




**Institut für Software & Systems Engineering**
Universitätsstraße 6a D-86135 Augsburg

# Efficient relational main-memory query processing for Hadoop Parquet Nested Columnar storage with HyPer and Vectorwise

Sebastian Wöhrl

**Masterarbeit im Elitestudiengang Software Engineering**

SOFTWARE ENGINEERING
Elite Graduate Program




**Institut für Software & Systems Engineering**
Universitätsstraße 6a D-86135 Augsburg

# Efficient relational main-memory query processing for Hadoop Parquet Nested Columnar storage with HyPer and Vectorwise

| | |
|---|---|
| Matrikelnummer: | 1276280 |
| Beginn der Arbeit: | 05. September 2014 |
| Abgabe der Arbeit: | XX. September 2014 |
| Erstgutachter: | Prof. Alfons Kemper, Ph.D., Technische Universität München |
| Zweitgutachter: | Prof. Dr. Thomas Neumann, Technische Universität München |
| Betreuer: | Jan Finis, Technische Universität München |
| | Prof. Dr. Peter Boncz, Centrum Wiskunde & Informatica, Amsterdam |

SOFTWARE ENGINEERING
Elite Graduate Program

## ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den XX. 09. 2014 Sebastian Wöhrl

# Abstract

Hadoop is the de-facto standard for building clusters for big data processing. Data is stored in a myriad of formats, most not optimized for large-scale efficient processing. With the development of Parquet by Cloudera and Twitter the companies aim to introduce a new general-purpose storage format to unify database-like data storage in Hadoop clusters with a format designed with nested data structures and distributed parallel processing in mind. Previous approaches for interconnecting the worlds of Hadoop clusters and classic relational database systems (RDBMSs) focused on data exchange using bulk-copying techniques. Recent developments try to close the gap by deploying RDBMSs on Hadoop clusters (Actian Vector-on-Hadoop) or providing the Hadoop ecosystem with RDBMS features (Cloudera Impala). Our approach uses the reverse angle and extends RDBMSs to read and process data from Parquet files in Hadoop clusters directly and integrating them completely into the relational model. Furthermore we specifically deal with the nested data model of Parquet files and use it to optimize query execution. Our experiments show that for certain use cases our implementation outperforms Parquet implementations of Hadoop systems like Impala and Hive and can even be nearly on par with relational database systems. Our approach therefore provides an already efficiently usable first step in integrating the worlds of Hadoop and RDBMSs.

# Contents

**List of Figures** iii

**List of Tables** iii

**1 Introduction** 1
- 1.1 Motivation . . . 1
- 1.2 Contribution & Scope . . . 2
- 1.3 Related Work . . . 2
- 1.4 Outline of the Thesis . . . 4

**2 Parquet Format** 5
- 2.1 Hadoop Ecosystem . . . 5
- 2.2 Parquet Schema . . . 8
- 2.3 Parquet File Format . . . 9
- 2.4 Repetition and Definition Levels . . . 11
- 2.5 Storage Layout . . . 13
- 2.6 Discussion of the Parquet Format . . . 15
- 2.7 Reconstructing Nested Records . . . 17
- 2.8 Parquet in Impala and Hive . . . 17
- 2.9 Parquet and Cluster Processing . . . 18
- 2.10 Comparison of Storage Formats . . . 18

**3 Implementation Concepts** 20
- 3.1 Mapping a Parquet Schema to a Relational Schema . . . 20
- 3.2 Constructing Tuples . . . 21
- 3.3 Extending SQL Syntax with Nested Tables . . . 22
- 3.4 Vectorwise and Nested Tables . . . 24
- 3.5 Building Scan Operators . . . 25
- 3.6 Combining Joins and Scans . . . 26
- 3.7 Excursion: Supporting Nested Formats other than Parquet . . . 28
- 3.8 Different Meanings of NULL-Values . . . 28
- 3.9 Writing Parquet Files . . . 29

**4 Implementation Details** 31
- 4.1 Implementing a Parquet Library . . . 31

4.2 Implementing Scan Operators . . . 32
4.3 Vectorized Approach for Vectorwise . . . 33
4.4 Quick Count for Vectorwise . . . 33
4.5 Code Generation for Hyper . . . 34
4.6 Combining Joins and Scans in Hyper . . . 37
4.7 Efficient Bit Unpacking . . . 37
4.8 Dealing with Different String Representations . . . 42

**5 Evaluation** **44**
5.1 The Power of Compilers . . . 44
5.2 Vectorwise and Parquet . . . 45
5.3 Hyper and Parquet . . . 47
5.4 Profiling a Query for Hyper . . . 49
5.5 Optimizing Bit Unpacking . . . 51
5.6 Parquet Format: Space vs Speed . . . 51
5.7 Parquet Format: Using Compression . . . 52
5.8 Using Bigger Datasets . . . 54
5.9 Comparing with other Parquet Implementations . . . 56

**6 Conclusion** **58**
6.1 Scale-Out . . . 59
6.2 Future Work . . . 59

**Bibliography** **61**

# List of Figures


2.1 MapReduce . . . . . 7
2.2 Parquet File layout [17] . . . . . 10

3.1 Original execution plan . . . . . 27
3.2 Optimized execution plan . . . . . 27

4.1 Unpacking 2-bit values . . . . . 41
4.2 In-place string conversion . . . . . 43

5.1 Impact of compiler optimizations on query runtime . . . . . 45
5.2 Runtime comparison Parquet TPC-H on Vectorwise . . . . . 46
5.3 Runtime comparison Parquet TPC-H on Hyper . . . . . 48
5.4 Compressed Parquet TPC-H on Vectorwise with local reads . . . . . 53
5.5 Compressed Parquet TPC-H on Vectorwise with HDFS reads . . . . . 54
5.6 Parquet TPC-H on Hyper scale-factor 10 . . . . . 55
5.7 Parquet TPC-H on Vectorwise scale-factor 10 . . . . . 55


# List of Tables


5.1 Comparison TPC-H on Hyper Native vs Parquet . . . . . 48
5.2 Parquet TPC-H on Hyper with Vectorized Approach . . . . . 49
5.3 Runtime comparison for unpacking bit-packed values . . . . . 51
5.4 Parquet TPC-H on Hyper with Fixed bitwidth . . . . . 52
5.5 Parquet Filesize for different compression algorithms . . . . . 53
5.6 Comparison of different Parquet implementations . . . . . 56

# 1 Introduction

## 1.1 Motivation

Clusters are one of the prominent and current trends in data processing, often used in conjunction with the buzzword big data [1]. The dominant system for building clusters is Apache Hadoop [2] and the software from its ecosystem like HDFS[1] and MapReduce [3]. One such part is Impala by Cloudera[2]. It is a distributed SQL query engine for Hadoop. Its primary storage format is Parquet[3], a column-oriented storage format for nested data. As other big players in the Hadoop environment like Hive[4] also start to integrate Parquet as a storage format it is becoming more and more the primary format for storage of complex data in a distributed Hadoop environment.

More and more companies build clusters using the Hadoop ecosystem and its software stack for analytical data processing (OLAP)[5] but concurrently also still use traditional relational database systems (OLTP). But despite this both worlds remain mostly separated. The only connection often being a regularly scheduled batch-loading-process for copying data from the relational database systems to the Hadoop cluster.

The primary authors of Parquet, Cloudera and Twitter, intended it to be a general-purpose format for the entire Hadoop ecosystem[6]. That means with its growing adoption there is a possibility to finally bridge the gap and allow traditional database systems access to Hadoop cluster data on the query level without having to account for a myriad of data formats.

On the side of relational database systems one of the trending topics of current database research is the concept of main-memory databases. One such research project is Hyper [4], developed at the Database chair of the Technical University of Munich. Its features not only include in-memory processing but also a data-centric approach using code generation for query execution [5].

Another trending topic, although already quite old, are column-oriented databases. The pioneer in that area was MonetDB [6], still developed at the Centrum Wiskunde & Infor-

[1] Hadoop Distributed File System: `http://wiki.apache.org/hadoop/HDFS`
[2] `http://impala.io`
[3] `http://parquet.io`
[4] `http://hive.apache.org`
[5] `http://www.itproportal.com/2014/02/03/big-data-trends-hadoop-is-the-clear-frontrunner-for-enterprises/`
[6] `http://blog.cloudera.com/blog/2013/03/introducing-parquet-columnar-storage-for-apache-hadoop/`

matica (CWI) in Amsterdam[7]. Also invented there was its successor called Vectorwise [7], now developed by Actian under the name Actian Vector.

## 1.2 Contribution & Scope

As explained above Parquet is on its way to become the dominant data storage format for Hadoop.

To integrate the worlds of relational databases and Hadoop Parquet we focus on the concept of external tables already widely used to provide database users with a table like access to structured files (CSV) residing outside of the database. Following the same concept we integrate Parquet files by representing them as special tables that still behave like relational tables so that users can use existing SQL query capabilities to access these files.

One challenge lies in the fact that the data inside Parquet files has a nested structure whereas relational tables are by design flat. So we have to map both concepts to each other. Also as of the time of this writing there is no publicly available open-source library for accessing Parquet files usable from C++. Therefore during the course of this thesis we develop a library designed to access Parquet files in an efficient way and provide its data in a way suitable for processing by relational database systems.

As mentioned in Section 1.1 Hyper and Vectorwise represent two different approaches to developing modern relational database systems. As such we chose those two as candidates for integrating Parquet query capability and to demonstrate the different approaches needed for both systems to achieve efficient query execution.

As such the main challenge of this thesis is the integration of Parquet files into these database systems so that queries can be handled in an efficient way.

## 1.3 Related Work

In Section 1.1 we already mentioned the database systems Hyper and Vectorwise as the targets for this thesis.

Hyper is a main-memory relational database system developed as a research project at the Database chair of the Technical University of Munich [4]. One of its main features based on the main-memory approach is that it is suitable for OLAP and OLTP workloads. It achieves that by using the process-fork mechanism of the underlying operating system to separate both workloads. The main database process handles OLTP queries. Whenever a OLAP query is sent to be executed (and therefore requires a consistent snapshot / state of the database) a fork of the original process is created and the query is executed by the forked process. As the memory of the processes is duplicated the memory of the forked process remains in a consistent state even when the main process is changing data. But despite that the fork does not take up the double amount of memory as the underlying

[7] Part of this thesis was written during an internship at CWI

operating system uses a copy-on-write approach to only duplicate those memory blocks that are being changed in one of the processes. For all other blocks only one instance exists which is shared by all forks. This is done transparently to the processes by the operating system and therefore requires no special implementation for the database.

Another more interesting feature of Hyper is the query execution model. Traditionally relational database systems use the iterator model [8]. In this model operators provide an iterator-like access to the tuple stream it produces using its inputs via a *next*-Method. As this model means a number of (possibly quite expensive due to virtual methods) function calls per tuple. In lieu of the fact that database systems get more and more dominated by CPU consumption and less by I/O this model is quite inefficient. Therefore Hyper uses a new model [5] where the algebraic operator tree is used to generate code for query execution that is data-centric so that values can be kept in CPU registers as long as possible. For this the operator tree is split into pipelines with pipeline breakers being operators that have to materialize tuples before they can continue processing (e.g. sort or aggregation operators, but also joins for at least one input side). Then code is generated to push tuples through the pipelines therefore maximizing the time a tuple can stay in CPU registers. For code generation the LLVM framework [9] is used to produce code in the LLVM intermediate representation that is then optimized and compiled into native machine code.

The implementation for Hyper in this thesis is also partially based on "Instant loading for main memory databases" [10] whose implementation was used as a template for our own implementation.

Vectorwise is a database system originally developed as a new kernel for MonetDB under the codename X100 [11] at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. MonetDB [6] is the pioneer for column-oriented database systems which store values not grouped by tuples/rows but grouped by column. Therefore queries that only use a (small) subset of columns can run faster as only these needed columns need be read whereas if tuples are stored in row storage the entire tuple with all columns would always have to be read. Vectorwise [7] adds on that and extends the query execution model (based on the classic iterator-model) to not process single tuples but an entire vector of tuples at a time. Due to the column-oriented nature the tuples are provided with one vector per column. Vectorwise was bought by the company Actian (which also bought the Ingres database system and uses it as the fronted for Vectorwise) and is now developed and marketed as a commercial product under the name Actian Vector.

The latest development effort for Vectorwise is an integration into Hadoop. The project is developed under the codename Vortex and promoted as Actian Vector-on-Hadoop [12]. Vortex extends Vectorwise to run as a distributed system (with parallel distributed query processing) on the nodes of a Hadoop cluster and to use HDFS as storage backend for the database (but it still uses its own storage format). The goal is that companies only need to have one cluster that can run both Hadoop and a traditional relational database system.

An introduction to the Hadoop ecosystem and the systems relevant to this thesis can be found in Section 2.1.

## 1.4 Outline of the Thesis

**Chapter 1: Introduction** The motivation behind integrating Parquet with relational databases and the contributions of this thesis are explained. Furthermore, this chapter presents an overview of the thesis and of related work.

**Chapter 2: Parquet Format** This chapter contains a detailed explanation of the Parquet format, its features and its usage in the Hadoop ecosystem. It also discusses the practicability of specific features.

**Chapter 3: Implementation Concepts** This chapter focuses on the conceptual aspects for implementing a Parquet library and the integration into the relational database systems Hyper and Vectorwise.

**Chapter 4: Implementation Details** We take a closer look at some interesting aspects of the implementation and focus on ways to optimize for query execution speed.

**Chapter 5: Evaluation** In this chapter we show the results of benchmarks that were run during different stages of the development process and also present experiments that highlight specific features of the implementation.

**Chapter 6: Conclusion** A summary of this thesis is presented and open features and questions for future projects are listed.

# 2 Parquet Format

Parquet is a file format designed to store nested data based on a strict schema in a column-oriented way. It is based on Google Dremel [13] and used as the main storage backend in Impala. It also can be used as a storage backend for a number of other systems in the Hadoop environment, including Apache Hive.

Section 2.1 gives a quick overview of the Hadoop ecosystem in relation to Parquet. A description of the strict schema for the format is given in Section 2.2. The format and layout of the file itself is described in Section 2.3 followed by explanations of the important features of the format in Sections 2.4 and 2.5. This is complemented by a discussion of some of the design choices of the format in Section 2.6. Also in this chapter are a description of the record reconstruction algorithm from the Dremel paper in Section 2.7, a look into how Parquet is used in the Hadoop ecosystem in Section 2.8 and its usability for cluster processing in Section 2.9, and finally a short comparison of the storage formats of Parquet and Vectorwise in Section 2.10.

## 2.1 Hadoop Ecosystem

**Apache Hadoop** Apache Hadoop[1] is the umbrella project and a framework for distributed data processing. Its built on HDFS[2] (Hadoop Distributed File System) for storage and MapReduce [3] for processing.

**Apache Hive** Apache Hive[3] is a data warehouse built on top of Apache Hadoop. It uses HDFS as storage backend and MapReduce as the tool for query processing. For managing and querying datasets stored in HDFS Hive provides a query language called HiveQL which is designed to be SQL-like in terms of syntax and features. Queries formulated in HiveQL are translated into MapReduce jobs which are then executed on the Hadoop cluster.

**Impala** Impala[4] (developed by Cloudera) is a sql query engine built on top of Apache Hadoop. It utilizes the same infrastructure as a Hadoop cluster, specifically HDFS and HCatalog (see below) for parallel processing of queries. It uses HDFS and file formats from Hadoop as storage but unlike Hive it does not use MapReduce for query processing but

[1] `http://hadoop.apache.org`
[2] `http://wiki.apache.org/hadoop/HDFS`
[3] `http://hive.apache.org`
[4] `http://impala.io`

instead uses its own execution engine processes running on the cluster nodes (and using HDFS local access whenever possible). The main format for data storage used by Impala is Parquet.

