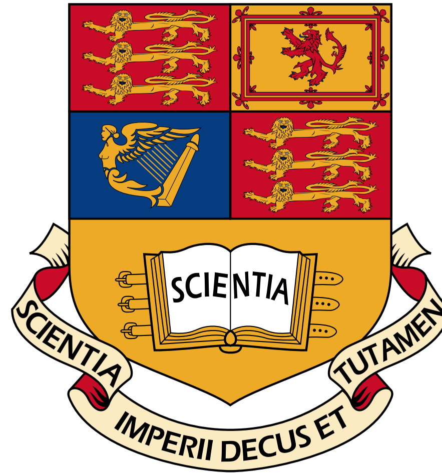
Imperial College London



Department of Computing

Master of Engineering in Computing
Individual Project Report

Scalable In-Memory Aggregation

Robert Jan Kopaczyk

Supervisor:
Dr. Peter Pietzuch

Second Marker:
Prof. Alexander Wolf

June 21, 2011

Abstract

OLAP (Online Analytical Processing) systems play an important role in many industries today. By aggregating the individual records of a data set, they provide an intuitive multi-dimensional view on the large volumes of data commonly stored by many organisations and are used for the purposes of analysis. Recently, the increasing availability of machines with large amounts of main memory and improving processor speeds have led to a surge in the popularity of in-memory OLAP systems, which can process multi-dimensional queries faster than their on-disk counterparts.

However, while hardware capabilities improve, the amount of data to be analysed continues to grow. We can imagine that technological innovation in the area of hardware resources may not be able to keep up with this growth and indeed could reach a halt. Rather than frequently buying the newest (and often most expensive) cutting-edge machines the market has to offer, a better answer to the problem of data volume growth can be to enable a solution to scale out. This means that computation happens in parallel within a cluster of relatively inexpensive commodity hardware machines. Additional machines can then be connected to handle even larger amounts of data.

The goal of this project was to build a prototype of a scalable in-memory aggregator. The result, a system called Simian, is able to distribute data and computation across a cluster of machines, providing fast response times for many types of multi-dimensional queries applied to large data sets.

Acknowledgements

I want to thank the following people – without their help, this project would likely not have been as successful:

- My supervisor, **Dr. Peter Pietzuch**, for agreeing to supervise my project proposal and for providing ideas and useful guidance throughout.
- My second marker, **Prof. Alexander Wolf**, for his useful comments about the background research carried out in the early stages of this project.
- My team at Deutsche Bank, **Market Risk IT**, for inspiring my original project proposal.
- Finally, my parents **Irena** and **Andrzej**, for helping with proof-reading and their support in the final year of my studies.

Table Of Contents

1 Introduction **7**
1.1 About OLAP . . . 7
1.2 The New Trend: In-Memory Analysis . . . 8
1.3 The Problem: Growing Data Volume . . . 8
1.4 The Solution: Distributed Computing? . . . 8
1.5 Contributions of this Project . . . 9
1.6 Outline of this Report . . . 10

2 Background Research **11**
2.1 OLAP: Theoretical Background . . . 11
2.2 OLAP Techniques . . . 15
2.3 Data Storage . . . 17
2.4 Run-Length Encoding Schemes . . . 20
2.5 Data Layout . . . 21
2.6 Data Partitioning . . . 23
2.7 User Interaction . . . 26
2.8 Distributed Programming . . . 29
2.9 Related Systems . . . 31
2.10 Goals for the Research Prototype . . . 32

3 Design Choices **35**
3.1 Choice of Implementation Language . . . 35
3.2 Feasibility Study: Column Orientation . . . 35
3.3 The Storage Layer . . . 38
3.4 Distributed Programming Frameworks . . . 39
3.5 Data Distribution . . . 41
3.6 Storage Layer Optimisations . . . 43

4 Simian: A Scalable In-Memory Aggregator **45**
4.1 Architecture . . . 45
4.2 Data Organisation and Bulk Loading . . . 46
4.3 Aggregation . . . 49
4.4 Post-Processing . . . 52
4.5 Aggregate Tables . . . 53
4.6 Indices . . . 55
4.7 Contrasting Aggregate Tables with Indices . . . 57
4.8 User Interface . . . 57

4.9 Implementation Details . . . 58

5 Evaluation **61**
5.1 Benchmarking Methodology . . . 61
5.2 Experiment #1: Scaling Up . . . 65
5.3 Experiment #2: Scaling Out . . . 67
5.4 Experiment #3: One Billion Records . . . 70
5.5 Qualitative Evaluation . . . 74

6 Conclusion **81**
6.1 Possible Extensions . . . 82

A The Query Language **85**

B The Star Schema Benchmark **87**
B.1 General Comments . . . 87
B.2 Query Flight 1 . . . 88
B.3 Query Flight 2 . . . 89
B.4 Query Flight 3 . . . 90
B.5 Query Flight 4 . . . 93

Bibliography **95**

Chapter 1

Introduction

In this chapter, I will outline the basic idea behind the goals of this project. Besides a general, high-level introduction to OLAP and multi-dimensional databases, I will also touch on the topic of data volume growth, and outline how this problem can be solved by way of distributed computing.

1.1 About OLAP

Online Analytical Processing (OLAP) systems play an important role across many industries in today's world. They are a part of the wider Business Intelligence (BI) range of data analysis techniques, a term which also covers other approaches such as data mining. The ability to access and manipulate a conceptualised *hypercube* containing aggregated information is the core functionality of such systems. A hypercube is one which contains several dimensions, not necessarily three as found in a classical cube we know from geometry. In an OLAP system, they represent attributes that can be used to distinguish data points, such as the year an order was made or the brand of an ordered item in a typical sales data set. Each distinct point in the hypercube then contains aggregated measures, such as profit or revenue.

A user can inspect a hypercube by using a language that expresses *multi-dimensional queries*. However, there are plugins for spreadsheet programs and web-based dashboards that can automatically generate such queries, thus freeing the user from the burden of having to enter them manually.

	Male	**Female**	**Total**
20 and under	129,227	155,729	284,956
21 to 24	17,022	18,065	35,087
25 to 39	10,989	17,478	28,467
40 and over	2,381	5,681	8,062
Total	159,619	196,953	356,572

Table 1.1: An example of a multi-dimensional query result [1].

Results of such queries are in the most basic form rendered as simple tables, with each combination of dimension values present in the data set and the associated aggregated measures forming one row of the output. Table 1.1 shows a more refined and visually intuitive result format. This is a count of students accepted to UK university courses in 2007, aggregated by age group and

gender, as reported by UCAS. This table has headings both horizontally (gender) and vertically (age group) which classify the aggregated values. The representation also includes totals for each age group and gender, and a total for the entire table. Aggregation can be done in other ways than just counting individual occurences matching some description; for example, a revenue measure is most commonly aggregated by adding up individual values to form a result. Also, other forms of representing the output from multi-dimensional queries can be used. It is common for aggregated values of revenue by market sector to be rendered as a pie chart, for instance.

Unsurprisingly, OLAP systems are heavily used in areas such as accounting, marketing or finance, where the ability to quickly summarise and analyse large amounts of data plays an important role.

1.2 The New Trend: In-Memory Analysis

In recent years, we have seen the widespread adoption of 64-bit hardware architectures and operating systems, enabling computers to access massive amounts of memory. A server with 32 gigabytes of RAM is not uncommon anymore, and additionally has become far more affordable. This has allowed entirely memory-based OLAP systems to expand their market share: by not being bound to the traditionally small main memory sizes, it is possible to fit more data into memory than ever before. This way, such systems leverage the inherent speed advantage that main memory has over disk storage, leading to shorter start-up times and quicker responses to queries.

Since many OLAP-type queries are very I/O-intensive and often consist of scanning all or a substantial part of the data set, performing queries on data held entirely in main memory can substantially improve performance and also provide additional interactivity to users. In fact, sub-second response times for large data sets are now becoming the norm.

1.3 The Problem: Growing Data Volume

While storage capabilities and access speeds for various media (not only main memory) have been growing dramatically, and processor speeds have continued to improve, the amount of data to be collected, stored and analysed has not remained at a constant level, either. Going back to the example of accepted university applicants in the previous section, the total number for 2009 was 416,531, around 16.82% more than in 2007 [1], meaning that more information about the individual students will be stored. And a recent (2009) study by Aberdeen Group has shown that Business Intelligence data, for the organisations surveyed, is growing at a weighted average rate of around 30% per year [2]. When looking at these figures, one may ask if this is bad news for memory-based OLAP systems. Will the advantages gained by recent improvements of hardware performance soon be dissipated by the growth in data volume?

1.4 The Solution: Distributed Computing?

A naive solution to the problem of data volume growth would be to just buy a better, more expensive machine if storage space runs out and/or processing time becomes inadequate. After all, one may argue, computing hardware gets better every year. However, there are good reasons why this solution may not prove to be viable in the long term:

- Who guarantees that technological innovation will keep up with the data growth rate? We can, after all, imagine that Moore's Law will not continue to hold forever. Also, this does not help if the rate of data volume growth exceeds the growth in matching hardware capabilities.

- Buying highly specialised cutting-edge equipment is bound to become expensive, as the newest products in the market are usually sold at higher prices.

There is a cheaper alternative. We can buy several low-cost commodity hardware machines and make them work together to achieve both higher storage capacity and better processing speed at an acceptable cost. This also solves the first problem: although there is a certain coordination overhead between nodes in a distributed system, we can continue to scale our system independently of the speed of innovation.

This approach has already been adopted by both academia and industry. The BOINC project utilises the processing power of hundreds of thousands of individual computers around the world, mainly for scientific computing tasks. On the industrial side, Google has patented the MapReduce concept, which helps applications to scale across a cluster of commodity hardware machines. The Apache foundation offers a free implementation of MapReduce called Hadoop. Scalable file (GFS and HDFS) and database systems (BigTable and HBase) have been implemented by Google and the Hadoop project, respectively, for use with such applications.

1.5 Contributions of this Project

The nature of OLAP systems (most aggregation operations we would be interested in can be done completely in parallel) naturally makes distributed computing a method to be considered. By distributing OLAP data to multiple machines, we gain the ability to scale *out*. This means that we can simply attach an additional machine to the system which will handle additional data, as opposed to just scaling *up* – increasing the amount of data at a single machine, while maybe adding additional memory to accomodate it.

As a part of this individual project, a research prototype of such a scalable OLAP system has been implemented. It allows the user to submit multi-dimensional queries, which will be processed using data sets held entirely in main memory. The final result of the implementation phase of this project is both fast, by providing sub-second response times for many common aggregation queries, and space-efficient, by not bloating up the version of the input data held in main memory to a multiple of its original size, as is the case with some OLAP solutions. At the same time, the system can scale out to multiple machines in order to be able to store and analyse large data sets – the largest data set which was successfully tried in the context of this project consisted of over one billion records. While increasing the overhead incurred from distributed operation, adding more nodes to the cluster did not result in excessive performance degradation.

The report details the background research carried out in the course of this project. It also explains the design choices, implementation specifics, challenges and reasoning that resulted in the prototype presented here, and finally tests and evaluates it using a standardised benchmark, while also attempting to pinpoint its strengths and limitations.

1.6 Outline of this Report

This report presents the work done in the course of this project. Apart from this introduction, it also contains the following chapters, listed in the order of appearance in this report:

Background Research. In the initial phase of the project, I studied the wider topic of this project in detail. The resulting literature survey is contained in this chapter. It starts with a general overview of the area, and proceeds to explain specific concepts which are immediately relevant to the goals of this project.

Design Choices. This chapter outlines the high-level design choices behind the research prototype. Often, I had to select between multiple approaches to realise some feature during the implementation phase, and explanations are given of the reasoning behind those choices.

