsPerWorker(workerSet)$
```

The above algorithm, Algorithm 4.7, tries to balance the *mpl* on top of the current workers-(sub)set and thus, to compute the *maximum threads per worker* for the query workload. From *lines 2 to 15* we try to distribute threads (bounded by *mpl*) to the worker nodes, proportionally to their local data (i.e. remaining partitions) and available CPU (#cores) resources (including an over-allocation). If we cannot succeed to elapse all threads and achieve the *mpl* with one shot, starting *line 20 to 32* we equally redistribute the remaining *mpl* to those workers that were left unassigned (with responsibilities) after running the (balanced) min-cost flow algorithm.

Choosing RMax' for the cost model

The decision of choosing $RMax' = 1$ (to restrict the *maximum responsibility degree* at runtime) and get a *1 responsibility assignment* instead was driven because of the following experiment. After implementing the *shrinking* (greedy heuristic) approach to minimize the optimal cost solution, we simply compared the consequences of the two different *responsibility assignments* that we can obtain with our cost model: *one-to-one assignment* ≡ *RMax = RMax' = 1* vs. *many-to-one assignment* ≡ *RMax = RMax' = 2*. We ran a TPC-H scale factor 100, with 12-partitioned tables and replication degree 2, on a 4-node Vectorwise on Hadoop cluster (internal nodes, see 2.5), from which 3 nodes were selected as workers. The *mpl* was set to 24, which means that we had at most 8 threads per node, 2 per partition (each node had 4 partitions). In next paragraphs we use ***sin*** abbreviation for the *single responsibility assignment* version and ***drm*** for the *multiple responsibility assignment* version.

Table 4.1: **sin** vs **drm**: TPC-H scale factor 100, with 12-partitioned tables and replication degree 2, on a 4-node Vectorwise on Hadoop cluster (internal nodes, from which 3 nodes were selected as workers. The *mpl* is set to 24.

Vers.	**sin**		**drm**	
Test-run	hot	cold	hot	cold
$Q1$	5.57 s	31.96 s	5.57 s	34.00 s
$Q2$	0.99 s	6.69 s	0.98 s	7.84 s
$Q3$	0.48 s	2.57 s	0.52 s	4.28 s
$Q4$	0.17 s	1.75 s	0.18 s	2.64 s
$Q5$	2.30 s	4.30 s	2.12 s	6.04 s
Q6	0.31 s	**0.49** s (1.0x)	0.31 s	**2.41** s (4.92x)
$Q7$	1.30 s	6.42 s	1.31 s	9.47 s
$Q8$	1.63 s	5.01 s	1.63 s	5.93 s
$Q9$	6.81 s	20.57 s	6.79 s	22.71 s
$Q10$	3.35 s	10.79 s	3.37 s	13.46 s
$Q11$	0.92 s	1.92 s	0.90 s	2.03 s
$Q12$	0.81 s	3.24 s	0.82 s	3.73 s
$Q13$	12.02 s	28.25 s	11.75 s	28.96 s
$Q14$	0.73 s	0.80 s	0.72 s	0.80 s
$Q15$	1.23 s	1.31 s	1.21 s	1.44 s
$Q16$	1.67 s	1.91 s	1.62 s	2.00 s
$Q17$	2.66 s	3.06 s	2.67 s	3.15 s
$Q18$	5.16 s	15.41 s	5.05 s	15.36 s
$Q19$	3.17 s	6.24 s	3.22 s	6.44 s
$Q20$	1.87 s	2.29 s	1.97 s	4.72 s
$Q21$	7.10 s	8.87 s	6.99 s	12.84 s
$Q22$	1.56 s	1.77 s	1.47 s	1.77 s
Total	61.81 s	**165.62** s	61.17 s	**192.02** s

Table 4.1 shows that the TPC-H cold-run for *drm* version is somehow slower by **1.15x** (after 3 averaged runs) than the equivalent run for *sin* version (despite the fact that the setup is exactly the same).

So, why the TPC-H cold-run is slower for the ***drm*** *version?* Only by investigating the X100 log section of **Q6** from below, we see a difference in the way *scan ranges* are assigned to the worker nodes. With *mpl = 24* and *12 partitions*, we get *2* (roughly) *equal ranges* per partition. The *sin* version assigns the whole partition range to the same node because it is not aware of other (partition) responsibilities in the system (*RMax = RMax' = 1*). Since 2 threads are used (at maximum) per partition, each range gets 1 thread assigned. So, in this case, it means that the full partition range will be shared by 2 threads. On the other hand, the *drm* version assigns the first range of a partition to one worker node and the second range of it to another ($RMax = RMax' = 2$), depending on who else is responsible for the same partition. Although this version assigns 1 thread per (equal-split) range as well, the ranges are now scattered over the entire worker-set. The consequence is that a full partition range may not be shared between 2 threads on the same node. We have marked with ***** (in the log section) the partition ranges which involve *lineitem@0*: with *sin* version we see both ranges [12648448, 17727488) and [17727488, 22806528) assigned to *node 0* (2 threads, 1 worker node), whereas with *drm* we see the first range [12648448, 17727488) assigned to *node 0* and the second [17727488, 22806528) assigned to *node 1* instead (2 threads, 2 worker nodes).

```
// Log section from X100 workers: node 0, node 1, node 2
=====================================
(1) TPC-H Q6, sin version
Responsibility assignment (single responsibilities):
node 0: 0, 3, 6,  9
node 1: 1, 4, 7, 10
node 2: 2, 5, 8, 11
=====================================
Query scheduler: computing the min worker-subset and (partition) responsibilities to
    involve in query execution (Algorithm 4.5)
Query scheduler results: chosen workers and responsibilities (based on the
    passignment metadata)
<host>: <partition>, ...
node 0: 0, 3, 6,  9
node 1: 1, 4, 7, 10
node 2: 2, 5, 8, 11
Query scheduler results: distribute 24 (mpl) threads
<host>: <partition> [#threads], ...
node 0: 0 [2], 3 [2], 6 [2],  9 [2]
node 1: 1 [2], 4 [2], 7 [2], 10 [2]
node 2: 2 [2], 5 [2], 8 [2], 11 [2]
===================================== Stats
REWRITER_XCHG:parallel_mode: available_cores 24, core_target 30, max_parallelism 24
REWRITER_XCHG:ccheduler: num_queries = 1, max_parallelism = 24, num_granted = 24
REWRITER_XCHG:xchg_rule: max_tds_per_node total = 24
REWRITER_XCHG:xchg_rule: max_tds_per_node = [8,8,8]
REWRITER_XCHG:xchg_rule: feasible_sol total = 24
REWRITER_XCHG:xchg_rule: feasible_sol = [8,8,8]
===================================== Ranges on node 0
BUILDER:MScan(lineitem@0) on thread [0] has range : [12648448, 17727488) ***
BUILDER:send producer 0 allocated.
BUILDER:MScan(lineitem@0) on thread [1] has range : [17727488, 22806528) ***
BUILDER:send producer 1 allocated.
BUILDER:MScan(lineitem@3) on thread [2] has range : [12648448, 17727488)
BUILDER:send producer 2 allocated.
BUILDER:MScan(lineitem@3) on thread [3] has range : [17727488, 22806528)
BUILDER:send producer 3 allocated.
BUILDER:MScan(lineitem@6) on thread [4] has range : [12648448, 17727488)
BUILDER:send producer 4 allocated.
BUILDER:MScan(lineitem@6) on thread [5] has range : [17727488, 22806528)
BUILDER:send producer 5 allocated.
BUILDER:MScan(lineitem@9) on thread [6] has range : [12648448, 17727488)
BUILDER:send producer 6 allocated.
BUILDER:MScan(lineitem@9) on thread [7] has range : [17727488, 22806528)
BUILDER:send producer 7 allocated.
===================================== ranges on node 1
BUILDER:MScan(lineitem@1) on thread [0] has range : [12648448, 17727488)
BUILDER:send producer 0 allocated.
BUILDER:MScan(lineitem@1) on thread [1] has range : [17727488, 22806528)
BUILDER:send producer 1 allocated.
BUILDER:MScan(lineitem@4) on thread [2] has range : [12648448, 17727488)
BUILDER:send producer 2 allocated.
BUILDER:MScan(lineitem@4) on thread [3] has range : [17727488, 22806528)
```

```
BUILDER:send producer 3 allocated.
BUILDER:MScan(lineitem@7) on thread [4] has range : [12648448, 17727488)
BUILDER:send producer 4 allocated.
BUILDER:MScan(lineitem@7) on thread [5] has range : [17727488, 22806528)
BUILDER:send producer 5 allocated.
BUILDER:MScan(lineitem@10) on thread [6] has range : [12648448, 17727488)
BUILDER:send producer 6 allocated.
BUILDER:MScan(lineitem@10) on thread [7] has range : [17727488, 22806528)
BUILDER:send producer 7 allocated.
===================================== ranges on node 2
BUILDER:MScan(lineitem@2) on thread [0] has range : [12648448, 17727488)
vBUILDER:send producer 0 allocated.
BUILDER:MScan(lineitem@2) on thread [1] has range : [17727488, 22806528)
BUILDER:send producer 1 allocated.
BUILDER:MScan(lineitem@5) on thread [2] has range : [12648448, 17727488)
BUILDER:send producer 2 allocated.
BUILDER:MScan(lineitem@5) on thread [3] has range : [17727488, 22806528)
BUILDER:send producer 3 allocated.
BUILDER:MScan(lineitem@8) on thread [4] has range : [12648448, 17727488)
BUILDER:send producer 4 allocated.
BUILDER:MScan(lineitem@8) on thread [5] has range : [17727488, 22806528)
BUILDER:send producer 5 allocated.
BUILDER:MScan(lineitem@11) on thread [6] has range : [12648448, 17727488)
BUILDER:send producer 6 allocated.
BUILDER:MScan(lineitem@11) on thread [7] has range : [17727488, 22806528)
BUILDER:send producer 7 allocated.
=====================================
// (2) TPC-H Q6, drm version
Responsibility assignment (multiple responsibilities):
node 0: 0, 3, 6,  9, 2, 5, 8, 11
node 1: 1, 4, 7, 10, 0, 3, 6,  9
node 2: 2, 5, 8, 11, 1, 4, 7, 10
=====================================
Query scheduler: computing the min worker-subset and (partition) responsibilities to
    involve in query execution (Algorithm 4.5)
Query scheduler results: chosen workers and responsibilities (based on the
    passignment metadata)
<host>: <partition>, ...
node 0: 0, 3, 6,  9, 2, 5, 8, 11
node 1: 1, 4, 7, 10, 0, 3, 6,  9
node 2: 2, 5, 8, 11, 1, 4, 7, 10
Query scheduler results: distribute 24 (mpl) threads
<host>: <partition> [#threads], ...
node 0: 0 [1], 3 [1], 6 [1],  9 [1], 2 [1], 5 [1], 8 [1], 11 [1]
node 1: 1 [1], 4 [1], 7 [1], 10 [1], 0 [1], 3 [1], 6 [1],  9 [1]
node 2: 2 [1], 5 [1], 8 [1], 11 [1], 1 [1], 4 [1], 7 [1], 10 [1]
===================================== stats
REWRITER_XCHG:parallel_mode: available_cores 24, core_target 30, max_parallelism 24
REWRITER_XCHG:scheduler: num_queries = 1, max_parallelism = 24, num_granted = 24
REWRITER_XCHG:xchg_rule: max_tds_per_node total = 24
REWRITER_XCHG:xchg_rule: max_tds_per_node = [8,8,8]
REWRITER_XCHG:xchg_rule: feasible_sol total = 24
REWRITER_XCHG:xchg_rule: feasible_sol = [8,8,8]
===================================== ranges on node 0
BUILDER:MScan(lineitem@0) on thread [0] has range : [12648448, 17727488) ***
BUILDER:send producer 0 allocated.
BUILDER:MScan(lineitem@2) on thread [1] has range : [12648448, 17727488)
BUILDER:send producer 1 allocated.
BUILDER:MScan(lineitem@3) on thread [2] has range : [12648448, 17727488)
BUILDER:send producer 2 allocated.
BUILDER:MScan(lineitem@5) on thread [3] has range : [12648448, 17727488)
BUILDER:send producer 3 allocated.
BUILDER:MScan(lineitem@6) on thread [4] has range : [12648448, 17727488)
BUILDER:send producer 4 allocated.
BUILDER:MScan(lineitem@8) on thread [5] has range : [12648448, 17727488)
BUILDER:send producer 5 allocated.
BUILDER:MScan(lineitem@9) on thread [6] has range : [12648448, 17727488)
```

```
BUILDER:send producer 6 allocated.
BUILDER:MScan(lineitem@11) on thread [7] has range : [12648448, 17727488)
BUILDER:send producer 7 allocated.
==================================== ranges on node 1
BUILDER:MScan(lineitem@0) on thread [0] has range : [17727488, 22806528) ***
BUILDER:send producer 0 allocated.
BUILDER:MScan(lineitem@1) on thread [1] has range : [12648448, 17727488)
BUILDER:send producer 1 allocated.
BUILDER:MScan(lineitem@3) on thread [2] has range : [17727488, 22806528)
BUILDER:send producer 2 allocated.
BUILDER:MScan(lineitem@4) on thread [3] has range : [12648448, 17727488)
BUILDER:send producer 3 allocated.
BUILDER:MScan(lineitem@6) on thread [4] has range : [17727488, 22806528)
BUILDER:send producer 4 allocated.
BUILDER:MScan(lineitem@7) on thread [5] has range : [12648448, 17727488)
BUILDER:send producer 5 allocated.
BUILDER:MScan(lineitem@9) on thread [6] has range : [17727488, 22806528)
BUILDER:send producer 6 allocated.
BUILDER:MScan(lineitem@10) on thread [7] has range : [12648448, 17727488)
BUILDER:send producer 7 allocated.
==================================== ranges on node 2
BUILDER:MScan(lineitem@1) on thread [0] has range : [17727488, 22806528)
BUILDER:send producer 0 allocated.
BUILDER:MScan(lineitem@2) on thread [1] has range : [17727488, 22806528)
BUILDER:send producer 1 allocated.
BUILDER:MScan(lineitem@4) on thread [2] has range : [17727488, 22806528)
BUILDER:send producer 2 allocated.
BUILDER:MScan(lineitem@5) on thread [3] has range : [17727488, 22806528)
BUILDER:send producer 3 allocated.
BUILDER:MScan(lineitem@7) on thread [4] has range : [17727488, 22806528)
BUILDER:send producer 4 allocated.
BUILDER:MScan(lineitem@8) on thread [5] has range : [17727488, 22806528)
BUILDER:send producer 5 allocated.
BUILDER:MScan(lineitem@10) on thread [6] has range : [17727488, 22806528)
BUILDER:send producer 6 allocated.
BUILDER:MScan(lineitem@11) on thread [7] has range : [17727488, 22806528)
BUILDER:send producer 7 allocated.
```

For a better understanding, we ran alone **Q6** and dived into its profiling files from all 3 worker nodes. This particular query is I/O bounded (for cold-runs) and it showed in Table 4.1 the worst performance degradation. As expected, the I/O was the one to be affected. The *sin* version reads (on all 3 workers) **3 x 440** = **1320 blocks** in total, whereas the *drm* version **3 x 456** = **1368 blocks** – **48 more blocks** in total, **roughly 10% more**. Given that the boundary of a range split may fall down in the same block – and with *drm* we divide the range scans across different workers – it means that a block might be read / fetched more than once (depending on the responsibility degree). This also explains the *2x* number, in total, of HDFS_SHORT_CIRCUIT requests for the *drm* version, comparing with the *sin* version: we need to open a file more times than usual, in order to (locally) fetch a particular block and read its range of tuples (note: the file is replicated and so, we still have local reads).

Table 4.2: **sin** vs **drm**: Q6 **cold-runs**, TPC-H scale factor 100, with 12-partitioned tables and replication degree 2, on a 4-node Vectorwise on Hadoop cluster (internal nodes, from which 3 nodes were selected as workers. The *mpl* is set to 24.

Vers.	**sin**		**drm**	
Prefetch	on	off	on	off
Q6	5.02 s	**7.58** s	5.84 s	**7.65** s

Table 4.2 shows the runtimes for Q6 with and without *io_prefetch*. With *io_prefetch* enabled the *sin* version may have already fetched the next blocks of a file (on the same node where

the 2nd range is assigned too) by the time another request for the same block arrives. So to speak, we can process the second block I/O request from the buffer-pool already. This does not apply to the *drm* version. It is even worse, we may pre-fetch blocks that we may not even use. We saw from our profile results, that there were just *6 to 30* blocks per single column scan (i.e. per thread). If there are 2 threads reading adjacent ranges on the same node, they are effectively reading from very close regions of the file and so, the locality of the file might play a role there (i.e. read-ahead / caching / or simply that a spinning disk behaves better). With *io_prefetch* disabled, the (total) I/O throughtput for Q6 is roughly the same (∼29.20 MB/s) for both versions, but, obviously, when compared with *io_prefetch* enabled (∼46.32 MB/s) is much lower. However, the buffer-manager's *syncmiss counters* for block I/O still differ (by a little) between the two versions. The *drm* version, as we already explained, might read some blocks twice than needed. In conclusion, the *dynamic resource management* algorithm should find a *thread-balanced* solution, if any, in order to benefit from I/O pre-fetching and caching. As a result, putting together the insights we got from both *sin* and *drm* versions, we always recommend for *RMax'* to be set to **1**. That makes $RMax \geq 1$ and $RMax' = 1$. The number of threads per partitions (implicitly the *mpl*) should scale-up according to the responsibility assignment that the min-cost flow algorithm returns with $RMax'$ set to 1 (Algorithm 4.5).

Conclusion

In this chapter we described the path towards achieving *dynamic resource management* in Vectorwise on Hadoop. Backed by a *min-cost flow algorithm* (Algorithm 4.4), and a *shrinking heuristic* (Algorithm 4.6) to minimize the optimal cost solution, our (threefold) approach succeeds in the end to: (1) *determine the Vectorwise worker-set*, (2) then *assign responsibilities to Vectorwise worker nodes* and finally, (3) *schedule the cluster resources that a query can use at runtime* by computing an optimal *resource footprint* (i.e. what particular nodes and the amount of resources, e.g. CPU and memory, to use). The outcome of this approach tries to improve the Vectorwise on Hadoop performance *during failover situations* and also, when concurrent Vectorwise (internal) and Hadoop (external) workloads *overlap over the same worker nodes.* Further results are presented in Chapter 6.

Chapter 5

YARN integration

Today's enterprises are looking to go beyond batch processing and integrate existing applications with Hadoop to realize the benefits of real-time processing and interactive query capabilities. As more organizations move from single-application Hadoop clusters to a versatile, integrated Hadoop 2 data platform hosting multiple applications, YARN is positioned as the integration point of today's enterprise data layer. At the architectural center of Hadoop, YARN provides access to the main resources of the platform. Hence, any MPP database system, whose architecture leverages the Hadoop capabilities and wishes to be a *'fair citizen'* of the ecosystem, should integrate with YARN.