### HCatalog

> Apache HCatalog is a table and storage management layer for Hadoop that enables users with different data processing tools – Apache Pig, Apache MapReduce, and Apache Hive – to more easily read and write data on the grid. [14]

To achieve this goal HCatalog acts as an abstraction between different storage formats stored in HDFS and processing tools. This abstraction is in the form of relational tables (with records and columns) which can be placed in databases. Each table can consist of multiple partitions which can be created over one or more keys (often done for a date column). For every file from HDFS registered in HCatalog a location and metadata are stored. This not only provides an abstraction on the storage formats used but also allows decoupling of data and processing (for example MapReduce jobs) definitions. The idea is that a processing tool (like Apache Hive) does not need to know the details for data files stored in HDFS (like path or storage format) but instead just queries the relational abstraction provided by HCatalog. The translation between the abstraction and the real storage format is done by format-specific drivers in HCatalog. There is also an extension point which allows developers to add drivers for new storage formats. That way processing scripts for Hive or Pig are independent of the real location or format of data and therefore do not need to be changed when the underlying storage format of the data is changed or if the files are just moved to another location. One more benefit is that with the necessary information stored in HCatalog different processing tools can access each others data. For example a table created in Impala can be read by Hive without the need to define anything as Impala stores the information about its tables in HCatalog.

**Apache Drill** Drill[5], an Apache project currently in the Incubator, is another SQL query engine for Hadoop. It is designed specifically for nested data formats and can also read schema-less formats like JSON. The architecture of Drill is based on massive parallel processing with low-latency queries and is intended to be used with existing Hadoop clusters. Just like Impala it does not rely on MapReduce for query execution, but can nonetheless still read data from Hive deployments. For testing and experimentation purposes it provides an embedded mode that can be run as a normal Java program without having to deploy it on a Hadoop cluster. In this mode Drill can also read files from local filesystems.

**ORC File** ORC (short for Optimized Record Columnar) [15] is a file format designed for and used by Apache Hive. An ORC file consists of stripes (also called rowgroups) and a file footer for auxiliary information (metadata). A stripe is comprised of a footer which

[5] `http://incubator.apache.org/drill/`

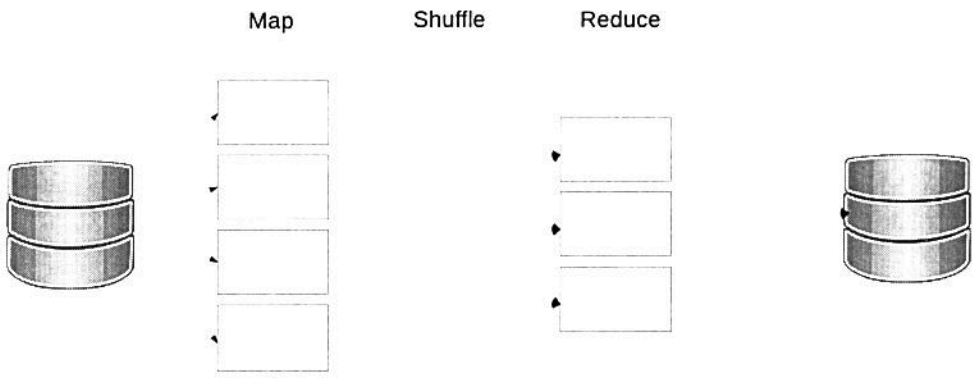


Figure 2.1: MapReduce

contains metadata, index data for all columns (min-max-values and row positions for each column) and the data itself organized by column. The data in the index allows for skipping rows not relevant to a query. By default each stripe has a size of 250MB and index data is written for every 10,000 values (so 10,000 rows can be skipped). In its design the format of ORC files is very similar to that of Parquet files and can be seen as sort of a predecessor.

**MapReduce** MapReduce [3] is the concept of splitting work into small parts so that it can be done in parallel. The work is split into two phases (see Figure 2.1): In the map-phase all nodes in the cluster working on the job process distinct parts of the input (usually files from HDFS) producing intermediate results (often in key-value form). These intermediate results are then shuffled and distributed between the nodes (all values with the same key end up on the same node) using the network and used as input for the reduce-phase when then produces the final result (normally written back to HDFS).

An often used example is the counting of words: Each node gets a part of the text and emits key-value pairs during the map-phase with the keys being the words and the values being the count. Then the pairs are shuffled so that the intermediate results for the same word from the different nodes are sent to one node. In the reduce-phase the nodes combine these pairs so that for every word only one key exists.

As mentioned above Hive does not have its own execution engine for queries but instead translates queries into a series of MapReduce jobs that produce the result of the query [16]. Hive is not implemented on the relational model although it translates queries into operator trees and then transforms them to be executed as MapReduce jobs. These jobs have an administration overhead and need additional time for distributing shuffle data over the network and for writing intermediate results to be used in other jobs to HDFS.

## 2.2 Parquet Schema

The schema for Parquet is based on the schema for Google Dremel described in [13]. Records consist of atomic and record types (called groups). Record types can recursively again consist of atomic and record types. All types have a multiplicity attribute which can have the values required, optional, repeated. The formal definition of the data model as given in the Dremel paper is as follows:

$$\tau = dom|\langle A_1 : \tau[*|?], ..., A_n : \tau[*|?]\rangle$$

Based on that and the informal description on the Parquet homepage we developed a formal grammar for the textual description of a Parquet schema in the Backus-Naur-Form:

```
<schema>     ::= 'message' '{' <fields> '}'
<fields>     ::= <field> | <field> <fields>
<field>      ::= <repetition> <def>
<def>        ::= <simple> | <group>
<simple>     ::= <type> <name> ';'
<group>      ::= 'group' <name> '{' <fields> '}'
<type>       ::= 'int32' | 'int64' | 'int96' | 'float' |
                 'double' | 'binary' | 'string' | 'boolean'
<repetition> ::= 'required' | 'optional' | 'repeated'
<name>       ::= [a-zA-Z_0-9]+
```

An example of a valid schema (comprising the region and nation part of TPC-H) is as follows:

```
message region {
    required int32 regionkey;
    required binary name;
    required binary comment;
    repeated group nation {
        required int32 nationkey;
        required binary name;
        required binary comment;
    }
}
```

The schema can be interpreted as a tree with groups/records as nodes and atomic fields as leaves. Since Parquet stores data organized by column every atomic field in the schema is a column in the file. In the graph sense every leave becomes a column. For that purpose the schema is flattened and the name of the column is the full path from the tree to the leave through the tree. So in the example above the following columns exist in the file:

- region.regionkey (type int32)

- region.name (type string)
- region.comment (type string)
- region.nation.nationkey (type int32)
- region.nation.name (type string)
- region.nation.comment (type string)

In C++-like class definitions this could be defined as

```
class Region {
    int regionkey;
    string name;
    string comment;
    vector<Nation> nations;
}

class Nation {
    int nationkey;
    string name;
    string comment;
}
```

Throughout this thesis record types will be called groups and atomic fields will be called fields (or columns). An object is a concrete instance of a nested structure conforming to a schema. For example a region object has regionkey, name and comment fields and an array or group of nation sub-objects.

## 2.3 Parquet File Format

A textual description of the Parquet file format can be found on the github-project parquet-format[6]. In the project there is also a picture depicting the file layout[7] (included as Figure 2.2).

For encoding metadata Parquet uses the compact binary protocol from Apache Thrift[8]. The Thrift definition files for the different metadata objects can be found in the parquet-format github project[9].

A Parquet file begins and ends with the 4-byte magic number "PAR1". The structure of the file is optimized for bulk writing, the metadata is placed at the end of the file followed

[6] http://github.com/Parquet/parquet-format/
[7] http://raw.githubusercontent.com/Parquet/parquet-format/master/doc/images/FileLayout.gif
[8] http://thrift.apache.org/
[9] http://github.com/Parquet/parquet-format/blob/master/src/thrift/parquet.thrift

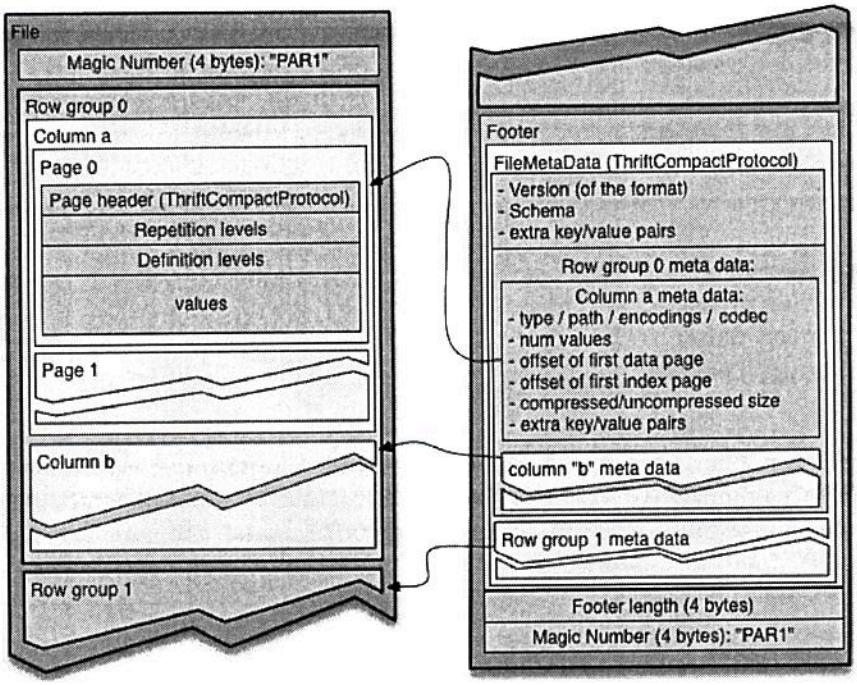


Figure 2.2: Parquet File layout [17]

by a 4-byte length field which specifies the length of the footer which is comprised of the file metadata. This metadata contains a compact representation of the schema, so every Parquet file is self-contained and can be read without the help of any external definitions.

The data storage of the file is organized into rowgroups which contain columns which contain pages. This hierarchy is represented in the metadata. Also they are stored in continuous memory. Interesting to know is that a Parquet file can stretch over several physical files as every column in a rowgroup can be stored in another file (the metadata for the column, called a ColumnChunk, has optional fields for file path and file offset). Every column contains at last one data page. A page contains a short header (also encoded using the Thrift compact protocol) which specifies the page size and the number of values. The value-part of each page can be compressed. Currently the algorithms GZIP, LZO and Snappy are specified.

## 2.4 Repetition and Definition Levels

Since the schema in Parquet is nested but the data is stored in a flat column-oriented way certain information needs to be encoded with the values to allow for reassembly of records (see also Section 2.7). This also comprises if a value is repeated or omitted (denoted by optional in schema). To achieve this the so-called repetition and definition levels were introduced into Parquet (also following the approach from the Dremel paper).

Every column has maximum repetition and definition levels (r- and d-levels for short) which can be calculated using the schema and a simple algorithm which traverses through the schema tree: Start at the root with both levels set to 0. Then go through the schema hierarchy:

- For an optional group/field d-level is d-level of parent plus one
- For a repeated group/field r-level is r-level of parent plus one and d-level is d-level of parent plus one

So in a flat schema (no nesting) with only required fields all r- and d-level values are always zero.

The complexity stems from the fact that values are stored in a flat manner column-wise but must be viewed as nested records with hierarchical data.

When storing a value it is annotated with its corresponding r- and d-levels. They are calculated as follows:

The process starts with the first object to be written at the top level and writes the fields from the root group with r-level 0 and d-level 0 (if the field is required or optional and null) or 1 (if the field is optional and not null). Then the first object of the next lower group is written, also with r-level 0 and corresponding d-levels. Then the first object of the next lower level and so on till the first object of the lowest level has been written with r-level 0. Then all other objects of the lowest level (belonging to the parent object) are written but with the maximum r-level of the corresponding group. After this the process goes back up

the chain to the next higher level and repeats the process. This goes on until all sub-objects of the top level object have been written (like a depth-first tree traversal). Then the process starts anew with the next top level object and r-level 0.

In Pseudocode (python-like):

```
for obj in messages:
    handleObj(obj, 0)

def handleObj(obj, r):
    for field in fields(obj):
        writeField(obj, field, r, d)
    for group in groups(obj):
        objs = list(obj, group)
        handleObj(obj[0], r)
        for o in obj[1:]:
            handleObj(o, r_level(group))
```

The first obj of a group is always written with the r-level of the parent, all the following are written with the defined maximum r-level of the group.

Because of the strict schema it is a necessity that each object always contains the full hierarchy. That means if in an object a sub-object, which is in the schema defined as optional or repeated, is not set a dummy object with all fields set to NULL must be placed instead, and for every group of that dummy object the same rule applies. When writing these objects to a Parquet file for all the NULL values the definition level of the (in the tree hierarchy) lowest parent group that is not NULL is used.

For example consider the following simplified schema:

```
message region {
    required int32 regionkey;
    repeated group nation {
        required int32 nationkey;
        repeated group customer {
            required int32 custkey;
        }
    }
}
```

If a region object contains no nations then a dummy nation object (with a nationkey set to NULL) and a dummy customer object (with custkey set to NULL) must be placed. When writing the values to a Parquet file a definition level of 0 will be used for the NULL values because the lowest defined parent is the region which has a definition level of 0.

Interesting to note is that even though nationkey and custkey are defined as required they are regardless still implicitly optional because they are in a group that is repeated.

## 2.5 Storage Layout

As mentioned a Parquet file is organized into rowgroups. Each rowgroup is independent from other rowgroups in a way that only complete top-level objects can be stored in them. This means that when a new rowgroup starts it starts with a new top-level object (r- and d-level are both 0) and can only end if all sub-objects of that object have been written. Following the TPC-H example all nations of a region must be contained in one rowgroup. There are no exact size requirements but sizes of 512MB or 1GB per rowgroup are recommended. This value should be aligned with the chunk size of the HDFS configuration.

Each rowgroup consists of columns which consist of pages. The metadata for the rowgroups and columns are stored in the global file metadata at the end of the file and contain file offsets for the beginning of each rowgroup/column. A column in a rowgroup is called a column chunk.

Each column consists of pages wich are written consecutively. Currently three types of pages exist: DataPages, DictionaryPages and IndexPages (not yet defined). Pages should be considered indivisible in terms of reading, for size about 8KB are recommended (to correspond to memory pages provided by the operating system) but again there are no exact size requirements.

### Pages

**DataPages** DataPages consist of a header (encoded using the Thrift compact protocol) which defines primarily the size of the page, the encoding used for the page and the number of values stored on the page. After the header follow the repetition and definition levels encoded using RLE-Encoding (see below). If either of the levels are by definition always 0 (for example a required field of a top-level object) then these levels are omitted entirely. After that follow the values which are stored back-to-back either plain or using an encoding, for example for integer values RLE-Encoding or for strings Dictionary-Encoding (also see below).

If compression is used only the definition and repetition levels and the data part of the page are compressed, the header is excluded as it is already stored in a compact manner. To read a page, first the header has to be read to find out how large the page is. Then using that size the rest of the page can be read and if necessary decompressed.

**DictionaryPages** DictionaryPages are used in conjunction with the Dictionary-Encoding for DataPages. A DictionaryPage consists of values stored as plain encoding. In the DataPages encoded using DictionaryEncoding only the entry ids referencing the values in the DictionaryPage are stored (using RLE-Encoding). There can only be one DictionaryPage per column chunk. Its position is referenced in the metadata for the chunk contained in the file metadata.

**IndexPages** As of time of this writing IndexPages are defined in the metadata but are still an empty concept. There exists a proposal on how to implement them (in a branch called "index" of the github-repository) which conceptually is like a B+-Tree: An IndexPage consists of a list of min-Value- and page-offset-pairs. The pages pointed to by the offsets can again be IndexPages (which would correspond to inner nodes in a B+-Tree) or DataPages (which would correspond to leaves in a B+-Tree). This index can only be built on one column and that has to be sorted. For all other columns the proposal recommends building indexes to allow finding a page for a given row-number. IndexPages are stored together with the DataPages and can be freely mixed (there is no given order). This allows writers to fill IndexPages as they are writing DataPages and write an IndexPage whenever it is full without having to buffer it.

## Encoding

For storing values space-efficiently the Parquet format defines and describes several encoding mechanisms. The most important ones are Plain, Dictionary Encoding and Run Length Encoding / Bit-Packing Hybrid (also called RLE-Encoding for short).

### Plain

This is the simplest encoding and is defined for all types. It just stores values back-to-back.