Simian: A Scalable In-Memory Aggregator. This chapter introduces the research prototype resulting from the implementation phase of this project and details its architecture and features. For the most part, this consists of abstract descriptions of how the system accomplishes various crucial tasks. The name "Simian" stands for "**S**calable **I**n-**M**emory **I**nformation **A**ggregation **En**gine".

Evaluation. After the implementation phase was finished, the system had to be evaluated. In the evaluation part of this project, I have carried out various tests of the system to answer questions about quantitative aspects, such as whether the system still delivers good performance while it scales out to more machines. Various qualitative questions are also considered, such as strengths and limitations of the system.

Conclusion. The report ends with a review of the project and possible extensions to the work presented in this report.

Chapter 2

Background Research

In this chapter, I will first go into more detail about the theoretical background of OLAP systems. Afterwards, I will outline considerations such as how data can be stored and operating an OLAP system in a distributed manner.

2.1 OLAP: Theoretical Background

2.1.1 Differences Between OLAP and OLTP

OLTP (Online Transactional Processing) is a term which is often contrasted with OLAP. Frequently cited examples for OLTP-like operations are ones reflecting business needs such as the management of bank accounts or flight bookings. Generally, such transactions read and/or write only a few rows of a table at a time. However, due to the fast pace of the modern business world, these transactions are often so frequent (and therefore, potentially overlapping) that concurrency and consistency of the data are major concerns.

The term OLAP (Online Analytical Processing) on the other hand refers to operations used for the purposes of performing data analysis. These types of operations allow a multi-dimensional view on the data (as described in the introductory chapter) and are often required by decision support and reporting systems. Contrasting with OLTP, OLAP-like operations are frequently very read-intensive and tend to access much larger amounts of data when they are run.

OLAP and OLTP have different uses, and it is not surprising that database systems are often purpose-built for either one or the other type of processing. In fact, in terms of standardised database benchmarks, specialised OLAP benchmarks exist, as do specialised OLTP benchmarks.

The term OLAP is distinct from the term data warehouse. While a data warehouse integrates data from various sources to allow users to analyse it, OLAP is specifically concerned with providing a multi-dimensional view on that data. On the other hand, data warehouses do not need to necessarily be created for OLAP purposes: other uses include decision support and data mining.

2.1.2 Dimensions, Measures and Aggregation

OLAP systems are populated from records of data which contain so-called dimensions and measures [18]. Measures are generally numerical in nature (for example, money). However, they do not necessarily need to be purely numerical. For example, in some applications, it can become necessary to use vectors of numbers as measures.

Measures can either come directly from the data or can be retrieved by post-processing. For example, we could have a measure which represents the value of a transaction in the native currency (e.g. in Japanese Yen). We could then create another measure which represents the value of this transaction in a reference currency (e.g. Euro), derived from the native currency value and a table of exchange rates.

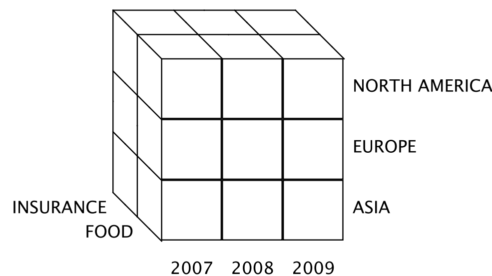


Figure 2.1: An example OLAP cube with dimensions region, year and market sector.

Measures are then categorised within the system by dimensions. These are attributes of the data, e.g. the country of residence of the customer who made an order, or the order year. In general, there is no automatic way of deciding whether any attribute of a data set should be a measure or a dimension. Some attributes could act as either a dimension or a measure, and this is left as a design decision to the user or database designer. A particular value a dimension can take is called a member. On an abstract level, the data in an OLAP system can be imagined to be arranged as a hypercube with N dimensions (called an N-cube here). An example of this can be seen in Figure 2.1. This cube has as many dimensions as the input data set, and each dimension is divided according to its member values. Each point within the cube contains aggregated measures, summarised from the records which have their attributes set to the corresponding values.

Aggregation is one of the most important concepts in OLAP: we can pick one or more dimensions of the N-cube and "collapse" it to form a lower-dimensional cube only containing these dimensions, which is more useful for our data analysis purposes. For an N-cube, we can generate up to 2^N such subcubes (also called aggregates or aggregations). For example, in Figure 2.2, with 3 dimensions we have $2^3 = 8$ possible aggregations. The possible sub-cubes we can derive for any given data schema can be arranged in a lattice, in which an aggregation with no dimensions (i.e. of all data points) is at the bottom and the aggregation of all dimensions together is at the top [3].

2.1.3 Concept Hierarchies

Some dimensions stored in an OLAP system are naturally hierarchical and can be arranged in such a manner. For example, let us consider the concept of a location. We generally consider a continent to encompass more locations than a country, which in turn is less specific than a province. In OLAP systems, concept hierarchies reflect such relationships. They constitute mappings from higher level

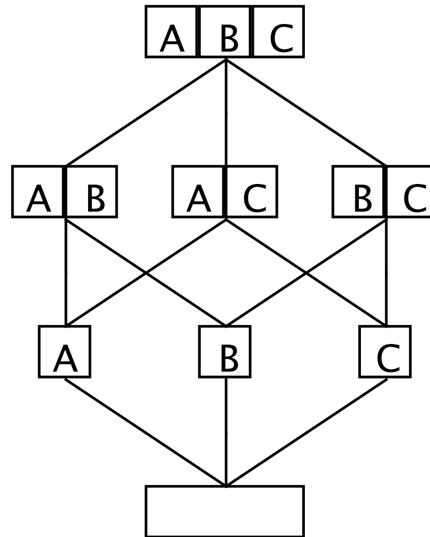


Figure 2.2: Lattice of aggregations for cube with dimensions A, B and C.

and more vague to lower level and more specific concepts [4]. This can be used to split a dimension into several layers.

2.1.4 Typical OLAP Operations

According to Gyssens and Lakshmanan [19], any OLAP system has a list of properties it should fulfill when providing its services to the users. These impact both the design of the conceptual model behind the system and the query language exposed to the outside world. The properties the paper identified were:

Querying. Users need to be able to formulate queries in some kind of language. The query language should ideally be both powerful and simple and allow us to receive results which can be rendered in an easy-to-understand format, such as a table.

Restructuring. This is the ability to restructure information in such a manner that it brings out different perspectives on a data set. This can be seen as adding or removing a dimension to an existing view on the multi-dimensional model. For example, going back to the UCAS example, we could imagine removing the "gender" dimension from the view in order to receive a view by "age group" only – with the aggregates measures being recalculated accordingly. Gyssens and Lakshmanan defined the fold (remove dimension) and unfold (add dimension) operators to model this operation.

Classification. This is the ability to be able to classify or group data sets in such a way that they can be easily summarised afterwards. This involves classifying the data into distinct groups before aggregation is carried out. Theoretically, this can be seen as mapping tuples of a relation into one or more distinct groups. This is an important feature, since we want to break data into these groups before aggregation happens.

Summarisation. This is the ability to summarise data in order to receive aggregate values. Theoretically, this can be seen as collapsing multi-sets of (usually numerical) values to a single,

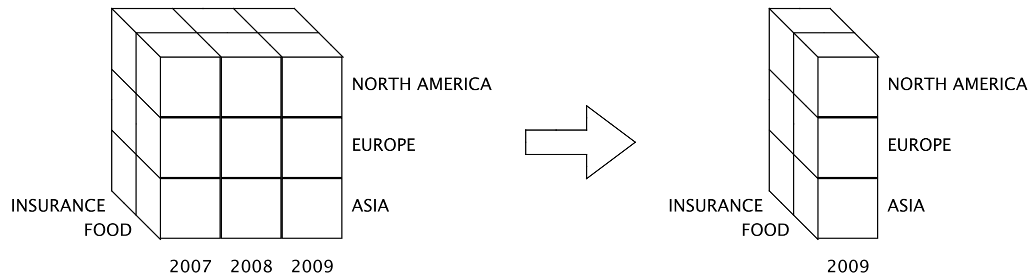


Figure 2.3: Slicing an OLAP cube.

consolidated value. For example, summarising the set {*1, 2, 3, 3*} using a sum function will yield the result of 9.

There are several operations a user will often carry out while successively changing queries to analyse the data. Some of these are defined in terms of consumer experience rather than the underlying implementation (e.g. both a roll-up and a drill-down change the view to align it with an aggregation). The following operations [4] are commonly carried out:

Roll-Up. A roll-up operation aggregates data in a cube. There are two main ways in which this is achieved: firstly, we can aggregate across a concept hierarchy, e.g. by collapsing countries into continents. Secondly, we can just remove a dimension from the sub-cube, which will gain us a more general view on the data. For example, in the UCAS example from the introduction, we could just drop the gender dimension and receive a view of accepted university applicants by age group only.

Drill-Down. A drill-down operation is the opposite of a roll-up: we either descend along a concept hierarchy, or add more dimensions to the view. The resulting view gives us more detail.

Slicing. This is similar to an SQL `SELECT` operation and returns a new sub-cube with only the specified values for a dimension. For example, we could select only data where the recorded year is 2009, as shown in Figure 2.3.

Dicing. Dicing is very similar to slicing, however, it performs a selection on two or more dimensions. An example would be e.g. `(CustomerCountry = "Germany" OR CustomerCountry = "France") AND (Year = "2009")`.

Dimension Browsing. This involves being able to explore dimension members and concept hierarchies. The results can then be used to further refine future queries.

Drill-Through. This involves retrieving and displaying the records from the input data set which have contributed to some particular value in an aggregation result. These records are called *contributors* by some OLAP systems.

2.1.5 The Extract-Transform-Load Process

Before being able to use a data warehouse, e.g. by utilising OLAP techniques, the data often has to go through an extract-transform-load (ETL) process. In the extraction phase, one or more data sources are accessed to retrieve the data. This can for example be an existing relational database which is not meant to be used as a data warehouse. It could also be data in another form – for example log files as produced by a web server. Afterwards, the data may be transformed:

examples include converting between time formats and changing the data to use uniform letter case conventions, and also cleaning data from aspects we are not interested in. Finally, the data is loaded into the system [25].

2.2 OLAP Techniques

There are two main techniques in use today for managing OLAP data: MOLAP and ROLAP. Both have advantages and disadvantages, and typical scenarios in which they are more appropriate. This section aims to give an overview of these.

2.2.1 MOLAP

MOLAP, or multi-dimensional OLAP, is one of the oldest ways to manage OLAP data. Aggregates are stored in a special multi-dimensional data structure. In general, MOLAP systems pre-compute all possible aggregates, which involves listing results for all combinations of dimension members. They therefore tend to have large storage requirements.

The main advantage of MOLAP is that the pre-computed aggregates can be easily accessed using an index (as they generally resemble multi-dimensional arrays), and retrieval is therefore very fast. In fact, once we have pre-calculated all possible values the user could ever request, not taking into account the time needed to perform the pre-computation, this is the fastest possible way to serve responses to user queries. However, one of the main problems which plagues MOLAP systems is that, with a growing number of dimensions, the resulting data structure becomes increasingly sparse – it will contain a lot of empty cells, because many of the dimension member combinations contain no data. This phenomenon is called data explosion in some sources and can result in huge storage requirements. Some MOLAP systems use methods such as compression or adopting a two-level-storage representation [6] to deal with this sparsity problem, however doing so can destroy the desirable indexing feature inherent to MOLAP [3]. In general, MOLAP is still a good choice for data sets with a low number of dimensions. However, for cases with large numbers of dimensions, or dimensions with a large number of members, there may be better choices.