Let us consider the following scenario: *suppose we want a set of processes to co-exist within a Hadoop cluster.* Furthermore, as requirement, these processes must run *out-of-band* from YARN, which means that YARN is not allowed to control their life-cycle. These processes use, dynamically, different amounts of CPU and memory based on their load (we do not consider yet I/O or network bandwidth resource allocation since YARN does not support that). Their CPU and memory requirements are managed independently from YARN. With other words, depending on their load, they might require more CPU but not more memory, or vice-versa. However, using YARN (Resource Manager) for scheduling out-of-band processes too, we should be able to share and utilize appropriately (e.g. without contention) the compute resources of an entire Hadoop cluster. Hence, we can let YARN be aware of all the allocated resources within the cluster, whether they were needed for Hadoop jobs or some out-of-band workloads. The out-of-band processes must run a *YARN Application Master* (see Section 1.2.4) in order to interact with YARN and negotiate for compute resources. When the Application Master is granted with all it required, we bump up their CPUs and/or memory utilization. Though this action is "hidden" from YARN, we do make sure that our resource utilization is still within the granted limits. Whenever these processes back off, their allocation should also be released by YARN. Basically, we have to kill their Application Master(s). Using an approach like this, Hadoop's running jobs and other out-of-band processes can fairly share the entire cluster's compute resources. YARN would be the only one that does the *resource bookkeeping.* The Application Master should run in *unmanaged mode* (on any of the Haoop nodes) in order to zero out the necessity of allocating one container within YARN (in *managed mode*). To note, the latter option needs the minimum resource allocation per container as predefined in the configuration files. Once the Application Master runs, it will fork *container processes* at the corresponding Node Managers to do nothing more than a basic *sleep-forever.*

5.1 Overview: model resource requests as separate YARN applications

As we briefly explained in the beginning of this chapter, the approach to integrate Vectorwise with YARN is to *model resource requests as separate YARN (native) Applications*. Implicitly, based on Section 1.2.4, every resource request will fork its own YARN Application Master. The idea is to focus towards allocating resources *per workload* instead of *per query*. It reduces the number of resource requests (i.e. request latency) and also decreases the query's runtime overhead. Starting a YARN Application Master means to request and allocate some set of resources, reserve specific containers, run the container processes, and monitor the application's status. Nonetheless, this usually comes with a big overhead: *5-7 seconds*, depending on the Resource Scheduler if it is busy or not. For instance, just to request and get some compute resources within YARN (w/o the other steps) it takes approx. 80% of the overhead time. And to do so *per query*, in which case we have to acquire before and release afterwards, is too overwhelming. Instead, a *per workload basis* (coarse-grained) approach is more suitable for a real-time integration. We see a Vectorwise *workload* as a *group of queries* with similar *maximum parallelism level* and *resource requirements* that run on the same (sub)set of workers for an *unpredictable amount of time*. From a workload management point of view, in time, these workloads will inevitable change the resource utilization of the Vectorwise database within the Hadoop cluster. Hence, it is important for us to detect such events in order to know *when* to *request* extra resources from YARN and achieve elasticity within the Hadoop ecosystem. To do so, we defined the notion of a *system context*. The *system context* refers to a moment in time when a query starts running on partitioned/non-partitioned tables and/or at the same time with other overlapping queries. Each time this context changes, e.g. one partitioned query starts running concurrently with a non-partitioned query, the (Vectorwise) *resource manager* will span a different *resource footprint* during the *query rewriting* phase. We recommend reviewing Section 1.3.2 and 4.3 to remember how the previous steps work. If we can detect these changes in time we could know whether or not to *request* extra resources. For the moment, it is easier to think of it as a 2-dimensional matrix $system - context := [partitionedQuery \in \{true, false\}][numQuery \in \{1, \ldots, \mathbb{N}\}] \leftarrow true, false$ from which we can remember if a context change actually occurred; at X100 bootstrap we initialize the matrix with *false* and reset it to the same value when every query workload has ended. A Vectorwise workload is created when the first of its queries starts running in the system; a *system context change* is recorded and cluster resources get requested from YARN. It is assumed that, within the same workload, the latency between two consecutive queries is small. Therefore, we can imply that a Vectorwise workload ends when its latest query finishes and the duration of the *idle (workload) state* passes over a certain threshold (set by the user). If that happens, the approach takes a *pessimistic* decision and ends the current workload by stopping its correspondent YARN Application Master. The next query will start a new workload and try to reclaim its *resource footprint*, though it is unfortunate when the query could have been related to the previous workload. However, queries running in a different system context or with different resource requirements are treated as *separate workloads*. If some workloads have to run concurrently, then the system will try to allocate more resources to fulfill both's footprint requirements (but all within the cluster's maximum capability). For implementation details we recommend reading Section 5.3.

Figure 5.1 illustrates an example of running multiple Vectorwise queries on 3 worker nodes with 8 cores each [8, 8, 8], plus 0.25 maximum core over-allocation. We use various colors (green, purple, red) to differentiate between different workloads. Important to notice are the two scenarios explained above: (1) if the timeout exceeds, the next query starts a new workload (green color) and (2) different overlapping queries need extra resources, within the maximum capabilities (green to purple to red, but not purple to violet). The example starts with a query workload (green color) that acquires [4, 4, 4] threads per node (*mpl = 12*). When it finished and the *idle_timeout* have passed, we release those resources. Then, it happens that another query (green color again), which could have been part of the previous workload, starts running in the system and targets the same *resource footprint* (*mpl = 12*). Since it starts after the *idle_-timeout*, it has to regain its amount of resources back from YARN. Next, we have a sequence of three queries: the first two queries (purple color) that are from the same workload, both asking for [2, 2, 2] threads per node (*mpl = 6*) and the third one (red color) that overlaps with all the other queries/workloads aiming towards [4, 4, 4] threads per node (*mpl = 12*); for the third query we need to use the 0.25 core over-allocation in order to achieve its *mpl*.

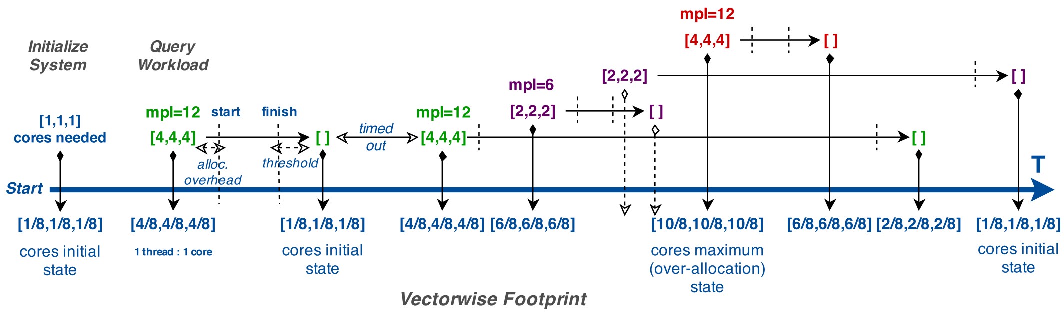


Figure 5.1: Timeline of different query workloads.

An overview of the approach is illustrated in Figure 5.2. Basically, the figure depicts a two-phase workflow. The integration starts with the Vectorwise worker-set initialization phase (**P1**), given that we previously decided the responsibility assignment. During this phase we allocate through YARN the initial amount of resources for the database to be operable, e.g. run distributed queries w/o any parallelism, 1 core + enough memory for query execution and buffer pool per worker. This phase is managed by the *WorkerSetInit* component (custom YARN Application Master). From this point on starts the second phase (**P2**). Any request to increase the worker-set amount of resources, i.e. extra CPU and memory to run a new workload, is sent directly to the *DbAgent* component (custom YARN ClientApplication). Besides what is described in the previous chapters, the *DbAgent* is also responsible for resource allocation within YARN (i.e. to acquire or release new containers). For each request that it receives, it spawns a new YARN Application Master (*IncreaseResources* component) that negotiates with the *ResourceManager* the demand for extra containers: it reserves and starts containers with specific resource requirements on a subset of (or all) worker nodes and then it manages their life-cycle. Previously to any resource request, the Vectorwise Master node asks the *DbAgent* for the current *resource state* of the cluster (e.g. available/used resources) in order to compute the *resource footprint* of a new query workload. Both, the *update worker-set state* and *acquire/release* send-receive requests, are part of the DbAgent to Vectorwise Master communication protocol. More details will be given in Section 5.2 and Section 5.3.

Under the hood, the DbAgent uses a set of YARN Application Masters that run as *unmanaged* JVM processes, i.e. they run *isolated* from YARN and need no resource allocation in particular. Each one negotiates specific resource requirements with the Resource Manager, reserves and runs one container per worker node, and then loops around to manage their life cycle. We note that, in our case, these containers do not run computational tasks as they should (e.g. as with Map-Reduce framework). Instead, the *"placeholder"* processes running inside the containers are just monitoring the co-located X100 servers, on the same worker nodes (e.g. poking the server to see if it is still alive). These processes run a *while-check-sleep* primitive and thus, their CPU and memory usage is negligible. The *unmanaged* YARN Application Master can be spawned by either *(1) an initial request* to allocate the minimum amount of (CPU and memory) resources needed to bootstrap Vectorwise and be operable to users (e.g. at least 1 core + enough memory for query execution per node) or *(2) new resource requests* for different (maybe overlapping) query workloads. Therefore, Vectorwise manages its own X100 back-end processes *out-of-band from YARN*; it uses YARN just for *resource bookkeeping* and for (Hadoop) *cluster-awareness.*

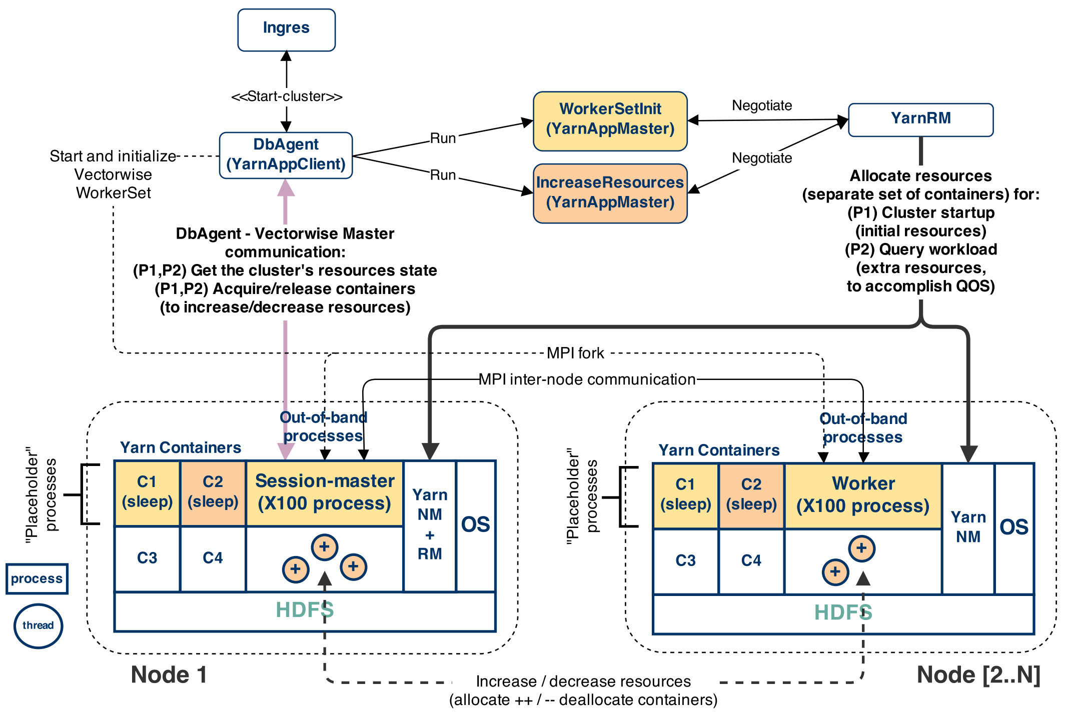

Ingres
<<Start-cluster>>
DbAgent (YarnAppClient)
Start and initialize Vectorwise WorkerSet
Run
Run
WorkerSetInit (YarnAppMaster)
IncreaseResources (YarnAppMaster)
Negotiate
Negotiate
YarnRM
Allocate resources (separate set of containers) for: (P1) Cluster startup (initial resources) (P2) Query workload (extra resources, to accomplish QOS)
DbAgent - Vectorwise Master communication: (P1,P2) Get the cluster's resources state (P1,P2) Acquire/release containers (to increase/decrease resources)
MPI fork
MPI inter-node communication
Out-of-band processes
Yarn Containers
"Placeholder" processes
C1 (sleep)
C2 (sleep)
Session-master (X100 process)
Yarn NM + RM
OS
C3
C4
HDFS
process
thread
Node 1
Worker (X100 process)
Yarn NM
Node [2..N]
Increase / decrease resources (allocate ++ / -- deallocate containers)

Figure 5.2: Out-of-band approach for Vectorwise-YARN integration. An example using 2 x Hadoop nodes that equally share 4 x YARN containers.

The distribution of resources is completely managed by YARN and we just control the node locations (i.e. worker nodes) where our resources should be allocated.

As stated in YARN-896 [1], there is significant work ongoing to allow long-lived services running as YARN applications. For example, the API's functionality to increase/decrease resource utilization within YARN containers (YARN-1197 [2]) is not yet implemented. This makes it impossible "per se" to gain full elasticity and to allocate/deallocate resources in a fine-grained fashion for Vectorwise. Though the idea to model *resource requests* as *separate YARN applications* achieves *elasticity* to a certain extent for *increasing* the amount of resources, we are limited when it comes to the opposite operation, *decreasing* resources. To explain why, let us consider the following scenario: we assume to have the same 3 workers with 8 cores each, but 2 queries running at the same time. If the first query needs all 24 available cores ($mpl = 24$ gets all the 3 workers), the other would get only 10 cores; the second query's mpl is limited to $\frac{16\ threads\ max + 16 * 0.25\ over-allocation}{2\ queries} = 10\ threads$. We reference Section 1.3.3 for the two (before-mentioned) scheduling policies and recommend reviewing Section 5.1 to understand what comes next. Assuming the two queries as being part of different workloads, e.g. first query runs on partitioned data and the second not, the first workload would then allocate the amount of resources it needs before it starts. On the other hand, the second workload does nothing because the available cores are all claimed already. At this moment, just one YARN Application Master (for the first workload) negotiates the *resource footprint* and manages the 3 *placeholder processes* (one per node). When both queries are still active or the second one finishes faster, that is fine. The problem appears if the first workload finishes earlier and we have to stop its YARN Application Master to release the amount of allocated resources (i.e. all cores). Moreover, this means to leave the second query (using 10 cores) running, but "outside" YARN /

[1]YARN-896: issues.apache.org/jira/browse/YARN-896
[2]YARN-1197: issues.apache.org/jira/browse/YARN-1197

without resource-bookkeeping. Besides, we cannot just deallocate the difference of resources between the first and second workload using the available API interface. With the existing YARN version this functionality can be translated into *(1) stopping* the existent Application Master (for the first query) and then *(2) starting* another one (for the second query), which adds a significant overhead and is not a worthwhile solution to implement (from a product-wise point of view). Therefore, our workaround is to release all requested resources at once, when the latest workload ends (*idle state* duration $>$ *threshold*) and there are no other queries still active in the system. What is changed from Figure 5.1 is shown below, in Figure 5.3.

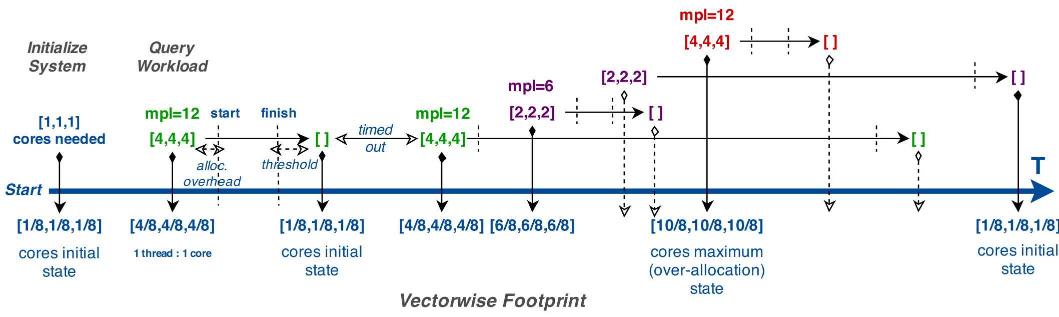


Figure 5.3: Timeline of different query workloads (one-time release).

However, as YARN-1197 is being pushed for the next release version of YARN, we are going to revise this approach in the near future. For instance, instead of creating one YARN Application Master per resource request (per workload), we could multiplex all the requests towards a single YARN application (or a pool of applications distributed over different scheduling queues) that dynamically increases/decreases its containers' amount of resources. Thereby we can avoid the application startup overhead entirely and reduce the request latency even more (i.e. a YARN Application Master can run in background and listen only for new requests).

Another alternative is to use Apache's Slider [1] platform, launched in beta-version around June 2014, which exposes a set of services that allow long-running applications, real-time and online applications to easily integrate into YARN (and be YARN-Ready [2]). Besides, it provides an interface for real-time communication, especially for database systems running on Hadoop or database-style workloads where a very high-speed transfer or data processing and response times are required. It complements Apache Tez [3], which is quickly gaining adoption as the batch and interactive engine of Hadoop. However, one thing to note, is that Slider, as well as the YARN 2.2 release, do not support resource extension (i.e. increase or decrease primitives) for containers yet. This means we would still need a work-around to achieve resource elasticity for Vectorwise on Hadoop, to allocate or deallocate resources when is required so.

[1] Apache Slider: slider.incubator.apache.org
[2] YARN-Ready Program: hortonworks.com/press-releases/hortonworks-announces-yarn-ready-program
[3] Apache Tez: tez.apache.org

5.2 DbAgent–Vectorwise communication protocol: update worker-set state, increase/decrease resources

The DbAgent–Vectorwise communication protocol uses 3 types of request/response messages in order to: *(1) update the worker-set (system's resource) state*, e.g. available cores/memory, allocated cores/memory, etc. and *(2) increase/decrease the worker-set current resources*, which are allocated through YARN. A simple example of a database session running one query is depicted in Figure 5.4.

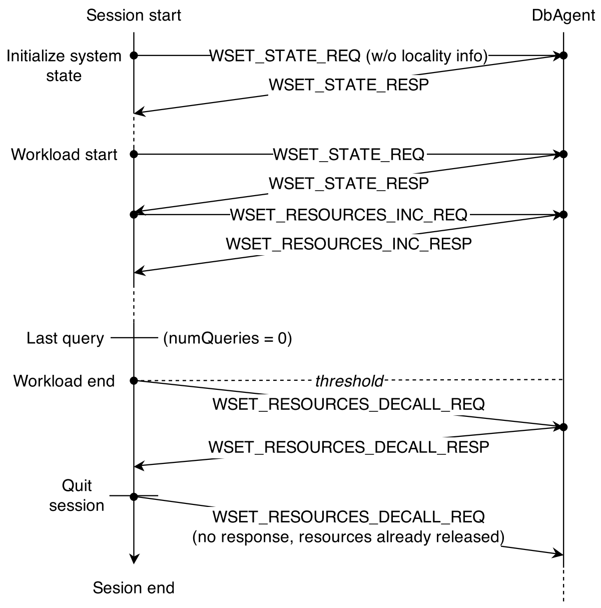


Figure 5.4: Database session example with one active query: the communication protocol to increase/decrease resources.

Update the worker-set (system's resource) state. The system's resource state includes the list of *worker nodes*, the *locality_info* of the existent Vectorwise data (i.e. local stored partition tables to determine the *partial replicas*), and the cluster's resource overview *cores_-{capability, allocated, available}*, plus *memory_{capability, allocated, available}*. However, the information that actually needs to be up-to-date for the *resource footprint* computation at query runtime is the *locality_info*, *{cores, memory}_allocated*, and *{cores, memory}_available*. To get these metrics and the data-locality information from YARN (through the DbAgent) we use the following request/response Json format messages:

```
// Get the worker-set (resource) state:
// WSET_STATE_REQ <-> WSET_STATE_RESPONSE
Request: {
  "message_type":"WSET_STATE_REQ"
}
Response: {
  "message_type":"WSET_STATE_RESP",
  "workers_set":["node01",...], // Hostnames
  "cores_capability":[8,..], // Number of cores per worker
  "memory_capability" [122880,...], // Amount in MB per worker
  "cores_allocated":[1,...], // Number of cores per worker
  "memory_allocated":[24576,...], // Amount in MB per worker
  "cores_available":[15,...], // Number of cores per worker
  "memory_available":[98304,...], // Amount in MB per worker
  "locality_info":{
    "partition-id":[local-nodes,...], // Workers location
    ...