**Boolean** One bit per value (bit-packed, least significant bit first)

**Int32** Stored as 4 bytes little endian

**Int64** Stored as 8 bytes little endian

**Int96** Stores as 12 bytes little endian

**Float** Stored as 4 bytes IEEE little endian

**Double** Stored as 8 bytes IEEE little endian

**ByteArray** First the length of the array as 4 bytes little endian followed by the bytes in the array (so unlike with c-strings no termination character is used)

**Fixed len byte array** Just the bytes in the array (as the size is defined by the schema)

### Dictionary Encoding

See also the description of DictionaryPages above. The values are stored back-to-back (usually strings as byte-arrays) on the DictionaryPage. On the DataPages only the indices to the dictionary are stored (using Plain encoding).

### RLE Encoding

This is a combined hybrid of run-length-encoding and bit-packing. The grammar is defined in the parquet-format github-project[10]:

```
rle-bit-packed-hybrid: <length> <encoded-data>
length := length of the <encoded-data> in bytes \
    stored as 4 bytes little endian
encoded-data := <run>*
run := <bit-packed-run> | <rle-run>
bit-packed-run := <bit-packed-header> <bit-packed-values>
bit-packed-header := varint-encode(<bit-pack-count> << 1 | 1)
// we always bit-pack a multiple of 8 values at a time,
//   so we only store the number of values / 8
bit-pack-count := (number of values in this run) / 8
bit-packed-values := *see 1 below*
rle-run := <rle-header> <repeated-value>
rle-header := varint-encode( (number of times repeated) << 1)
repeated-value := value that is repeated, \
    using a fixed-width of round-up-to-next-byte(bit-width)
```

## 2.6 Discussion of the Parquet Format

During implementation of our library for accessing Parquet files we found out a lot about the usefulness of the Parquet file format design which we will discuss in the following.

One thing that is quite clear when contemplating the high level file design is that it was optimized for bulk writing in one go. The usage of rowgroups limits the number of values for the columns that need to be kept in memory. Because the storage is column-oriented first all values for one column need to be written before the values of the next column can be written which means when creating Parquet files using a non column-oriented input (like rows from a database or a CSV file) all the columns need to be buffered in memory. By separating the file into rowgroups only the values for one rowgroup need to be buffered. A way to avoid buffering values at all would be to write each column to its own file which is allowed by the format. But depending on the number of columns this could get confusing and also hinder read performance as reading many small files is less efficient than reading one big continuous file.

This leads to the already mentioned design decision of allowing a Parquet file to consist of several physical files. With all necessary information on how the files are connected stored in the metadata of the main file such a group of files is self-contained. Splitting them in such a way that every rowgroup is written to its own file allows for easy parallelization when processing a Parquet file by reading each rowgroup in parallel.

[10] http://github.com/Parquet/parquet-format/blob/master/Encodings.md

In contrast to this good high-level design, during implementation we found some aspects of the Parquet format which we think are not optimal. The first aspect is the fact that the size of a page is stored in the metadata of the page and the metadata is stored with the page itself which means that reading a page consists of several steps: The beginning of a page is known either because it is the first page and its file offset is part of the column metadata or because it follows directly after another page. The first part of the page is the encoded metadata (encoded using Thrift compact protocol) which is of unknown size. So first a guess has to be made of how big the pageheader containing the metadata is and that much has to be read from the file. Then it has to be decoded using Thrift which can fail if the guess was too low which would lead to another read (possibly in a loop) until the header can be successfully decoded. Then using the information in the metadata the size of the page is known and can be read. This means at least two read call are made to read a page which - depending on the operating system implementation of the file reading - will most likely be more expensive (in terms of wait time) than just one call for the entire space.

Currently the page metadata can optionally contain statistics which specify the minimum and the maximum value stored in the page (sort of like a min-max-index). But since these values are stored in the pageheader which (as explained above) is expensive to read, skipping a page is not very efficient as at least the pageheader has to be read and decoded.

Another aspect is that there can only be one DictionaryPage per column / rowgroup. And if the page is full then the system has to fall back to plain storage which means that the needed space increases dramatically. Assumptions: Average length of word: $l$, average repetition of word: $x$, size of page: $p$. For the length of the index number referencing the entry on the DictionaryPage we assume 2 bytes for simplicity. Then the increase in space of falling back to plain storage instead of adding a second DictionaryPage is defined by the following function:

$S(l, x) = (p/l + 1) * l * x - p/l + 1 * 2 * x - 2 * p$ (Space needed for storing one more value than can fit on one page minus space needed for storing index references for those values minus the space needed for two DictionaryPages)

Plotting this function shows that even when words are only repeated on average 5 times ($x = 5$) and have an average length of only 8 characters ($l = 8$) we already need four pages more for falling back to plain storage than we would need for adding another dictionary page.

Yet another aspect is the compression. The reason behind its use is the expectation that the overhead of decompression is smaller then the slowdown of reading data over a network connection. However our benchmark showed that when reading files from local disk the added decompression slows down queries rather then speeding them up (see Section 5.7). Thus assuming current fast networks approaching the bandwidth of local disks compression could prove to be more of a hassle than a benefit.

The last aspect is the design of the RLE encoding. As will be shown in section 4.7 this can be done quite efficiently for bitwidths that are divisors of 8 using SSE instructions. But for other bitwidths that span byte boundaries decoding them is rather slow. In our opinion it would have been better - especially for storing the repetition and definition levels - to

always use full bytes, our benchmark in section 5.6 shows that this gives a measurable speed improvement.

## 2.7 Reconstructing Nested Records

Since we want to process Parquet files using relational databases this means we need to be able to reassemble the records stored in the columns. For this the Dremel paper [13] describes an approach using finite state machines (FSMs) which allow reconstructing the records efficiently using only a subset of the columns (if not all columns are needed). Each field / column is represented by one state in the machine and corresponds to the reader for that field. The transitions between the states are labeled with the possible repetition levels of the originating column / state.

The transition algorithm is described in pseudo-code in detail in the paper. It can be described as follows for each state ( = field): The so-called barrier level for two fields is the repetition level of their lowest common ancestor. So for each field that comes before the current field and that has a larger repetition level then the current barrier level one transition is added between the current field and that previous field, the transition is labeled with the barrier level of those two fields.

Then for each repetition level between the barrier level (not included) and the maximum repetition level of the field the transition from the next lower level ($level - 1$) is copied. Finally for all repetition levels between zero and the barrier level (inclusive) transitions to the next field are added.

## 2.8 Parquet in Impala and Hive

Parquet can be used as a storage format in Impala[11] and is recommended for large queries or queries that only need a small subset of all tables. Impala can create Parquet files, load data into them and query them. But it only supports flat files, meaning complex or nested data types are not supported. For compression Impala currently supports Snappy and GZIP with a default for Snappy. When creating a Parquet file and loading data into it Impala will create a new file for every rowgroup (taking advantage of the fact that Parquet files can be split over several physical files) and aims for a filesize of 1 GB.

Hive also supports Parquet[12] for reading and writing and can also create Parquet files with nested structures. Parquet files created by Hive can be read using Impala as both use HCatalog to store meta information about tables. But for Impala this only works if the Parquet file is flat or else the file can not be accessed.

[11] `http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH5/latest/Impala/Installing-and-Using-Impala/ciiu_parquet.html`

[12] `http://cwiki.apache.org/confluence/display/Hive/Parquet`

## 2.9 Parquet and Cluster Processing

As Parquet is primarily intended for usage in the Hadoop environment it was designed for being used in clusters with parallel processing. In this section we want to highlight that.

The primary feature for parallelism are the rowgroups. Each rowgroup is independent of the others in terms that records never skip rowgroup boundaries.

Also each rowgroup can be stored in a different file. This ties in with MapReduce operations. Each node running a reduce operation writes its data to its own file as a separate rowgroup. Then after the job is completed the supervisor node creates the main Parquet file which just contains the file metadata with the pointers to the rowgroups in the different files. And when reading Parquet files the same mode for distributing work can be used by assigning each worker node one rowgroup to process.

This ability to split data across files also is beneficial to the often used batch mode for loading data. Many companies use Hadoop clusters as OLAP systems and traditional relational database systems for OLTP. Data from the OLTP systems are then batch-loaded into the Hadoop cluster on a regular basis (e.g. every night). When using Parquet files every batch run just produces one or more Parquet files with the new data that can then be tied together with the old data just by changing the master file with the file metadata and adding new rowgroups. If all the rowgroups itself are kept in separate files only the master file (which should be small as it only contains metadata) needs to be rewritten when new data is loaded.

Another feature intended for use in Hadoop clusters is the compression feature. Operating in a clustered way often means that data needs to be transferred between nodes using the network. In these use cases the slowdown from the additional CPU load for decompressing the DataPages can be compensated by the smaller files that need to be transferred over the network (see Section 5.7).

## 2.10 Comparison of Storage Formats

For storage of its data Vectorwise uses a column-oriented storage format based on PAX ([18], [7]). PAX stands for Partition Attributes Across and is different from traditional slotted disk pages (called N-ary Storage Mode or short NSM [19]). In NSM the values of a record are stored together in a sequential manner so that every data page contains a number of records. But this has the disadvantage that if only a subset of the columns are selected unnecessary data is read from disk into memory and possibly takes up cache space (especially CPU caches). The opposite of this is the Decomposition Storage Model (DSM, [20]) which stores values grouped by column and not by record therefore providing sequential reads for accessing the values of one column. PAX aims to be a combination of these two methods. It stores records on data pages like NSM but inside the pages stores the values column-wise like in DSM. According to [18] PAX executes TPC-H queries up to 40% faster and has far fewer cache-misses for main-memory workloads (up to 70%).

Vectorwise models its storage format based on PAX ([7]). Relations are stored in one or more PAX partitions which each contains a group of columns. The usage of the partitions can be configured and is normally auto-tuned and allows for all usage scenarios between all columns in one PAX partition (like original PAX) and every column in its own partition (like DSM) and everything between. Which one is used depends on the type and usage of columns and is normally self-tuned by the database system.

In contrast to this the Parquet storage format is more or less based on DSM. Values are organized in data pages but each page only contains values of one column. The concept of rowgroups is remotely similar to the PAX and NSM concept of storing records in pages (and not distributed over pages) but does so on a quite larger scale with rowgroups being in the scale of 512MB while data pages normally are in the range of 10-60KB.

# 3 Implementation Concepts

In this chapter we describe the design and its evolution behind our implementation efforts to integrate Parquet files into the relational database systems Hyper and Vectorwise.

For integrating Parquet files into our relational database systems we decided on an approach similar to the feature of external tables. Parquet files are represented by special tables which behave like normal relational tables and therefore can be accessed using normal sql queries. Since we wanted to do this integration for both Hyper and Vectorwise we decided to establish a separate Parquet library that implements all the Parquet specifics and provides abstracted interfaces for the database systems to use. Aside from that the implementation in the database systems consisted of mainly two parts: Extending the internal data structures of the database systems with special Parquet tables and providing implementations of scan operators for query execution to use instead of the normal operators for reading tables. The scan implementations would then use the Parquet library to read Parquet files.

Section 3.1 describes how we mapped Parquet schemas to relational schemas, and Section 3.2 how to construct tuples fitting that mapping. We describe the extensions for SQL syntax we used in Hyper (Section 3.3) and in Vectorwise (Section 3.4) for interacting with nested Parquet files while Section 3.5 gives an insight on how we implemented the scan operators. This approach is improved on in Section 3.6 by taking advantage of the nested structure of Parquet files. The rest of the chapter focuses on special topics: How we support other nested formats in Hyper (Section 3.7), the different meanings NULL values can have in the nested Parquet context (Section 3.8) and finally we describe how we created our own Parquet files (Section 3.9).

## 3.1 Mapping a Parquet Schema to a Relational Schema

Standard-compliant SQL syntax provides no concept of nested structures or datatypes. Tables can only have columns with simple datatypes like integers or strings. So we had to find a way to represent a Parquet file with its nested schema in relational table / column terms. To that end we decided to logically split the Parquet schema into groups each represented by their own relational table. To keep the information on the hierarchical structure we included special columns to represent the relationships between the groups as foreign key constraints with id and foreign key columns. These columns could then in queries be used to join the tables together and reproduce the hierarchy. These joins along the hierarchy could then in a later state be detected by the database system and merged

with the different scans into just one combined operator to directly produce all the columns from the different levels.

For that we devised a simple mapping algorithm to generate relational tables out of a Parquet schema. The algorithm follows the tree structure of the schema. Every inner node ( = group) of the tree becomes a relational table (its name is the concatenated names of all nodes from the root to this node). Every atomic field of that group becomes a column in the table. A required field is defined as "not null", an optional field as nullable. Additionally each table gets an artificial primary key column (which serves as a simple object counter) and a foreign key column to the table of the parent group (unless the group is top-level). Both columns will be filled using the repetition and definition levels. If an atomic field is designated as repeated then this field becomes its own table with just the value column and the artificial primary and foreign key columns. So conceptually a repeated atomic field is treated as a repeated group with just one required field.

As an optimization groups that are designated as required or optional can be merged into its parent group as the relationship between parent and child would always be 1:1 or 1:0.

## 3.2 Constructing Tuples

To implement the scan operators we needed a way to read a column subset from the Parquet file and reassemble those into tuples that can be processed by the database system. In section 2.7 we described a way of reconstructing nested records using finite state machines as it is used by Google. For the purpose of this thesis this algorithm could not be directly used. To emit records suitable for processing in a relational database system the possibly nested records need to be transformed into flat tuples. Conceptually for every nesting level a foreign key join has to be done to create flat tuples. Therefore we devised a simple algorithm that - using a subset of all columns in the Parquet file - emits flat tuples. The only requirement is that all selected columns need to be on one path in the schema tree. That means columns on the same nesting level (their groups have the same parent) need to be all in the same group otherwise when producing tuples a cross-product of these columns would have to be created. Conceptually this is like a 1:N Sorted-Merge-Join.

As an example consider the following schema:

```
message example {
    repeated group g1 {
        required int32 field1;
    }
    repeated group g2 {
        required int32 field2;
    }
}
```

If both field 1 and field2 are selected then the result would need to encompass a cross-product of all values of field1 with all values of field2 as these fields are in different groups

on the same level.

The algorithm keeps track of a global repetition level which is initialized with 0. To construct a new tuple the algorithm goes through all selected columns and gets the current repetition level for that column. If the repetition level is greater or equal to the current global level the next value for the column is retrieved and used for the tuple, otherwise the old buffered value is reused. The new global repetition level is calculated by retrieving the repetition level for the next value in every column and setting it to the maximum encountered level. In pseudocode:

```
def next():
    tmp_level = 0
    for col in columns:
        if col.r >= glob_level:
            putValue(col)
            advance(col)
        tmp_level = max(tmp_level, col.next_r)
    glob_level = tmp_level
```

The background is quite simple: As long as values in a column are repeated (which means their repetition level is set to the maximum repetition level of the column) only advance these columns, the others stay the same. Once the next value in that column belongs to another object its repetition level is smaller than the maximum level and denotes on which nesting level a new record is started (with 0 meaning a new top level record starts). So the values of that nesting level and all that are (in the tree view) below or on that level need to be advanced. And since the repetition level of the first value (in a repeated column) is set to the repetition level of the level on which a new record is started using the algorithm described above all these columns will be advanced.

## 3.3 Extending SQL Syntax with Nested Tables

In the beginning of the chapter we talked about representing Parquet files as special tables that still behave like normal relational tables. We also decided to represent each group of the Parquet schema as its own table. But at the same time we wanted to present the user with a simple interface. As Hyper allows for easy extension of its syntax we decided to introduce the concept of nested tables. For Vectorwise this approach was unfortunately not possible (see Section 3.4).

We extended the internal data structures of Hyper with a new database schema element called nested table and provided a new statement to create them:

```
create nested table <name> from parquetschema <filename> [with (<options>)]
```

This statement registers a new nested table in the database. The provided file can either be a text file containing a textual representation of a Parquet schema following the grammar

defined in section 2.2 or a Parquet file, in that case the schema definition is extracted from the file metadata. The Parquet schema is then attached to the nested table. Additionally options can be provided to the statement that specify whether virtual columns for foreign key relationships should be provided. Also optionally a Parquet file can be specified that will be taken as standard input for queries. Otherwise a Parquet file can be specified with each query. This provides the flexibility of reading different Parquet files with the same schema without having to change the nested table.