2.2.2 ROLAP

ROLAP, or relational OLAP, takes advantage of relational database technology to provide analytical functionality. Data is frequently stored in a specialised form of schema, such as a star or snowflake schema. This aspect of database design will be covered in detail in a subsequent section, but in general just consists of specialised tables in a relational database. This also handles most sparsity issues which could arise for us, since relational databases have mechanisms in place to deal with this [3]. Generally, ROLAP systems arrange their data in such a way that standard relational query language expressions (for example join, select, grouping in SQL) are used to respond to queries in a multi-dimensional description language. In general, ROLAP systems do not require pre-computation of large amounts of data in the same way MOLAP systems do. Thus, these systems are better equipped when dealing with data that has a large number of dimensions, since they do not suffer from data explosion of the same scale. However, for performance reasons, aggregation results can still be pre-computed and stored in what is called aggregate tables by such systems.

These are normally frequently-accessed, user-defined reports, and usually low in number. When compared with MOLAP systems, the general consensus is that ROLAP systems respond to queries more slowly, but on the other hand, scale better with an increasing number of dimensions and generally allow more flexibility with respect to query complexity and data update [7].

2.2.3 Comparison and Hybrid Techniques

Although MOLAP is still a good choice for smaller data sets, especially ones with few dimensions and dimensions with few distinct values, ROLAP systems tend to scale better in general. Due to the more efficient nature of the underlying storage mechanism and lack of pre-computation by default, ROLAP systems usually are able to store data with more and larger dimensions and can handle cube redefinitions and frequent updates in a more flexible manner.

Some hybrid approaches are employed as well (frequently called HOLAP, although there seems to be no consensus over what this means in detail as there is for ROLAP and MOLAP). For example, it is possible to use an MOLAP model for coarser sub-cubes with few dimensions, while a fine-grained view is obtained using ROLAP techniques. We could for instance pre-calculate MOLAP data structures for all aggregations containing three or less dimensions, and if we cannot respond to a query from this, use the ROLAP way to calculate all aggregations which cannot be derived from the MOLAP data structures.

I have decided to reject the use of an MOLAP technique for the purposes of this project. The decision is influenced by the requirements we have for the finished system.

Firstly, since the goal of the project is to have a distributed system, we need to invest some thought into ways of distributing the data across machines. While thinking up and comparing the performance of ways to distribute data held in a logical view such as a (relational) database is comparatively straightforward - plenty of literature exists on the topic of distributed databases, and they are widely used in industry - adequately partitioning a custom data structure as needed for MOLAP will probably be harder, and such cases will be described in fewer sources.

Secondly, MOLAP systems seem to have problems with combinatorial explosion, especially in the context of schemas with many dimensions. This could affect the flexibility of the final system, and managing this phenomenon would require complex techniques such as compression. Although one may argue that we will have plenty of space available in the desired setting (multiple, possibly dozens, of distributed machines), we would still rather have the data volume grow in a roughly linear rather than in an exponential (or worse) fashion compared to the amount of base data. In comparison, ROLAP systems seem to be able to deal with this issue in a more graceful way.

Finally, in an industry survey paper, Gorla [7] defines some features an organisation should look for when deciding whether to use an MOLAP or ROLAP system. Some of these consider the general user-friendliness of these solutions, which is not a major focus of this project, however other points are relevant. According to the paper, MOLAP systems in general are better suited to scenarios where a user will mainly concentrate on simpler, pre-defined reports, since the queries that can be generated for these systems are not usually very flexible. Also, MOLAP data structures are usually regenerated periodically and this process takes significant amounts of time. On the other hand, ROLAP systems are more suited to large volumes of dynamic data as common in retail, manufacturing and finance. The queries that can be defined for them are more flexible than the ones for the MOLAP variants. Most organisations eventually prefer ROLAP systems because of

the reasons mentioned.

For this project, the same reasons apply: the implementation will deal with large amounts of data distributed across several nodes. The amounts of data I am planning for are common in industries which make heavy use of ROLAP (e.g. finance, retail). Using MOLAP could easily lead to combinatorial explosion, especially considering the high cardinality nature of some dimensions in such data sets – for example, an organisation may store a lot of data about its customers, and customer attributes such as their name will be unique for each of the (potentially many) customers. The reasons mentioned by Gorla, together with my own reasons as outlined above, have convinced me to adopt an ROLAP-like approach.

2.2.4 Aggregate Tables

Although I am rejecting the MOLAP approach, there are nevertheless some advantages to be found in pre-computing data structures to be used with at least some queries: sometimes, these do not take up much space, and answering a query using such a pre-computed data structure offers a substantial speed-up when compared to a full scan over the data set.

An approach that is often chosen in ROLAP systems is to pre-aggregate some data into *aggregate tables*. Aggregate tables have a coarser granularity than the source data (i.e. contain a lower level of detail and have fewer rows). The measures of importance are still included, but are pre-aggregated to fit the new view on the data. In practice, this often allows users and administrators to define aggregates of importance, for which suitable aggregate tables are computed. Identifying these aggregates is based on the queries that are expected to happen in a system and whether computing the suitable aggregate tables is feasible in terms of overall size. This often requires input from domain experts. However, in the literature, approaches to calculating an optimal subset of aggregates have been studied, including greedy [27] and genetic algorithms [28].

A query planner will have to decide when a query can make use of an aggregate table. This is often based on the lattice of aggregations as illustrated in Figure 2.2. We can note how aggregates further down the lattice (towards the empty box) can be derived by collapsing aggregates further up in the hierarchy. Aggregate tables are often a good choice for aggregations which use only few dimensions of low cardinality.

2.3 Data Storage

2.3.1 Relational Databases

Generally, as the name already implies, ROLAP data is usually stored in relational databases. The relational model allows us to store data in relations, i.e. predicates involving several columns of values. First introduced as an idea by E.F. Codd in 1970 [12], it draws its inspiration primarily from set theory and seeks to protect the user from having to understand the underlying physical structure of the data (which the database system is managing) via abstraction. Relational algebra is a formal language forming the basis for many relational query languages, such as the different flavours of SQL. Today, relational databases are used widely in industry, and provide important advantages such as order independence and the ability to normalise the schema, thereby giving us a feasible method to reduce data redundancy and inconsistency. Many relational database systems

are publicly available and some also allow holding data entirely in memory, and thus would be interesting to consider for our purposes. Also, some publicly available databases can be directly started and managed entirely from code in a major language, by e.g. abstracting the database server as a Java object. I will explain more on the feasibility of re-using these in this project in Chapter 3.

2.3.2 Non-Relational Databases / "NoSQL"

Recently, the concept of non-relational databases is becoming a popular topic in the IT industry. Another term these are often known as is "NoSQL", however, *non-relational* is a more accurate term to distinguish them from "traditional", relational database systems. These are often highly specialised to certain environments (e.g. distributed programming or querying documents of data held in a special form such as XML). Notable examples include Cassandra, a distributed database management system (DBMS) built to deal with very large amounts of data, and CouchDB, a system which stores its data in the form of JSON documents.

Since these databases generally have less expressive query languages than relational ones, it is potentially even simple to write such a system ourselves for a particular project. For example, the GQL query language designed for the BigTable system by Google is modelled after SQL but only supports queries on a single table, avoiding joins for performance reasons [29]. In fact, many distributed non-relational databases lack built-in joins, because they do not operate quickly enough in a distributed architecture. Non-relational databases are thus frequently more code-intensive to the application developer than their relational equivalents, with the developer instead writing code to accomplish this task, or designing the schema in such a way that no joins are needed – often by de-normalising the data. Such a database system tailored to our needs may offer us advantages, especially performance-wise. This could be interesting in the context of this project, especially since joins are something we would typically want to avoid anyway in a distributed environment. Implementing a complete relational database, on the other hand, with all of its defining features and guarantees is a daunting task.

On the other hand, there already are some criticisms of many common non-relational databases. As would be expected, the reliance on manual query programming has been criticised, since although this may be easy for simple operations, it can get very hard for complex queries. Also, non-relational databases have less of an emphasis on the ACID (Atomicity, Consistency, Isolation, Durability) properties which are guaranteed by virtually any relational DBMS. Atomicity and durability together mean that either all actions in a transaction will be committed and the results permanently stored, or none will and the transaction is cancelled, while consistency implies that the transaction must obey all legal protocols and leave the database in a consistent state. Isolation requires that the effects of a transaction cannot be accessed by other transactions before a commit occurs [16] [17]. These are almost expected to be obeyed by any modern DBMS, yet neglected by many new, non-relational systems. However, this is just one side of the coin, as it may not be such a big concern in certain environments, and contributes to scalability and performance in these [15].

Although building a *relational* OLAP system using a *non-relational* data model may seem paradoxical at first, it can conceivably work. However, maybe using the term ROLAP for this is a bit of a misnomer, and the term seems to be mainly used to contrast it with MOLAP systems, which generally use multi-dimensional arrays. I prefer not to attach too much importance to labels, and to just decide what is really needed in the context of this project.

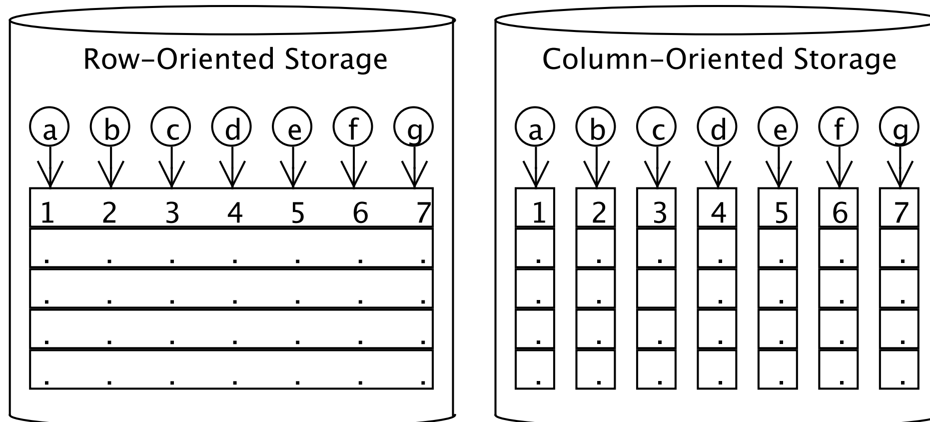


Figure 2.4: Column-oriented storage contrasted with row-oriented storage.

2.3.3 Column-Oriented Databases

An interesting concept which can be considered for use in this project is column-orientation, a concept which is gaining popularity recently and is used in systems such as LucidDB, which is a DBMS built specifically for business intelligence and data warehousing [40]. Such databases store their data arranged by columns rather than by rows, and thus tend to have better performance for common operations such as scanning a subset of columns in a table. Whereas the data layout of traditional row-oriented databases is fits entire records of a table next to each other, a column-oriented store has several sections for each table, each of which is dedicated to only storing values of a particular column next to each other. This is especially advantageous in an OLAP context: while row-oriented stores may still be more suitable for OLTP applications, column-oriented ones have been reported to be more suitable to OLAP-like settings. In the case of a full scan over a subset of columns, a row-oriented DBMS will read in blocks of records, only to discard all of the irrelevant attributes – while a column-oriented DBMS will read in blocks of values only for columns it is interested in. Clearly, the column-oriented DBMS will read in less blocks when there are fewer columns to be considered. On the other hand, column oriented databases have been reported as expensive for operations such as updates or insertions [13]. Therefore, they seem to be more suitable to read-intensive systems, which often only receive an initial bulk of data to insert. However, most OLAP systems fit this description, and in fact, many are based on column-oriented data storage modes. Whether the performance gains of disk-based column-oriented databases are similarly present with in-memory databases will be investigated in Chapter 3.

2.3.4 Optimisation Approaches

Column-oriented storage alone already allows efficient full scans over a data set. However, this can potentially be optimised even more. In particular, we could consider storing columns sorted in a lexicographic manner, with e.g. lower-cardinality columns taking preference in the defined ordering. Then, run-length encoding could be applied, greatly reducing the size of the column [24]. Additionally, and perhaps paradoxically, run-length encoding could result in shorter query processing times - we can still iterate over a run-length encoded column, and highly compressed columns can take much less time to iterate over, as we can skip long streaks of values we are not interested in.