  }
}
```

Increase/decrease the amount of resources. To increase the current amount of resources we specify into a Json format message how many extra cores and memory we need per node, depending on the *resource footprint's* specification. To decrease them we just send a request to the DbAgent at the end of the latest query workload (after the *idle_ timeout* threshold) and everything will go back to the initial resource state (i.e. as it was at startup).

```
// Increase worker-set resources:
// WSET_RESOURCES_INC_REQ <-> WSET_RESOURCES_INC_RESP
Request: {
  "message_type":"WSET_RESOURCES_INC_REQ",
  "cores_plus":[15,...], // Number of cores per worker
  "memory_plus":[98304,...] // Amount in MB per worker
}
Response: {
  "message_type":"WSET_RESOURCES_INC_RESP",
  "response":"SUCCEEDED"|"FAILED"
}
// Decrease worker-set resources:
// WSET_RESOURCES_DECALL_REQ <-> WSET_RESOURCES_DECALL_RESP
Request: {
  "message_type":"WSET_RESOURCES_DECALL_REQ",
}
Response: {
  "message_type":"WSET_RESOURCES_DECALL_RESP",
  "response":"SUCCEEDED"|"FAILED"
}
```

5.3 Workload monitor & resource allocation

For *resource allocation* within YARN we make use of the query's *resource footprint (workersSubset, responsibilities, maximum threads per worker)* tuple, which implicitly contains information on how many threads per worker node need to be claimed (in total) for a query at runtime. Based on the communication protocol from Section 5.2, when we request extra resources (from the Resource Manager) for a workload the idea is to fill-out the cores_-plus:[...] list from the message-body with the difference between the *maximum threads per worker* (member of the *footprint* tuple) and the current state of the *allocated resources*: $footprint.maximumThreadsPerWorker - allocatedCores$ (1 thread : 1 core). In other words, the *resource footprint* tells us how many *cores/threads* we need more when compared to the current state of the system. Obviously, if the difference is negative then extra resources are not available for the moment and we should not even try to send a request to increase those. However, we record this "failure" in the system such that, after a system config *failure_ timeout*, another query from the same workload tries again to acquire from YARN what was previously needed. The *workload monitor's* and *resource allocation's* workflow is illustrated in Figure 5.5. The figure depicts two possible scenarios: **(a)** when the *resource request* was ***successfully***, being able to *acquire* the *resource footprint*, and **(b)** if it ***failed*** because of an external (Hadoop) workload overlapping on the same set of resources. For both, we show in particular (using distinct colors) what is the YARN footprint at each step, for **(a) on left** and **(b) on right**. Following the workflow's state transitions (from left to right) the reader can notice that the first 4 steps (*enter*, *attempt on (YARN) resources*, *compute the resource footprint* and *acquire exact resources*) are common to all query workloads. A "branch-out" may happen after a *resource* *request* is sent to the *DbAgent*. Depending on the *resource availability* within the Hadoop cluster, the request can either *succeed*, case in which we *update* (green color) the system state to the current level of allocated resources, or *fail* because of a situation with *no more available resources* (e.g. overlapping external workloads). For the latter we should *rollback* (red color) to the previous system state (i.e. to what resources were available before the *attempt*) and record this "failure" (light blue color). For **(a)** we release the workload's extra resources when this ends (and the *idle_ timeout* is passed), whereas for **(b)** we do not have to release anything. Yet, we do have to *retry* (light blue color) to obtain our resources after a certain time; typically this "waiting" time is equal with the before mentioned *failure_ timeout*.

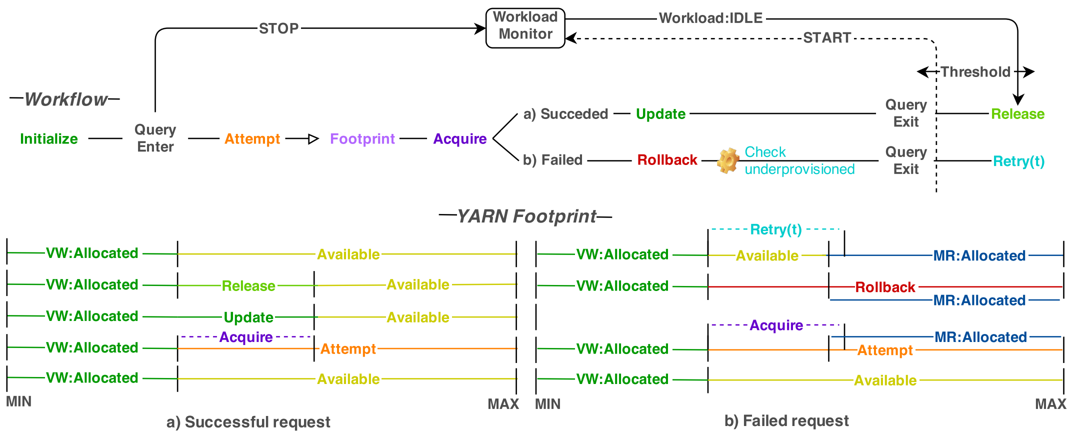


Figure 5.5: The *workload monitor's* and *resource allocation's* workflow for: **a) successful** (resource) **request** and **b) failed** (resource) **request** (VW – Vectorwise, MR – Map-Reduce)

Omitted from the *resource footprint* are the *memory requirements*, which are computed separately from the resource scheduler's implementation. The standard way of specifying the amount of memory for Vectorwise is by setting static values for the *buffer pool* and *query execution* memories inside the configuration file. Since the buffer pool is relatively independent from the query execution, there is no relation as such to allow us increasing its memory size dynamically at query runtime. Therefore, this value is read from the configuration file and set as a X100 initialization argument in the worker-set startup command; initial resources are allocated from YARN before running the X100 (worker-set) backend processes. For query execution we defined a linear function to increase the memory size at runtime; the total amount to increase is proportionally to the footprint's demand of core/thread resources and is limited by the (maximum) available memory on each of the worker nodes.

The implementation of the *resource allocation* functionality follows the approach described in Section 5.1, a *per workload* resource allocation with a *one-time* release, and uses the communication protocol from Section 5.2 to *update/acquire/release* resources. A high-level pseudocode of the runtime *resource allocation* routine is presented in Algorithm 5.1. Algorithm 5.2 shows the pseudocode implementation of the *update/acquire/release* primitives. In addition, the *workload monitor* uses a thread to monitor the workload's *start/stop* state changes and also to simulate the system's *idle state*, knowing when to release the extra amount of resources. The monitoring thread is *created* during the system's initialization phase, *starts* measuring the *idle state* each time the latest query workload (i.e. *numQueries=0*) leaves Vectorwise and it *stops* when the duration of the *idle state* passes a certain *threshold (or idle timeout)* configured by the user. When this thread stops, a request to *release* all resources is sent to the DbAgent. The psuedocode implementation of the *workload monitor's* functions is given in Algorithm 5.3.

Algorithm 5.1 High-level pseudocode: *workload-monitor* & *resource allocation*

```
    $partitioned \in \{true, false\}$
    $numQueries \in \{1 \dots \mathbb{N}\}$
 1: **(1) Query Enter**
 2: $numQueries \leftarrow numQueries + 1$
 3: $stopTimeoutThread()$
 4: ...
 5: **(2) In Rewriter Phase:** inside the parallel rule where the footprint
    and the feasible solution are computed
 6: $attemptIncrease \leftarrow attemptIncreaseResources(partitioned, numQueries)$ // update the
    system's state with all the available resources from YARN
 7: do
 8:     $rollback \leftarrow false$
 9:     // if attemptIncrease is true, it means that the system's state is updated "temporarily" with the
        resource information we get from YARN (e.g. allocated resources, available/extra resources, etc.)
10:     $footprint \leftarrow footprint(...)$ // is the resource scheduler's job to compute the current $footprint$
11:     if $attemptIncrease \wedge \neg acquireExtraResources(footprint, partitioned, numQueries)$
12:         $rollback \leftarrow true$
13:         $rollbackSystemState()$
14:         $attemptIncrease \leftarrow false$
15:     end if
16: while $rollback$
17: if $attemptIncrease$ then
18:     // make the attempt consistent with the sate of the system
19:     $updateSystemState(footprint)$
20: else
21:     // (1) the attempt failed, or (2) there were not enough available resource at the moment
22:     $checkSystemState(partitioned, numQueries, footprint)$ // check if the system is underprovi-
        sioned
23: end if
24: ...
25: **(3) Query Exit**
26: $numQueries \leftarrow numQueries - 1$
27: if $numQueries = 0$ then
28:     $startTimeoutThread()$
29: end if
```

The above high-level pseudocode, Algorithm 5.1, introduces the resource allocation at query runtime and it complies with Figure 5.5, the logical workflow of the resource allocation and workload monitor. When a *query enters* the system we increase the number of active queries and *stop* the workload monitor from its sleeping-condition that simulates the *idle state*, *lines 2 and 3*. From *line 6 to 24* is the runtime resource allocation routine: we make an attempt over all YARN's available resource and we try to acquire the current *resource footprint's* requirements. If the request failed we fall into a *rollback* procedure in which we change the system's state to the previous state and recompute the *resource footprint*; *mpl* should be decreased if that happens. However, during the *checkSystemState(...)* function at *line 22* will make sure that, after a certain *failed_ timeout* time, a next query from the same workload is going to try again and acquire what resources were needed initially. Last but not least, when a *query exits* the system we decrease the number of active queries and if there are no others we can *start* the workload monitor, *lines 27 to 30.*

Algorithm 5.2 High-level pseudocode for the *attempt-increase, acquire, rollback, update and check* functions used in Algorithm 5.1

```
attemptIncreaseResources(partitioned,numQueries):
  $yarnAcquiredFailed[partitioned][numQueries] \leftarrow timestamp$, records the timestamp when
  we failed to acquire extra resources
  $yarnRequestSent[partitioned][numQueries] \leftarrow \{true, false\}$, records with $\{true, false\}$ if
  we have sent (or not) a resource request to the DbAgent
Input: partitioned: whether the current query runs on partitioned/non-partitioned tables,
    numQueries: the number of running/active queries in the system
Output: true if is still possible to get extra resources, false otherwise
1: if $\neg yarnRequestSent[partitioned][numQueries] \vee (yarnAcquiredFailed[partitioned][numQueries] \wedge$
   $failed\_timeout \leq (current\_timestamp - yarnAcquiredFailed[partitioned][numQueries]))$
   then
2:   $state \leftarrow yarnRequestWorkerSetState()$ // request for available/extra resources
3:   $attemptIncrease \leftarrow increaseStateTo(state)$ // "temporarily" increase all resources; we store
     this amount (or excess) separately so that we can rollback to the previous state afterwards
4:   return $attemptIncrease$
5: end if
6: return $false$
acquireExtraResources(footprint,partitioned,numQueries):
Input: partitioned, numQueries, footprint: the resource footprint structure (includes the max-
    imum threads per node the current query needs to achieve its mpl)
Output: true if we could acquire extra resources, false otherwise
1: // try to allocate extra resources from YARN: send a request to the DbAgent and wait for response
2: $yarnRequestSent[partitioned][numQueries] \leftarrow true$
3: if $yarnAcquireExtraResources(footprint)$ then
4:   $yarnAcquireFailed[partitioned][numQueries] \leftarrow 0$ // reset timestamp
5:   return $true$
6: end if
7: $yarnAcquireFailed[partitioned][numQueries] \leftarrow current\_timestamp$
8: return $false$
rollbackSystemState():
  $plusResources$ keeps the attempt we did to increase the amount of resources, so that we can
  rollback (i.e. decrease the excess of resources from the current state) if needed
1: $rollbackState(plusResources)$ // rollback to the previous state by decreasing the excess of resources
   from the current state, state which was updated in advance during the attemptIncreaseTo(...) function
updateSystemState(footprint):
Input: footprint
1: // merge the resource footprint with the state before the attempt to increase resources, $stateBefore \leftarrow$
   $allocatedResources - plusResources$
2: $mergeStateWith(footprint, allocatedResources, plusResources)$ // basically we remove the ex-
   cess and update the older state with the exact amount of resources needed by the footprint, amount that
   we know it was successfully claimed from YARN
checkSystemState(partitioned, numQueries, footprint):
Input: partitioned, numQueries, footprint
1: // check if the CPU resources are underprovisioned
2: if $underprovisionedSystem(footprint)$ then // this function looks at how many of the current
   allocated cores are still available for internal query workloads and it compares that with the footprint's
   CPU resources per worker node; also we take in account the maximum capability of a worker node before
   considering the system as being undeprovisioned
3:   $yarnAcquireFailed[partitioned][numQueries] \leftarrow current\_timestamp$
4: end if
```

Algorithm 5.3 High-level pseudocode for the workload-monitor's *run, start, stop, and exit* functions used in Algorithm 5.1

```
the $timeoutThread$ checks whether all Vectorwise workloads have finished, so that we know
when to release the amount of extra resources; it starts running when the database session
is initialized for the first time
$timeoutThread.state \in \{START, STOP, EXIT\}$, where $START/STOP$ are related to the
workload's $IDLE/RUNNING$ states and $EXIT$ is used when the system shuts down
runTimeoutThread():
1: $acquireLock()$
2: while $\neg timeoutThread.state = EXIT$ do
3:   $waitOnCondition(timeoutThread.condition, getLock())$
4:   if $\neg timeoutThread.state = EXIT$ then
5:     $now \leftarrow current_timestamp$
6:     $timedWaitOnCondition(timeoutThread.sleepCondition, getLock(), now + idle\_timeout)$ // simulate the idle-state: wait until the $idle\_timeout$ threshold is passed
7:   end if
8:   if $timeoutThread.state = START$ then
9:     // release all resources
10:    $reinitializeSystem()$ // rollback to the initial (resource) state of the system
11:    $yarnReleaseExtraResources()$ // send a WSET_RESOURCES_DECALL_REQ to the DbAgent
12:  end if
13: end while
14: $releaseLock()$
startTimeoutThread():
1: $acquireLock()$
2: $timeoutThread.state \leftarrow START$
3: $awake(timeoutThread.condition)$
4: $releaseLock()$
stopTimeoutThread():
1: $acquireLock()$
2: $timeoutThread.state \leftarrow STOP$
3: $releaseLock()$
exitTimeoutThread():
1: $acquireLock()$
2: $timeoutThread.state \leftarrow EXIT$
3: $awake(timeoutThread.condition)$
4: $awake(timeoutThread.sleepCondition)$
5: $releaseLock()$
```

Algorithm 5.2 brings up the *attempt-increase, acquire, rollback, update and check* functions, used at runtime by the X100 Master to *allocate resources* from YARN. The *attemptIncreaseResources(...)* function temporarily updates the state of the system, *lines 2 and 3*, with all the available YARN resources (if any) as an optimistic approach to achieve the query's *mpl*. We note that some *resource footprints* may need less cores then all the available. Another important function is *acquireExtraResources(...)* in which we try to claim the extra cores we need (from YARN), compared to the current allocated cores per worker nodes. This is done by sending an *increase request* to the DbAgent, *lines 3 to 6*. The previous two functions use $yarnRequestSent[..][..]$ and $yarnAcquiredFailed[..][..]$ matrices to verify (at $attemptIncreaseResources : line : 1$) and record (at $acquireExtraResources : lines : 2, 4, 7$) if any *requests* for specific workloads were *sent* and, respectively, if they *failed* due to resource unavailability (i.e. an external Hadoop workload got the available resources in the meantime). In Algorithm 5.3 we depict the workload-monitor's *run, start, stop and exit* primitives. Important to emphasize is the $runTimeoutThread(\ldots)$ function, called during the Master's system initialization. This function simulates the *workload's idleness* when the thread is set on *START* state, *lines 4 to 7*, and releases the amount of resources afterwards if this state has not been changed to *STOP*. The rest of the pseudocode is self-explanatory. Because of space limits, we leave the remaining *utility functions*, such as the $yarnRequestWorkerSetState(\ldots)$, $yarnAcquireExtraResources(\ldots)$ and $yarnReleaseExtraResources()$ functions, which implement Section 5.2's protocol in order to *update*, *request* and *release* resources, or the $increaseStateTo(\ldots)$, $mergeStateWith(\ldots)$, $rollbackState(\ldots)$, $underprovisionedSystem(\ldots)$ and $reinitializeSystem()$ system's helper functions uncovered.

YARN resource allocation overhead

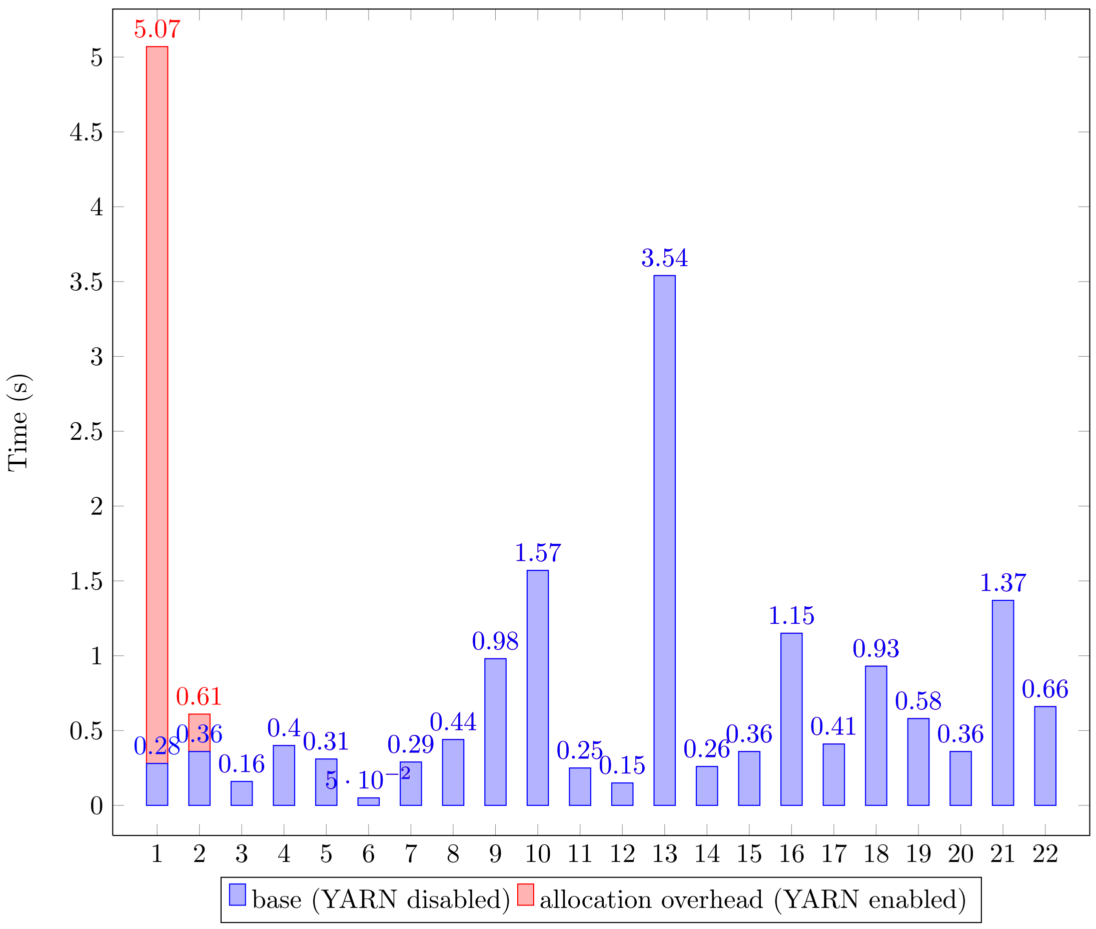


Figure 5.6: YARN's allocation overhead: TPC-H scale factor 100, 32 partitions, 8 "Stones" nodes, 128 *mpl*.