The next extension point was to enable the user to specify a Parquet file for each query. We implemented two possible ways to do that. One is to specify the file directly as a table reference in the query:

```
select * from (parquetfile <filename> with schema <nestedtable>)
```

Instead of just providing a table name in this case the user provides a pair of a reference to a nested table and a file name pointing to a Parquet file.

The other way was inspired by the with-clause:

```
with <name> as (parquetfile <filename> with schema <nestedtable>)
    select * from <name>
```

This way specifying the file name and the query itself can be separated and the file can be referenced several times without having to provide the full specification each time.

If a Parquet file was already specified during the creation of the nested table then both of these ways are not necessary. The name of the nested table can just be used as normal table reference and the database will automatically use the pre-provided file:

```
create nested table foo from parquetschema <schemafile>
    with (parquetfile <parquetfile>);

select * from foo
```

All this syntax still does not address the problem of representing and using nested schemas in queries. To solve this we also extended the SQL syntax with what we call nested table names or references: Using the dot-syntax (as used in many object-oriented languages to access potentially nested attributes) the user can specify a nested name as a table reference:

```
create nested table customer from parquetschema "tpch.schema"
    with (parquetfile "tpch.parquet");

select c_name, o_orderkey, l_quantity
from customer, customer.orders, customer.orders.lineitem
where ...
```

The starting point of the nested name has to be a valid reference to a nested table. The rest of the name is used as a path through the Parquet schema to define a group inside the

schema, with each dot denoting a new level in the hierarchy of the schema. This way the user can use the nested table like a normal table but using the nested name syntax still reference the nested structure of Parquet.

This syntax extension of course also works with the above explained ways to specify a Parquet file directly inside the query:

```
create nested table customer from parquetschema "tpch.schema";

select o_orderkey from
    (parquetfile "tpch.parquet" with schema customer).orders;

with customer as (parquetfile "tpch.parquet" with schema customer)
    select o_orderkey from customer.orders;
```

Last but not least we also wanted to provide a way to copy the data from a Parquet file into relational tables (like the *copy into*-statement in many database systems). But we still wanted the user to have the benefits of using nested tables names. Therefore we provided the *create nested table*-statement with an option to specify a storage mode. For this storage mode the user can specify any of the storage modes implemented in Hyper for relational tables (like column or row). When the statement is executed the database internally creates a set of relational tables that represent the nested schema of the provided Parquet schema. Then when the user issues a query containing a nested table (possibly with a nested table reference) the query will be rewritten to use the corresponding relational table instead. That way a user can get the query speed of Hyper but still use it like a nested table.

To fill the created relational tables with data we implemented a *copy*-statement:

```
copy <name> from parquetfile <filename>
```

An example usage could be:

```
create nested table foo from parquetschema "tpch.schema"
    with (storage "column");
copy foo from parquetfile "tpch.parquet";

select * from foo.orders;
```

## 3.4 Vectorwise and Nested Tables

Originally we wanted to implement the same syntax extensions and features for Vectorwise. But this was not possible. First of all Vectorwise is a commercial product so introducing new features requires more effort, planning and organisation than was possible during the course of this thesis. Also a show-stopper was that Vectorwise itself is only a backend while the frontend is provided by Ingres which is not developed by the Vectorwise team in Amsterdam

but by the Ingres team in the USA. Introducing the propsed syntax changes would have meant extending the frontend and the backend. The necessary coordination would have delayed the thesis so we decided to forego the syntax extensions and accompanying features and provide just the necessary features using a workaround.

The workaround requires the user to manually create all the relational tables with their columns. Then the user can use a syscall to register these tables to the Vectorwise backend as aliases for a Parquet file. A syscall in this context is a way for the user to send commands using a special statement in the Ingres frontend directly to the Vectorwise backend without the frontend having to be aware of the command. So we extended the internal data structures of Vectorwise so that relational tables can be registered as aliases for a part of a Parquet file schema. Then we implemented a syscall to give the user the ability to create such an alias:

```
CALL VECTORWISE(PARQUET_REGISTER '''<name> <parquetfile> <schemaroot>''');
```

In the statement *name* must refer to an already created relational table, *schemaroot* denotes a path through the nested schema of the Parquet file mentioned by *parquetfile* (or nothing if it references the top-level group of the schema). An example would be (assuming the Parquet file has a schema of customer - orders - lineitem):

```
create table customer (...);
create table orders (...);
create table lineitem (...);

call vectorwise(parquet_register '''customer tpch.parquet''');
call vectorwise(parquet_register '''lineitem tpch.parquet orders''');
call vectorwise(parquet_register '''lineitem tpch.parquet orders.lineitem''');
```

## 3.5 Building Scan Operators

With the introduction of nested tables and syntax extensions for them we provided the administrative half of the Parquet integration. For the other half we had to implement scan operators so that queries could be run on the Parquet files via the nested tables.

### Hyper

Hyper query execution is based on the produce-consume model [5] and utilizes code generation to compile queries into LLVM assembly code for efficient execution. The code generation is data-centric (see Section 1.3).

This means the scan operator needs to be implemented in such a way that it produces one tuple at a time to be consumed by the overlying operators in the execution tree. As such we also implemented the Parquet library to provide a simple next-iterator interface to retrieve tuples and wrote the scan operator to use that interface.

### Vectorwise

In contrast to Hyper Vectorwise does not use code generation but instead utilizes vectors. As such it does not operate on single tuples but on vectors of tuples. And as Vectorwise is also column-oriented the operators need to produce a vector of values for every column. So for Vectorwise we implemented the Parquet scan operator in such a way that it would request tuples from the Parquet library in a loop to fill result vectors.

## 3.6 Combining Joins and Scans

With the approach above alone we would succeed in allowing queries to Parquet files from relational databases. But as already mentioned one of the interesting aspects of Parquet is the nested structure of data. This can be used as an advantage when dealing with queries which contain joins following the hierarchy of the Parquet file.

Joining tables that represent the nested structure of a Parquet file along that hierarchy path means basically reassembling the nested objects (or at least parts of them). The algorithm for constructing tuples described in Section 3.2 is already designed to deal with fields from different groups of the hierarchy and assembles them in accordance to the nested structure. Therefore on a conceptual level reading different groups of the hierarchy using several scan operators and joining them together is equivalent to reading all columns in one single scan operator.

So the idea is to modify query execution plans in a way that joins (following the hierarchical structure) with Parquet scans (belonging to the same file) as inputs are combined into a single scan operator producing all requested columns and therefore avoiding (possibly costly) join operations.

A simple example: The query

```
select sum(l_quantity)
from customer, orders, lineitem
where c_custkey = o_custkey and o_orderkey = l_orderkey
      and c_custkey % 5 = 0
```

would produce an execution plan like in Figure 3.1. Assuming all three tables are nested tables from a Parquet file the optimizer could merge this plan into just one scan with a selection on top (see Figure 3.2). Thereby eliminating two HashJoins.

### Parquet Natural is Relational Outer Join

Taking into account the nested structure and the explicit storing of null values for not-existing subgroups in a Parquet file, a Parquet scan following the hierarchy structure is conceptually a one-sided outer join (e.g. on a join of customer and orders it would be a left outer join). This means that the different join types (inner, outer, semi, anti) need to be handled specially:

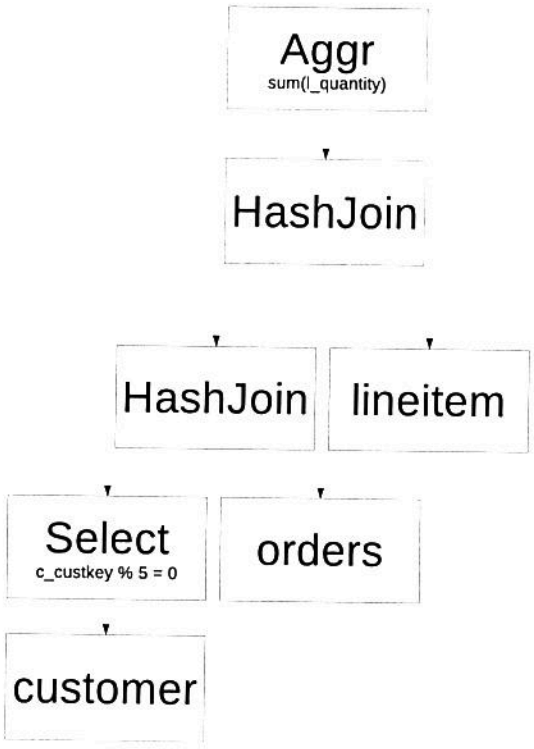


Figure 3.1: Original execution plan


Figure 3.2: Optimized execution plan

**Left / Right outer join** Assuming the outer side of the join is a parent of the other side in the nested schema this is by default done by the Parquet scan operator. The reversed roles (outer side is a child in the nested schema) could not be done but would make no sense because considering the hierarchical structure it would produce the same results as an inner join. The same goes for a full outer join which would behave just like a one-sided outer join.

**Inner join** The outer join can be made to behave like an inner join by adding a selection on top to filter out tuples will null values (meaning missing subgroups) on the child side.

**Semi join** This is similar to the inner join but the other side is just ignored in the ongoing processing. Special considerations need to be taken in the implementation so that at least one column from the side that is ignored higher up is nonetheless still produced by the scan to ensure the join behaviour.

**Anti join** An anti join can not be done by the Parquet scan without additional implementation and therefore this join should be handled by the existing implementation in Hyper.

## 3.7 Excursion: Supporting Nested Formats other than Parquet

As a side-experiment we also wanted to provide Hyper with the capability to read other nested formats such as JSON. So we made the syntax extensions described above more generic and gave the user the possibility to specify JSON files instead of Parquet files in the queries. The restriction was that the JSON file had to follow a strict schema and the schema definition had to be provided using the Parquet schema definition grammar.

We also implemented classes to read JSON files using the same interface as for Parquet files. That way we only had one implementation of the scan operator relying only on the generic interface that would use one or the other implementation from the library depending on the file type provided for the query.

```
create nested table foobar from parquetschema "somejson.schema";

select * from (jsonfile "data.json" with schema foobar);

with foo as (jsonfile "data.json" with schema foobar)
    select * from foo.nested.name;
```

## 3.8 Different Meanings of NULL-Values

Fields in Parquet can be optional, if the value is not set then a special NULL value is stored (using the d-levels). The same applies if an optional group is not set, then NULL values

are stored for each child field (and the sub-groups) to signal that the group is not there. This creates a problem when converting data like TPC-H, which is not nested, into Parquet format. Following the TPC-H example and a schema of region-nation-customer-orders it can be that there are customers that have no orders. In relational databases this fact is not explicitly stored but merely implicitly by the fact that the orders table has no rows with a foreign key referencing that customer. In contrast to that in Parquet this fact is explicitly stored by storing an order object with all fields set to NULL. When reading Parquet files from a relational database as in our implementation by treating each nesting level as its own table this poses a problem because when reading for example only the orders all the null objects denoting non-existing orders will be read as well. This can be dealt with by two possible solutions: The first is to just accept that these rows of NULL-values exist and deal with them on the level of queries. This also means declaring all column types for relational tables representing Parquet files as nullable. The alternative is to filter out these values during the scan. With our basic implementation this is quite easy as during reading of a tuple all r- and d-levels are read anyway so we just have to handle the special case that the values for all columns read are NULL and in that case skip to the next row. In our vectorized implementation this is no longer that easy as the columns are read independently of each other so a check for this special case would need to be done on a higher level and after filling the vectors which would counteract the speedup provided by vectorized reading.

## 3.9 Writing Parquet Files

During the writing of this thesis we needed a way to create Parquet files to use as test files for our implementation and as input files for our benchmarks. As we wanted and needed to create Parquet files with nested schemas, using Impala was not possible. The other option was to use Hive. But after some tries we had to give up as Hive had problems in writing data in complex nested structures with several levels. Hive provides two aggregate operator functions called *collect_list* and *collect_set*[1] intended to create an array from input values, which both were not able to handle complex structures. Also Hive used a higher abstraction level which lacked the necessary flexibility to create specifically-constructed test files. So we set out to create our own tool to write Parquet files.

The tool was designed to take a Parquet schema definition and input data in a supported format and generate a Parquet file from this. As input formats we support JSON, XML and CSV files. Implementing support for JSON and XML was easy as both formats are already nested. The tool requires the files to follow the Parquet schema definition in terms of names, nesting and types (including required/optional/repeated definitions). Additional fields/nodes/elements in the input files are possible and are just ignored so it is possible to only use a subset of the input data.

On startup the tool creates page buffers and vectors for the repetition and definition levels for every column in the schema. Then the input file is parsed using external libraries

[1] `http://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF`

(rapidjson[2] and rapidxml[3]). Both are small C++ header-only-libraries which claim to be very fast. The interface for both libraries was quite easy to use and for rapidjson also partly mimicked the C++ Standard library (STL) interfaces for vectors and maps in respect to iterators. After parsing the input files the resulting nested structures are stepped through recursively following the structure of the Parquet schema definition. Encountered values are written to the corresponding page buffers. Whenever a page reaches a defined maximum size (we mostly used 8 KB as this is the page size suggested from the Parquet authors) it is cached and new values are written to a new buffer. Whenever a new top level record is written (for example in the TPC-H case a new region) beforehand the current size of the rowgroup (the accumulated size of all finished and current page buffers) is evaluated. If it reaches another defined maximum size (we used 512 MB) all page buffers are written to disk as a rowgroup and the buffers are emptied. This process can only happen when a new top-level record starts because by definition rowgroups must be self-contained and therefore child records of a record can not be stored in different rowgroups.

As we wanted to create Parquet files using TPC-H schema and data we also added support for CSV files. As these are inherently flat we needed to find a way to create nested data from flat files (for example a structure of orders and lineitems from the csv files for both tables). For our solution we designed a two-step process. In addition to the Parquet schema definition our tool is provided with a mapping of CSV columns to Parquet schema fields and also the columns containing primary and foreign keys are marked (in the TPC-H example o_orderkey as a primary key in the orders table and l_orderkey as the corresponding foreign key in the lineitem table). Also the CSV input files need to be sorted based on foreign key. In the first step our tool reads all input files and generates a list of foreign keys and their corresponding file offsets (the position in the file where the first record with this foreign key starts). As the files are sorted by foreign key all records with the same foreign key are stored continuously. These offsets are written to disk to allow reuse. In the second step the tool iterates through all top-level records. For every record it finds the corresponding child records using the offset table created in the first step. This process is recursively repeated for every nesting level. So that a record and all its child records (following full recursion) are written before the next record is read.

Our tool also supports compressing pages using any of the defined compression algorithms. Also to allow us to create Parquet files with different characteristics (pagesize and compression algorithm) for our benchmarks and experiments these properties can be changed using commandline arguments.

[2] `http://code.google.com/p/rapidjson/`
[3] `http://rapidxml.sourceforge.net/`

# 4 Implementation Details

In this chapter we concentrate on several detail aspects of the implementation and its process. Section 4.1 deals with implementation of the Parquet library itself whereas in Section 4.2 we describe our first approach to implementing scan operators and its limitations. Then in Sections 4.3 and 4.4 we focus on improving the implementation for Vectorwise and in Sections 4.5 and 4.6 for Hyper. Finally Sections 4.7 and 4.8 deal with optimizing certain aspects of the Parquet library.

## 4.1 Implementing a Parquet Library

At the beginning of the implementation phase for this thesis there did not exist any open-source library[1] for accessing Parquet files in C++. Most of the code for Hadoop software concerning Parquet was written in Java which is no viable alternative considering Hyper and Vectorwise are implemented in C(++) and one of the main aspects of this thesis is speed. The only existing code in C++ was the Parquet implementation for Impala. But it was not really usable as it was tightly integrated into Impala and also only supported flat schemas. Therefore we decided to implement our own library in C++ which would then be used by Hyper and Vectorwise.

The structure tries to follow basic object-oriented principles. The main class is *ParquetFile* which provides the interface for interacting with a Parquet file and takes care of opening and reading the file on the memory level. Possibly reading the file from HDFS is encapsulated into the class so that other parts of the library do not need to care about the physical location of the file. *ParquetFile* represents a file as a simple memory region.