A further approach that is useful in databases, in particular to speed up scans, is bitmap indexing. Bitmap indices are known to substantially improve scan times for columns with a small range of values, for example ones encoding concepts like gender and binary yes/no flags [24]. This involves creating a bit-string for each value in the column, with each bit representing whether this position in the column contains the value or not. This can speed up scans, as specifying a filter on some column allows us to use the bitmap to skip sections of the column which do not contain values relevant to the query. Similarly to run-length encoding the columns, a sorted data set will both reduce the size of a run-length encoded bitmap index (because the runs of values are larger) and also improve performance of scans (as longer runs of values we are not interested in can be skipped).

Enforcing a lexicographic ordering through sorting, with lower cardinality columns taking precedence in the ordering, is a good heuristic to use instead of trying to find the best possible ordering. In fact, the problem of finding the best ordering is NP-hard [30].

It has been demonstrated that bitmap indexing can also be applied to columns whose cardinality is not low. The general challenge is that traditionally, for a column with e.g. 100 distinct values, 100 separate bitmaps are required, each indicating the presence or absence of one of these values at a each position in the column. This can quickly become very space-demanding. A solution to this is to partition the values of the column into several disjoint sets – in this example, we could e.g. define 10 sets with 10 values each. Now, for each of these sets, a bitmap index is created. If the bitmap index is set to "1" at any position, this means that any one of these 10 values is contained in the column at that position. This allows us to still skip large parts of the column which are not relevant to the queries – false positives do not usually matter, as we can still check whether the value found at a position indicated by the index is one we were looking for, or one that is irrelevant [31].

2.4 Run-Length Encoding Schemes

In this section, I will give a brief overview of two run-length encoding schemes which could be applied in this project in order to optimise bitmap indices.

2.4.1 Word-Aligned Hybrid (WAH)

The WAH scheme [42] is based on words and splits the bitmap into clean words (sequences of either all ones or all zeroes) and verbatim words (all other words). An initial marker bit indicates which type of word follows. If this is a verbatim word, the remaining 31 bits store the content of this word. In the case of a clean word, an additional bit indicates which type of clean word follows, followed by a 30-bit sequence representing the run-length of this type of clean word. Although this scheme can drastically decrease bitmaps in size, it can also in the worst case increase the size of a bitmap by 3% – as 32 bits would be used to represent 31 bits of verbatim data. The theoretical worst case scenario thus occurs when the entire bitmap data consists of what will become verbatim words in the "compressed" bitmap.

2.4.2 Enhanced Word-Aligned Hybrid (EWAH)

The EWAH scheme [30] is derived from the WAH scheme and seeks to offer an improvement for the cases where WAH fails to compress a bitmap (and increases its size instead). It again uses two types of words, and the verbatim type is unchanged from WAH. The second type of word is called a marker word: it is of 32-bit length and contains a bit indicating the type of the clean word that will follow, and the rest of the bits stores the number of clean words of this type (16 bits) and the number of verbatim words (15 bits) which will follow this marker word. EWAH bitmaps start with a marker word. Although EWAH may be less efficient than WAH in cases where there are many consecutive clean words, it is very unlikely that the compressed bitmap will be larger than the original, as long runs of verbatim words are dealt with more efficiently.

2.5 Data Layout

Now that we have established the choice of the ROLAP method (or something of that kind) for the project, I will expand on some of the data layout possibilities that follow from this.

2.5.1 Flat Tables

In the simplest form, we could use a flat file database, where all data is kept in one large, unnormalised table. Aggregate tables could also be stored in unnormalised tables. The early literature on database theory is full of criticism of this approach, and suggests different modes of normalisation. Among the problems identified by the authors are the tendency for such unnormalised data sets to have large storage requirements, since the form generally gets redundant and does not exploit repetition well to decrease the space required. For example, a customer frequently has made several orders, and a record about each order would, in a normalised database, usually contain a foreign key to a seperate table storing customer records. In a flat table of orders, however, each detail about the customer would be replicated in full. Additionally, it has been pointed out that such schemas do not preserve consistency very well – it is fairly easy to go wrong and e.g. assign different addresses to a customer, when the constraint is that a customer should only have one associated address. Also, flat tables may require us to use (and deal with) `NULL` values, which is cumbersome. Normalisation, on the other hand, can help remove the need for `NULL` values.

However, as we shall see later, some aspects of denormalisation can be helpful especially in the context of distributed databases. Here, data which is denormalised to some degree can reduce or eliminate the need for join operations, which can get expensive, and especially so when data is stored at different sites.

2.5.2 Star Schemas

As mentioned previously, data in ROLAP systems is often arranged in star schemas, a specialised table layout to enable multi-dimensional database features in a relational database system. At the centre of a star schema, we have a fact table: it contains the measures from the original data (e.g. transaction value) and also, references/foreign keys to dimension tables, hence giving it the shape of a star. The dimension tables represent the dimensions we have defined, which the user can use

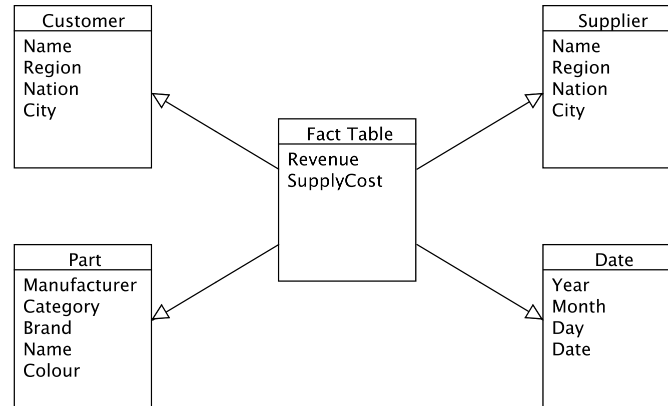


Figure 2.5: An example star schema.

in the analysis process. A dimension table can also represent a concept hierarchy of dimensions, i.e. a progression such as "continent, country, province", and frequently groups together related attributes – e.g. the name and country of a customer. In this case, the dimension table simply contains more columns to fit the additional attributes [6]. Star schemas are used by applying standard relational database operations such as joins and selection – aggregation is achieved by using SQL expressions such as `GROUP BY` in conjunction with SQL aggregate functions like `SUM` or `AVG`. Figure 2.5 shows an example star schema – the dimension table names used in this particular example form a highly popular example in the literature. Listing 2.1 shows an example of a star join query, which classifies an aggregated profit measure by order year and customer nation, with certain restrictions in place. A star schema is a simplification of the more complex snowflake schema, which will be covered in the next section.

Listing 2.1: Example of a star join query [44]. The table `LINEORDER` is the fact table.

```
SELECT
  d_year,
  c_nation,
  SUM(lo_revenue - lo_supplycost) AS profit
FROM
  date, customer, supplier, part, lineorder
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_partkey = p_partkey AND
  lo_orderdate = d_datekey AND
  c_region = 'AMERICA' AND
  s_region = 'AMERICA' AND
  (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
GROUP BY
  d_year, c_nation
ORDER BY
  d_year, c_nation
```

2.5.3 Snowflake Schemas

Unlike what is the case with star schemas, in snowflake schemas the dimension tables are normalised to a further degree. For example, a progression such as "continent, country, province" could be modelled as a table of provinces, referenced by a table of countries, which in turn is referenced by a table of regions. In comparison to star schemas, snowflake schemas can have significantly lower storage requirements, especially in cases where dimension tables are large and contain complex concept hierarchies. They are also (relatively) simple to use, and the schema remains flexible: we can still easily update the dimensions. Additionally, snowflake schemas can be derived from entity-relationship modelling diagrams using a semi-automated methodology [9]. Snowflake schemas are suggested in many data warehouse design methodologies and are widely adopted across industry [8]. However, there are disadvantages to this method: usually more join operations are needed to execute queries, which could negatively impact performance.

2.5.4 Discussion

While star and snowflake schemas offer us a more normalised view on the data, saving space and helping to ensure consistency, flat tables have the advantage of being easy and fast to use in a distributed context. However, one may argue that from the point of view of a data warehouse, consistency is not such a big criterion – frequently, data warehouses are populated from data sources used for other purposes anyway, which may be normalised and should at least observe consistency. So the consistency found in a data warehouse will mainly depend on the correctness of the extraction process.

As will be outlined in Chapter 3, I decided to choose a storage mode which by default stores denormalised data, reaping the benefits we get from this in a distributed environment, but also keeps dimensions as seperate entities and allows normalisation in certain cases.

2.6 Data Partitioning

In this section, I will consider some choices for partitioning the data with the goal of storing (not necessarily disjoint) partitions on separate nodes. There are two main features we look for when distributing our data. Computation should preferably happen where the data is to reduce the amount of resulting network traffic. For example, sending/receiving an aggregation result is far less expensive in terms of time than sending/receiving the entire base data and then performing aggregation.

2.6.1 Parallelising Aggregation

The process of aggregation is, in most cases, simple to parallelise. This means that we can split the input data set into several non-overlapping parts, perform aggregation on each of them independently, and then combine the results in such a manner that the output from parallel processing is the same as if the processing happened sequentially.

There may be special design requirements when parallelising the aggregation process, and this mainly depends on the aggregation function that we want to run in parallel. The case for simple

(but still very common) functions such as taking the sum, minimum or maximum of measures is trivial. All of these functions are binary operators which satisfy the associative property. In other words, the order in which the operations are performed does not matter, and also allows us to set any arbitrary bracketing in a sequence of applications of the same operator.

The case is less trivial, but still fairly simple to solve, for the problem of computing the (arithmetic) mean of a set of values. Clearly, we will not necessarily receive the same results by applying the binary or even n-ary mean operator with different orderings. A solution to this problem is to not store the result of the mean calculation directly. Instead, we store two values: the sum of the values we want to take the mean of and the total number of contributors – since an arithmetic mean is just the former divided by the latter. If we wish to combine this with another result, we simply add up both values – as addition is commutative, we will receive the same result.

Certain other functions are less trivial to parallelise, among them for example standard deviation. However, in most contexts (as judged from some benchmarks I have seen and from prior work experience), these functions are less frequently or even never required, and thus do not have a high priority for the purposes of this project.

Some OLAP application frameworks allow developers to define their own aggregate functions, often by implementing special interfaces for this purpose. The aggregation result will be able to keep track of a *history* which allows the developer to define an appropriate course of action for a variety of scenarios when computing non-trivial aggregation functions such as the arithmetic mean. Cases which need to be addressed include e.g. computing the binary operator for two values, for an aggregation result with a history and another value, and for two aggregation results. Obviously, the cases for trivial binary operators such as sum are simple, and generally the same code is used for all of them in each case.

2.6.2 Horizontal Partitioning and the Shared-Nothing Architecture

When considering methods to spread data across nodes, horizontal partitioning is an obvious choice. It is an approach common in distributed databases. Here, entire records are distributed onto nodes, often according to some pattern - for example, if a record has an attribute for the country it "belongs" to, we could have a pattern that each node is responsible for a certain country. However, other approaches can be employed, such as basing the distribution on a hash function.

Vertical partitioning is a different concept: the data is not broken up into sets of records, but rather into sets of columns. These columns are then assigned to be stored on designated nodes. For example, one node could store a column representing a country, and another could store a date column. This is a good option when considering a system in which users are frequently only interested in a subset of the attributes alone (such as a particular country), however it is not a good option when considering multi-dimensional queries. Many combinations of attributes could be interesting to a user, and we could sooner or later require a distributed join, tremendously decreasing performance.

It would be great for our system to have a shared-nothing architecture: this would mean that the nodes themselves are totally independent of each other – none of them has a need to access the memory available to another node in the system [23]. Since there are no points of resource contention between nodes, this allows operations on the nodes to be carried out in a faster manner, since this significant bottleneck found in many systems is missing. OLAP systems in particular can

be designed to be shared-nothing, since aggregation is generally parallelisable as discussed in the previous section.