To measure YARN's *resource allocation overhead*, i.e. negotiating resources and reserving/running specific containers, in a mixed scenario of queries we have run a TPC-H scale factor 100 (*Lineitems* and *Orders* 32-partitioned) on a 8 "Stones" cluster (see 2.5), of which all of them were selected for the worker-set. We have initialized Vectorwise with 1 Core + 24 GB per worker and the *mpl* was set to 128, the maximum amount of cores in the cluster. No other external (Hadoop Map-Reduce) jobs were running at the same time on the cluster; experiments using Hadoop-busyness are shown in Chapter 6.

In our case, the first 2 queries from TPC-H can already be classified into 2 different workloads: Query-1 reading *Lineitem* that is partitioned and Query-2 running on *Part, Partsupp, Supplier, Region, Nation* non-partitioned tables. Hence, all the required resources are allocated at the beginning of the benchmark's run (during the first 2 queries' run). Yet, the (resource allocation) overhead is still not negligible. Results are shown in Figure 5.6.

As we can see from Figure 5.6, when **Query-1** starts, there is an overhead of **4.79s** between the two of the runs that is used (in the YARN enabled version) to allocate *15 more cores* and the rest of the available memory per worker node. On the other hand, **Query-2** *requests only a worker-set (resource info) update* given that there are no resources left in the cluster. The later adds an overhead of **0.25s** at query runtime.

Conclusion

In conclusion, we have shown how one can achieve *resource elasticity* for Vectorwise on Hadoop, alongside with the *dynamic resource management* described in the previous chapter, Chapter 4. Our approach to *allocate* resources per workload, instead of per query, and *deallocate* at once when no query has been active for an *idle_timeout* reduces the overall overhead of *resource management* through YARN; the current's YARN API release version adds a big overhead when starting an Application Master to negotiate resources and manage containers, Figure 5.6. However, as we explained in Section 5.1, we expect the previous issue to be fixed by the code commit of YARN-1197 [1] such that we could improve our approach and gain *more elasticity* in the near future.

[1]YARN-1197: issues.apache.org/jira/browse/YARN-1197

Chapter 6

Evaluation & Results

6.1 Evaluation Plan

We have divided the evaluation of our research project in three separate phases, following the order of which the project's approach was presented with technical details throughout this thesis (in Chapter 3, Chapter 4, and Chapter 5):

- **P1**: HDFS custom block-placement policy

  – Testing the HDFS custom block replication, which replicates and collocates Vectorwise data, during a system fail-over (i.e. a node crash).

- **P2:** Dynamic resource scheduling algorithm

  – Testing the min-cost flow (implicitly the cost-model) approach for reliability and performance during node failovers.

  ** When a node fails the database session must be restarted and thus the buffer pool will be empty. In this case, Vectorwise needs to access the HDFS layer in order to read its data.

- **P3:** YARN integration

  – Testing the min-cost flow (implicitly the cost-model) approach while running other external (Hadoop Map-Reduce) jobs on the same cluster nodes (i.e. simulating Hadoop busyness).

We expect from each evaluation phase to derive the following conclusions:

- **P1:** The runtime performance should not degrade after data recovery. At the end of the HDFS data re-replication process, all missing replicas must be replicated and collocated on the new worker nodes (or the remaining workers). The system's I/O throughput in this situation should be at least as it was during the cluster's "healthy" state.
- **P2:** We achieve a better runtime performance (or at least the same as the baseline) by favoring local reads vs remote (over TCP) reads.
- **P3:** We achieve a better runtime performance (and resource utilization) by steering the Vectorwise query computation to the least loaded workers, with respect to their data-locality. Before running a query workload, we should have allocated the workload's *resource footprint* through YARN in order to reduce the resource contention.

Experimental Setup

All the experiments we present in this section were carried out on the available clusters of computers that were described in Section 2.5. Table 6.1 shows the exact values we used for the relevant configuration parameters of Vectorwise on Hadoop (X100 engine).

Parameters	**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, depending on the chosen cluster to test.	4 or 16
bufferpool_size	The amount of memory the I/O buffers can use to cache data from disk, depending on the chosen cluster to test.	10GB or 48GB
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 1.3.2)	0.3
enable_profiling	Flag specifying whether detailed per-query profiling information should be collected.	TRUE

Table 6.1: Parameters configuration values

Both **P1** and **P2** try to highlight the importance of *data locality* in a system that suffers from a fail-over. Hence, we perform our tests on so-called "cold" I/O buffers (i.e. empty buffer pool, or cache). At the beginning of each test we *clear* the OS File System's cache and before each query we use the X100's *clear_bm syscall* to empty the I/O buffers.

On the other hand, **P3**'s goal is to emphasize the benefits of *dynamic resource management* through YARN and is performed on "hot" I/O buffers. This means that MScan operators will always read data from memory and there is no disk access whatsoever. As such, before running any tests, we performed prior cold-runs (w/o calling the clear_bm syscall beforehand) to ensure all the required data would be available in the I/O buffers. We allowed each of the worker nodes used for testing to store up to its maximum buffer pool size of data. Since **P3** requires more computation power because of the concurrent (Vectorwise) internal and (Hadoop Map-Reduce) external workloads, for this particular phase we used the more powerful "Stones" cluster nodes.

We note that only queries expressed in the Vectorwise algebra were used and, therefore, just the X100 engine was put under stress. Nevertheless, since our solution does not modify the way Ingres interacts with the Vectorwise (X100) Master, there should be no issues regarding how Ingres parses the SQL statements and sends the optimized X100 query plans.

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 [1], 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

[1]www.tpc.org/tpch

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 300, which produce approximately 32GB and 96GB of compressed data respectively. *Order* and *Linitem* are the only partitioned tables, 16 or 32 partitions. 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) for **P1, P2** and just a subset of them, a workload composed of Q1 and Q13, for **P3**. The two of the queries target opposite characteristics of the system, Q1 being I/O bounded for cold-runs and CPU bounded for hot-runs, whereas Q13 is both network and CPU bounded in the latter case (because of the big Join/DXchgHashSplit between Orders and Customer tables and the afterwards Aggregations).

Simulating Hadoop Busyness

For testing how the min-cost flow (implicitly the cost-model) approach behaves while running external workloads that overlap with the Vectorwise worker-set, we used the TeraSort [1] standard MapReduce sorting benchmark and ran different sort of jobs. TeraSort is a benchmark that combines testing the HDFS and MapReduce layers of a Hadoop cluster. As such it is not surprising that this benchmark suite is often used in practice [2], which has the added benefit that it allows people – among other things – to compare the results of their own cluster with the clusters of others. One can use the TeraSort benchmark, for instance, to iron out his Hadoop configuration. Typical areas where it is helpful, is to determine whether the resource container assignments are sound (as they depend on the variables such as the number of cores per node and the available memory), whether other MapReduce-related parameters are set properly, or whether the FairScheduler configuration someone came up with really behaves as expected. For us, TeraSort simulates Hadoop *busyness* in the cluster. We use this benchmark especially for its the *Map* phase, which takes about 40% of the runtime and generates (for bigger data sets) a CPU avg. load of 80-90%. More details about our busyness workloads are given in Table 6.2. To make it even more interesting, we have configured our jobs to run either on the *full* worker-set (8 nodes, not. *full-set*) or on the *half* of it (4 nodes, not. *half-set*). This makes the workload's *Map* phase to be CPU and I/O intensive, respectively CPU and network intensive (since Hadoop does not always match all Mappers with their right data-locality and needs remote data access). Previously, we loaded our (5GB, 10GB and 100GB) data sets in two directories, each one for its own *full-set* (loaded on 8 nodes) or *half-set* (loaded on 4 nodes) run.

Table 6.2: Workload runtime and CPU avg./peak load for *TeraSort* benchmark runs on different data-set sizes, number of Mappers/Reducers and node sets.

Busyness	**Full-set**			**Half-set**		
Config.	596.8MB x 8w	1.2GB x 8w	11.6GB x 8w	1.2GB x 4w	2.4GB x 4w	23.3GB x 4w
$16M/16R$	35 s	56 s	461 s	108 s	207 s	1613 s
$32M/32R$	30 s	43 s	266 s	63 s	112 s	628 s
$64M/64R$	26 s	36 s	218 s	40 s	57 s	376 s
CPU load	40 – 50%	60 – 70%	70 – 80%	30 – 40%	50 – 60%	70 – 80%
CPU peak	–	–	–	–	80%	95%

[1] TeraSort package: hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html

[2] TeraSort competition: sortbenchmark.org

6.2 Results Discussion

Instrumenting Block Replication

In Section 2.1 we have discussed the two of the HDFS local read options, through DataNode (out-of-box DataTransferProtocol) and direct local reads (short-circuit config), showing the results of a performance comparison using the Test-DFS I/O benchmark in Table 2.1. Given that, it make sense to enable the *short-circuit* configuration during the cold-run tests and benefit from "real" data locality. We prove the later by running now a TPC-H cold-run on 8 "Rocks" workers with 32 *mpl*. As we can see from Figure 6.1, the overhead of sending data through TCP sockets (though the data is stored locally) increases the individual query execution time by **50% to 100%** in some cases. The average I/O throughput per query drops down as well, Figure 6.3 (*w/ block collocation* enabled), from **31.81 MB/s** to **21.65 MB/s**.

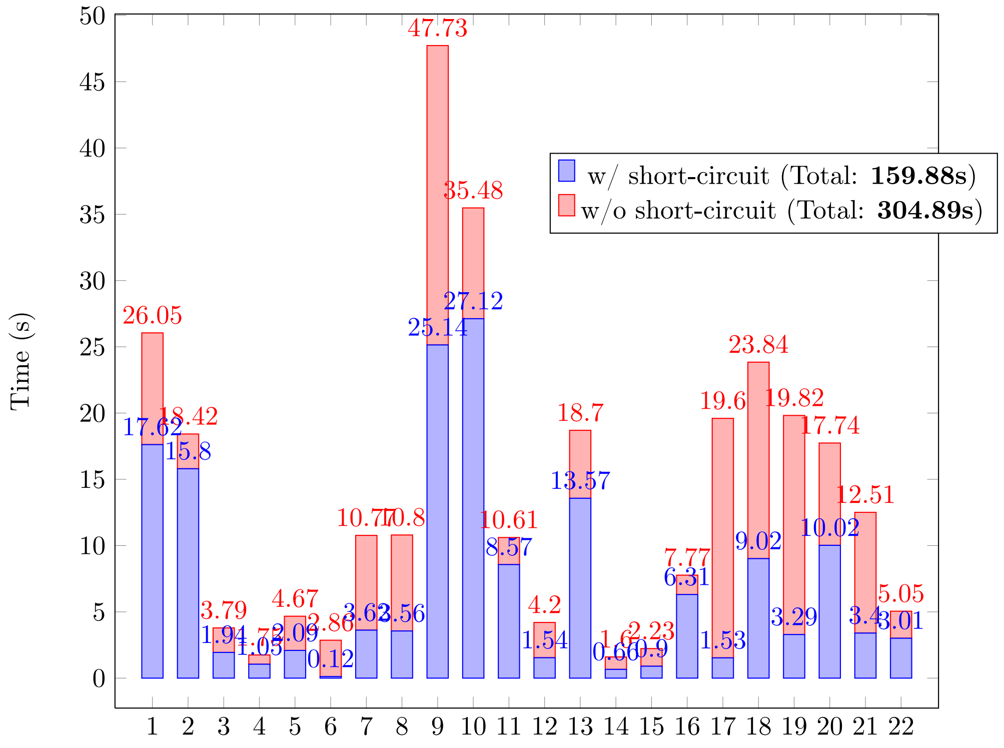


Figure 6.1: Running *with* (w/) and *without* (w/o) *short-circuit* when *block colloc.* is enabled. TPC-H benchmark: 16-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl.

Focusing on how we can *instrument HDFS block replication*, Chapter 3 describes our *Custom Block Placement Policy* implementation for Vectorwise on Hadoop. At the end it also embodies some preliminary results (Tables 3.1 and 3.2 and Figure 3.3) showing the benefits of this policy while running just a sub-set of the TPC-H queries (1, 4, 6, and 12). Based on the preliminary results, we concluded that our custom policy is better (by **1.58-1.78x**) than the default HDFS placement after re-replication, having control over data (co)locality to worker nodes, whereas the default behavior performs better during node failures (**1.14-1.41x**), due to new worker nodes (replacing the failed ones) having some of the local data that Vectorwise queries may need. To strengthen this conclusion, we have extended the same test to all TPC-H queries. New results are described in the remaining of this section. Some of the results consider the case when the entire cluster's size is restricted to the exact number of Vectorwise workers (not. workers/hadoop-nodes(total-nodes) = 8/8(10)) to point out the consequences of limiting the default replication just to the current worker-set.

Table 6.3: Query execution times: *baseline* (default policy) vs. *collocation* (custom policy) versions, in 3 situations (1) healthy state, (2) failure state (2 nodes down), (3) recovered state. TPC-H benchmark: 16-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl (W/N = workers/hadoop-nodes).

State	**Healthy**			**Failure**			**Recovered**		
Vers.	baseline	baseline	collocation	baseline	baseline	collocation	baseline	baseline	collocation
W/N	8/10	8/8(10)	8/10	8/8	8/8(8)	8/8	8/8	8/8(8)	8/8
$Q1$	17.59 s	17.11 s	17.62 s	19.57 s	26.32 s	29.70 s	29.23 s	25.76 s	17.25 s
$Q2$	27.16 s	27.72 s	15.80 s	29.43 s	27.49 s	16.29 s	29.96 s	25.70 s	16.42 s
$Q3$	2.40 s	2.31 s	1.94 s	3.06 s	2.89 s	4.04 s	3.47 s	3.35 s	1.98 s
$Q4$	1.14 s	1.04 s	1.05 s	1.86 s	1.51 s	2.06 s	2.11 s	1.79 s	0.90 s
$Q5$	1.95 s	2.03 s	2.09 s	4.36 s	4.62 s	4.76 s	5.18 s	4.79 s	1.93 s
$Q6$	0.12 s	0.11 s	0.12 s	2.09 s	2.28 s	2.32 s	1.92 s	1.70 s	0.12 s
$Q7$	3.38 s	3.66 s	3.62 s	8.42 s	9.67 s	10.42 s	10.33 s	9.32 s	3.69 s
$Q8$	3.39 s	3.42 s	3.56 s	7.82 s	10.17 s	10.01 s	8.87 s	8.62 s	3.85 s
$Q9$	33.33 s	34.08 s	25.14 s	43.51 s	53.48 s	47.48 s	47.70 s	47.60 s	26.54 s
$Q10$	36.70 s	36.74 s	27.12 s	37.90 s	46.43 s	32.61 s	39.91 s	40.81 s	28.80 s
$Q11$	14.56 s	13.59 s	8.57 s	13.98 s	14.38 s	10.21 s	14.37 s	14.16 s	9.22 s
$Q12$	1.47 s	1.50 s	1.54 s	3.82 s	4.43 s	4.32 s	4.68 s	3.99 s	1.54 s
$Q13$	13.87 s	14.41 s	13.57 s	15.13 s	16.00 s	23.42 s	26.29 s	23.57 s	13.64 s
$Q14$	0.79 s	0.80 s	0.66 s	0.88 s	1.33 s	1.63 s	1.33 s	0.98 s	0.70 s
$Q15$	1.15 s	1.38 s	0.90 s	1.45 s	1.69 s	2.74 s	1.72 s	1.71 s	1.02 s
$Q16$	9.69 s	10.18 s	6.31 s	10.26 s	11.24 s	7.91 s	11.14 s	9.96 s	7.24 s
$Q17$	2.54 s	2.01 s	1.53 s	10.55 s	17.34 s	16.28 s	10.59 s	8.52 s	1.54 s
$Q18$	8.72 s	9.23 s	9.02 s	19.38 s	17.97 s	22.82 s	17.58 s	17.70 s	9.18 s
$Q19$	4.03 s	3.95 s	3.29 s	12.96 s	17.62 s	18.14 s	13.05 s	12.43 s	3.67 s
$Q20$	15.99 s	15.20 s	10.02 s	16.18 s	19.40 s	17.17 s	17.72 s	17.31 s	11.60 s
$Q21$	3.33 s	3.53 s	3.40 s	11.49 s	10.50 s	11.80 s	10.43 s	11.07 s	2.91 s
$Q22$	2.77 s	4.20 s	3.01 s	2.62 s	3.95 s	5.17 s	3.01 s	3.79 s	3.14 s
Total	206.07 s	208.2 s	**159.88** s	**276.72** s	320.62 s	301.30 s	310.59 s	294.63 s	**166.68** s
	(1.28x)	(1.30x)	**(1.0x)**	**(1.0x)**	(1.15x)	(1.08x)	(1.86x)	(1.76x)	**(1.0x)**

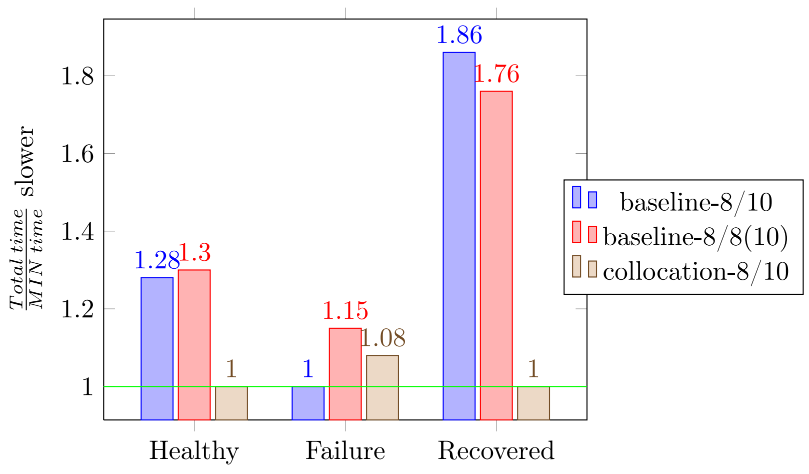


Figure 6.2: Overall runtime performance: *baseline* vs. *collocation* versions, based on Table 6.3

From Table 6.3's new runtime results and Figure 6.3's query I/O throughput measurements, we can derive the same conclusion: **our custom policy still outperforms the default behavior of HDFS** by **1.86x** (better runtime performance) after data recovery. The major difference in the current setup is that some queries may read non-partitioned tables (i.e. access remote data for "cold-runs"). Non-partitioned tables are in principle buffer-cached all over the worker-set, but we first need a prior "cold-run" for that to happen. This is similar with a *single-writer to multiple-readers* behavior. During bulk-loading, non-partitioned tables are written down on HDFS by a single worker (in general the *session-master*). Afterwards, during a "cold-run", all the other workers will read and buffer-cache the existent non-partitioned tables from wherever their HDFS blocks got replicated in the cluster. However, the default placement scatters the remaining *R-1 replicas* all over the entire cluster, requiring a reader to open and manage many short TCP connections for copying such data locally. If the HDFS block size is too small or too few blocks are stored on the same node, the numerous I/O requests to DataNodes will follow a *random-access pattern* with *short sequential-reads* [1]. This results in a poor I/O throughput performance, Figure 6.3, as HDFS performs better during *sequential-reads* of bigger and contiguous blocks [54]. We can see when some queries read non-partitioned data by looking at Table 6.3's individual query times during the *Healthy* state and comparing everything with the *collocation* version. The worse times are for *baseline* and *baseline-restricted*, Query-2 is a good example. The implemented custom policy, in contrast with the default HDFS behavior, stores the block replicas of non-partitioned tables onto the same *R nodes*, i.e. a reader will maintain exactly 1 opened connection per file. Moreover, the before-mentioned advantage of the default behavior during node failures is effectively lost if we restrict the number of Hadoop nodes to the worker-set size, i.e. number of workers equal with the total number of nodes. We see from Figure 6.3 that (for the *baseline-restricted* experiment) the average I/O throughput per query drops from **30.94 MB/s** to **23.20 MB/s** during the two nodes *failure* and then increases to **27.35 MB/s** after *recovery*. New worker nodes, chosen to replace the two failed ones, will no longer have local data and they must access remote data instead. That aside, during HDFS recovery, missing data is replicated over the entire cluster and just some of it will get stored on the two new nodes. Our custom block replication policy exhibits a similar behavior. Nevertheless, the later policy controls the replication of missing data and makes sure that new nodes (or the remaining ones) will gain *full data (co)locality*. This scenario really emphasizes all the differences between the custom HDFS policy and the default out-of-box placement during the three *Health, Failure and Recovered* system states. At the bottom side of the table, in Figure 6.2, we compare the total runtime performance between the *baseline, baseline-restricted, and collocation* versions.