The rest of the class structure follows the structure of the Parquet file format. The classes *ParquetRowgroup*, *ParquetColumn* and *ParquetDataPage* each deal with a rowgroup, a column in a rowgroup or a page in a column respectively. These classes mostly just hold context and delegate any operations to the appropriate classes. The different encodings used to store data each have their own implementation class which share a common interface and are used by the *ParquetDataPage*-objects to actually read values from the pages.

All these classes only provide access to values in columns. To provide an interface that produces entire tuples we introduced the *ParquetTupleReader*-class. It encapsulates all the logic necessary to combine the different columns from the nested structure into flat tuples

[1] During the course of this thesis devlopment of a library was started by the Parquet project: `http://github.com/Parquet/parquet-cpp`

that can be processed by database systems. In the first implementation this class only had a simple interface modelled after next-iterators. Its intended usage was like this:

```
while (reader.next()) {
    for (col : columns) {
        val = reader.getValue(col);
        // read val
    }
    // process tuple
}
```

In addition to the columns from the Parquet file the *ParquetTupleReader* also provided virtual columns, namely an id column and foreign key columns for every parent in the schema hierarchy. So for example in the nested schema of customer - orders - lineitem when selecting columns from lineitem the *ParquetTupleReader* can provide an id for each tuple (like a row number) and foreign key columns referencing the ids of orders and customer. This way it is possible to formulate joins between the hierarchy levels in queries without having to explicitly include any intermediate levels.

## 4.2 Implementing Scan Operators

Based on the Parquet library we implemented scan operators for Hyper and Vectorwise.

**Hyper** The implementation for the scan operator itself is straightforward and based on the implementation of the CSV scan operator from [10]: It uses the *ParquetTupleReader* to retrieve one tuple and then calls a generated function that contains all the consumer code. Therefore the runtime part of the operator code is very small as the logic for reading Parquet files is encapsulated in the library.

**Vectorwise** The implementation for Vectorwise is similar but instead of just producing one tuple it calls the *next*-Method repeatedly to fill the result vectors that are provided by the overlying operator.

For Vectorwise there was one more workaround to implement. As explained in section 3.4 the concept of nested tables could not be introduced due to the fact that the frontend for Vectorwise is provided by Ingres. As such queries on tables that have a Parquet alias defined are still seen as normal relational tables by the Ingres frontend and also by the Vectorwise backend. To change this we extended the query plan rewriter with a rule for these Parquet tables. The rule matches every scan operator in the query execution tree (called *MScan*), checks if a Parquet alias is defined for the table and then replaces the *MScan* with our own scan operator which we called *PScan* (for ParquetScan).

## 4.3 Vectorized Approach for Vectorwise

The first implementation for Vectorwise was suboptimal as it broke the vectorized approach that was responsible for its high speed. Therefore we extended the implementation of the library to directly produce vectors of values. This included the entire class hierarchy, especially the decoder classes and the *ParquetDataPage* which were extended to read an arbitrary number of values and fill them into a provided vector (which ultimately would come from the Vectorwise runtime).

One restriction applied: The requested columns had to form a flat subschema, meaning that all columns had to be in the same group and not have the repetition type of REPEATED. This was because with the vectorized approach it was not convenient to apply the logic required for columns from different groups as it would again break the vector model. But we managed to work around that restriction by abandoning the idea of performing joins inside the scan operator and instead use existing join operators:

Vectorwise has for foreign key joins a special operator called MergeJoin1 which uses a clustered index on the foreign key column (which must be sorted) to conceptually do a Sort-Merge-Join with the sort phase done during data insertion. To do this efficiently Vectorwise creates a special join-index column for the table with the foreign key (for the join orders - lineitem this would be the lineitem table) which just specifies the row number of the parent record. As the table is sorted based on that key (as is the parent table) these row numbers are also implicitly sorted. With this Vectorwise can implement a very efficient MergeJoin by adding the row number column (called TID) and the join-index column to the selected columns (they are later in the query execution tree filtered out and are only used for the join).

To achieve this we extended the vectorized approach to also provide the virtual id and foreign key columns that the first implementation provided. Then we extended the scan operator to recognise the special columns that would be requested by the Mergejoin operator (row number and join-index column) from the scan operator and rewrite these to use the provided virtual id and foreign key columns. As the id column is generated anew for every query and is relative to the tuples emitted in the current query it corresponds with the row number of the emitted tuples. The same is true for the generated foreign key.

## 4.4 Quick Count for Vectorwise

During implementation of the scan operator for Vectorwise we had to handle one special case. When a query contains a count aggregation for the entire table without any predicates to limit the result set (*SELECT count(*) FROM table*) a normal scan is issued for a special virtual column. But in this case the overlying operators ignore the content of the resulting vector and only use the returned number of values for further processing.

The first implementation of the Parquet scan changed the requested column set so that instead of the virtual column (which obviously did not exist in the Parquet file) one of the

existing columns from the schema (from the group represented by the relational table) with the REQUIRED option would be selected. But this was rather inefficient as all the values were read and produced into the vectors but never used by the overlying operators.

So we extended our *ParquetTupleReader*-class with a special *count*-Method. This method uses the information available in the metadata of the Parquet file to quickly produce the number of values for a column without reading or processing any actual data. For a column that is a required element from the top-level group of the schema the number of values can be taken from the number of records from the file metadata. For any other column the rowgroup-column metadata needs to be read. It provides the number of values for a column in the rowgroup. These numbers just need to be added up. Since the metadata for the rowgroups is also stored in the global file metadata no actual data pages need to be read to get this information which makes it very fast as the metadata is already read at the beginning when opening the file.

One drawback to this implementation is that for columns that have a repetition type of OPTIONAL or REPEATED or are inside a group with one of these types the number of values provided also contains null values stemming from the nested structure which (for example when comparing normal TPC-H with our nested TPC-H Parquet variant) can produce deviating results.

## 4.5 Code Generation for Hyper

As previously mentioned one of the main aspects of Hyper is the fact that for every query specific code is generated that produces the query result. For the first implementation of the Parquet scan operator in Hyper this was not really used as the generated LLVM code just called the *ParquetTupleReader* to produce the next tuple. The logic inside the class had to be generic to accommodate all possible combinations of requested columns. Also the logic only produced one tuple at a time and also only read one value from each of the data columns making it rather inefficient. For Vectorwise we made the decision to use the already used MergeJoin to provide efficient joins between the Parquet tables (see Section 4.3). That meant that the logic for producing tuples was rather simple as with the joins handled externally the selected schema was flat and therefore handling of the hierarchy was not necessary.

For Hyper we wanted a scan operator that had the ability to handle columns from different groups of the schema. The requirements for the second implementation try were that the logic-heavy part should be done by generated LLVM code adapted to the current query. Also values from the data pages should be read in groups (vectors).

Therefore we first extended the *ParquetTupleReader*-implementation with the ability to read all the values and the repetition and definition levels from a column in a rowgroup into a vector (*readColumn*). For that the already implemented extension for Vectorwise to produce vectors could be reused and simplified so that it not only read a specific number of values but all the values from all the pages in the column. The method returns three

vectors, one with the actual values and the other two with the corresponding repetition and definition levels.

The second part was to write code that would generate LLVM code that reimplemented the tuple-generating logic tailored to the specific query. This is heavily influenced by and based on [5].

Conceptually the code generated looks like this (as C++ pseudocode):

```
for (rg=0; rg < numRowgroups; rg++) {
    // for every column
        reader.readColumn(colindex, rg, values, rlevels, dlevels);
    cur_r_level = 0;
    new_r_level = 0;
    while (at least one column has a value left) {
        // one block for every column
        if (*rlevels >= cur_r_level) {
            if (*dlevels == maxDForColumn) {
                // produce value
                values++; // advance vector
            } else {
                // produce null value
            }
            rlevels++; dlevels++; // advance level-vectors
        }
        if (*rlevels > new_r_level) new_r_level = *rlevels;
        // ... repeat for the other blocks

        cur_r_level = new_r_level; // set for next iteration

        // code for consumer operators
    }
}
```

This design ensures that the C++-code is only called once per column/rowgroup (to read the values from the column) and the logic to produce the tuples is encoded entirely in LLVM. Also as much information as possible can be included as static information (using constants) into the generated code instead of using method calls each time (e.g. the number of rowgroups or the maximum definition level for each column).

The code to produce a value is specific to the underlying datatype but in general for every column that is to be produced a struct with the following fields is created and filled for every value:

```
struct Slot {
```

```
    uint8_t* buffer;
    uint32_t length;
    bool null;
};
```

For fixed-size datatypes (integers, floats) the length is omitted as it is constant. The *buffer*-field is either a pointer to the underlying value or for strings a pointer to a character array.

Optimizations to the code can be done if a column is in the top-level group and is required. Then there can be no null values therefore the checking of the definition level can be omitted as can the null indicator field in the slot. If all columns are in the same group (flat) and all have a repetition type of REQUIRED or OPTIONAL then the entire logic of comparing the repetition levels of the columns can be omitted, greatly simplifying the logic.

In addition to producing values from real columns we also wanted the operator to have the ability to produce virtual id and foreign key columns. In the first implementation these columns were intended to serve as join columns to use for foreign key joins. Although these were no longer necessary as the joins would be handled by the Parquet scan internally (see next subsection) we still included the functionality in the second implementation to accommodate cases which the combined scan-join-operator could not handle. The functional implementation is very easy. For the id column it is just a simple implementation of an incrementing row counter. For the foreign key columns the implementation is also quite easy, it is represented by the following C++-code:

```
if (fk_r_level >= cur_r_level) {
    fk++;
    // produce fk value
}
```

The logic behind it is as follows: *fk_r_level* is a constant representing the maximum repetition level of the parent, and *cur_r_level* indicates on which level new values are produced (taken from the main implementation). So only for these levels of the hierarchy for which new values are produced can the foreign key be incremented.

An example: Assuming a nested schema of customer - orders - lineitem. Then the respective maximum repetition levels are 0 for customer, 1 for orders and 2 for lineitem. Now if the scan currently (from an abstract join perspective) loops through the orders of one customer, the current repetition level transitions between 1 (new order) and 2 (new lineitem for an order). This means the foreign key for the customer stays the same (repetition level 0) and the foreign key for the order only increases when a new order is read (which means a repetition level of 1) whereas when only a new lineitem is read the current repetition level is at 2, so neither of the foreign keys are incremented.

## 4.6 Combining Joins and Scans in Hyper

As mentioned above with the second implementation we also wanted the ability to combine joins between Parquet scans that follow the hierarchical structure of the schema into one unified Parquet scan. We explained the concept in general in Section 3.6. The consideration being that in many cases for joins Hyper would generate code to do a HashJoin which would be rather slow. Instead a join along the schema hierarchy could be done inside the operator with just minimal overhead of some additional If-statements and repetition level tracking (see explanations above). With the implementation utilizing code generation the operator itself was already prepared to handle joins but the database system still needed to be told to combine joins and scans into one scan. This was implemented into the query tree optimization phase.

Hyper has four different optimization phases which are run consecutively. These phases are named Unnesting, PredicatePushdown, Reordering and ChoosePhysical. Unnesting tries to get rid of (dependent) subqueries by merging it into its outer query or possibly extracting it completely into some sort of view. PredicatePushdown tries to push down selection predicates as far down as possible to reduce the number of tuples having to be processed by operator higher up in the execution tree. Reordering means to reorder the order of join so that simple (in the sense of number of tuples) joins are done first to where possible avoid joining big tables together. ChoosePhysical finally tries to select the best join method (e.g. Hash- or IndexJoin).

For our purpose we extended the ChoosePhysical optimization pass for the join operator as at this time cross products have been transformed into joins. One drawback however is that at that point the join reordering could have destroyed joins following the Parquet hierarchy.

The implementation itself is as follows: All the join operators are walked through from bottom to top and each operator is analysed. We iterate downwards both sides of the operator. If both sides end with a ParquetScan and have only Selections in between then the two scans and the join are combined into one ParquetScan. The operator tree is then rearranged so that the selections from both sides of the join are put above the new combined scan. This process is repeated for every join. Due to the bottom-to-top-approach the join tree is recursively merged.

## 4.7 Efficient Bit Unpacking

For storing the repetition and definition levels Parquet uses the RLE-encoding which - as explained earlier - is a hybrid of bit packing and run-length encoding that can be combined to achieve very space-efficient storage of vectors of small numbers. Using the tool callgrind of the valgrind toolsuite[2] we found out that when using the rowgroup-vectorized approach for Hyper between 80% and 90% of the time needed to read the values of a column in a

[2] `http://valgrind.org`

rowgroup and produce the vectors to be processed by the Hyper scan operator was spent reading the repetition and definition levels which were stored using bit packing. So we used this as an area of possible optimization.

The code was already written in such a way that each possible bitwidth (for this implementation we only implemented bitwidths up to 8 as we only used RLE-Encoding for the repetition and definition levels) had its own implementation instead of one generic implementation for all widths. This was coupled with the design decision to read and process all values at once and provide an array of bytes (*uint8_t*) instead of reading one value at a time on request. Together the implementation should in theory allow for efficient processing due to minimal branching (only one big while-loop, no ifs).

As an example the implementation for the two bit width was as follows (buffer is a pointer to the raw bit packed data, valptr is a preallocated array to store the resulting byte values, count contains the number of encoded values):

```
while (count > 0) {
    uint8_t val = *buffer;
    *valptr = val & 0b00000011;
    ++valptr;
    *valptr = (val >> 2) & 0b00000011;
    ++valptr;
    *valptr = (val >> 4) & 0b00000011;
    ++valptr;
    *valptr = val >> 6;
    ++valptr;
    count -= 4;
    ++buffer;
}
```

But as it turns out, although the code looked efficient, gcc (version 4.9) could not exploit pipelining and parallel processing because of a data dependency on valptr. Rewriting the code using index access for valptr gave a significant speedup (see Section 5.5) as it removed the data dependency and therefore allows modern CPUs to execute the statements using execution pipelines in a semi-parallel mode:

```
while (count > 0) {
    uint8_t val = *buffer;
    valptr[0] = val & 0b00000011;
    valptr[1] = (val >> 2) & 0b00000011;
    valptr[2] = (val >> 4) & 0b00000011;
    valptr[3] = val >> 6;
    valptr += 4;
    count -= 4;
    ++buffer;
```

```
}
```

Using instructions from the SSE extensions (SIMD, Single Instruction Multiple Data) of modern x86-CPUs we were able to gain an additional speedup. We will illustrate it using the implementations for the one and two bit widths. The SSE instructions were called using intrinsics[3] provided by the compiler.

**1 bit** In this implementation we handle two bytes at a time (amounting to 16 values) stored in an 128bit register. First these two bytes are shuffled so that the first byte is duplicated and stored in the first 8 bytes of the register and the second byte in the other 8 bytes. Then the bitwise AND of the register and a special mask is computed. We call this mask the exponential mask. In the first byte of the mask only the first bit is set. In the second byte only the second bit is set and so on. For the second 8 bytes it starts again with the first bit. As a result in every byte of the result register only the bit we are interested in is (possibly) set. Then the *cmpeq* function is applied to the result register using the same mask. As a result the bit we are interested in is duplicated to every bit of its byte. As a last step another bitwise AND is applied using a mask we call one mask because only the first bit of every byte is set. Thus using only 4 instructions (+ 2 instructions for loading/storing the input and the result) we have extracted 16 values.

The resulting code looks like this:

```
__m128i EXP_MASK = _mm_setr_epi8(0x1, 0x2, 0x4, 0x8,
    0x10, 0x20, 0x40, 0x80, 0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80);
__m128i SCATTER_MASK = _mm_setr_epi8(0, 0, 0, 0, 0, 0, 0, 0,
    1, 1, 1, 1, 1, 1, 1, 1);
__m128i ONE_MASK = _mm_set_epi8(1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1);

while (count >= 16) {
    __m128i rega = _mm_insert_epi16(__m128i(), *buffer, 0);
    __m128i res = _mm_shuffle_epi8(rega, SCATTER_MASK);
    __m128i res2 = _mm_and_si128(res, EXP_MASK);
    __m128i res3 = _mm_cmpeq_epi8(res2, EXP_MASK);
    __m128i res4 = _mm_and_si128(res3, ONE_MASK);
    _mm_storeu_si128(reinterpret_cast<__m128i*>(valptr), res4);
    count -= 16;
    buffer+= 2;
    valptr+= 16;
}
```

[3] for C and C++ by including the header x86intrin.h

**2 bit** For the 2 bit implementation we read 4 bytes at a time which also produce 16 values. These bytes are also shuffled so that the first byte is copied to the first 4 bytes of the 128bit register, the second byte to the second 4 bytes and so on (we refer to the result register of this operation as A). Then the following operations are done (EM refers to a mask similar to the first exponential mask but this time with two bits set per byte):

$$\begin{aligned} RES = ((A \quad)\&\&(EM \quad))\| \\ ((A << 2)\&\&(EM << 2))\| \\ ((A << 4)\&\&(EM << 4))\| \\ ((A << 6)\&\&(EM << 6)) \end{aligned}$$

The result is then right-shifted by 6 and another bitwise AND is applied using a mask where for every byte only the first two bits are set.