2.6.3 Data Warehouse Striping (DWS)

The Data Warehouse Striping (DWS) technique [5] involves distributing fact table entries for a star schema in a (typically) round-robin fashion while loading. This means that each node will have roughly the same number of fact table rows (± 1). The dimension tables themselves are replicated on each node. An implicit assumption is made here that while the facts will take up a lot of space, dimension tables are comparatively small. Queries to each node are submitted, and partial results are returned. Merging partial results is not a large overhead here – it is the alternative, sending a lot of raw data, which we want to avoid. Different variations of the distribution algorithm have been tried: apart from trying forms of round robin with varying window sizes (e.g. loading 1, 10, 100, ... rows before moving on to the next node), random distribution and hashing were investigated in the literature [10], with mixed results depending on the type of query used.

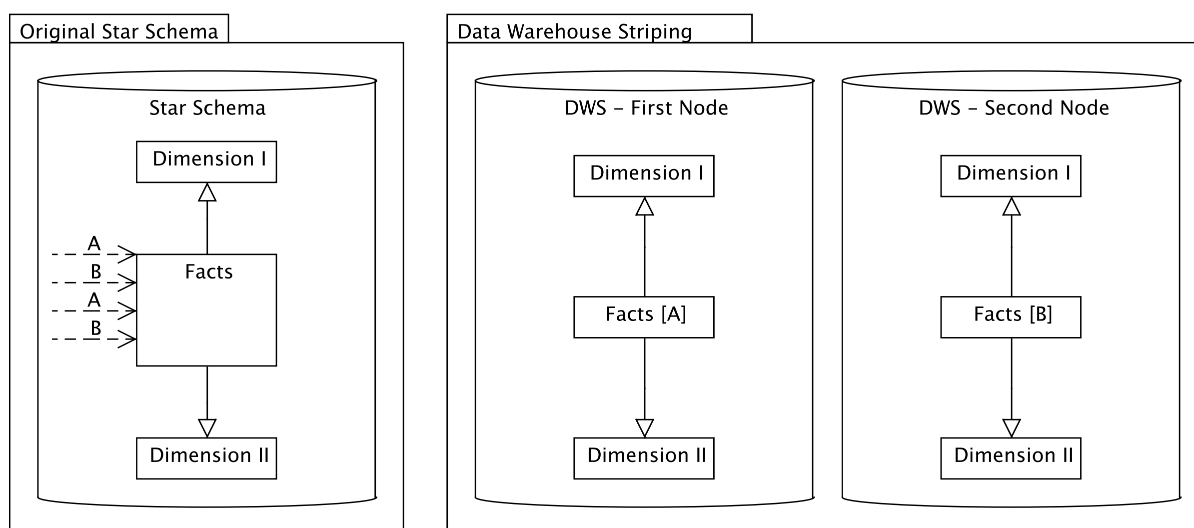


Figure 2.6: Illustration of the Data Warehouse Striping partitioning scheme.

This approach can result in near-optimum load balancing. However, we are assuming a small set of values for each dimension, allowing the dimension tables to be easily replicated across all nodes. Thus, the main drawback of this technique is that it may not be applicable in contexts with large dimensions. For example, a large dimension table which occupies hundreds of megabytes of space would be replicated on each node, cutting into the main memory provided on this node which we would rather use for the fact table.

So while this approach allows us to distribute data across nodes, it could potentially not scale very well. An issue to note is that while fact table data grows (corresponding to, for example, an increase in the volume of transactions a company is involved in), so does dimension data (some of the new transactions will involve first-time customers). Thus, we could hit the limits of a node not necessarily because of the size of the fact table data, but instead because of the dimension data.

2.6.4 DWS Extension: Selective Loading

In practice, if we have a large dimension which is causing trouble when using the DWS technique, each entry in the dimension table is often only related to a few facts. For example, consider a customer dimension (which will be a "large" dimension: an airline often has millions of customers) with individual flight tickets being facts for an airline OLAP system: each single customer will only have few facts associated with themselves, these being the flight tickets they bought. From this observation, we can improve the DWS technique: for a large dimension, we only load a row into the corresponding dimension table on a node if a fact referencing this row exists on the node. This is called Selective Loading [5].

There is one small problem with this, but also a solution: it is rather complicated to perform a dimension browsing query, i.e. to just look up the entries in the big dimension. The solution is to have a part of the full dimension table on each node as well, in addition to keeping a local dimension table with only the entries needed in the local fact table. This way, we have a way to get a full view of the dimension with a still acceptable performance. It is worth investigating whether deliberately keeping dimension tables as small as possible on each node via Selective Loading improves performance in terms of speed, and I may do so later on in the project.

On the other hand, the referenced paper indicates that even though this approach manages to save storage space on the nodes, they will still require more space to accomodate the data than would be the case with a single node. I tried to implement the Data Warehouse Striping and Selective Loading techniques for the prototype and was able to replicate this issue. This is because as dimension data grows, the number of co-occurences found in the partitions of a star schema tends to grow as well; thus, it is extremely likely that the dimension tables of different nodes will share a large number of entries. An analysis of this issue, and the resulting reasons for my rejection of both of these approaches, can be found in Chapter 3.

2.7 User Interaction

As discussed in Section 2.1.4, a query language is needed to bridge the gap between the user and the system. This should provide the user either directly, or indirectly via some kind of visual tool, with the ability to quickly formulate queries and receive results. The language should contain constructs which will allow the user to restructure, classify and summarise data. The typical workflows a user may be interested in (roll-ups, drill-downs, slicing and dicing) should be easy and intuitive to accomplish in this language. Of course, using SQL to perform star join operations can be seen as as an example of this; however other, often simpler ways to achieve interaction have been introduced. The concepts discussed in this section should add ease of use and better representation to basic star joins.

2.7.1 The Datacube Operator

Jim Gray et al. [22] introduced a set of extensions to SQL in order to better handle multi-dimensional queries. The traditional `GROUP BY` statement used in conjunction with aggregate functions only generates the core of a particular view on the datacube, but does not automatically generate other desirable data such as totals. This leads to problems when we want to show information such as histograms, roll-up totals and sub-totals for drill-downs and cross tabulations.

Thus, two new keywords were added to the SQL language: `ROLLUP` and `CUBE`. These are extensions to the normal `GROUP BY` mechanism which also compute sub-totals. For example, a `ROLLUP` with `CustomerNation` and `AgeGroup` would also contain totals for the customer countries of residence, while a `CUBE` with the same parameters would additionally contain totals for the age groups.

2.7.2 The MDX and XMLA Standards

Multi-Dimensional eXpressions (MDX) is a query language first introduced by Microsoft. It is specifically designed for use with multi-dimensional databases and resembles SQL. It simplifies the submission of multi-dimensional queries and the rendering of the results. Apart from using it in "raw" form, graphical tools exist which can be manipulated by a user to generate MDX expressions. These are then automatically sent to a server and the results are rendered in the graphical tool – the user does not even need to be aware of the existence of the MDX query language. Frequently, OLAP servers are based on the *XML for Analysis* (XMLA) standard. This involves wrapping an MDX query together with additional properties and parameters into an XML request, with the response also being in XML format [21]. MDX alone, or in conjunction with XMLA, offers a flexible way to represent standard OLAP concepts such as roll-up/drill-down and slice-and-dice.

Listing 2.2: Example MDX Query [20].

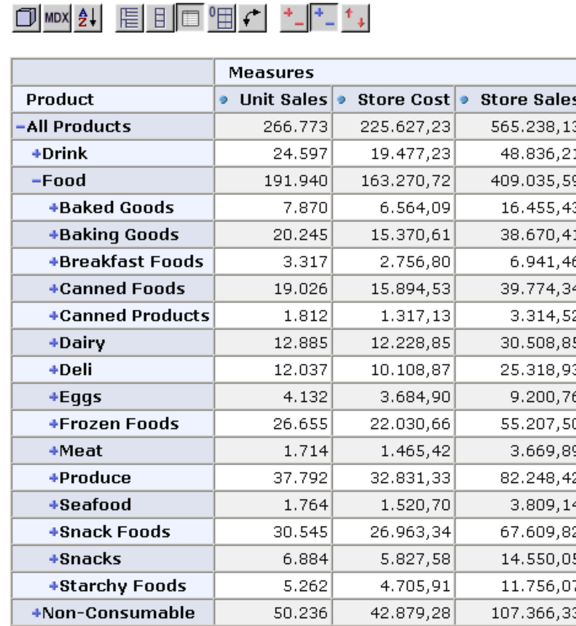
```
SELECT
        {[Store Type].MEMBERS} ON COLUMNS,
        {[Store].[Store City].MEMBERS} ON ROWS
FROM
        [Sales]
WHERE
        (MEASURES.[Sales Average])
```

In the example shown in listing 2.2, we can see some of the crucial features the language can express. The query operates on a cube called `Sales`. Members of dimensions are assigned to being displayed on either the columns or the rows of a result set, to be appropriately rendered by the front end. A member of a dimension is a value that resides in it: for example, the value `United Kingdom` can be a member of a `Country` dimension. MDX allows us to utilise concept hierarchies, as used for example in the expression `[Store].[Store City]`. We can also specify the measures and functions we are interested in using for aggregation: here, this is the sales average. Tools exist which can translate MDX into SQL queries suitable for use with a star schema, for example Mondrian.

2.7.3 Interactive Tools

Many OLAP solutions supply visual, interactive tools which assist the user when analysing the data. As mentioned previously, these generally generate MDX queries based on manipulation of the interface by the user, submitting them to the server in the background and rendering the received results.

Usually, a user will be able to select dimensions (and maybe dimension hierarchies) and allocate them such that the dimension members appear either at the top (horizontally) or on the side (vertically). The user is also able to choose measures and the functions to be used for aggregation. The aggregates are then displayed at the intersection of the relevant dimension member values. In



	Measures		
Product	Unit Sales	Store Cost	Store Sales
-All Products	266.773	225.627,23	565.238,13
+Drink	24.597	19.477,23	48.836,21
-Food	191.940	163.270,72	409.035,59
+Baked Goods	7.870	6.564,09	16.455,43
+Baking Goods	20.245	15.370,61	38.670,41
+Breakfast Foods	3.317	2.756,80	6.941,46
+Canned Foods	19.026	15.894,53	39.774,34
+Canned Products	1.812	1.317,13	3.314,52
+Dairy	12.885	12.228,85	30.508,85
+Deli	12.037	10.108,87	25.318,93
+Eggs	4.132	3.684,90	9.200,76
+Frozen Foods	26.655	22.030,66	55.207,50
+Meat	1.714	1.465,42	3.669,89
+Produce	37.792	32.831,33	82.248,42
+Seafood	1.764	1.520,70	3.809,14
+Snack Foods	30.545	26.963,34	67.609,82
+Snacks	6.884	5.827,58	14.550,05
+Starchy Foods	5.262	4.705,91	11.756,07
+Non-Consumable	50.236	42.879,28	107.366,33

Figure 2.7: An example use of JPivot, illustrating drilling down on a concept hierarchy [37].

the case of the user using a dimension hierarchy as opposed to a "flat" dimension, this will initially only display the highest level of the hierarchy. The user can then expand the hierarchy downwards, gradually drilling down deeper into the data. This is rather similar to the pivot table functionality offered by spreadsheet programs.

An example of a publicly available visual component which provides this functionality as described is JPivot, shown in Figure 2.7. JPivot uses Mondrian as the OLAP server, and supports the MDX and XMLA standards. Most commercial OLAP systems provide their own implementation of this type of user interface and/or repackage third party components for this purpose. Also, these do not always need to be in the form of a web application as is the case with JPivot; another very popular choice are plugins for spreadsheet applications such as Excel, rendering results in a similar way to a pivot table.