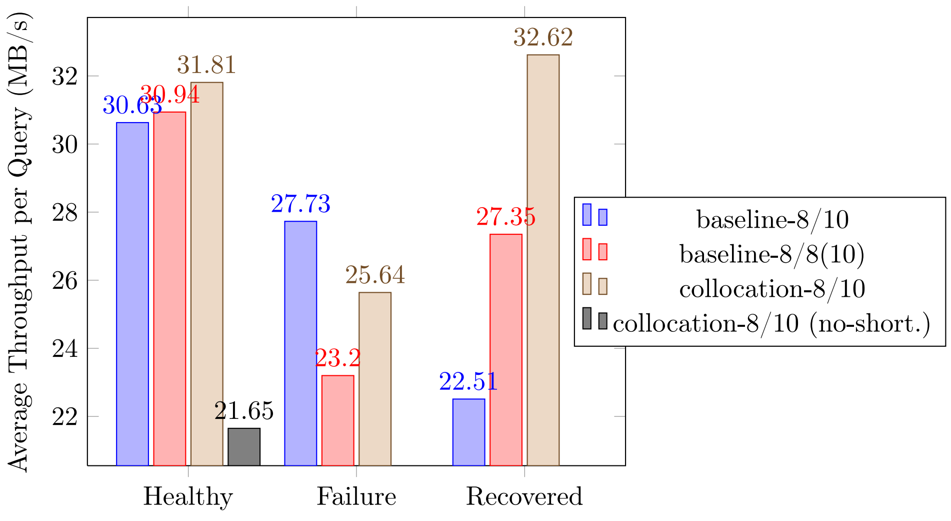


Figure 6.3: Average throughput per query: *baseline* vs. *collocation* versions, related to Table 6.3.

[1]Discussion: stackoverflow.com/questions/13993143/hdfs-performance-for-small-files

Table 6.4: Query execution times: *baseline* (default policy) vs. *collocation* (custom policy) versions, in 3 situations (1) healthy state, (2) failure state (2 nodes down), (3) recovered state. TPC-H benchmark: 32-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl (W/N = workers/hadoop-nodes).

State	**Healthy**		**Failure**		**Recovered**	
Vers.	baseline	collocation	baseline	collocation	baseline	collocation
W/N	8/10	8/10	8/8	8/8	8/8	8/8
$Q1$	19.13 s	19.91 s	20.88 s	22.86 s	25.78 s	18.66 s
$Q2$	32.84 s	18.27 s	32.13 s	22.91 s	28.80 s	18.48 s
$Q3$	3.01 s	2.74 s	3.47 s	4.05 s	4.13 s	2.91 s
$Q4$	1.84 s	1.70 s	2.21 s	3.30 s	2.71 s	1.63 s
$Q5$	1.94 s	1.77 s	4.04 s	4.61 s	3.94 s	1.82 s
$Q6$	0.12 s	0.13 s	2.07 s	2.50 s	1.37 s	0.12 s
$Q7$	3.62 s	3.79 s	7.34 s	9.45 s	7.79 s	3.33 s
$Q8$	3.97 s	3.63 s	8.14 s	10.25 s	9.60 s	3.42 s
$Q9$	37.34 s	27.80 s	50.28 s	44.57 s	46.97 s	27.54 s
$Q10$	40.98 s	30.73 s	31.57 s	39.58 s	30.24 s	32.32 s
$Q11$	13.82 s	9.58 s	15.42 s	12.14 s	13.40 s	10.53 s
$Q12$	1.94 s	1.92 s	3.64 s	4.52 s	4.03 s	1.82 s
$Q13$	14.09 s	14.18 s	14.80 s	18.35 s	23.21 s	13.72 s
$Q14$	1.01 s	0.74 s	1.24 s	1.63 s	1.29 s	0.72 s
$Q15$	2.56 s	1.05 s	1.54 s	1.86 s	1.96 s	1.22 s
$Q16$	11.20 s	7.18 s	11.68 s	9.36 s	10.11 s	8.29 s
$Q17$	2.81 s	1.48 s	12.72 s	14.21 s	11.20 s	1.54 s
$Q18$	10.02 s	9.38 s	16.92 s	20.64 s	20.08 s	8.69 s
$Q19$	4.40 s	3.81 s	14.25 s	16.42 s	13.99 s	3.77 s
$Q20$	18.01 s	12.30 s	20.58 s	17.80 s	18.77 s	13.55 s
$Q21$	3.55 s	3.48 s	10.41 s	12.17 s	11.35 s	3.74 s
$Q22$	5.02 s	4.43 s	4.11 s	3.93 s	4.06 s	3.64 s
Total	233.22 s	**180** s	**289.44** s	297.11 s	294.78 s	**181.11** s
	(1.29x)	**(1.0x)**	**(1.0x)**	(1.02x)	(1.62x)	**(1.0x)**

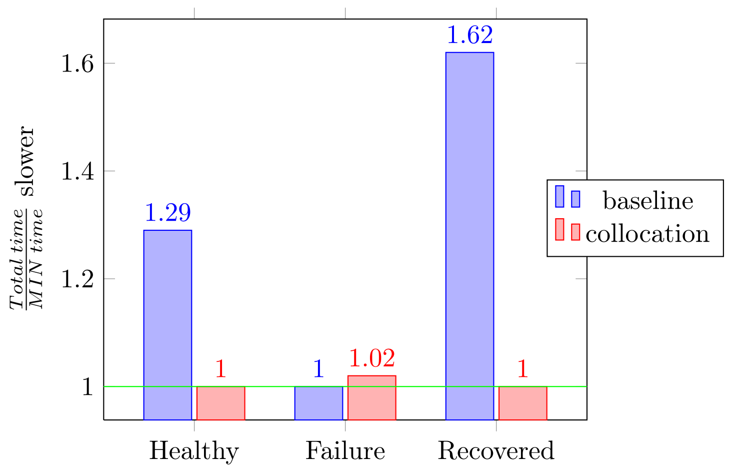


Figure 6.4: Overall runtime performance: *baseline* vs. *collocation* versions, based on Table 6.4

Though the previous tests were performed on a database schema with 16-partitioned tables, the same behavior reproduces for 32-partitioning as well (Table 6.4 with Figure 6.4).

Dynamic Resource Management

Backed with results from the previous section, we prove that we do achieve data-locality to Vectorwise worker nodes after re-replication. However, as Chapter 3 concludes at the end, there is still some room left to improve the performance of Vectorwie during failovers by favoring *local reads* over *remote* (over-the-network) *reads*. To understand how this is done, we recommend (re)reading Chapter 4, which explains in the very detail (including all of our algorithms) how *dynamic resource management* works in Vectorwise on Hadoop. Indeed, the outcome of this approach should improve the performance of Vectorwise on Hadoop *during fail-over situations* and also, when *concurrent* Vectorwise (internal) and Hadoop (external) *workloads overlap* over the same worker nodes / cluster resources. For now we focus on the first direction and only later (in next section) discuss the second. In the following paragraphs we present TPC-H test results, *with* and *without* dynamic resource management enabled, during *two* typical *failover* scenarios.

Table 6.5 shows TPC-H query execution times for Vectorwise on Hadoop during a *2-node failover* situation, with *dynamic resource management* (not. *drm*) and without; both have block collocation enabled. We run one "cold" and then a "hot" test on 8 "Rocks" workers (simply labeled from *worker1* to *worker8*), using *32 mpl* and a schema with *Order* and *Lineitem* in *16 partitions* (from 0 to 15, we omit the *c16* suffix from Section 3.2). The HDFS *replication* degree (R) is left to default value *3*. In total, 48 (3 x 16, including the replicas) partitions are placed on 8 workers, each stores 6 (equal) partitions. Moreover, the *responsibility* degree (RMax), see Section 4.2, is set to *2*. Based on the same section's insights, the later choice makes every worker node being responsible of 4 partitions and *each partition* to have exactly *2 responsible workers*. Also, as mentioned in Section 4.3, this adds a certain elasticity in computing an optimal *resource footprint* for a query workload. We can vary the available resources from *1* thread per node to the *maximum* (available) threads and, at the same time, choose what worker nodes to involve in the computation. To make sure we can do so, we use a maximum core *over-allocation* of *0.25* for more flexibility. With $RMax = 2$ and the *default round robin placement* in place, is possible to form *two separate worker-subsets: 1, 3, 5, 7* and *2, 4, 6, 8* that can get involve independently in query execution (depending on their data-locality and load-factors). Therefore, we construct *two* different *failover* scenarios for *drm*: in **(a)** we fail one worker from a subset (*worker7*) and one from the other (*worker8*), whereas in **(b)** we fail two workers from just one of the subsets (*worker6* and *worker8* from the second subset). In Figure 6.5 we show which **worker nodes** and (partition) **responsibilities** were **involved** in query execution during **(a)** and **(b)** using the 0.25 core over-allocation. Or, simply said, the *resource footprint* for Table 6.5. We remind from Section 4.2 that the *cost-model* tries to change as little as possible the ex-responsibility assignment once the *DbAgent* restarts the worker-set due to *node failovers*. We normalize our (node) naming convention to make it easier to associate between a *failed* and a *new* worker. New worker nodes (which replaced the failed ones) have the suffix *n* added to their label, e.g. *worker6* is replaced by *worker6n*.

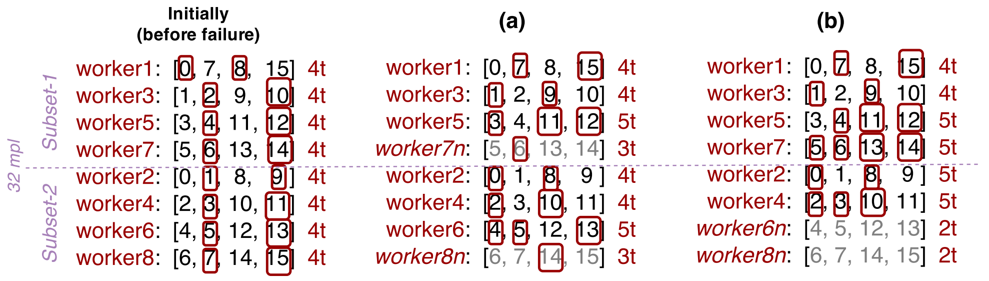

Initially
(before failure)
(a)
(b)
32 mpl
Subset-1
Subset-2
worker1: [0, 7, 8, 15] 4t
worker3: [1, 2, 9, 10] 4t
worker5: [3, 4, 11, 12] 4t
worker7: [5, 6, 13, 14] 4t
worker2: [0, 1, 8, 9] 4t
worker4: [2, 3, 10, 11] 4t
worker6: [4, 5, 12, 13] 4t
worker8: [6, 7, 14, 15] 4t
worker1: [0, 7, 8, 15] 4t
worker3: [1, 2, 9, 10] 4t
worker5: [3, 4, 11, 12] 5t
worker7n: [5, 6, 13, 14] 3t
worker2: [0, 1, 8, 9] 4t
worker4: [2, 3, 10, 11] 4t
worker6: [4, 5, 12, 13] 5t
worker8n: [6, 7, 14, 15] 3t
worker1: [0, 7, 8, 15] 4t
worker3: [1, 2, 9, 10] 4t
worker5: [3, 4, 11, 12] 5t
worker7: [5, 6, 13, 14] 5t
worker2: [0, 1, 8, 9] 5t
worker4: [2, 3, 10, 11] 5t
worker6n: [4, 5, 12, 13] 2t
worker8n: [6, 7, 14, 15] 2t

Figure 6.5: The *worker nodes* and (partition) *responsibilities involved* in query execution during **(a)** and **(b)**. We use *red* color to emphasize the latter, plus the amount of threads per worker to satisfy the *32 mpl*. Besides, we also show the state *before the failure*. The *local responsibilities* are represented with *black* color and the *partial* ones, from the two new nodes, with *grey* color.

Table 6.5: Query execution times during node failures: colloc. *without* dynamic resource management (not. *collocation*) vs. colloc. *with* dynamic resource management (not. *drm*). TPC-H benchmark: 16-partitions for *Order* and *Lineitem*, 8 "Rocks" workers, 32 mpl, 0.25 core over-allocation, RMax resp. degree 2. Since RMax = 2 we can form two separate worker-subsets. In **(a)** we fail 1 worker from a subset and 1 from the other, in **(b)** we fail 2 workers from just one of the subsets.

Test-run	**Hot**			**Cold**		
Vers.	collocation	drm-**a**-0.25	drm-**b**-0.25	collocation	drm-**a**-0.25	drm-**b**-0.25
$Q1$	0.70 s	1.32 s	1.31 s	29.70 s	27.37 s	27.50 s
$Q2$	0.26 s	0.26 s	0.26 s	16.29 s	21.20 s	20.03 s
$Q3$	0.16 s	0.17 s	0.18 s	4.04 s	2.72 s	2.95 s
$Q4$	0.04 s	0.06 s	0.06 s	2.06 s	1.42 s	1.59 s
$Q5$	0.27 s	0.39 s	0.40 s	4.76 s	3.54 s	3.55 s
$Q6$	0.04 s	0.07 s	0.07 s	2.32 s	1.69 s	1.15 s
$Q7$	0.29 s	0.43 s	0.44 s	10.42 s	8.22 s	7.48 s
$Q8$	0.40 s	0.54 s	0.55 s	10.01 s	7.49 s	5.98 s
$Q9$	1.77 s	2.87 s	2.96 s	47.48 s	40.70 s	32.02 s
$Q10$	1.83 s	2.06 s	2.05 s	32.61 s	34.63 s	28.86 s
$Q11$	0.22 s	0.22 s	0.22 s	10.21 s	11.60 s	9.91 s
$Q12$	0.15 s	0.24 s	0.26 s	4.32 s	2.47 s	2.57 s
$Q13$	3.50 s	4.87 s	5.04 s	23.42 s	21.19 s	21.44 s
$Q14$	0.26 s	0.29 s	0.29 s	1.63 s	1.33 s	1.04 s
$Q15$	0.34 s	0.40 s	0.42 s	2.74 s	1.28 s	1.89 s
$Q16$	0.45 s	0.47 s	0.45 s	7.91 s	8.67 s	6.92 s
$Q17$	0.52 s	0.83 s	0.84 s	16.28 s	11.51 s	4.77 s
$Q18$	1.17 s	1.62 s	1.62 s	22.82 s	14.43 s	14.44 s
$Q19$	0.58 s	0.95 s	1.03 s	18.14 s	15.17 s	7.93 s
$Q20$	0.47 s	0.43 s	0.44 s	17.17 s	16.80 s	12.06 s
$Q21$	1.50 s	2.31 s	2.33 s	11.80 s	8.68 s	7.24 s
$Q22$	0.43 s	0.49 s	0.48 s	5.17 s	3.70 s	2.95 s
Total	**15.35** s	21.29 s	21.70 s	301.30 s	265.81 s	**224.27** s
	(1.0x)	(1.38x)	(1.41x)	(1.34x)	(1.18x)	**(1.0x)**

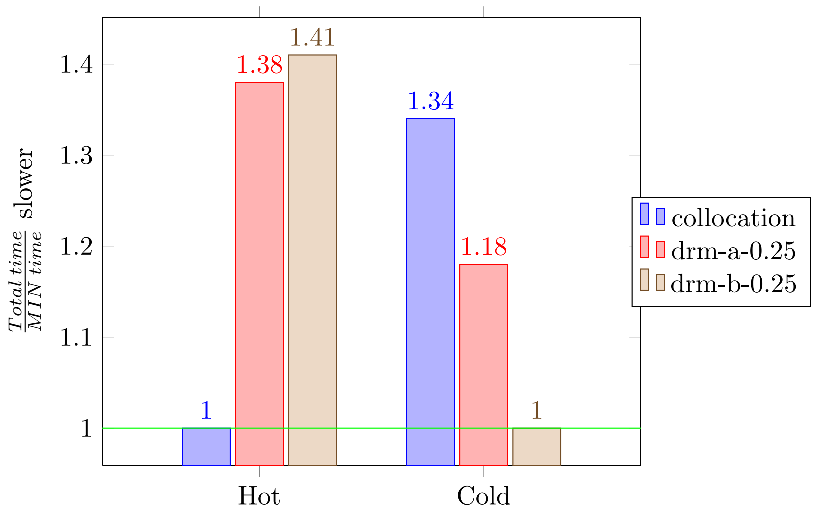
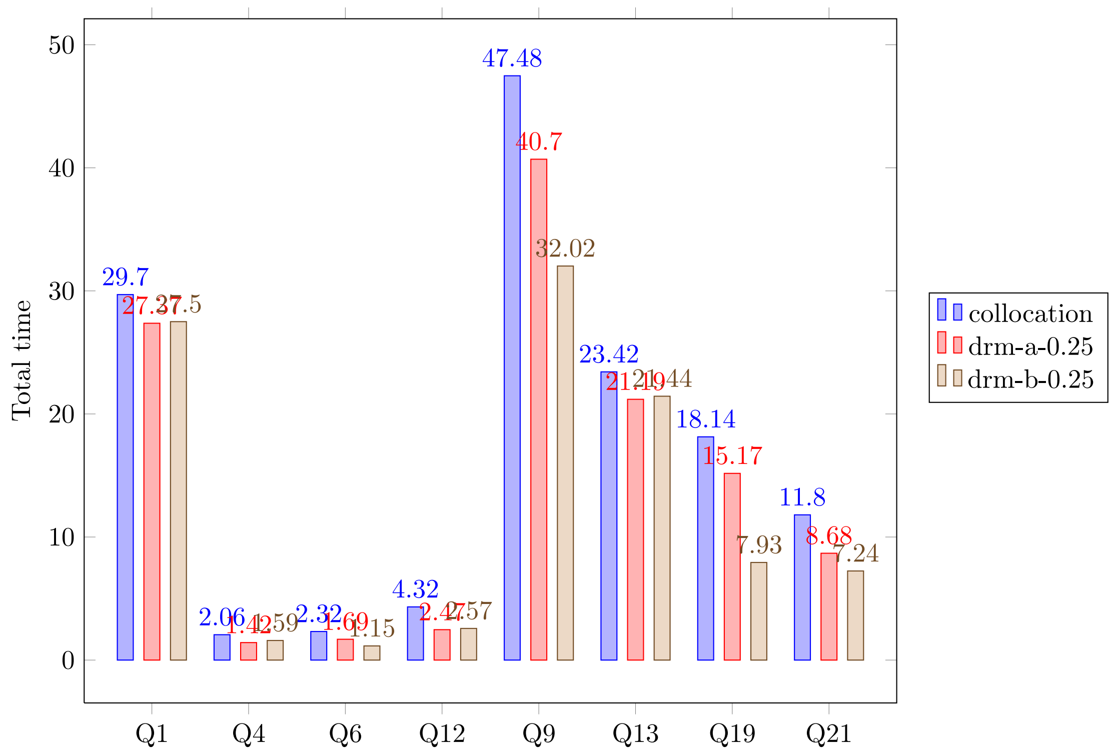
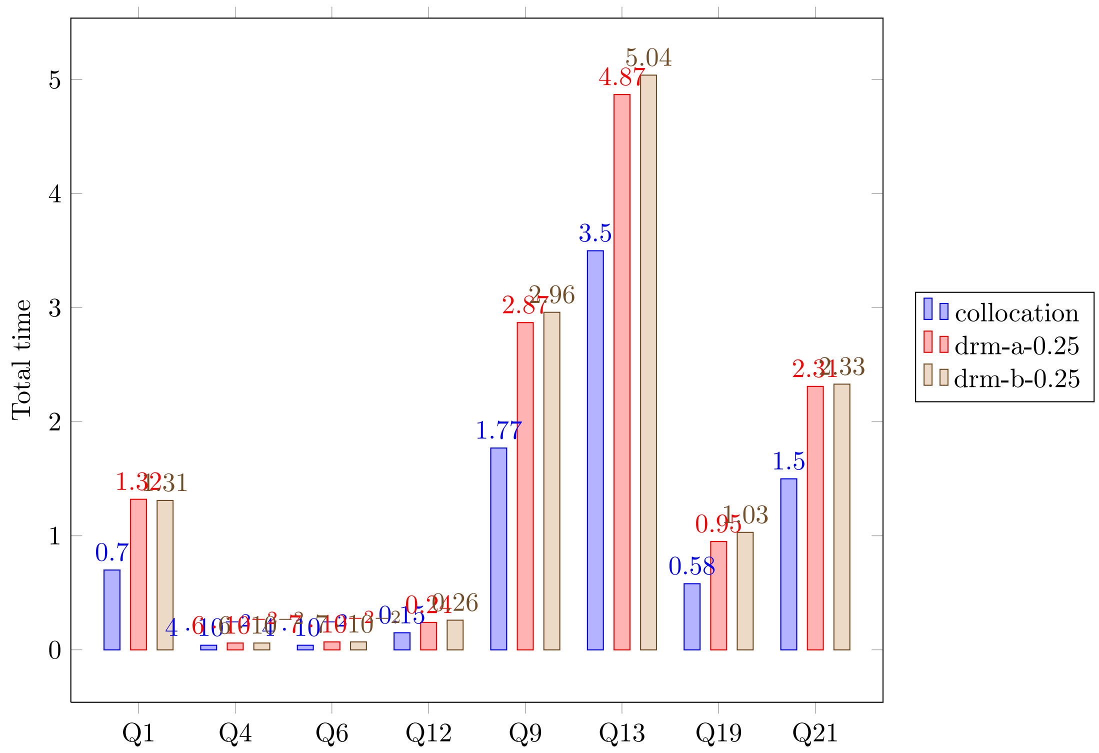


Figure 6.6: Overall runtime performance: *collocation* vs. *drm* versions, based on Table 6.5.