**Other bit widths** For a bitwidth of 4 the same algorithm as for 2 bits can be used. It works on 8 bytes and needs fewer steps (EM is again similar to the other exponential mask except 4 bits are now set per byte, 15MASK is a mask with the 4 first bit set of every byte):

$$(((A\&\&EM)\|((A << 4)\&\&(EM << 4))) >> 4)\&\ 15MASK$$

For a bitwidth of 8 no action needs to be taken as the values already have the right bitwidth (We assume that repetition and definition levels are small and therefore can be stored in one byte).

The bitwidths that are more complicated are the others (3, 5, 6 and 7) because in these cases result values are comprised of bits from different bytes which means reassembly is more difficult and can not be easily done using SIMD instructions.

### Runtime estimations

In the following we want to estimate the possible speedup of using SSE instructions over normal instructions for the one and two bit widths. We do this by estimating the number of clock cycles needed per value. For simplicity we ignore overhead of control structures (while-loops) and assume normal instructions (like load, store, shift, logical and/or) each take one cycle. For the SSE instructions we refer to the Intel Intrinsics Guide[4].

**1 bit** Without SSE instructions unpacking of one bit values is implemented as follows:

```
while (count > 0) {
    uint8_t val = *buffer;
    for (uint i=0; i < 8; ++i) {
        valptr[i] = (val & (1 << i)) >> i;
```

[4] http://software.intel.com/sites/landingpage/IntrinsicsGuide/

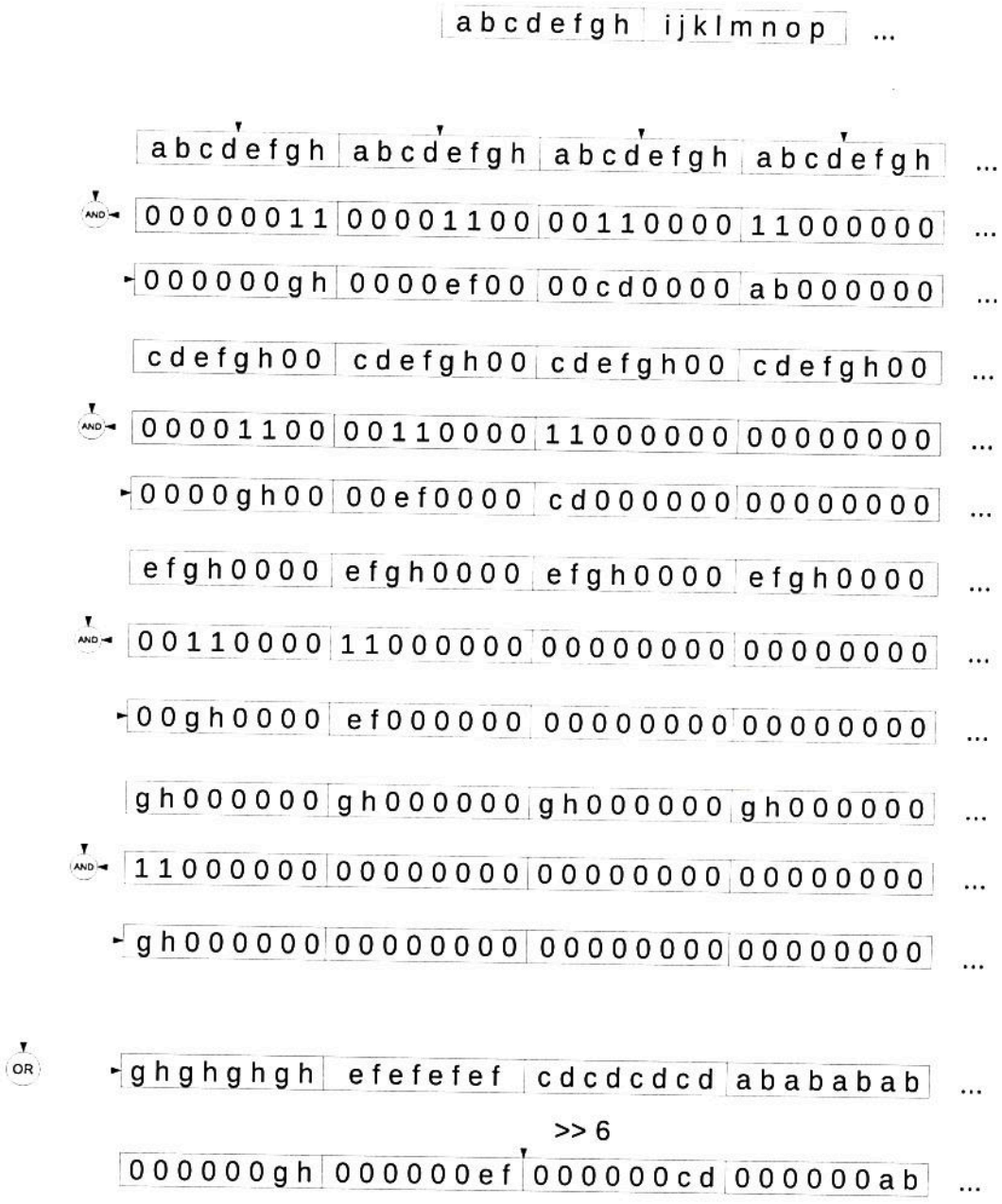


Figure 4.1: Unpacking 2-bit values

```
    }
    valptr += 8;
    count -= 8;
    buffer++;
}
```

We assume the inner loop is unrolled by the compiler and the left shift ($1 << i$) is calculated at compile time. Then for each bit we have the following operations: load, and, right shift, store which amounts to 4 cycles per value. For the SSE version we have one load and one store (each two cycles), two AND operations, one shuffle and one compare instruction which each cost one cycle. This sums up to 8 cycles per 16 values which means less than one cycle per value so in theory using SSE instructions should speed up operations by a factor of about 8.

**2 bit** Without SSE instructions the implementation takes around 11 cycles for four values with three ands, three shifts, one load and 4 stores. With SSE instructions we have around 17 cycles for 16 values. So in this case SSE instructions should speed up the process by a factor of two to three.

## 4.8 Dealing with Different String Representations

In the Parquet format strings are represented as a two-tuple of length and a byte array. If the values are stored in plain encoding than the values are just stored back-to-back on a DataPage.

For Hyper this is convenient because on the level of LLVM assembly code for executing a query strings are represented as a char pointer and a length field. So neither the library nor the scan operator need to do conversion but can just provide pointers into the memory of the ParquetDataPage. This is of course under the assumption the the provided strings are treated as read-only.

Vectorwise on the other hand uses normal C-style character arrays with a zero-termination character. So some sort of conversion needs to take place when providing string values from a Parquet file to Vectorwise operators. The naive approach would be to allocate a new character array for every string that has a length of one greater than the string length, copy the data to the new memory and put a termination character at the end. But this would be very inefficient as it means many small memory allocation and copy operations. Therefore we decided on a more efficient approach that takes the storage structure of the strings into consideration. As the values are stored back-to-back it means that after the byte array for one string follows the length field for the next string. This means that after reading the length field of the following string we can just replace the leftmost byte of the field with the termination character of the first string (see Figure 4.2). This way no memory allocation and copying is necessary, all the work is done in-place. But it has one caveat: After the page has first been read it can not be read again in subsequent queries / scans without rereading

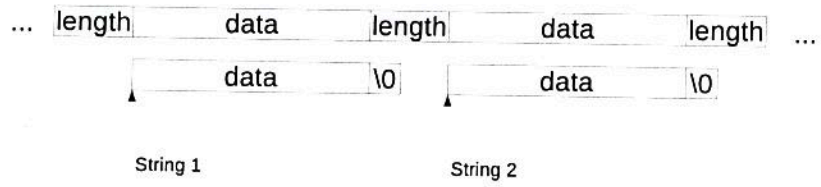


Figure 4.2: In-place string conversion

the memory of the page from disk because the internal structure is damaged. This can be circumvented if the page keeps an intermediate structure that just stores pointers to all the strings. With this subsequent reads would be even faster as all the work of conversion is already done. And if the pointers are stored in the same way that they have to be written to the result vectors than a subsequent read can just copy the pointers as a block in one operation.

# 5 Evaluation

Using our implementation for reading Parquet files we did a series of benchmarks using both Vectorwise and HyPer. As the basis for our benchmarks we used the TPC-H benchmark. As data basis we used the region, nation, customer, orders and lineitem tables from a scale-factor 1 version (also referred to as 1 G version beause of the filesize). In most of the benchmarks we compared query runtime between Native (using normal relational tables) and Parquet (using Parquet files). For this we created a Parquet file using our tool *csv2parquet* with a nested schema of region-nation-customer-orders-lineitem and for the native version we loaded the csv files of the tables into the database using normal *copy into*-statements.

If not noted otherwise runs were done on machines of the Scilens cluster[1] with a Intel Xeon E5-2650 v2 CPU[2] and 256 GB of RAM (DDR3-1866). Every query was executed 10 times in a row and the minimum of the runtimes was used as result (best out of 10).

This chapter is divided as follows: Section 5.1 shows the speedup of compiler optimizations, Sections 5.2 and 5.3 show the evolution in terms of runtime speed during the different stages of the implementation. In Section 5.4 we look closer at the distribution of runtime inside the query execution for Hyper. The following Sections 5.6 to 5.7 focus on some special optimization aspects of the Parquet format and library. To finish off Section 5.8 shows how both Vectorwise and Hyper deal with bigger datasets and Section 5.9 does runtime comparisons with other systems from the Hadoop ecosystem.

## 5.1 The Power of Compilers

Early on in the development and integration for Vectorwise and Hyper we did a trial run using variants of the TPC-H queries 1 and 6 with an unfinished version of the Parquet library to get a feeling for behaviour and runtime. For that first run we had compiled the library with gcc (Version 4.8) and no compiler optimizations enabled (*-O0*). For Vectorwise the runtimes were about 19.5 and 10.5 seconds, for Hyper they were about 13 seconds for both queries (see Figure 5.1).

Then we did another try but this time the library was compiled with all compiler optimizations enabled (*-O3*). This decreased runtime significantly: For Vectorwise the runtime went down to about 8 and 4 seconds, for Hyper down to about 3.5 seconds.

[1] A cluster for scientific computing run by the CWI in Amsterdam: `http://www.scilens.org`

[2] `http://ark.intel.com/products/75269/`

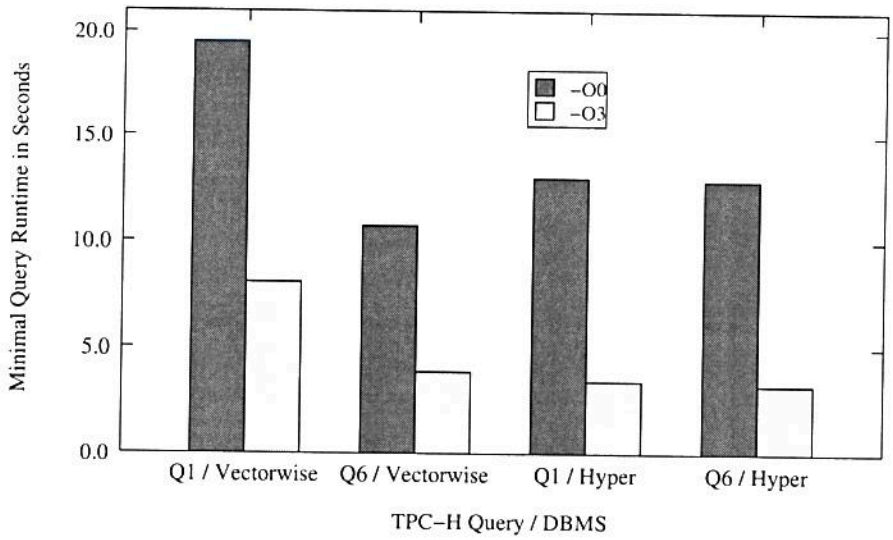


Figure 5.1: Impact of compiler optimizations on query runtime

This meant a decrease in runtime between 59% (Vectorwise query 1) and 75% (Hyper query 6) with no code changes, only by enabling compiler optimizations. This shows how effective modern compilers can be when optimizing code.

## 5.2 Vectorwise and Parquet

As of writing of this thesis Vectorwise still holds the performance record for TPC-H for 100 GB and 300 GB non-clustered database sizes[3]. In our benchmarks we deliberately disabled some of the performance features of Vectorwise when running against native relational tables as the Parquet version did not have those features and we wanted to do a fair comparison. The most important features we disabled were range queries and parallel processing: When run on default settings Vectorwise will try to split the query execution plan into parts that can be run in parallel. Specifically it will split the tables to be scanned into ranges and give each range its own scan operator to be run in parallel. Disabling this feature increased the query runtimes for native queries considerably. For example for query 1 with enabled parallelism runtime for the 1 G version was around 0.75 seconds, with this feature disabled it went up to to around 3.16 seconds. Therefore Vectorwise will seem slower in these experiments than it could be due to the disabled performance features.

During implementation of the Parquet library and its integration into Vectorwise and the subsequent optimizations we ran several benchmarks to measure speedup. Basis was a scale-factor 1 TPC-H Parquet datafile with a nested structure of region-nation-customer-

[3] `http://www.tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster`

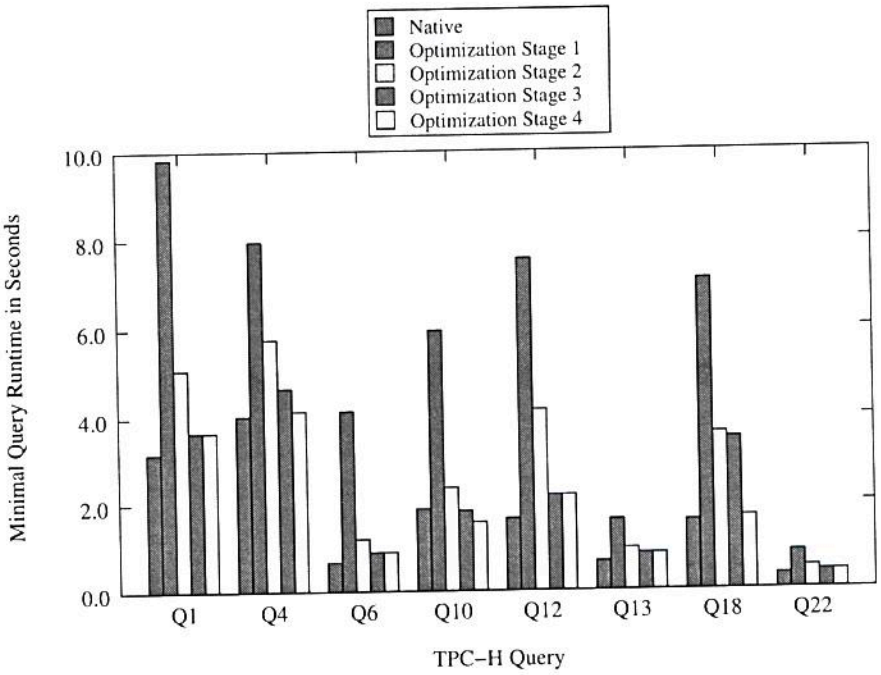


Figure 5.2: Runtime comparison Parquet TPC-H on Vectorwise

orders-lineitem using queries 1, 4, 6, 10, 12, 13, 18 and 22. Basis for the comparison was a run with the same queries against the same data loaded into relational tables (Native). The results can be seen in Figure 5.2.