The manner in which multi-dimensional query results are presented in front ends as discussed in this section is often highly configurable. A developer can define properties which will be obeyed by standards-compliant user interfaces. An instance of this is prohibiting aggregation of certain dimensions. For example, we may want to curtail aggregation between records with monetary measures (i.e. sums of money) when some currency dimension would have different values for each record. The front end will then generally adhere to this rule by asking the user to enter a value for the dimension for which aggregation is prohibited (to be used as a filter) before any further actions are possible. This will prevent the user from receiving incorrect results which could be the outcome of e.g. adding a value representing Japanese Yen to one representing Euros. A slightly more mundane issue which can be configured are number formatting specifics, such as the minimum and maximum number of fractional digits that should be displayed.

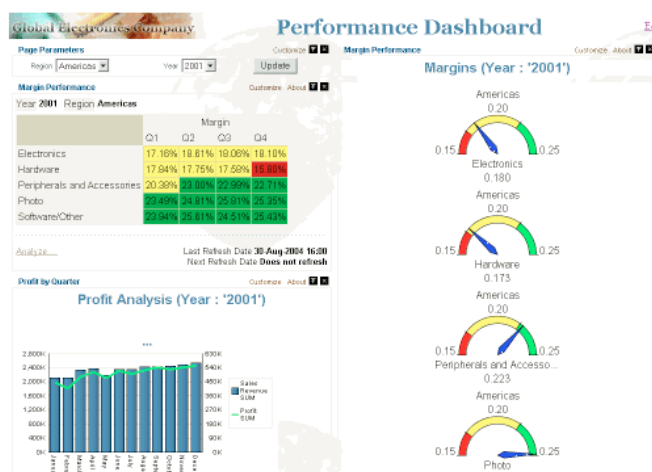


Figure 2.8: A Business Intelligence dashboard by Oracle [36].

2.7.4 Dashboards

Another way for the user to interact with the system is by using a *dashboard*. This is generally provided in the form of a web application and contains a variety of reports. Dashboards are generally assembled from components which query an OLAP system and then render the information graphically; for example, as pie or bar charts. These are useful for analysts, as they can go to a website and inspect information that has already been transformed into an easy-to-follow representation. They do not need to first tell the system which representation they want, as the reports are generally predefined. Of course, it is in principle possible to provide components which allow some degree of interactivity (e.g. we could display a table of aggregated measures by city, and the user could switch the country by selecting another value from a combo box). Figure 2.8 shows such a dashboard.

2.8 Distributed Programming

In this section, I will briefly describe a few concepts which can help us to build a distributed system. These concepts are general in scope and not specific to this project; they are implemented by distributed programming frameworks available to programmers, as opposed to only distributed OLAP systems.

2.8.1 Remote Procedure Calls

Implementing communication between nodes in a network directly via TCP/IP without any layer of abstraction is a very cumbersome process. A programmer will generally have to define their own protocol, and make sure that the code that will be run on each node will be consistent with this definition. This will get very hard to coordinate in most situations, especially non-trivial ones. Even worse, apart from dealing with only the part of the process in which actual communication happens, a programmer will generally also have to write their own serialisation/deserialisation routines in order to transmit more complex data structures.

Remote procedure call (RPC) frameworks help with such issues, by abstracting network communication to "feel" more like calling a procedure in the code – rather than having to open a socket and write complicated code to coordinate interaction with the other end of the line. What happens is that a server node exports an interface on some port, which defines the procedures which can be invoked remotely. A client node can import this interface and then invoke these procedures, generally as if they were local parts of the program. RPC frameworks also perform serialization/deserialization of arguments to procedures and results; on top of that, they handle error checking. This way, the programmer does not have to hand-code a complex communication protocol anymore and can use constructs which are far more intuitive.

Java provides a remote procedure call framework by default, called *Remote Method Invocation* (RMI). It works by exporting a Java interface on a port, which is also implemented by some class on the server node. The client node can then import a handle to this as a reference to the respective implementing objects, and invoke methods on this reference as if it were a normal local object – with the small detail that RMI introduces additional exceptions to highlight error conditions.

2.8.2 MapReduce

MapReduce is a framework and a concept created by Google to help with distributed processing on a large number of nodes. It is intended to be used with parallelisable problems which can be broken up into several constituent parts. MapReduce computation happens in two steps:

1. **Map:** a master node decomposes the input into several smaller chunks, which can be processed in parallel. These chunks are then distributed to a number of worker or nodes, or *mapped* in this terminology, which start processing the requests. The problem decomposition process is developer-defined.

2. **Reduce:** once the master node has received replies with partial results from all worker nodes, it will proceed to combine (or *reduce* in this terminology) these results according to some (again) developer-defined merging procedure.

The theory behind MapReduce was inspired by functional programming and the staple map and reduce functions present in virtually all functional programming languages. In the actual MapReduce framework, the developer writes the map and reduce functions, and the framework automatically deals with distributing the workload. This relatively simple concept allows us to operate clusters with potentially thousands of nodes. Hadoop is a project maintained by the Apache foundation which aims to replicate the mode of operation as described in the original paper by Dean and Ghemawat [14], and an interesting choice for this project which I will consider in Chapter 3.

2.8.3 Discussion

Both of these frameworks provide some abstraction from the low-level aspects of communication in networks. It is definitely important for this project to use some such abstraction for the research prototype, as otherwise the implementation process will become complicated, lenghty and error-prone. On the other hand, it is important to consider the options carefully, as choosing one such framework becomes binding: it may be very hard to switch to another model in the middle of the implementation process.

2.9 Related Systems

While I am not aware of many implementations of specifically a *distributed in-memory aggregator*, similar systems have existed for a long time. In general, what falls under the term OLAP when describing such a system can be a description for what I perceive to be several types of features, and it can provide some or all of them. In general, we can distinguish the following concerns:

Data Storage. This deals with how and where the data is stored and the representation of the data. For example, a column-oriented database provides this feature – and in this case, the chosen representation is quite suitable for OLAP-like workloads.

Data Analysis. This deals with providing adequate capabilities to perform the required analysis of the data; apart from the general ability to carry out aggregations, this would also include other aspects such as managing pre-computed data structures and providing a more specialised query language than the storage layer alone provides, such as MDX.

User Interaction. This feature describes the parts of a system which deal immediately with the user – i.e. the functions the user interface provides to allow the user to define the results they desire and also dealing with achieving the representation the user specified.

As we shall see, systems labelled as "OLAP" will support one or more of the above features, and can also be combined in different ways to achieve all three of these features.

2.9.1 A Selection of OLAP Systems

With the above classification in mind, I will introduce a few existing OLAP systems, which may implement some or all of the above functions.

Mondrian [33] is an example of an OLAP server which provides additional data analysis capabilities to an existing mode of data storage. Mondrian delegates aggregation operations to one of several supported relational database management systems. On top of this, Mondrian implements the MDX query language, parsing it into SQL expressions to be applied to the back-end database. Additionally, Mondrian allows a developer to define schemas (with a choice between star and snowflake schemas) and to control other features such as aggregate tables and output formatting. Since Mondrian does not provide a user interface of its own and it will only work on top of a back-end database, it is an example of a system that enhances data analysis capabilities of an existing database.

JPivot [32] provides a user interface to MDX data sources. Besides allowing a user to perform operations such as drill-downs and roll-ups on a visual representation, it can also be used to provide charts visualising the data. Since JPivot deals with neither data analysis nor data storage, it is an example of a system that only offers user interaction capabilities.

Palo [34] is an OLAP system developed by a company called Jedox. It is a memory-based solution, similar to what I want to achieve in this project. The back-end is MOLAP-based [35], while front-end functionality and ETL tools are also supplied by the same vendor. Thus, it is an example of a system which provides all three of the features mentioned before: data storage, data analysis and user interaction. However, Palo does not seem to support scaling out beyond a single machine.

2.9.2 DBToaster: Efficient View Maintenance

Materialised views are commonly used in databases to speed up query processing, and consist of the results of some query. The specific case where such a materialised view consists of aggregation results is an aggregate table. When the data stored in a database changes, the materialised views need to change accordingly. In some situations, doing so efficiently is important, and for this purpose incremental view maintenance algorithms have been proposed. Since a materialised view often represents a join between several base relations, these algorithms try to propagate *deltas* (changes) from the base relations in order to compute the overall change of the view.

The DBToaster project [46] has created an SQL compiler which generates database engines for in-memory stream processing and offers incremental view maintenance for continuous queries on streaming data. The view maintenance operations are compiled into native code. This is done *recursively*: by considering combinations between base relations, it tries to transform delta forms (which are themselves queries) into even simpler queries, attempting to receive very simple and fast procedural statements to be used for view maintenance. If an OLAP application is to receive frequent updates, then some form or another of incremental view maintenance would be an important consideration to implement, and reacting to new data quickly is an interesting area of research. However, the goal of this project is primarily to study read-intensive systems which store large amounts of data, not systems which perform stream processing. That being said, efficient maintenance of views is a useful feature even in such systems.

2.10 Goals for the Research Prototype

In this project, I want to explore the wide topic area I have introduced in the background research chapter. For this reason, I want to build a research prototype which will perform all three of the functions mentioned before: data storage, data analysis and user interaction. Obviously, since this scope is very wide, and I also want to primarily build a scalable architecture (an angle most other systems do not consider), the research prototype will not have *all* of the functions which are available for such systems, but rather a good selection of *some* of them. The goal will be, essentially, to build a research prototype consisting of a vertical slice of an end-to-end system which satisfies the following features:

An adequate amount of features. The resulting implementation should allow a selection of analytical functionality which allows us to e.g. implement and run benchmarks. However, we do not want to get too focused on features which, while they improve user experience and are tedious to implement, also do not add much overall complexity the system would have to handle – i.e. do not add much value to this project.

A vertical slice of functionality. The resulting implementation should have components which satisfy each of the three requirements discussed earlier: data storage, data analysis and user interaction capabilities should all not rely on external systems.

A good testing ground for algorithms. The resulting implementation should allow us to implement algorithms which deal with various aspects of the system, the performance of which we can quantify by using some kind of benchmark.

Acts as a framework. The resulting implementation should ideally constitute a framework which allows a developer to build an OLAP application with relatively little additional code required.

> This should e.g. take the form of the developer writing a bulkload process in a language such as Java, and using some configuration files, the rest of the framework should handle all other concerns – i.e. basic aggregation capabilities and more complex features such as pre-computation.

The vertical slice should be narrow, but fill out the entire depth of existing systems. In other words, our research prototype does not need to have all of the functionality users may wish for in industry, as long as it demonstrates the underlying concepts well and serves as a good basis for the evaluation chapter.

Additionally, the other requirement I want to add to the final research prototype is that it is **fast** and **memory-efficient**. We need to quantify these adjectives in order to know what they mean in this context. A fast system here is one which is not frustrating while we use it; it is a frequent occurence in OLAP systems that impatient users will resubmit a query several times, proceeding to click arbitrary UI components because they think it will make the result (or in fact, some sign of life) appear faster. Clicking on a button and waiting for a minute or two for something to happen provides a sub-optimal user experience. Thus, the response time for most queries a user would be interested in should be no more than a few seconds, with sub-second response times clearly being the optimum. In terms of memory usage, the research prototype should not bloat up the on-disk source data to occupy a multiple of the original space in main memory; although the research prototype will be intended to run on many machines, we should bear in mind that the system is not practical if the amount of hardware needed to run it would be too expensive for an industrial user to buy.

Chapter 3

Design Choices

In this chapter, I will outline some design choices which had to be made, either before or during the implementation phase, along with the reasons for making each respective choice. Some of my initial design choices also proved to result in insufficient performance and I reconsidered them, re-implementing that particular part of the research prototype. In such cases, I will also describe the history behind the final decision that was taken.