(a) Cold-runs: Q1, Q4, Q6, Q12 (I/O bounded) and Q9, Q13, Q19, Q21 (I/O bounded), based on Table 6.5



(b) Hot-runs: Q1, Q4, Q6, Q12 (now CPU bounded) and Q9, Q13, Q19, Q21 (now CPU & network bounded), based on Table 6.5

Figure 6.7: Overall runtime performance: Q1, Q4, Q6, Q9, Q12, Q13, Q19, and Q21, based on Table 6.5

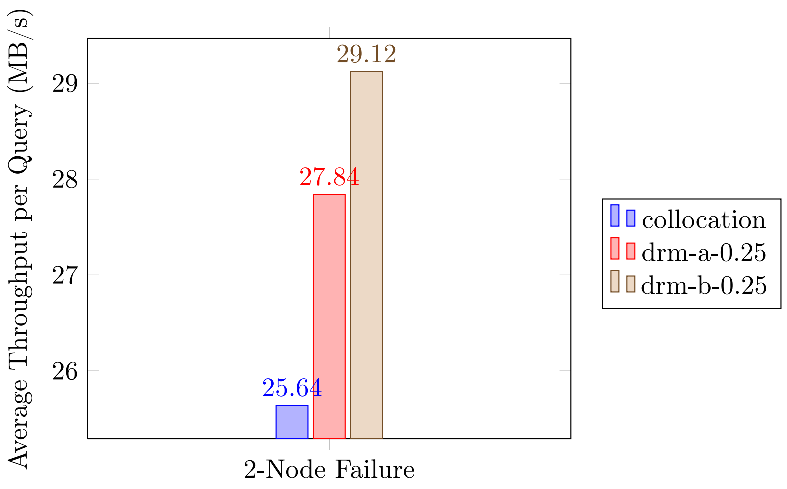


Figure 6.8: Average throughput per query: *collocation* vs. *drm* versions, related to Table 6.5.

Figure 6.5 clearly shows the difference between how the new *resource scheduling* algorithm (Section 4.3) picks the right worker nodes, responsibilities and number of threads (per worker) for query execution during **(a)** and **(b)**. Although we try with our cost-model to only get local reads for **(a)**, yet we are still forced to involve *worker7n* and *worker8n* in I/O for the *6c16* and *14c16* partial (partition) responsibilities. It is important to observe that the two failed workers were consecutive in the list of nodes used by the *round-robin placement*, plus the *responsibility assignment* algorithm (see Chapter 4), which made them the single nodes responsible for *6c16* and *14c16*. Failing two consecutive worker nodes (when replication degree $R = 3$) is probably the worst to happen. In contrast, case **(b)** is free of the previous (partial responsibility) constraints. *Worker6* and *worker8* belong to the same *Subset-2* and so, implicitly, they are no longer consecutive in the input list. This means that the *resource scheduling* algorithm is not forced to involve partial responsibilities from the two new nodes anymore. Instead, it can find other workers (e.g. from *Subset-1*) that have those (partition) responsibilities already stored. For instance, *worker6n* and *worker8n* have responsibilities in common with *workers 1, 5 and 7*, but the latter three have them locally. With *core over-allocation = 0.25* in this case, the *resource scheduling* algorithm finds a worker-subset composed only of *workers 1, 2, 3, 4, 5, and 7*, which has all our data and achieves the *32 mpl*). It also distributes the *mpl* (32 threads) proportionally with each of the local responsibilities involved in query execution, e.g. *worker7* has to read table partitions from its all *4 local responsibilities* using (the maximum number of) *5 threads*. The *mpl* distribution applies to **(a)** as well. We exclude *6n and 8n* only from the I/O operations by replacing them with "healthy" workers that have *full data locality* and can support their burden. Note that we leave them 2 threads each for intermediary processing (e.g. intermediary aggregations). Because of this reason, **(b)** outperforms **(a)** by **1.18x**. We see this from their runtime performance, Table 6.5, and average I/O throughput, in Figure 6.8. Nonetheless, yet **(a)** and **(b)** are each one better than the plain *collocation* version (i.e. without dynamic resource management), by **1.13x** and, respectively, **1.34x**. The latter difference is depicted in Figure 6.6, compared with *drm-b-0.25* version. Since both **(a)** and **(b)** *outperform* the plain *collocation* version on *cold-runs* during *failures*, it implies that overall (based on Table 6.3) we improve upon the *baseline* version – with **(a)** by 1.04x and **(b)** by 1.23x. The difference can be seen in their average I/O throughput too, Figure 6.3 and Figure 6.8. Hence, we can say that we have achieved our goal to *improve* the performance of Vectorwise during *node failures* by *favoring local reads* over *remote (access) reads.*

Obviously though, for *hot-runs*, where almost all TPC-H queries become *CPU or network bounded*, the assignment from Figure 6.5 is no longer balanced. We took the time results of queries Q1, Q4, Q6, Q9, Q12, Q13, Q19 and Q21 and compared them for cold- and hot- runs separately, in Figure 6.7. During cold-runs (Figure 6.7a) all these queries are I/O bounded and so, *drm-b-0.25* performs the best. On the other hand, during hot-runs (Figure 6.7b) we exhibit the opposite – Q1, Q4, Q6, and Q12 are now CPU bounded, whereas the remaining Q9, Q13, Q19 and Q21 are network bounded. These 8 queries (and especially the latter 4) have the worse degrading performance for the *drm* versions during the hot-run. For example, because of the assignment from Figure 6.5, *workers 4, 5, and 7* are now a bottleneck during query execution in *drm-b-0.25*. This is a tradeoff that the before-mentioned approach has to deal with it during the timespan of *HDFS data recovery*. We could diminish this imbalance effect by *increasing* the *core over-allocation* (e.g. 0.5 over-allocation) and the *degree of partitioning* (e.g. 32 partitions), it should provide the new *resource scheduling* algorithm more *flexibility* in balancing the *mpl* over the chose worker-subset. Nevertheless, for the previous experiment, it takes around 8-9 minutes to recover the 3rd missing (approximately 26 GB of data) replicas. This time can be easily amortized considering the side effect of controlling data-locality, i.e. reading local data during cold-runs or when is not enough buffer pool to cache all the tables.

YARN Integration

As we mentioned earlier in this section, we are now going to present the performance results of the Vectorwise on Hadoop database system when *internal* workloads overlap with *concurrent* (Hadoop) *external* workloads over the same worker nodes / cluster resources. Hence, we focus on the improvements brought by the *dynamic resource scheduling* approach (implicitly the min-cost flow algorithm, plus our cost model) and YARN integration for cluster awareness and resource management, while simulating Hadoop *busyness* over the worker-set with the TeraSort benchmark. To remember the essentials, we recommend looking into Section 4.3. Summarizing, the approach dynamically schedules cluster compute resources according to the worker-set resource availability, load-balancing constraints, and data-locality. This should reduce the overall resource contention that exhibits when two or more running systems, coexisting on the same cluster, are not aware of each others workloads. As we pointed out at the end of Section 6.1, we can configure our TeraSort (Hadoop) workloads to run with 16, 32, or 64 *mappers* and *reducers*, either on the *full worker-set*, or on just on a *half* of it (both with their own input data directory). Extra details (i.e. total runtime, CPU load) are given in Table 6.2.

In Tables 6.7, 6.8 and 6.9 we present the runtime results of executing a sequence of 11x TPC-H Q1 and Q13 queries in Vectorwise on Hadoop, *with* and *without* YARN integration, side by side with each of the 6 workload combinations (started first), but only for the 100GB dataset because our purpose is to increase the *Map* phase duration and to prolong the resource contention. We also ran the same experiment without any busyness, Table 6.2, to have a baseline for comparison and also to highlight the resource allocation overhead that shows up at the beginning of each test. The correspondent assignments of workers, (partition) responsibilities and threads per node (out of max available for Vectorwise) involved in query execution during the no-busyness, 16M/16R, 32M/32R, and 64M/64R **half-set** runs are depicted in Figure 6.9, Figure 6.10, Figure 6.11, and respectively, Figure 6.12. Before each of the experiments we clear the File System's and buffers manager's caches and execute a no-busyness cold-run, Figure 6.9 tells us what responsibilities are "hot" before the tests. By doing this we can measure the performance degradation of *drmy* when its runtime assignment dynamically changes because of the cluster's busyness and thus, "cold" partitions are being read from the disk. In our tables we show both the results when *drmy's* buffers are "semi-hot" (not. *drmy-*) for the experiment and "hot" (not. *drmy+*) as well. Note that from the *drmy-* time results we have excluded the overhead of allocating resources.

In the end of this section we measure the average CPU load (not. from *[min* to *Max]*) of the Vectorwise workers on which both internal and external workloads overlap during the *Map* phase, Figure 6.13 and Figure 6.14; we can identify that by looking at the assignment figures below the tables (i.e. if max available cores for Vectorwise $<$ max worker capability, which is 16 cores, then it means other workloads are running on the same node). Something to note in our experiments is that our system's networking layers do not interleave, Hadoop uses Ethernet, whereas Vectorwise uses the Infinband fabric.

Table 6.6: Query execution times without Hadoop busyness, no overlapping external workloads: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2.

Busyness	**None**		
Vers.	drm	drmy-	drmy+
$Q1$	0.74 s	5.69 s	0.73 s
$Q13$	7.47 s	7.26 s	6.96 s
$Q1$	0.71 s	0.71 s	0.70 s
$Q13$	6.92 s	7.21 s	6.85 s
$Q1$	0.70 s	0.67 s	0.73 s
$Q13$	6.43 s	7.16 s	6.72 s
$Q1$	0.71 s	0.72 s	0.68 s
$Q13$	7.41 s	7.19 s	6.91 s
$Q1$	0.72 s	0.72 s	0.73 s
$Q13$	6.60 s	6.56 s	6.69 s
$Q1$	0.70 s	0.66 s	0.67 s
$Q13$	7.45 s	7.10 s	6.82 s
$Q1$	0.71 s	0.68 s	0.72 s
$Q13$	6.59 s	6.93 s	7.04 s
$Q1$	0.70 s	0.71 s	0.71 s
$Q13$	6.69 s	6.75 s	6.99 s
$Q1$	0.70 s	0.69 s	0.72 s
$Q13$	6.64 s	7.05 s	6.72 s
$Q1$	0.70 s	0.71 s	0.69 s
$Q13$	6.45 s	7.11 s	6.74 s
$Q1$	0.71 s	0.71 s	0.70 s
$Q13$	6.40 s	6.83 s	6.64 s
Total	82.85 s	89.82 s	82.86 s
	(1.0x)	(1.08x)	**(1.0x)**

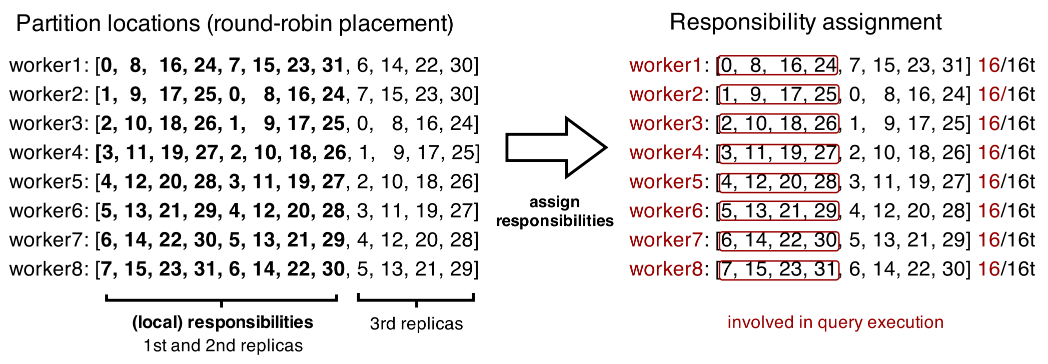

Partition locations (round-robin placement)
worker1: [0, 8, 16, 24, 7, 15, 23, 31, 6, 14, 22, 30]
worker2: [1, 9, 17, 25, 0, 8, 16, 24, 7, 15, 23, 30]
worker3: [2, 10, 18, 26, 1, 9, 17, 25, 0, 8, 16, 24]
worker4: [3, 11, 19, 27, 2, 10, 18, 26, 1, 9, 17, 25]
worker5: [4, 12, 20, 28, 3, 11, 19, 27, 2, 10, 18, 26]
worker6: [5, 13, 21, 29, 4, 12, 20, 28, 3, 11, 19, 27]
worker7: [6, 14, 22, 30, 5, 13, 21, 29, 4, 12, 20, 28]
worker8: [7, 15, 23, 31, 6, 14, 22, 30, 5, 13, 21, 29]
(local) responsibilities
1st and 2nd replicas
3rd replicas
assign responsibilities
Responsibility assignment
worker1: [0, 8, 16, 24, 7, 15, 23, 31] 16/16t
worker2: [1, 9, 17, 25, 0, 8, 16, 24] 16/16t
worker3: [2, 10, 18, 26, 1, 9, 17, 25] 16/16t
worker4: [3, 11, 19, 27, 2, 10, 18, 26] 16/16t
worker5: [4, 12, 20, 28, 3, 11, 19, 27] 16/16t
worker6: [5, 13, 21, 29, 4, 12, 20, 28] 16/16t
worker7: [6, 14, 22, 30, 5, 13, 21, 29] 16/16t
worker8: [7, 15, 23, 31, 6, 14, 22, 30] 16/16t
involved in query execution

Figure 6.9: The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.6. We use *red* color to highlight the latter information. The metadata for *partition locations* (see Section 3.2) is shown on the left and the *responsibility assignment* on the right.

Table 6.7: Query execution times during overlapping external (Hadoop) workloads, 16 Mappers / 16 Reducers: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2.

Busyness	**Full-set**			**Half-set**		
Vers.	drm	drmy-	drmy+	drm	drmy-	drmy+
$Q1$	0.77 s	7.69 s	0.71 s	1.78 s	8.89 s	1.92 s
$Q13$	7.89 s	7.39 s	7.67 s	10.99 s	7.56 s	8.19 s
$Q1$	0.78 s	0.73 s	0.73 s	1.19 s	1.88 s	1.89 s
$Q13$	7.73 s	7.13 s	7.06 s	10.08 s	7.99 s	7.70 s
$Q1$	0.78 s	0.69 s	0.71 s	1.06 s	1.91 s	1.90 s
$Q13$	7.66 s	6.92 s	7.29 s	10.09 s	8.84 s	8.39 s
$Q1$	0.74 s	0.73 s	0.69 s	1.13 s	1.94 s	1.88 s
$Q13$	7.65 s	7.27 s	6.90 s	9.78 s	8.23 s	7.68 s
$Q1$	0.73 s	0.74 s	0.77 s	1.40 s	1.91 s	1.98 s
$Q13$	7.84 s	6.91 s	6.92 s	10.05 s	7.87 s	8.21 s
$Q1$	0.79 s	0.71 s	0.68 s	1.30 s	1.90 s	1.91 s
$Q13$	7.72 s	7.03 s	6.81 s	10.07 s	7.78 s	8.31 s
$Q1$	0.75 s	0.86 s	0.73 s	1.23 s	1.92 s	1.94 s
$Q13$	7.63 s	7.13 s	7.42 s	10.44 s	7.70 s	7.69 s
$Q1$	0.75 s	0.69 s	0.73 s	0.96 s	1.92 s	1.96 s
$Q13$	7.62 s	6.90 s	7.13 s	9.86 s	7.40 s	7.72 s
$Q1$	0.74 s	0.68 s	0.69 s	0.89 s	1.90 s	1.86 s
$Q13$	7.57 s	7.22 s	7.32 s	10.21 s	8.22 s	7.57 s
$Q1$	0.74 s	0.69 s	0.80 s	1.11 s	1.92 s	1.92 s
$Q13$	7.63 s	6.88 s	7.60 s	10.51 s	8.04 s	7.85 s
$Q1$	0.78 s	0.72 s	0.69 s	1.42 s	1.88 s	1.90 s
$Q13$	7.77 s	7.37 s	7.73 s	9.74 s	7.77 s	8.26 s
Total	93.06 s	93.08 s	87.78 s	125.29 s	115.37 s	108.63 s
	(1.06x)	(1.06x)	**(1.0x)**	(1.15x)	(1.06x)	**(1.0x)**

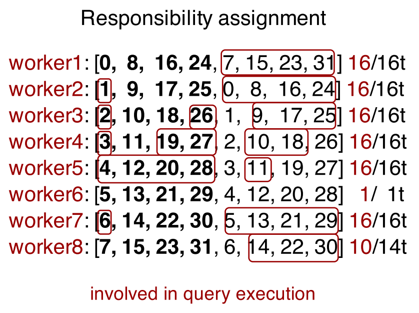

Responsibility assignment
worker1: [0, 8, 16, 24, 7, 15, 23, 31] 16/16t
worker2: [1, 9, 17, 25, 0, 8, 16, 24] 16/16t
worker3: [2, 10, 18, 26, 1, 9, 17, 25] 16/16t
worker4: [3, 11, 19, 27, 2, 10, 18, 26] 16/16t
worker5: [4, 12, 20, 28, 3, 11, 19, 27] 16/16t
worker6: [5, 13, 21, 29, 4, 12, 20, 28] 1/ 1t
worker7: [6, 14, 22, 30, 5, 13, 21, 29] 16/16t
worker8: [7, 15, 23, 31, 6, 14, 22, 30] 10/14t
involved in query execution

Figure 6.10: The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.7's **half-set** busyness runs. We use *red* color to highlight the latter information.