The different implementation / optimization stages:

**Stage 1** Basis was the first implementation (one tuple at a time filled into vectors).

**Stage 2** Implementation from Section 4.3, all values are read in a vectorized fashion.

**Stage 3** In addition to stage 2 conversion of string values (see Section 4.8) is done in-place, so values are no longer copied

**Stage 4** MergeJoin support enabled for inner joins

As can be seen the runtime for queries with stage 1 implementation is - depending on the query - several times the runtime of the same query on native tables. This is mainly due to the fact that at this stage reading tuples was not yet optimized for vectors. So the result vectors were filled by getting one tuple from the Parquet library and doing it in a loop until the vector was full.

Stage 2 implementations greatly improves on this. At this stage all values are read from the file using the vectorized approach where the Parquet library uses vectors internally for

reading from the DataPages. This improves runtime for all queries quite dramatically, for example for query 6 the runtime drops to about one third whereas query 1 only sees a drop of about 40%. This can be explained with the fact that although both only do a scan on the lineitem-table query 6 only has one *sum*-aggregation whereas query 1 has several *sum*- and *avg*-aggregations and a *count*-operation. Stage 3 enhances that by handling the string conversion (see Section 4.8) in-place. This avoids allocating memory and copying the data for each value. The speedup of this optimization depends on the usage of string columns in the query. Quite the improvement can be seen in query 12 with the runtime dropping about 50% from 4.1 seconds to 2.1 seconds. Queries 1 and 4 also show some improvement, whereas for the other queries the impact is negligible as hardly any string handling is done in them. Finally stage 4 adds MergeJoin-support (see Section 4.3). This has no impact for queries without joins that can be handled by a MergeJoin, such as querys 6, 12, 13 and 22. The best improvement can be seen for query 18 as it consists only of inner and semi joins which can be handled efficiently by a MergeJoin.

When comparing the runtime with stage 4 optimizations to native tables we can see that the performance is almost as good. Query 10 is even a little faster then with native tables. Other queries are nearly identical in runtime, like queries 4 and 18. The other queries are slower by about 10% to 30%.

This slowdown is caused by a combination of factors: The first is that for every page read from the Parquet file the page header has to be decoded using Thrift which takes time. Profiling showed that about a third of the time needed to produce a vector of values from a page is spent in parsing the page header.

Another factor is that in addition to reading the actual values the repetition and definition levels have to be read and acted upon. There are cases where this can be omitted (all fields are required and are at the top level) but generally (especially if the virtual id or foreign key columns are requested) the repetition- and definition-levels have to be read and processed which takes additional time.

## 5.3 Hyper and Parquet

For Hyper we used the same queries from TPC-H with the same Parquet datafile as for Vectorwise. We compared the runtime of the queries running against native relational main-memory-tables to nested Parquet tables using the different implementations outlined in Section 4.2, 4.5 and 4.6. As with Vectorwise we disabled parallelization features in Hyper to allow for a fair comparison. These features normally try to execute the different operator pipelines from a query in parallel using a scheduler to keep dependencies intact.

With the first implementation which called the Parquet library for every tuple (see Section 4.2) runtimes were higher by a factor of between 8 (query 13) and 55 (query 22) than that of native relational tables (see Table 5.1). For this comparison the Parquet file was preloaded into memory before running the queries to avoid disk access.

This bad runtime behaviour of the first implementation was to be expected as it involved

| Query | Native | Parquet |
|---|---|---|
| Q1 | 0.142 | 3.427 |
| Q4 | 0.273 | 4.261 |
| Q6 | 0.060 | 3.235 |
| Q10 | 0.111 | 4.064 |
| Q12 | 0.164 | 4.236 |
| Q13 | 0.102 | 0.781 |
| Q18 | 0.236 | 7.812 |
| Q22 | 0.013 | 0.719 |

Table 5.1: Comparison TPC-H Hyper Native vs Parquet First implementation (runtime in seconds)

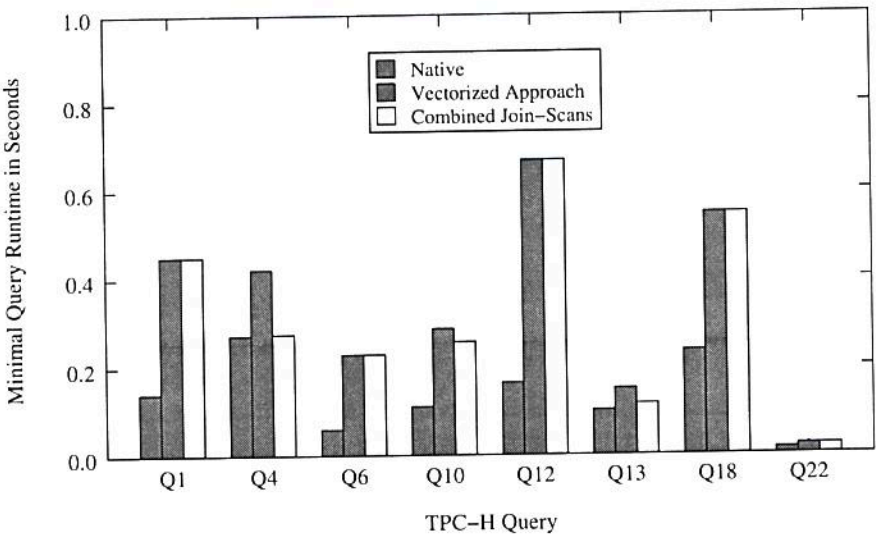


Figure 5.3: Runtime comparison Parquet TPC-H on Hyper

| Query | Vectorized | Omitted r/d-levels | Speedup |
|---|---|---|---|
| Q1 | 0.502 | 0.449 | 11% |
| Q4 | 0.462 | 0.421 | 9% |
| Q6 | 0.272 | 0.229 | 16% |
| Q10 | 0.345 | 0.287 | 17% |
| Q12 | 0.712 | 0.669 | 7% |
| Q13 | 0.163 | 0.150 | 8% |
| Q18 | 0.592 | 0.548 | 7% |
| Q22 | 0.024 | 0.021 | 12% |

Table 5.2: Runtime comparison TPC-H Hyper Vectorized approach without and with omitting handling of repetition or definition levels (runtime in seconds)

a series of method calls from LLVM assembly into C++-code for every tuple. Also all of the logic for handling tuple reconstruction was still a generic implementation.

The runtime could be reduced by about a order of magnitude (see Figure 5.3) using the second implementation. As outlined in Section 4.5 it involved generating LLVM assembly code specific for the query and reading values from the Parquet file in a vectorized manner. Optimizations to this approach include that reading and handling of the repetition levels can be omitted as long as all columns to be selected are from the same group and all have a repetition type of OPTIONAL or REQUIRED (flat subschema). And reading and handling of the definition levels can be omitted for a column if the field for the column is in the top-level group. This optimization alone gained a significant speed improvement for some queries (see Table 5.2).

With the introduction of the merging of joins and scans into a single scan (Section 4.6) some queries showed additional improvement. Best it can be observed for query 4 which sees a reduction in runtime of about 35%. This is because Hyper rewrites the query (which originally has an *exists*-clause with a related subquery) so that it can be handled by a normal inner join which is then for the Parquet version combined into one big scan. Other queries like 1 and 6 show no improvement at all as the query plans do not contain any joins.

## 5.4 Profiling a Query for Hyper

Before we started implementing the optimized version of our Parquet library for Hyper we wanted to get a feel for how the generated code of a query would look like and how the the interface for the Parquet library should be designed in respect to that. To do that we took one simple query containing a join and wrote the code to run that query by hand.

The query was

```
select sum(l_quantity)
from customer, orders, lineitem
```

```
where c_custkey = o_custkey and o_orderkey = l_orderkey
      and c_custkey % 5 = 0
```

The code consists of one outer loop that goes through the available rowgroups. For each rowgroup the necessary columns are read using the Parquet library. Then the inner loop is started which loops until the values from all columns are used up. Inside the inner loop is the logic that determines which vectors to advance using the repetition and definition levels and then provides these values to the code that filters out null values (inner join), checks the condition (*c_custkey % 5 = 0*) and does the aggregation (sum of *l_quantity*). Conceptually this is all one operator pipeline.

Running this query against native relational tables the scan operator would read the following columns: *c_custkey*, *o_orderkey*, *l_orderkey* and *l_quantity*. When running this query against a Parquet file with the scans and the joins combined into one Parquet scan the *o_orderkey* and *l_orderkey* columns can be omitted as they are no longer needed for the join.

In the fully optimized version running the query took 31ms. Running the same query in Hyper using native relational tables took 26ms with parallelism enabled and about 94ms single-threaded.

We took a profile of the execution of that query with our manual implementation using callgrind. This gave us some interesting insights:

- More than 80% (in some queries up to 90%) of the runtime is spent in the pipeline code itself, only about 10% to 15% are spent in the method calls to the Parquet library.
- About one third of the time spent to read and process a DataPage is spent decoding the Thrift encoded page header. This means only two thirds are actually spent processing data from the page.
- Decoding the bit-packed repetition and definition levels takes up to 80% of the time spent to process the data from a page.

We were also able to confirm these results using the fully implemented version running in Hyper.

We also ran two different versions of the code: One simulates selection pushdown by checking the condition as soon a new value for that column is read, the other reads the values from all columns thereby producing the entire tuple and then checks the condition. Conceptually selection pushdown checks the predicates inside the scan operator thereby avoiding unnecessary read operations by skipping values that would be filtered out anyway. This pushdown is implemented as an optimization for the normal table scan operator in Hyper. So we tested both implementations to see if implementing it would gain a significant advantage. The version with selection pushdown implemented took about 5% to 10% less time than the version without it. In the end we decided against implementing selection pushdown in our Parquet scan operator as in our opinion the improvement in runtime did not warrant the additional implementation cost.

| Bitwidth | Unoptimized | Without data dependencies | With SSE instructions |
|---|---|---|---|
| 1 bit | 18312ms/GB | 4784ms/GB | 1600ms/GB |
| 2 bit | 9168ms/GB | 3036ms/GB | 888ms/GB |
| 3 bit | 5480ms/GB | 2018ms/GB | |
| 4 bit | 4614ms/GB | 2292ms/GB | |
| 5 bit | 3397ms/GB | 1558ms/GB | |
| 6 bit | 2819ms/GB | 1359ms/GB | |
| 7 bit | 2980ms/GB | 1279ms/GB | |
| 8 bit | 169ms/GB | 169ms/GB | |

Table 5.3: Runtime comparison for unpacking bit-packed values

## 5.5 Optimizing Bit Unpacking

In section 4.7 we described how we optimized the unpacking of bit-packed values with a focus on the bitwidths 1 and 2. To measure performance we fed the algorithm an input vector with a size of 1 GB and stopped the time it took to decode the values using different bitwidths. We compared the unoptimized version, the version with reduced data dependency and the version using SSE instructions (only for bitwidths 1 and 2). The results can be found in Table 5.3.

Rewriting the code just to reduce data dependency without changing the executed statements reduces runtime to between one half (4 bit) and one quarter (1 bit) of the unoptimized versions. Using SSE instructions again reduces runtime to about one third.

## 5.6 Parquet Format: Space vs Speed

As mentioned in Section 4.7 unpacking bit-packed values with bitwidths that do not align with byte boundaries is hard to optimize as bits of the values are distributed between bytes meaning additional bit shifting is necessary for decoding. So we wanted to see if using a fixed bitwidth of 8 bit would translate to a significant speedup in queries. To that end we modified our Parquet library to always use a fixed bitwidth of 8 bit for all bit-packed values, created a new Parquet file with that property and ran our query suite with it. The results (see Table 5.4) show a reduction in runtime of on average 10%. The Parquet file itself grew bigger by about 12% from using more space for the repetition and definition levels. In our opinion the achieved speedup is significant enough to warrant wasting space for the bigger file.

In light of that experiment we recommend the following changes to the storing of bit-packed values in Parquet files: The format should be extended (by adding a new field to the file metadata) to allow the creator of the file to set a fixed bitwidth of 8 bit for the bit-packing of all repetition and definition levels. This option could be set by the users of the database system (just like they can choose if they want to enable compression for

| Query | Variable | Fixed | Speedup |
|---|---|---|---|
| Q1 | 0.502 | 0.444 | 12% |
| Q4 | 0.284 | 0.260 | 9% |
| Q6 | 0.269 | 0.224 | 17% |
| Q10 | 0.310 | 0.268 | 14% |
| Q12 | 0.695 | 0.652 | 6% |
| Q13 | 0.127 | 0.117 | 8% |
| Q18 | 0.585 | 0.550 | 6% |
| Q22 | 0.024 | 0.022 | 8% |
| Avg | | | 10% |

Table 5.4: Runtime comparison TPC-H Hyper Variable bitwidths vs Fixed bitwidth 8 bit (runtime in seconds)

the file and if they do which algorithm to use) so they can do the trade-off between space and speed. Furthermore we think the bit-packing algorithm should be modified to only use bitwidths that align with byte boundaries (1, 2, 4 and 8 bits). So for example if 3 bits would be needed then the algorithm would choose a width of 4 bits or if 5 bits would be needed a width of 8 bits would be chosen. That way for small values space is not wasted while they still can be unpacked efficiently.

## 5.7 Parquet Format: Using Compression

As explained in Section 2.3 Parquet files support several different compression algorithms (currently GZIP, LZO and Snappy). To test the impact of these we created version of the already used Parquet file with a 1GB version of TPC-H for each of the compression algorithms. Then we ran our set of queries against the file using Vectorwise. For one run we had Vectorwise read the files from local disk, for the other we placed the files on a 3-node Hadoop HDFS cluster and had Vectorwise read them remotely (the machines were all connected in a locally switched network using Gigabit Ethernet cards). Table 5.5 shows the reduction in filesize gained by the different compression algorithms. Because of the many string columns and number columns with only small numbers (which in Parquet still need to be stored using at least 4 bytes) the filesize can be reduced significantly with all three algorithms. As expected GZIP produces the smallest file but takes significantly lower to compress and decompress.

Reading the compressed file from local disk shows (see Figure 5.4) that despite the smaller filesize the smaller cost of reading the file from disk is outweighed by the increased cost for decompression. The file compressed with GZIP shows the clearest increase in query runtime due to the fact that - compared to the other algorithms - filesize and compression time are not directly proportional. Reading the same files remotely from HDFS shows a reduction in query runtime for the compressed file variants of between 3% (query 18) and 12% (query

| Algorithm | Filesize | Percentage of uncompressed filesize | Compression cost (cycles/value) | Decompression cost (cycles/value) |
|---|---|---|---|---|
| Uncompressed | 1230MB | 100% | | |
| Gzip | 280MB | 24% | 397 | 55 |
| LZO | 441MB | 39% | 42 | 13 |
| Snappy | 434MB | 38% | 56 | 15 |

Table 5.5: Parquet Filesize for different compression algorithms

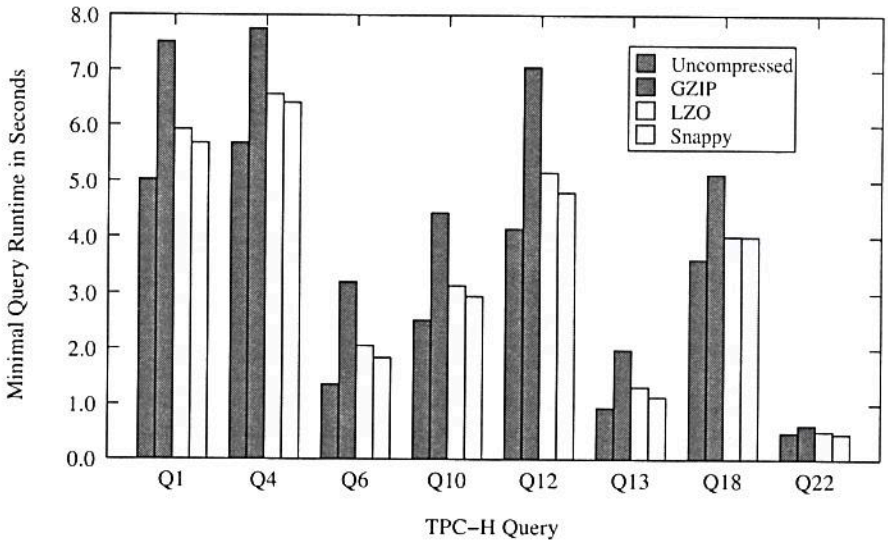


Figure 5.4: Runtime comparison TPC-H queries using Vectorwise with different compression algorithms for a Parquet file read from local disk

12). In most cases the file compressed using Snappy has the lowest runtime, meaning that Snappy seems to provide the best trade off between filesize and decompression cost.