3.1 Choice of Implementation Language

I have chosen Java to be the implementation language. Java is very suitable for the purposes of building large, distributed systems, such as the research prototype of this project, with lots of library support available for such projects. Unlike what the general impression used to be in the 1990s, Java today cannot be counted to be among the slow languages. The speed of the Java Virtual Machine has greatly improved ever since the introduction of just-in-time compilation, which today is standard for most JVM implementations.

Apart from being fast, Java offers great advantages in terms of clarity of code, when compared to other languages such as C and its derivatives. A great advantage, for instance, is the complete absence of explicit pointers. Java offers pointers only in the form of references to objects – there is no distinction between a reference, a value object, and references to references, as is the case in some other programming languages. Thus, in my opinion, a large Java codebase is easier to grasp than an equivalent one in C. Many libraries and frameworks exist for the Java platform, ranging from relatively simple concerns like time and date to more advanced ones such as distributed programming. This means that many trivial problems will not have to be directly dealt with, and I can concentrate on the important and interesting parts of the implementation.

3.2 Feasibility Study: Column Orientation

For queries whose results it can or did not precompute, an OLAP system will have to do a scan over the entire data set. Although a bitmap index can help in some cases, efficient scans are still a requirement to provide good performance for such complex queries. As discussed in Chapter 2, column-orientation promises to help with this issue, decreasing the time needed to scan over a large

data set. Before adopting this approach, I wanted to establish just how fast this approach is when contrasting it with row-orientation, especially since the in-memory case may be different than the on-disk case. For this purpose, I carried out a small feasibility study.

3.2.1 Experiment Description

Column-oriented storage is based on the concept of storing values of the same column in a contiguous manner on some storage medium. As the research prototype is meant to operate in memory only, we can implement this by e.g. associating a column with an array of values. Thus, for a data set with several columns, we can store this as an array of arrays – for example, the array could have 5 sub-arrays, each of which in turn has 10,000,000 values, for a data set with 10,000,000 records arranged in 5 columns. Conversely, row-oriented storage can also be seen as a multi-dimensional array: we could implement this by storing an array with 10,000,000 sub-arrays, each of which in turn has 5 distinct values for each of the columns. To the naive observer, this might seem not to make much of a difference – surely, they might say, this is just syntactic sugar which allows programmers to access an array with two index values as opposed to just one, and only spares them the hassle of having to combine the two values in some manner to calculate an array offset. However, Java treats these two definitions very differently. In a two-dimensional array in the Java language, each sub-array is an entirely different instance of an array. To account for this implementation detail of the Java compiler, I threw in an additional data layout mode, in which all data is stored in a single array, albeit in a row-oriented way: the records are still stored contiguously.

The task uses a data set which has 10,000,000 records, arranged in 5 columns. The expressions I have used to create the data structures associated with each storage mode differ: for column-oriented storage, this is `new int[5][10000000]`. For two-dimensional row-oriented storage, the expression `new int[10000000][5]` is used and for one-dimensional row-oriented storage, this is `new int[50000000]`.

I then proceeded to prepare the data structures by filling them with random integers between 0 (inclusive) and 20 (exclusive). The task which was to be performed in each test case is as follows: scanning over the data structure, compute a sum of the values in the column with index 2, but only include the record in the calculation if the value contained in the column with index 1 is even. This models more “useful” instances of aggregation quite closely, by considering both the summarisation and classification aspects. Since the values the data structures are filled with are the same in all three cases – the respective parts representing records are filled in the same order in each case, and the same random seed is used – all three test cases return the same sum. The time of each such mock aggregation operation is measured. The experiments were run with JVM parameters `-Xms1024m -Xmx1024m`, giving them an additional, larger amount of memory to fit the data. The computer used was a 3.06 GHz Intel Core 2 Duo MacBook Pro with 4 gigabytes of RAM running Mac OS X.

3.2.2 Results

In each case, the data structures were loaded with the random values, prior to performing the operation described in the previous section. Because the time taken to perform this operation is generally low and shows some variance, 100 runs are carried out – the computation step of each run is timed and an average is taken.

Storage Mode	Average Computation Time
Column-oriented	48.68
Row-oriented (two-dimensional)	129.73
Row-oriented (one-dimensional)	65.52

Table 3.1: Results from the storage mode feasibility test.

As can be seen in Table 3.1, the column-oriented storage variety fared best in this test; it is followed by the one-dimensional row-oriented variety. The two-dimensional row-oriented variant ranks last, on average taking more than two and a half times as long to compute the same result as the column-oriented one.

3.2.3 Discussion

As we have seen in this feasibility study, column-oriented storage does make a difference when it comes to scan times, and this advantage is still present if the data is held in main memory. The column-oriented storage mode implemented here with arrays is similar to what is used in on-disk column-oriented databases: values for the same column are contiguous. The one-dimensional row-oriented test case, on the other hand, is most representative of an average relational database: entire records are arranged in a contiguous manner. Finally, the two-dimensional row-oriented test case shows a misconception we should aim to bear in mind during the implementation phase: although it looks like the records would be arranged in a contiguous manner in such a scheme, in reality they will not end up being arranged like this; even worse, since each sub-array is an entirely separate instance of an array, a lot of dereferencing is implicitly added by the compiler. This is to ensure that we could, for example, assign the sub-array to a field of a one-dimensional array type. However, this incurs a performance hit. The column-oriented test case avoids this, since the dereferencing of a column will usually be cached. The way Java arranges the respective multi-dimensional arrays is illustrated in Figure 3.1. Also, column-orientation seems to be faster than one-dimensional row-orientation, as reading adjacent values in an array is generally cheaper in terms of time cost than accessing non-contiguous regions of memory. This improvement is likely due to the effects of techniques which exploit locality of reference for performance reasons (in CPU caching).

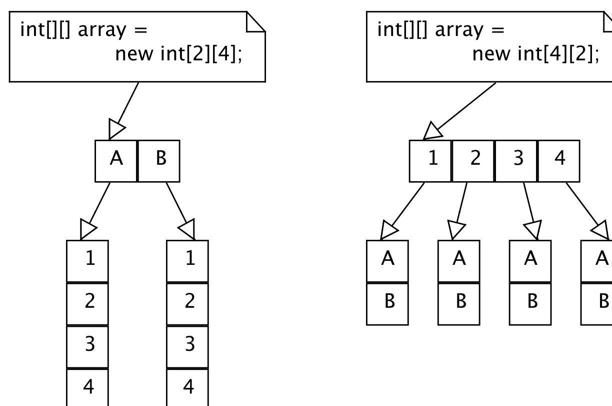


Figure 3.1: Illustration of how Java arranges two different definitions of a two-dimensional array in memory.

These findings extend to other ways in which we could structure our data, such as lists of lists – built using some kind of list implementation, such as the built-in Java `ArrayList`. It might also be tempting to create record objects, containing fields which hold dimension values and measures we want to access during aggregation, and storing them in some kind of collection data structure. However, this would again require dereferencing the objects first, most likely incurring an additional overhead.

3.3 The Storage Layer

I have decided to write a column-oriented storage layer from scratch. This option was competing with the alternative of using a freely available database to form the back-end. There were multiple reasons for this:

Simpler and Faster Interface. If I write the storage layer in e.g. Java, and the rest of the system is also written in Java, it will be much simpler and faster for the rest of the system to interface with it. Since many databases require communication via TCP/IP, providing database functionality via a tidy interface wrapping around another part of the code would eliminate the additional overhead caused by network communication. However, there are databases (for example H2) which have an embedded mode, skipping communicaton via TCP/IP, while also providing good integration with Java.

More Flexibility. Writing my own storage layer gives me the flexibility of deciding which algorithms and data structures to use. The developers of most available solutions have often already made these choices ("so you don't have to"), but this does not necessarily mean that they are the best choices for our particular use case. A simple example of this is that many DBMSs were written to handle an OLTP-like workload, not an OLAP-like one, and this may be hard to change and/or account for – for example, the types of indices provided may support efficient lookup, but not efficient scanning.

Deeper Understanding. By writing the storage layer myself, I can better understand and analyse what is really happening behind the scenes. For example, I will be able to better gauge how much memory a particular data structure is using, and why alternative ways of computing results yield different performance measurements. This can benefit the evaluation phase of the project. Additionally, implementing features such as e.g. bitmap indices myself provides a valuable learning experience, as opposed to just setting a property in some configuration file to enable this functionality.

Lack of an In-Memory Mode. The research prototype resulting from this project should store all of its data structures in the main memory of a machine. Most database management systems are written for on-disk operation only. We must also note that large parts of most on-disk DBMSs are written with a hard disk in mind – for example, most databases allow us to tune the database block size, and this generally needs to have some correspondence to the filesystem block size for best performance [38]. This is a strong hint suggesting that it will also not be possible to quickly and effortlessly adapt an existing on-disk database to use only main memory.

A strong candidate for use with this project was the freely available H2 database management system. It is written in Java, and as such can be managed from within a Java program – it offers an embedded mode, where communication with the database does not happen via TCP/IP, and is

thus faster. Stored procedures, written in Java, are also supported. H2 provides a dialect of SQL as its query language. Crucially, it also can run in an in-memory mode – just as required for the purposes of this project.

After discovering the reasons for why H2 would be a good idea for use with this project, I carried out some testing with this system. The experiments again loaded a randomly generated data set with few dimensions into the database, and I tried out different aggregation queries while measuring time performance and storage requirements. Query performance was fair – it was not excellent, but good when considering that this is presumably an OLTP-optimised system, and I could imagine such performance of the storage layer to be acceptable within the wider system I am attempting to build. However, I have noticed that the resulting database had high storage requirements – this was much more than storage of a comparable data set would require in a file. In a distributed system, this would translate into an increase of the number of nodes required to accomodate the data. I suspect that the poor space utilisation was due to additional data structures, especially indices, being created to improve the performance of lookups. When considering configuration options, I additionally discovered that H2 does not implement bitmap indexing. Given the popularity of bitmap indexing to support efficient scans, I wanted to have some form or another of this indexing scheme available in the implementation to explore and understand. Of course, I could modify the H2 source code to add support for bitmap indexing, but that would cause a lot of hassle, as I would need to familiarise myself with a large code base I did not write and then make changes to it, and it would likely not be a trivial task. Thus, I decided to implement my own storage layer.

3.4 Distributed Programming Frameworks

This section mirrors the one on distributed programming found in Chapter 2, and again contains a discussion of general distributed computing methods with reference to implementations of these, not specific ones directly related to the topic of this project.

3.4.1 Hadoop

The first idea I had for the implementation was to use the Hadoop distributed computing framework. It is a MapReduce framework, in the manner described in the background research chapter, and is one of the projects maintained by the Apache Foundation. On top of Hadoop, several other systems can be operated, including HadoopFS, a distributed filesystem with fault tolerance, and also HBase, a non-relational database modelled after BigTable as employed by Google.

To see how Hadoop could help us to build a distributed in-memory OLAP system, consider that most operating systems allow us to create a virtual RAM disk – a section of memory that will be treated as if it were just another storage medium such as a hard drive. The original idea I had for the project was to use Hadoop to coordinate distribution of data and computation, while making any systems we use (e.g. HadoopFS, HBase) store their data on a virtual RAM disk. This would allow us to achieve the goal of making the implementation entirely memory-based and distributed. Crucially, this may even allow us to use Hadoop as an infrastructure for the project, without making any modifications to the Hadoop codebase to add an in-memory mode.

However, there were some doubts over whether Hadoop is a good infrastructure for such work. Hadoop jobs seem to suffer from a high scheduling overhead, and thus the framework is probably

more suited to long-running jobs. In the OLAP realm, queries should ideally not take more than a few seconds to return results, with users favouring sub-second response times in the average case for obvious reasons – they lead to more interactivity and therefore less frustration. This property of Hadoop seems to impact systems such as Apache Hive, which aims to create a data warehousing infrastructure on top of Hadoop. The Hadoop wiki states that Hive jobs generally have a latency of minutes, even for small data sets. Although a part of this may be explained by the fact that the data is generally held on disk or that the data is not reorganised into a much cleverer form, the number one contributor to this lag seems to be Hadoop itself, and this idea is supported by the Hive developer community [11]. Therefore, I decided to reject Hadoop for the purposes of this project.