Table 6.8: Query execution times during overlapping external (Hadoop) workloads, 32 Mappers / 32 Reducers: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2.

Busyness	**Full-set**			**Half-set**		
Vers.	drm	drmy-	drmy+	drm	drmy-	drmy+
$Q1$	0.96 s	7.20 s	1.05 s	1.76 s	18.47 s	1.94 s
$Q13$	8.75 s	7.80 s	7.40 s	9.91 s	15.13 s	7.82 s
$Q1$	0.89 s	0.99 s	1.09 s	1.89 s	1.90 s	1.88 s
$Q13$	8.37 s	7.82 s	7.88 s	8.84 s	7.91 s	7.62 s
$Q1$	0.83 s	0.99 s	1.00 s	1.28 s	1.96 s	1.97 s
$Q13$	8.00 s	8.00 s	7.83 s	8.56 s	8.11 s	7.55 s
$Q1$	0.84 s	1.03 s	1.00 s	0.80 s	1.88 s	1.93 s
$Q13$	8.30 s	8.05 s	7.92 s	8.33 s	8.53 s	7.53 s
$Q1$	0.87 s	1.02 s	1.03 s	1.05 s	1.91 s	1.92 s
$Q13$	8.33 s	7.95 s	7.84 s	8.55 s	8.94 s	7.48 s
$Q1$	0.88 s	1.01 s	1.01 s	0.79 s	1.92 s	1.90 s
$Q13$	8.18 s	8.01 s	7.86 s	8.86 s	8.50 s	7.96 s
$Q1$	0.88 s	1.01 s	1.04 s	0.73 s	1.91 s	1.80 s
$Q13$	8.15 s	7.93 s	7.70 s	8.05 s	7.99 s	7.10 s
$Q1$	0.87 s	1.02 s	1.00 s	0.93 s	1.94 s	1.88 s
$Q13$	8.16 s	7.63 s	7.67 s	8.04 s	6.74 s	6.82 s
$Q1$	0.89 s	1.03 s	1.04 s	0.77 s	1.86 s	1.88 s
$Q13$	8.10 s	7.77 s	7.64 s	8.16 s	7.20 s	6.78 s
$Q1$	0.90 s	1.02 s	1.09 s	0.81 s	1.95 s	1.90 s
$Q13$	8.10 s	7.58 s	7.81 s	8.06 s	6.50 s	7.01 s
$Q1$	0.81 s	1.01 s	1.03 s	0.88 s	1.87 s	1.89 s
$Q13$	8.10 s	7.32 s	7.53 s	8.41 s	6.67 s	6.76 s
Total	100.16 s	103.19 s	96.46 s	105.46 s	129.79 s	101.32 s
	(1.03x)	(1.06x)	**(1.0x)**	(1.04x)	(1.28x)	**(1.0x)**

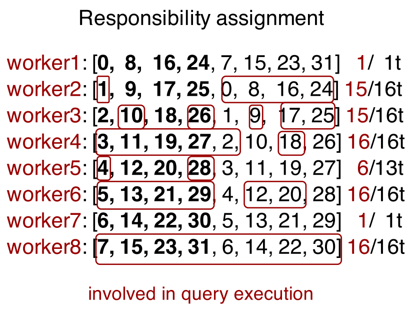
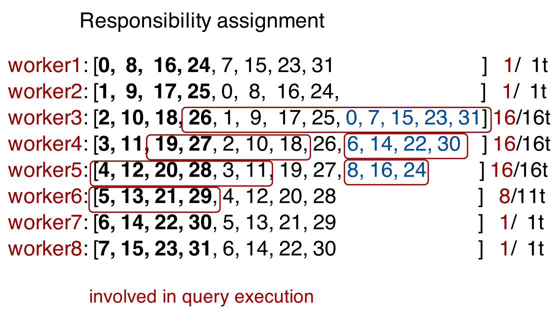

Responsibility assignment
worker1: [0, 8, 16, 24, 7, 15, 23, 31] 1/ 1t
worker2: [1, 9, 17, 25, 0, 8, 16, 24] 15/16t
worker3: [2, 10, 18, 26, 1, 9, 17, 25] 15/16t
worker4: [3, 11, 19, 27, 2, 10, 18, 26] 16/16t
worker5: [4, 12, 20, 28, 3, 11, 19, 27] 6/13t
worker6: [5, 13, 21, 29, 4, 12, 20, 28] 16/16t
worker7: [6, 14, 22, 30, 5, 13, 21, 29] 1/ 1t
worker8: [7, 15, 23, 31, 6, 14, 22, 30] 16/16t
involved in query execution

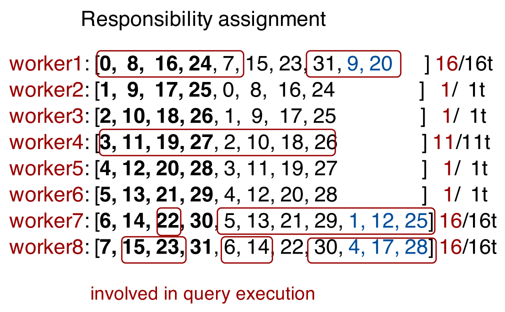
Figure 6.11: The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.8's **half-set** busyness runs. We use *red* color to highlight the latter information.

Table 6.9: Query execution times during overlapping external (Hadoop) workloads, 64 Mappers / 64 Reducers: testing Vectorwise *without* YARN integration (not. *drm*) vs. *with* YARN integration (not. *drmy*). TPC-H benchmark: Q1 and Q13 hot-runs, 32-partitions for *Order* and *Lineitem*, 8 "Stones" workers, 128 mpl, no core over-allocation, RMax resp. degree 2.

Busyness	**Full-set**			**Half-set**				
Vers.	drm	drmy-	drmy+	drm	drmy- (#1)	drmy+ (#1)	drmy- (#2)	drmy+ (#2)
$Q1$	1.29 s	9.03 s	2.07 s	6.92 s	46.30 s	2.48 s	44.35 s	1.94 s
$Q13$	13.26 s	9.09 s	9.52 s	63.91 s	33.81 s	14.38 s	32.94 s	11.25 s
$Q1$	1.34 s	1.93 s	1.98 s	5.59 s	1.99 s	2.08 s	1.92 s	1.99 s
$Q13$	12.37 s	9.20 s	9.53 s	21.10 s	11.45 s	14.11 s	10.02 s	11.07 s
$Q1$	1.37 s	1.89 s	2.02 s	5.84 s	1.97 s	2.12 s	1.89 s	1.93 s
$Q13$	11.58 s	8.04 s	8.62 s	11.17 s	11.49 s	16.71 s	9.98 s	10.55 s
$Q1$	1.16 s	1.91 s	1.95 s	2.83 s	1.94 s	2.88 s	1.91 s	1.99 s
$Q13$	12.08 s	7.23 s	7.87 s	8.53 s	11.31 s	10.60 s	10.12 s	10.64 s
$Q1$	1.68 s	2.00 s	1.89 s	1.03 s	1.94 s	1.92 s	1.98 s	1.97 s
$Q13$	8.25 s	5.76 s	6.69 s	8.31 s	10.28 s	10.58 s	9.97 s	10.31 s
$Q1$	0.83 s	2.07 s	2.12 s	0.89 s	1.96 s	1.95 s	1.91 s	1.93 s
$Q13$	8.31 s	5.39 s	5.23 s	8.98 s	10.34 s	9.83 s	9.95 s	10.35 s
$Q1$	0.88 s	2.04 s	2.12 s	0.80 s	1.95 s	2.00 s	1.93 s	1.94 s
$Q13$	8.54 s	5.25 s	5.31 s	8.48 s	10.18 s	10.77 s	10.09 s	10.32 s
$Q1$	0.81 s	2.08 s	2.09 s	0.87 s	1.92 s	1.96 s	1.89 s	1.94 s
$Q13$	7.99 s	6.05 s	5.58 s	8.16 s	10.71 s	10.00 s	10.25 s	10.57 s
$Q1$	0.86 s	2.07 s	2.11 s	0.87 s	1.95 s	1.95 s	1.90 s	1.95 s
$Q13$	8.56 s	5.35 s	5.38 s	8.62 s	10.92 s	10.33 s	9.96 s	10.30 s
$Q1$	1.05 s	2.04 s	1.99 s	0.85 s	1.91 s	1.94 s	1.87 s	2.06 s
$Q13$	8.49 s	6.75 s	4.73 s	8.63 s	10.28 s	10.10 s	9.94 s	10.92 s
$Q1$	0.89 s	1.93 s	2.02 s	1.05 s	1.89 s	1.94 s	1.88 s	2.00 s
$Q13$	8.29 s	6.73 s	4.67 s	8.88 s	10.76 s	11.06 s	10.06 s	10.76 s
Total	119.88 s	103.83 s	95.49 s	192.31 s	207.25 s	151.69 s	196.71 s	138.68 s
Full -set	(1.25x)	(1.08x)	**(1.0x)**	–	–	–	–	–
Half-set (1)	–	–	–	(1.26x)	(1.36x)	**(1.0x)**	–	–
Half-set (2)	–	–	–	(1.38x)	–	–	(1.41x)	**(1.0x)**

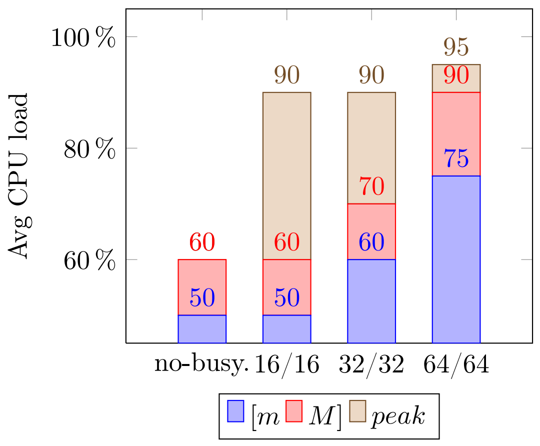

Responsibility assignment
worker1: [0, 8, 16, 24, 7, 15, 23, 31] 1/ 1t
worker2: [1, 9, 17, 25, 0, 8, 16, 24,] 1/ 1t
worker3: [2, 10, 18, 26, 1, 9, 17, 25, 0, 7, 15, 23, 31] 16/16t
worker4: [3, 11, 19, 27, 2, 10, 18, 26, 6, 14, 22, 30] 16/16t
worker5: [4, 12, 20, 28, 3, 11, 19, 27, 8, 16, 24] 16/16t
worker6: [5, 13, 21, 29, 4, 12, 20, 28] 8/11t
worker7: [6, 14, 22, 30, 5, 13, 21, 29] 1/ 1t
worker8: [7, 15, 23, 31, 6, 14, 22, 30] 1/ 1t
involved in query execution

For run (1)

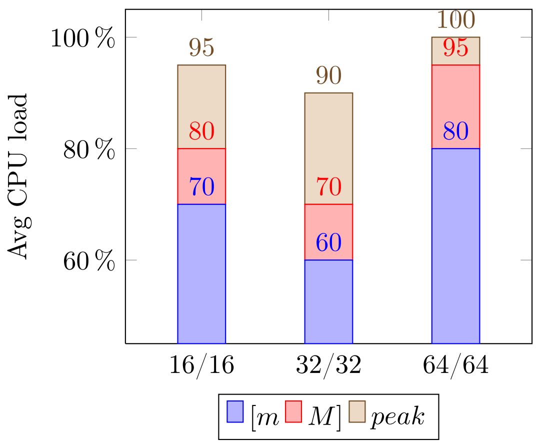

Responsibility assignment
worker1: [0, 8, 16, 24, 7, 15, 23, 31, 9, 20] 16/16t
worker2: [1, 9, 17, 25, 0, 8, 16, 24] 1/ 1t
worker3: [2, 10, 18, 26, 1, 9, 17, 25] 1/ 1t
worker4: [3, 11, 19, 27, 2, 10, 18, 26] 11/11t
worker5: [4, 12, 20, 28, 3, 11, 19, 27] 1/ 1t
worker6: [5, 13, 21, 29, 4, 12, 20, 28] 1/ 1t
worker7: [6, 14, 22, 30, 5, 13, 21, 29, 1, 12, 25] 16/16t
worker8: [7, 15, 23, 31, 6, 14, 22, 30, 4, 17, 28] 16/16t
involved in query execution

For run (2)

Figure 6.12: The *worker nodes*, partition *responsibilities* and the number of *threads per node* (out of max available) involved in query execution, for Table 6.9's **half-set** busyness runs. right. We use *red* color to highlight the latter information. With *blue* color we mark the *remote* responsibilities being assigned at runtime due to overloaded workers.

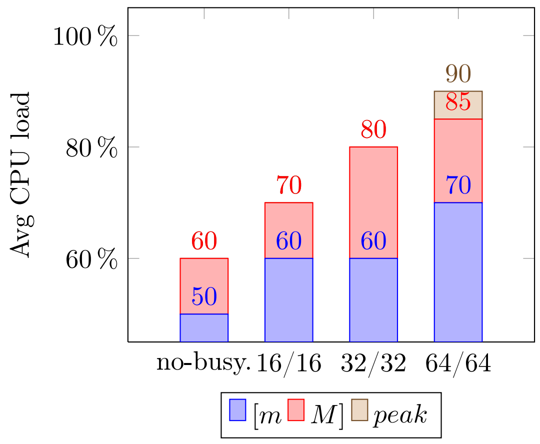


(a) Measuring CPU load during the **full-set** run, overlapping workers/cores.

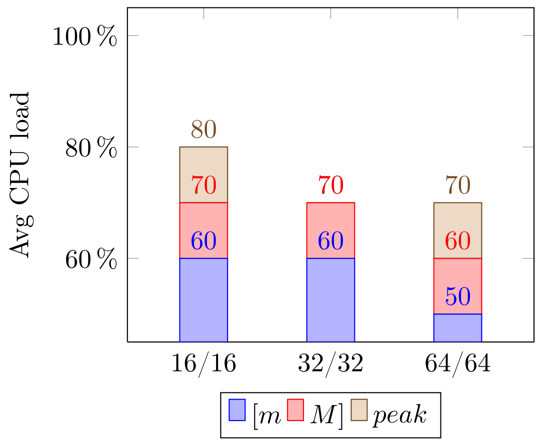


(b) Measuring CPU load during the **half-set** run, overlapping workers/cores.

Figure 6.13: Average CPU load (not. from *[min* to *Max]*) for *drm*, while running Q1 and Q13 side by side with 16, 32 and 64 Mappers and Reducers, on a **full-set** in left and **half-set** in right.



(a) Measuring CPU load during the **full-set** run, non-overlapping resources.



(b) Measuring CPU load during the **half-set** (#1) run, non-overlapping resources.

Figure 6.14: Average CPU load (not. from *[min* to *Max]*) for *drmy*, while running Q1 and Q13 side by side with 16, 32 and 64 Mappers and Reducers, on a **full-set** in left and **half-set** in right.

For the result discussion, we start with our *full-set* runs. For these tests, *drmy* (with YARN integration and, implicitly, with the new *resource scheduling* algorithm) uses the same assignment as if the cluster had no busyness, that is the assignment from Figure 6.9. Our *full-set* TeraSort jobs are using the full-set of workers in a balanced manner, i.e. 2, 4 and 8 M/R per node, hence the runtime assignment (i.e. the resource footprint) has to be balanced as well. The only thing that differs, compared with *drm* that had no YARN awareness, is the *mpl* that we try to achieve when different workloads overlap on the same set of resources. Because *drmy* is aware that our CPU available resources are much lower during each 16, 32 to 64 M/R test, it decreases the *mpl* from *128* (w/o busyness) to *128 - 17 = 111*, *128 - 33 = 95* and, respectively, *128 - 65 = 63* threads. In this situation, the algorithm acts more as a *throttling* mechanism. As we can see from the 64M/64R *drm* tests, for the first 10 queries or so (exactly while the *Map* phase of TeraSort happens) the performance degrades, i.e. spikes to 95% in the CPU load. This does not happen to the *drmy* version for instance, or if it happens is affecting Vectorwise less. It is important to mention that 1 core is used by the TeraSort (managed) Application Master and this actually used to influence our experiments for the *full-set* runs, i.e. the core is allocated and it appears so in YARN, but never used intensively as such. We had to literally force our algorithms to ignore this glitch when deciding the *resource footprint* at runtime. Yet, overall, we achieve better results compared with the *drm* version (w/o YARN integration). Tables 6.7, 6.8 and 6.9 show better (individual and total) times for the 11x Q1,Q13 workload, though we suffer a lot due to the allocation overhead as seen in the first query's time result. Also, few to non-existent CPU peaks are recorded during the *full-set* runs for *drmy*, Figure 6.14, as compared to *drm* in Figure 6.13. Note that in all our *drmy* tests we do not have overlapping resources (i.e. each workload, internal and external, uses its own reserved resources). The CPU load measurements were done for those cores allocated exclusively to Vectorwise on the worker nodes which also happened to run a small part of the external Hadoop workload. Hence, is perfectly normal for the CPU load to be somewhat bigger compared to a normal (w/o any busyness) run given that we read and process the same amount of data, but on less cores (and implicitly with less parallelism).

For the *half-set* tests we are just going to focus on the 64M/64R runs, as the others (with 16M/16R and 32M/32R) exhibit the same behavior. We start by noticing the same contention problem with the *drm* version, which also occurred in the *full-set* runs. During the first 80-90s of our *half-set* runs (i.e. the first 8 queries or so) the external workload's *Map* phase severely effects the performance of the *drm* version. In Figure 6.13 we see peaks at max CPU load. Moreover, the average minimum load per all cores (over the entire *Map* phase) is $\geq$ 80%. The impact is bigger than previously occurred in the experiments since now, TeraSort workloads ran at their full 64M/64R parallelism on just 4 of our workers. This means there were 16 Mappers/Reducers (= 16 cores) per worker participating in the resource contention; in our settings, one out of 16 Mapper/Reducer uses a single core at 70-80% load (Table 6.2). On the other hand, *drmy* is only a little affected by the overlapping workload (few peaks around 70% load) and that is happening on 1 out of 4 workers, on which we do not omit to *throttle* our parallelism and use just the amount of reserved resources – 8 threads for (#1) and 11 threads for (#2) out of 11 max available for Vectorwise. However, the *dynamic resource scheduling* algorithm determines a different assignment of workers, partition *responsibilities*, and threads per node (out of max available) at runtime for all 16, 32 and 64 M/R busyness tests (i.e. different *resource footprints*). Therefore, due to our experimental setup (see the beginning of this section), the Buffer Manager must read some "cold" partitions from disk. That aside, the algorithm involves *remote* responsibilities in query execution making the system use even remote reads (over TCP) for a part of the "cold" data. Overall, this issue makes the first two of our queries' to run really slow, the allocation overhead counts too within the first query's time. Around 70GB of compressed data (Lineitem and Orders tables are 76GB in total) have to be read until the buffer pool becomes hot again. We do not know any specifics, such as I/O or CPU load, for other worker nodes than the ones which overlapped with the external (Hadoop) workload, but looking at Table 6.9 and based on the previous two sections, we do not expect the I/O throughput to exceed 21-22 MB/s. Extrapolating the problem of "cold" buffer pools to a bigger set of queries (e.g. entire TPCH-H benchmark), this would make the resource contention issue fade whenever we deal with a smaller overlapping workload. However, if we look at the *drm+* ("hot" buffers and w/o YARN overhead) runtimes during the *Map* phase (i.e. first 8 queries or so) we could also conclude that for bigger workloads it actually makes sense for each of the two systems to allocate enough resources within the ecosystem they both run. As it seems so, this is a tradeoff between the HDFS I/O performance (for local/remote reads) and resource contention. Unfortunately though, for the latter problem there is nothing more someone can do than to avoid using the same resources. On the other hand, the earlier aspect can be improved, alongside with our *resource scheduling* policies and cost models. Some of the improvements are going to be discussed in our last chapter.