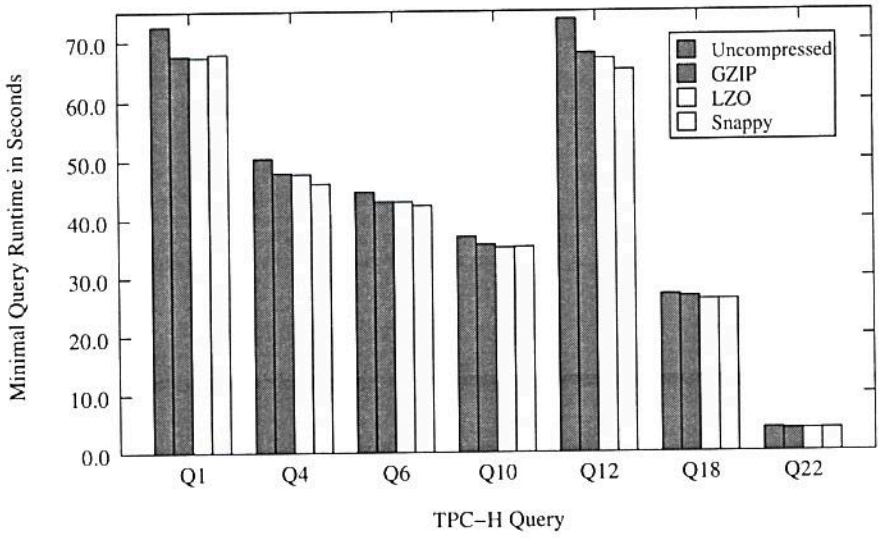


Figure 5.5: Runtime comparison TPC-H queries using Vectorwise with different compression algorithms for a Parquet file read from HDFS

## 5.8 Using Bigger Datasets

Most of our benchmarks were run using a scale-factor 1 TPC-H database. To test the performance for larger datasets we created a scale-factor 10 Parquet version. To keep rowgroup size within the desired bounds we split the data into two sets/files. One being customer-orders-lineitem and the other being region-nation. Would we have packed the entire structure into one file the rowgroup size would have been about 2 GB per rowgroup (5 regions, about 10 GB of data in total). Using these files we ran the usual query suite on Hyper (see Figure 5.6) and Vectorwise (see Figure 5.7).

The results show that our implementation not only works for small datasets but can also keep up with increasing workloads while still maintaining its query execution speed in relation to native relational queries.

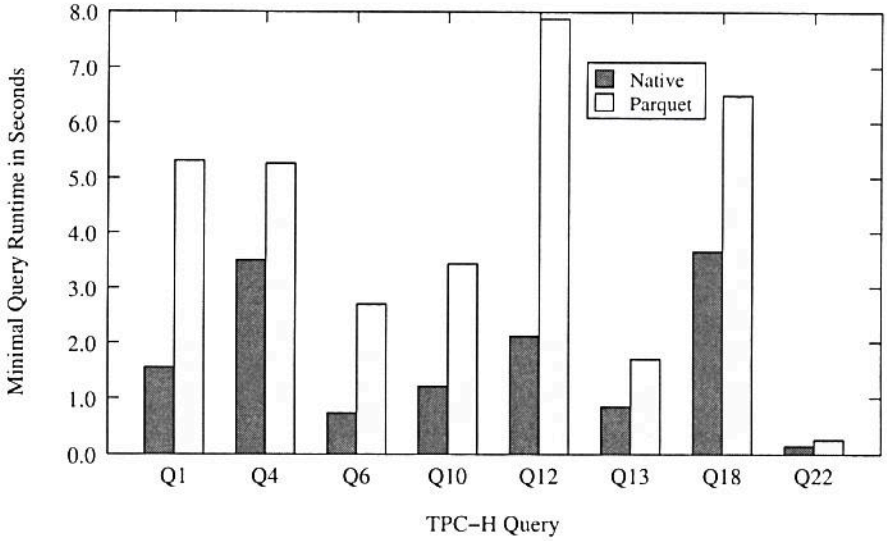


Figure 5.6: Runtime comparison TPC-H queries using Hyper with scale-factor 10

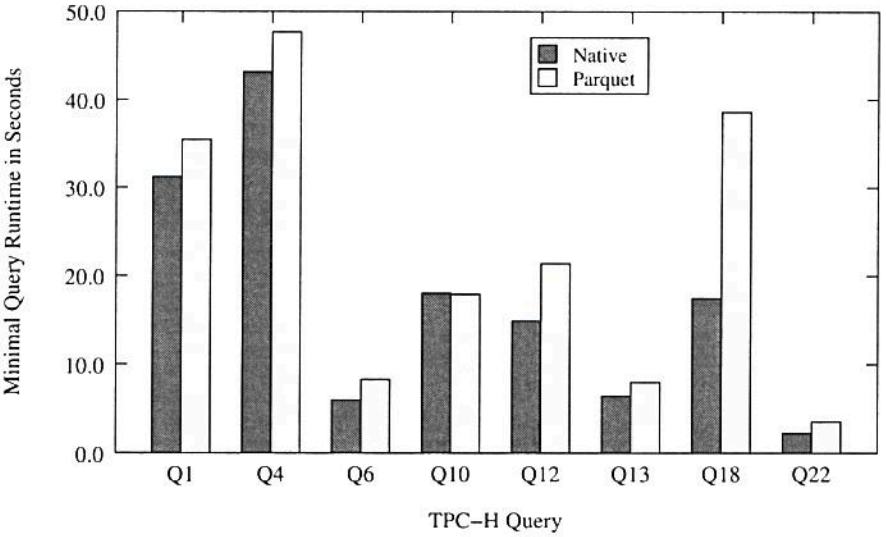


Figure 5.7: Runtime comparison TPC-H queries using Vectorwise with scale-factor 10

## 5.9 Comparing with other Parquet Implementations

In the previous sections we only compared our Parquet implementation with native relational tables of the database systems. But more interesting is to compare it with other Parquet implementations. Therefore we ran some tests using Impala, Hive and Drill. We ran these benchmarks in the quickstart virtual machine provided by Cloudera for its Hadoop distribution[4]. The VM is based on CentOS 6.2 and CDH 5.1 with Hadoop installed in pseudo-distributed mode with all services running on one node. We used the same scale-factor 1 TPC-H dataset as we used for the other benchmarks and ran the same query set. Queries 4, 18 and 22 had to be rewritten because neither system does support subqueries inside *EXISTS*- and *IN*-clauses. The rewrite consisted of extracting these subqueries into separate queries (using temporary views) and joining them to the main query. For query 22 additionally the anti-join had to be implemented using an outer join and a check-for-null-predicate.

The results of the query suite can be seen in Table 5.6.

**Impala** Impala has its own Parquet implementation in C++, but at the moment only supports Parquet files with flat structures. Therefore we created separate Impala tables stored as Parquet for every TPC-H table (region, nation, customer, orders, lineitem). Impala reads all data from HDFS but has optimized behaviour for reading data locally when the execution node runs on the same machine as the data storage node[5]. In comparison to our Parquet implementation running on Hyper Impala is between 1.5 times (query 12) and 6 times (query 13) slower. Part of this can be attributed to the fact that Impala is designed for parallel processing and was run with only one node. Slowdown from reading from HDFS should be negligible as all data was read locally.

[4] CDH: `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh.html`

[5] Cloudera recommends to install Impala nodes on all nodes of the HDFS cluster

| Query | Hyper Native | Hyper Parquet | Impala | Drill |
|---|---|---|---|---|
| Q1 | 0.191 | 0.622 | 1.510 | 20.022 |
| Q4 | 0.342 | 0.381 | 1.120 | 13.072 |
| Q6 | 0.069 | 0.343 | 1.070 | 5.990 |
| Q10 | 0.153 | 0.413 | 1.430 | 15.381 |
| Q12 | 0.193 | 0.897 | 1.330 | 15.013 |
| Q13 | 0.256 | 0.294 | 1.870 | 10.748 |
| Q18 | 0.518 | 0.738 | 4.370 | 23.696 |
| Q22 | 0.014 | 0.027 | 1.150 | |

Table 5.6: Comparison of different Parquet implementations (runtimes in seconds)

**Drill** Apache Drill is also implemented in Java but unlike Impala and Hive can also read data files from the local filesystem. It can be deployed in embedded mode without any external server and as a distributed system of nodes on a cluster. For our test we chose the embedded mode as it is similar to the mode of operation for Hyper. It can be seen that Drill is far slower than our Parquet implementation and also Impala. In comparison to Impala Drill is between 5.5 times (queries 6, 13 and 18) and 13 times (query 1) slower. Query 22 could not be run with Drill because some subquery functionality was not implemented.

**Hive** Hive uses the Java reference implementation for Parquet provided by the Parquet project. As explained in section 2.1 it converts each query into a series of MapReduce jobs that produce the results. These jobs also read data only from HDFS. Due to this approach queries with a small dataset (as is our benchmark) will have - in relative terms - a higher overhead for creating and managing the jobs than queries with large datasets (in the TB range) running on large clusters in parallel where this overhead can be more or less neglected in comparison to the job runtime. We did not include the Hive runtime results in the graph (Figure 5.6) because the numbers were - in comparison to the other systems - way higher. For example for query 1 Hive reports a query runtime of about 114 seconds. Interestingly enough the cumulative CPU time reported by the MapReduce jobs was about 29 seconds (which in orders of magnitude is still about the same as Drill with 20 seconds)

In conclusion we showed that our implementation using Hyper beats Impala, Drill and Hive in the single-node configuration. As we expected Impala is by far the fastest of the three systems due to its implementation in C++ whereas Hive is the slowest due to the overhead of the MapReduce jobs (see Section 2.1).

# 6 Conclusion

We designed and developed extensions for the relational database systems Hyper and Vectorwise to efficiently query Parquet files. For this we bridged the worlds of flat relational tables and nested data Parquet files, providing users with the illusion of dealing with normal tables with full SQL query capabilities while still providing efficient and fast query processing.

We started off with one generic implementation for both database systems employing the iterator-model. Based on the experiences gained with that first try we developed specialized implementations for both database systems that are closely adapted to the respective query execution models while both have in common that they read and process values in vectorized form. For Hyper we read the needed columns from an entire rowgroup into vectors and then produce tuples for the operator pipeline using specifically on a per query basis generated and optimized LLVM assembly integrated into the Hyper code generation model. We also take advantage of the flexibility of that approach and the natural nested structure of the Parquet files and optimize joins by handling them inside our specialized scan operator code. For Vectorwise we produce vectors of values that can directly be processed by the other operators. The handling of joins we leave to the MergeJoin operator by providing specific join columns derived efficiently from the nested structures of the Parquet files.

We finished off development with a series of experiments proving that for Vectorwise in a restricted set of conditions our implementation can achieve largely the same query runtimes as native operators. For Hyper we showed that although our Parquet implementation could not match native query runtimes, it still had impressive execution speed.

Along the way we also provided a comprehensive description of the Parquet format and also analysed strengths and weaknesses of the format from an implementation perspective focusing on speed.

With Hadoop and cluster processing assembling under the buzzwords big data and NoSQL the end of the relational model and relational databases is regularly proclaimed. But our evaluation paints the current state of database evolution with different colors. Even with Parquet, a storage format explicitly designed for efficient query execution, Hadoop-based database systems are no match for an optimized modern relational database like Hyper system in terms of query execution speed. That is true for Hive running on the MapReduce-model and also for Impala which has sort of a relational model and was developed for efficient and fast query execution but still fails to beat a real relational database system. It has to be granted that Impala shows way better execution runtimes than Hive. But Impala is still way slower than the main-memory approach of Hyper. Based on our evaluation we can conclude that this is mostly not the fault of the Parquet format, but more so the fault

of the chosen design and execution model. Because our implementation which integrated Parquet into relational databases (and was developed in only a few months by one person and still has some room for additional optimizations) is also faster than Parquet on Impala, a system using Parquet as its primary format which is under public development by Twitter and Cloudera since March 2013.

## 6.1 Scale-Out

It might be argued that the above claim is not fair as relational database systems are primarily designed as single-instance systems whereas Impala and Hive only show their advantages when run in big cluster environments with large datasets. But both relational contenders in this thesis have current efforts of extending the systems with massive parallel processing capabilities. For Vectorwise this is done under the already mentioned Vortex project which not only runs Vectorwise on Hadoop but also gives it the ability to run queries in a distributed and parallel fashion [12]. For Hyper [21] shows that queries can be efficiently run in a distributed fashion on parallel main-memory database clusters.

Using these developments it should be easy to take our approach of integrating Parquet with relational databases and the relational model and employ it in a distributed and parallel cluster-context while still retaining our margin in terms of query performance.

## 6.2 Future Work

Although our implementation provides a usable and already quite efficient way for reading and querying Parquet files for both Hyper and Vectorwise, still a number of open features and questions remain.

An interesting feature for both database systems would be the capability to not only read Parquet files but also write them. This offers some challenges as Parquet files are designed to be written in one go whereas classic table row insertion is done row by row. In this context it would also be interesting to extend the Parquet format with a way to update existing data. A possible angle could be using some sort of differential structures that are stored alongside the data files. Also the implementation of the Parquet library still misses some features such as processing Parquet files distributed across several physical files or some possible encodings for DataPages that are specified in the format.

As shown query execution speed for Hyper can be improved by implementing selection pushdown. But even more can be gained by utilizing the already available techniques for parallel query execution in Hyper for the Parquet implementation. And in the long term supporting massive parallel processing across nodes in a cluster (as is currently researched and developed for Hyper) will be necessary to compete with other Parquet implementations on a practical level.

The current approach for Vectorwise still lacks proper integration and some features to allow for a user experience needed for a commercially sold product. One point missing is the

extension of the SQL syntax that allows Parquet files to be treated as normal external tables without the workarounds currently in place. Also in that regard the capability to read and process the HCatalog from a Hadoop cluster and directly integrate Parquet files / tables stored there (created by Impala or Hive systems) would allow a flowing interconnection between both worlds from a user experience point of view. From a backend perspective the Parquet scan operator should be extended with a parallel processing mode to fit with the distributed execution model of Vectorwise. The extension should be done in a way that takes into account the existing alignment of the Parquet format for cluster processing instead of simply forcing the existing model of range queries. In the wake of that special considerations should be taken in the context of Vortex for reading Parquet files directly from HDFS. For an efficient way of reading Parquet files processing should be done locally - wherever possible - on the node storing the corresponding data, just like it is already implemented for normal backend storage in Vortex.

# Bibliography


[1] Wei-Hsiu Weng and Wei-Tai Weng. Forecast of development trends in big data industry. In *Proceedings of the Institute of Industrial Engineers Asian Conference 2013*, pages 1487–1494. Springer, 2013.

[2] Ashish Nadkarni and Laura DuBois. Trends in enterprise hadoop deployments. Technical report, 10 2013.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

[5] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[6] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. Monetdb: Two decades of research in column-oriented database architectures. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(1):40–45, 2012.

[7] Marcin Zukowski and Peter A Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[8] Raymond A Lorie. *XRM: An extended (N-ary) relational memory*. IBM, 1974.

[9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[10] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, 2013.

[11] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[12] Peter A Boncz. Announcing vortex - actian vector-on-hadoop. Hadoop Summit - San Jose, June 3 2014.

[13] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[14] HCatalog. `http://hortonworks.com/hadoop/hcatalog/`, accessed on 2014-09-18.

[15] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in apache hive. 2014.

[16] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[17] Parquet Github project. `http://github.com/Parquet/parquet-format/`, accessed on 2014-09-18.

[18] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.

[19] Jignesh M Patel and David J DeWitt. Partition based spatial-merge join. In *ACM SIGMOD Record*, volume 25, pages 259–270. ACM, 1996.

[20] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[21] Wolf Rodiger, Tobias Muhlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 592–603. IEEE, 2014.