Conclusions

Contributions

In this Master's thesis we undertook the challenge of creating the groundwork towards enabling *dynamic resource management* in Vectorwise on Hadoop MPP database system. The milestones we tried to reach were deciding on the high-level system architecture and creating a *proof of concept* that demonstrates (1) improved data-locality, (2) better performance and reliability during node failures, as well as (3) the ability to cope with overlapping external (Hadoop) workloads, all *due to* our custom HDFS block replication policy, min-cost flow algorithms (plus cost-models) for dynamic resource scheduling, and YARN-integration for cluster resource management. Thereby, we consider that we achieved (to a certain extent) all our research goals from Section 1.5.

Answers to Our Research Questions

Scheduling and resource management are not only important, but also very challenging. Clusters are exposed to a wide range of applications, that have highly diverse characteristics and performance requirements. For example, Map-Reduce applications are usually characterized by long running jobs. These jobs desire shorter turnaround time - time from submission to completions. Queries in database systems for analytical or transactional workloads, on the other hand, are much shorter-lived and much more interactive. Further, these environments need to support a large number of users simultaneously. As a result, the underlying system needs to respond to these kind of workloads as soon as they arrive, even at the cost of stretching their overall turnaround time slightly. Managing both types of workloads at the same time makes it even more difficult for a cluster resource manager, especially when each workload belongs to a different applications (e.g. running Map-Reduce jobs and OLAP queries). Scheduling and resource management needs to take numerous system parameters, such as CPU speed, memory size, I/O bandwidth, network bandwidth, context switch overhead, etc, into account. These metrics are often inter-related and can affect the choice of an effective scheduling strategy.

Coming back to our initial research questions, we argue that with an easy, non-intrusive design and only by using the existing meanings, i.e. YARN and HDFS native APIs, X100's existing code base, etc., it is still possible to implement *dynamic resource management* into Vectorwise on Hadoop. Besides the current API big issues regarding YARN's container resource extension and its inability as a service to monitor in real time an extensive set of metrics, other than CPU and memory availability, perhaps our major limitation has to do with the latter problem of *effective scheduling.* Though the *min-cost flow* approach has proven to be a viable option in literature for resource management and scheduling purposes, the cost-models we use flattens the relationship between the numerous system parameters, which means that we normalize beforehand all our cost values to the same "nominator". This makes it even harder for us to define relationships between different metrics or prioritize some of them more than the others.

Future Work

Apart from the enhancement opportunities we identified and implemented during this project, we also envision and address the challenge of providing in the near future the following features and improvements to our master's thesis approach.

Perfecting the cost-models

Since both the *worker-set selection* (Section 4.1) and *responsibility assignment* (Section 4.2) are performed only once at database startup and work with (more ore less) *static system properties*, we cannot improve by a lot the cost-models for these algorithms. What we can rather do is to account for a few more YARN metrics in our formulas, such as disk speed, network bandwidth, etc., whenever they are made available in the API. However, besides the latter additions of new metrics, for the *dynamic resource scheduling* (Section 4.3) algorithm we propose the following improvements that could bring it to completion:

- When defining the penalty cost for *partial* responsibilities (i.e. not yet local partitions), have instead a value that corresponds to *how much* from that partition was replicated locally already; it could be a percentage value (e.g. 80% replicated) that is normalized correspondingly to our cost-model (e.g. cost of 0.8). Overall this would make the cost-model take smarter decisions.
- We could also take in account the historical information about what (partition) responsibilities were involved during query execution and introduce another layer of costs, e.g. *hot* responsibilities, as they *become "hot"* in the buffer. With this we can alter a bit our cost-model and *avoid hitting the disk* to overcome Hadoop's local I/O poor performance. However, it would be hard to know when partitions are evicted from the buffer pool, such that we change our costs from *hot* to *local* again. We do something similar for *responsibility assignment*, where we account in the cost-model for ex-responsibilities (labeled as *local+*, instead of *local-*) that could have been cached by the worker node's OS File System. But there, data caching is just a side-effect and no assumptions can be made in that sense. The File System's cache is not under our control, whereas the Buffer Manager is.
- Last but not least, is to try to achieve some data-locality for *remote* responsibilities when we must involve them in query execution. The idea is to differentiate the penalty costs of *remote* responsibilities that match locally with some worker's $[RMax, \ldots, R]$ interval of remaining replicas from those who do not (R replication factor, RMax responsibility degree). It will be transparent for the *resource scheduling algorithm* as *remote* responsibilities will still be considered as such. Nevertheless, in this way we can extend our view towards all R replicas without increasing the $RMax$ degree and overloading the worker nodes with an extra degree of responsibility.

Full resource elasticity within a Hadoop cluster

To achieve full resource elasticity within a Hadoop cluster we still need to revise our approach for a neat YARN integration and to perfect the internal Vectorwise workload monitoring, alongside with our resource management policies. We devise the work that has to be done in two directions, each one with its own roadmap of alternative ideas for improvements:

YARN integration

As stated in YARN-896 [1], there is significant work ongoing to allow long-lived services running as YARN applications. For example, the API's functionality to increase/decrease resource utilization within YARN containers (YARN-1197 [2]) is not yet implemented. This makes it impossible "per se" to gain full elasticity and to allocate/deallocate resources in a fine-grained fashion for Vectorwise. Though the idea to model *resource requests* as *separate YARN applications* (Section 5.1) achieves *elasticity* to a certain extent for *increasing* the amount of resources, we saw that is limited when it comes to the opposite operation, *decreasing* resources. We cannot just deallocate the difference of resources between different overlapping workloads by just using the available API interface. With the existing YARN release this functionality would be translated into *(1) stopping* the existent Application Master (for the first workload) and then *(2) starting* another one (for the second workload), which adds a significant overhead and is not a worthwhile solution to implement from a product-wise point of view. Therefore, our workaround for the moment is to release all requested resources at once, when the latest workload ends (*idle state* duration $>$ *threshold*) and there are no other queries still active in the system.

However, with YARN-1197 being pushed in a next release, we plan to revise our approach in the future. For instance, instead of creating one YARN Application Master per resource request (per workload), we could multiplex all the requests towards a *single YARN Application Master* (or a pool of applications distributed over different scheduling queues) that is delegated to Vectorwise and can dynamically increase/decrease the amount of resources for its own containers. Thereby we can avoid the application startup overhead entirely and reduce the request latency even more (i.e. a YARN Application Master can run in background and listen only for new requests).

A second idea could be to use Apache's Slider [3] platform, launched in beta-version around June 2014, which exposes a set of services that allow long-running applications, real-time and online applications to easily integrate into YARN (and be YARN-Ready [4]). Besides, it provides an interface for real-time communication, especially for database systems running on Hadoop or database-style workloads where a very high-speed transfer or data processing and response times are required. It complements Apache Tez [5], which is quickly gaining adoption as the batch and interactive engine of Hadoop. However, one thing to note, is that Slider, as well as the YARN 2.2 release, do not support resource extension (i.e. increase or decrease primitives) for containers yet. This means we would still need a work-around to achieve resource elasticity for Vectorwise on Hadoop, to allocate or deallocate resources when is required so.

Another alternative is to use the so called *Container Delegation* [6], a very new capability that is being added to YARN. Currently, the *Standard Container Model* looks as such: a YARN application (via the Application Master) negotiates a container from the Resource Manager and then launches the container by interacting with the Node Manager to utilize the allocated resources (CPU, memory etc.) on the specific machine. Now, YARN-1488 [7] is adding a new model called the *Delegated Container Model*. It helps to understand the distinction between a Service X that is already running in YARN containers or externally such as Vectorwise, and an application that needs to utilize this service, for example a Vectorwise query. In the *Delegated Container Model*, the query can negotiate containers from the Resource Manager (via the Application Master) and, rather than launching them to utilize the new allocated resources, it *delegates* all these resources to Service X instead, which then *gets additional resources* from

[1] YARN-896: issues.apache.org/jira/browse/YARN-896
[2] YARN-1197: issues.apache.org/jira/browse/YARN-1197
[3] Apache Slider: slider.incubator.apache.org
[4] YARN-Ready Program: hortonworks.com/press-releases/hortonworks-announces-yarn-ready-program
[5] Apache Tez: tez.apache.org
[6] Container Delegation: hortonworks.com/blog/evolving-apache-hadoop-yarn-provide-resource-workload-management-services/
[7] YARN-1488: issues.apache.org/jira/browse/YARN-1488

its Node Manager(s) to *be used on behalf of the query.* This scheme contrasts with the *Standard Container Model*, basically the only way of building a YARN application today, in which the application would launch and manage the containers itself to use the allocated resources. From an implementation perspective, the Node Manager(s), on delegation, will expand the target containers of Service X, by modifying them to allow for more CPU, memory etc. When the containers are no longer needed, Service X can release them back to YARN, so the resources can now be used by other applications.

Workload monitor and resource management

An alternative approach to our workload monitor (Section 5.3), which should contribute to our efforts on achieving full resource elasticity, is as follows. Through specific formulas, at regular intervals we would measure the internal load of the system. An element to the computation would be to establish whether queries get to a certain target of parallelism level – the current *maximum parallelism level*, or as well a new environment variable such as *target parallelism level* or *minimum parallelism level.* If queries on average get significantly less parallelism than desired (by the target), then the system will be considered *overloaded.* If they get significantly more, will be classified as *underloaded.* The workload monitor should respond to this information over time (i.e. every few minutes), trying via YARN to gradually increase the amount of allocated resources if the system is "overloaded", or yielding them otherwise (if "underloaded"). The described alternative for the workload monitor is better mostly in the sense that resource elasticity would be managed in the background and never influence the latency of an individual resource request. This is a drawback of our current Master's thesis approach that needs to be fixed. Also, our current approach may not be better as such in identifying all the *context* (see Section 5.1) changes that determine when to ask YARN for more resources.

Another feature that is not yet supported would be to modify our (custom) HDFS block replication for Vectorwise such that it tries to create R node sets $S_1, \ldots, S_R$ (R the replication degree, e.g. 3) with every S_i having each table partition exactly once. This would allow us to *put to sleep* different node sets. *Putting to sleep* implies that we maintain the X100 backend processes running on the worker nodes, but entirely avoid using these nodes for query execution, and as such, deregister them from YARN. This means we can still run our database on the $\frac{N}{R}$ workers (N number of nodes) that are awake, and use one core on each as the *minimal resource footprint.* Note that, assuming C cores per worker node, in the latter situation we are just using $\frac{1}{C*R}$ CPU resources. Nonetheless, we can scale all the way up to $N * C$ cores.

Acknowledgements

This thesis was made possible thanks to Actian Corporation, which hosted the project and provided the necessary development environment (with extra support from Centrum Wiskunde & Informatica), as part of a 8-months internship.

Second, I would like to thank my supervisor, Prof. Dr. Peter Boncz, for his high devotion for the project, day-by-day guidance and productive discussion sessions. I am also thankful to my second reader, Prof. Dr. Henri Bal, for contributing to the final shape of my thesis.

I worked in an open-minded and friendly environment, and I owe this great experience to the entire Vectorwise team in Amsterdam, especially to (in the alphabetical order of their last name) Andrei Costea, Adrian Ionescu, Bogdan Raducanu, and Michal Szafranski, who always took the time to answer my questions and help me to fulfill the project.

Last but not least, I would like to thank my family and friends for their everlasting support.

Cristi

Bibliography

[1] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* In Proceedings of the OSDI, 2016.

[2] Parquet columnar storage for Hadoop. blog.cloudera.com/blog/2013/03/introducing-parquet-columnar-storage-for-apache-hadoop

[3] ORC file format for Hadoop. hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance

[4] V. Kumar Vavilapalli et al. *Apache Hadoop YARN: Yet Another Resource Negotiator.* In Proceedings of the SoCC, 2013.

[5] S. Babu and H. Herodotou. *Massively Parallel Databases and MapReduce Systems.* Foundations and Trends in Databases, Vol. 5, 2012

[6] G. Graefe. *Volcano - an extensible and parallel query evaluation system.* In Proceedings of the IEEE Transactions on Knowledge and Data Engineer, 1994.

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

[8] P. Boncz and M. Kersten. *MIL primitives for querying a fragmented world.* In Proceedings of the VLDB Journal, 1999.

[9] P. Boncz, M. Zukowski, and N. Nes. *MonetDB/X100: Hyper-Pipelining Query Execution.* In Proceedings of CIDR, 2005.

[10] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. *Block Oriented Processing of Relational Database Operations in Modern Computer Architectures.* In Proceedings of the IEEE ICDE, 2001.

[11] M. Zukowski, P. A. Boncz, N. J. Nes, and S. Heman. *Monetdb/X100 - A DBMS In The CPU Cache.* IEEE Data Engineering Bulletin, 2005.

[12] Kenneth C. Sevcik. *Application scheduling and processor allocation in multiprogrammed parallel processing systems.* In Performance Evaluation, 19:107 140, 1994.

[13] E. Rahm. *Dynamic load balancing in parallel database systems.* 1995.

[14] H. Märtens, E. Rahm, and T. Stöhr. *Dynamic query scheduling in parallel data warehouses.* In Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Euro-Par, 2002.

[15] E. Rahm. *Parallel query processing in shared disk database systems.* 1993.

[16] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. *Hive - A Petabyte Scale Data Warehouse Using Hadoop.* In Proceedings of the IEEE ICDE, 2010.

[17] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. *Major Technical Advancements in Apache Hive.*, In Proceedings of the SIGMOD, 2014.

[18] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. *HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads.* In Proceedings of the VLDB Endowment, 2009.

[19] A. Abouzied, D. Abadi, and A. Silberschatz. *Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems.* In Proceedings of the EDBT/ICDT, 2013.

[20] Hawq whitepaper. *Pivotal HD: HAWQ. A true SQL engine for Hadoop.*

[21] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, J. Cohen, C. Welton, G, Sherry and M. Bhandarkar. *HAWQ: A Massively Parallel Processing SQL Engine in Hadoop.* In Proceedings of the SIGMOD, 2014.

[22] EMC Greenplum presentation, F. Was. *Beyond Conventional Data Warehousing.* Greenplum Inc.

[23] Impala distributed SQL query engine. blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real

[24] Inside Cloudera Impala: Runtime Code Generation. blog.cloudera.com/blog/2013/02/inside-cloudera-impala-runtime-code-generation/

[25] J. Sompolski, M. Zukowski, and P. Boncz. *Vectorization vs. compilation in query execution.* In Proceedings of the DaMoN, 2011.

[26] Distributed SQL query engine for big data. prestodb.io

[27] M. Traverso. *Presto: Interacting with petabytes of data at Facebook.* www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920

[28] B. Schroeder and M. Harchol-Balter. *Achieving class-based QOS for transactional workloads.* In Proceedings of the IEEE ICDE, 2006.

[29] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. *How to determine a good multi-programming level for external scheduling.* In Proceedings of the IEEE ICDE, 2006.

[30] B. Niu, P. Martin, W. Powley, P. Bird, and R. Horman. *Adapting mixed workloads to meet SLOs in autonomic DBMSs.* In Proceedings of the IEEE ICDE, 2007.

[31] U. Dayal, H. Kuno, J. L. Wiener, K. Wilkinson, A. Ganapathi, and S. Krompass. *Managing operational business intelligence workloads.* In proceedings of the SIGOPS, 2009.

[32] S. Krompass, U. Dayal, H. A. Kuno, and A. Kemper. *Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls.* In Proceedings of the VLDB Endowment, 2007.

[33] S. Krompass, Harumi A. Kuno, Janet L. Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Managing Long-running Queries. In Proceedings of the International Conference on Extending Database Technology, ACM, 2009.

[34] Hadoop's Capacity Scheduler. hadoop.apache.org/core/docs/current/capacity_scheduler.html

[35] M. Zaharia, *The Hadoop Fair Scheduler.* developer.yahoo.net/blogs/hadoop/FairSharePres.ppt

[36] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. *Improving MapReduce Performance in Heterogeneous Environments.* In Proceedings of the OSDI, 2008.

[37] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. *Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling.* In Proceedings of the EuroSys, 2010.

[38] B. Palanisamy, A. Singh, L. Liu, and B. Jain. *Purlieus: Locality-aware resource allocation for MapReduce in a cloud.* In Proceedings of the SC, 2011.

[39] M. Li, D. Subhraveti, Ali R. Butt, A. Khasymski, and P. Sarkar. *CAM: a topology aware minimum cost flow based resource manager for MapReduce applications in the cloud.* In Proceedings of the HPDC, 2012.

[40] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. *Quincy: fair scheduling for distributed computing clusters.* In Proceedings of the SOSP, 2009.

[41] G. Graefe. *Encapsulation of parallelism in the Volcano query processing system.* In Proceedings of the SIGMOD, 1990.

[42] E. Parsons and K. C. Sevcik. *Multiprocessor scheduling for high-variability service time distributions.* In Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, 1995.

[43] A. Costea, A. Ionescu, *Query Optimization and Execution in Vectorwise MPP.* MSc thesis, Actian Corp., 2012.

[44] S. Heman, M. Zukowski, N. J. Nes, L. Sidirourgos, P. Boncz. *Positional update handling in column stores.* In Proceedings of the SIGMOD, 2010.

[45] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms.*

[46] Introduction to Minimum Cost Flows, 15.082J, 6.855J and ESD.78J MIT Courses, October 26, 2010. ocw.mit.edu/courses

[47] Parquet presentation: Hadoop Summit 2013. www.slideshare.net/julienledem/parquet-hadoop-summit-2013

[48] Parquet's 1GB block size, insert [shuffle]. www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_parquet.html

[49] Topcoder. Minimum Cost Flow, Part 2: Algorithms. community.topcoder.com/tc?module=Static&d1=tutorials&d2=minimumCostFlow2

[50] B. Thirumala Rao, N. V. Sridevi, V. Krishna Reddy, L. S. S. Reddy. *Performance Issues of Heterogeneous Hadoop Clusters in Cloud Computing.* In Global Journal of Computer Science and Technology, May 2011.

[51] Set and Partitions: Covering, Hitting, and Splitting. www.nada.kth.se/~viggo/wwwcompendium/node142.html

[52] Weighted set covering problem. en.wikipedia.org/wiki/Set_cover_problem

[53] Konstantinos Vasileiadis. *A Proof that Hitting Set is NP-Complete.* 2013.

[54] Tien Duc Dinh. *Hadoop Performance Evaluation.* Research report (practicum stage), *Ruprecht-Karls* Universitat Heidelberg (Institute of Computer Science), 2009.