3.4.2 Remote Method Invocation

Remote Method Invocation (RMI) is the remote procedure call framework included with Java by default. The basic concept is that an object in one virtual machine can invoke a method of an object situated in another virtual machine. The framework is very flexible, and shows some of the advantages of dynamic linking in Java: it allows to e.g. transmit an absent class definition from another virtual machine [39].

In order for an object to offer remote methods for invocation, it has to implement a remote interface, which in turn has to extend the `java.rmi.Remote` interface. Additionally, each method appearing in this remote interface must be declared as throwing at least a `java.rmi.RemoteException`. When these constraints are met, an object implementing this remote interface can be exported. This is done using the RMI registry; it assigns locations to exported objects which can be accessed by clients.

Once a client has successfully imported a remote object, communication with the other end of the line is extremely simple. The client only needs to invoke the remote methods while also handling exceptions should they arise. Using such a remote object is therefore not very different from using a local one. Issues such as serialisation/deserialisation are handled automatically. Java also offers the additional keyword `transient`, which when used in a field definition of a class will prevent this field from being serialised/deserialised [41]. This may be required in some situations where serialisation of a certain field is not desirable, as would be the case with e.g. file handles or database connections.

Prior to starting this project, I already had some experience with remote procedure call mechanisms and also with RMI in particular. Although the examples I worked on before with RMI were rather small, I was rather confident that I understand the main concepts behind it and will know how to use them effectively while implementing the research prototype. As such, I have decided to base the implementation on RMI, as there are overwhelming advantages in using this approach.

3.5 Data Distribution

As discussed in the background research chapter, there are several ways in which we can distribute our data across nodes. Here I will, with reference to my own implementation attempts, outline their advantages and disadvantages, and explain the way in which the research prototype resulting from the implementation phase of this project distributes data.

3.5.1 Data Warehouse Striping

I initially decided to make the system use Data Warehouse Striping. In this scheme, the fact table of a star scheme is horizontally partitioned, while each node in the system replicates all dimension tables in their entirety. At the beginning, this seemed to be a very good scheme – each node would have the full dimension data available, and would perform a star join operation to compute answers to queries. Additionally, the memory usage on each node was not too large with my initial, low-dimensional data set which I used for testing purposes during programming. However, problems became clear when I started to use a benchmark for OLAP systems. The particular benchmark I have used in this project provides developers with very large dimension tables – their size grows together with the size of the fact data, depending on some developer-specified scale factor. For the more challenging and interesting scale factors, most of these dimension tables will have entries numbering in the millions. Thus, at larger scale factors, the ratio of fact table data to dimension data on a worker node would decrease dramatically. Essentially, a lot of space would be wasted to replicate dimensions on each node, negatively impacting scalability. After a back-of-the-envelope calculation analysing the number of nodes required for my target scale factor, I decided to reject this approach, as the amount of hardware resources needed would have been prohibitive.

3.5.2 Selective Loading

With Data Warehouse Striping not bringing the expected results, I decided to try Selective Loading instead. In this scheme, the dimension tables on a node are only loaded with an entry if the fact data references it. Selective Loading was not hard to implement on top of the existing structures used for Data Warehouse Striping, as the two techniques are very similar. The main change was that instead of transmitting all dimension tables at the end, I would simply transmit the associated dimension data together with each set of records during bulk loading. This additional data would then be used to update the dimension tables on a worker node. Although this increased bulk loading times slightly, the scheme successfully avoided storing irrelevant dimension data on nodes.

I again repeated my trials with a more complex data set (the benchmark). Although Selective Loading helped to reduce the overall memory usage on nodes, mainly through reducing the size of large dimensions (small dimensions would usually have all of their entries on each node), it still did not scale well enough. Part of the reason was that even though the worker nodes would not store all of the dimension data, a large share of it would still be needed on each of them. This share would increase with larger data sets: the chosen benchmark does not only increase fact table data, it also grows dimension tables at the same time, and uses the new entries in the fact table. Although one would expect that for e.g. a system with two nodes, only half of the dimension data would be stored on each of them, this is not the case; in such a case, more than half of the data will be stored. However, the main problem was that the alphanumeric values of the dimension members

had to be copied onto each node: while the fact table would generally contain (numerical) foreign keys to the dimension tables, the values themselves had to be stored somewhere. Additionally, in order to ensure fast lookup, some kind of indexing scheme had to be used – I used one based on a hash-table. However, such indexing schemes will always increase the overall storage requirements by some amount. Additionally, it seems that sometimes there was a significant overhead in dealing with the dimension tables while scanning the fact table.

3.5.3 My Chosen Approach

The above findings about Selective Loading gave me another idea: since alphanumeric strings take up so much space, why waste even more by replicating them? We could just as well store them centrally on the master node. Additionally, I would store all data as a flat table, while allowing some storage space optimisation in cases where this brings benefits.

In this approach, I would first convert the original data (it could e.g. be a star schema) into a flat table. The reason for this is that certain dimension tables are only large because of the presence of one or more high-cardinality columns. For example, in the benchmark mentioned before, the decision was taken to store the country of residence of a customer together with their name; while there are only 25 possible countries of residence, there could potentially be hundreds of thousands, if not millions of customers, each with distinct names. However, if we put these together, then we need to additionally store the country column, with a lot of repetition. Splitting out the country column and storing it separately will not require dealing with this dimension table when all we want is to know what the country of residence of a customer is. While this increases the storage requirements of the fact table (we have to store an additional column), the research prototype resulting from this project supports more than just blind flattening of the data: lookup tables, which are similar in nature and function to dimension tables, can still be transmitted to a worker node; then, a "virtual" column in the table can be derived from this lookup table and only one additional column of the fact table. Using such lookup tables is beneficial with small dimensions, as the resulting dimension table will still be comparatively small. This mechanism, called post-processing, will be explained in depth in Chapter 4, but for now the crucial finding is that it can improve space usage. Now, whether the fact table actually stores it or whether it is derived, each attribute which was stored in a dimension table before is now logically part of a single table – the fact table. Thus, dealing with large dimension tables is no longer an issue.

The second important part of this approach is that the alphanumeric dimension data is only stored centrally on the master node. Each dimension member is assigned a numerical key, which is then stored on the worker node. Aggregation now works as an enrichment process: filter values of a query are annotated with the keys of the dimension members they permit before being sent to worker nodes, and the results we receive back consist only of numbers – they have to be enriched with the labels of the actual dimension members. This yields the same results, but without storing the huge amount of alphanumeric data on each worker node that we needed before to process queries. Since the data is now stored as a flat table, we can still perform aggregation, classification and filtering, but using the numerical keys only, and without any joins.

A minor point is that since now the columns in the fact table will be frequently low in cardinality, the size of each entry can be specified – for instance, while a `byte` variable can store less different values than a `short`, it will often be sufficient, and require less space.

3.6 Storage Layer Optimisations

In this section, I will go into more detail about possible optimisations which can be applied to the storage layer. In particular, this section is about run-length encoded bitmap indices, and how sorting can be implemented to improve their performance.

3.6.1 Run-Length Encoded Bitmap Indices

As we have seen in the background research chapter, bitmap indices are frequently used when full scans of the data set are required, as they allow us to skip over many records which do not match the restrictions provided in a query. Run-length encoding schemes can be applied to bitmap indices to improve their performance. Sorting improves run-length encoded bitmap indices, as a long chain of zeroes will generally result in less space usage, and we can therefore detect its length with less computational cost than just inspecting every single position in the index or perhaps checking whether a bit-wise operation indicates that each of the bits consists of zeroes – we can generally just use the run-length to determine whether a section of the bitmap can be skipped.

Thus, I have decided to use run-length encoding of bitmap indices, created from potentially pre-sorted columns, to increase scan speed where possible. I chose a library called JavaEWAH [43] to implement bitmap indexing, also used in the Hive data warehousing system and based on the Enhanced Word-Aligned Hybrid (EWAH) compression scheme discussed in the background research chapter.

3.6.2 Sorting in Low-Memory Conditions

Since sorting seems to improve the performance of run-length encoding schemes, such as ones for bitmap indices, it would be quite helpful to offer the option to sort the fact table. Unfortunately, after loading a large amount of data into the fact table of a worker node, this worker node will likely not have much free space at its disposal anymore; therefore, it is helpful to use a sorting algorithm which crucially only uses a small amount of additional space to perform the operations. Sorting algorithms which fit this description are called *in-place*: only a constant amount of memory is used outside of the data structure to be sorted, i.e. it does not depend on the size of the input.

I could not use standard Java implementations such as `java.util.Collections.sort`, as these generally expect the data structure to follow a particular format. According to the definitions given in *Introduction to Algorithms* [48], a data structure to be sorted consists of *records*, which have a *key* according to which we want to sort the data, and some *satellite data* which is irrelevant to the sorting. Now, many Java implementations want the data structure holding the records to implement `java.util.List`, while the records themselves should abstract key comparisons by implementing the `Comparable` interface. Although a custom data structure such as the fact table could be brought into this form, it is probably easier to implement a sorting algorithm ourselves.

The first algorithm I tried out, mainly due to the ease of implementation and because it satisfies the in-place sorting property was bubble sort. The algorithm works by repeatedly scanning the list to be sorted, and swapping adjacent items which are in the wrong order. This process stops when the list is ordered. Although it allowed me to sort small fact tables for experimentation, this did

not scale well to larger amounts of data, since bubble sort has an average case time performance of $O(n^2)$.

Thus, I decided to use heap sort instead. Heap sort is again an in-place sorting algorithm: the amount of additional memory used is constant. Heap sort sorts data by converting the input data structure into a heap – note that this is a type of data structure which can be viewed as an almost complete binary tree, and *not* a heap in the sense of "garbage collected storage" like the Java heap. Its worst case time performance is $O(n \log n)$, which is acceptable, as the $\log n$ part grows at a much slower rate than the input size, while its space usage is $O(1)$ (constant). Thus, heap sort allows us to sort a data structure with little additional memory usage while still being reasonably fast.

Chapter 4

Simian: A Scalable In-Memory Aggregator

In this chapter, I will introduce the research prototype (called Simian) built as part of this project. The implementation specifics presented in this chapter are the result of the background research in Chapter 2 and the design choices of Chapter 3.

4.1 Architecture

Simian is not a stand-alone application; rather, it acts as a framework, allowing an application developer who wishes to create an analytics application for a particular data set to do so with little additional code (for the remainder of the report, this user will be referred to as **the developer**). Being a distributed system, the resulting application runs on a number of interacting nodes. These nodes are sub-divided into one master node and one or more worker nodes. The master node handles a diverse range of functions, starting with coordination of the initial bulk loading process, parts of which can be developer-defined and generally depend on the data set which is meant to be analysed with the application being implemented using the framework. Later, the master node also has the responsibility of storing the dimension data. Queries can be submitted to the master node via the user interface; these are parsed and then mapped onto worker nodes. Once results are received, these are combined and enriched with dimension data – a process which will be explained in more detail later. The master node is also responsible for instructing the worker nodes which aggregate tables to use, if any. A worker node, on the other hand, has the primary responsibilities of storing fact data and responding to aggregation requests by carrying out the necessary computation. Depending on the settings defined by the developer, a worker node can also store aggregate tables (precomputed aggregates for queries involving a predefined set of dimensions) and bitmap indices, which it will use during the aggregation process to substantially speed up computation.

The user interface is web-based and by default allows queries to be entered, while also providing information about the state of the system. Developer-defined pages can be added which generally will submit a query to the master node, displaying the results in some manner the developer considers suitable. This can be used by a developer to create, for example, a dashboard for the users. The user interface also allows to start a bulk load and to follow its progress.

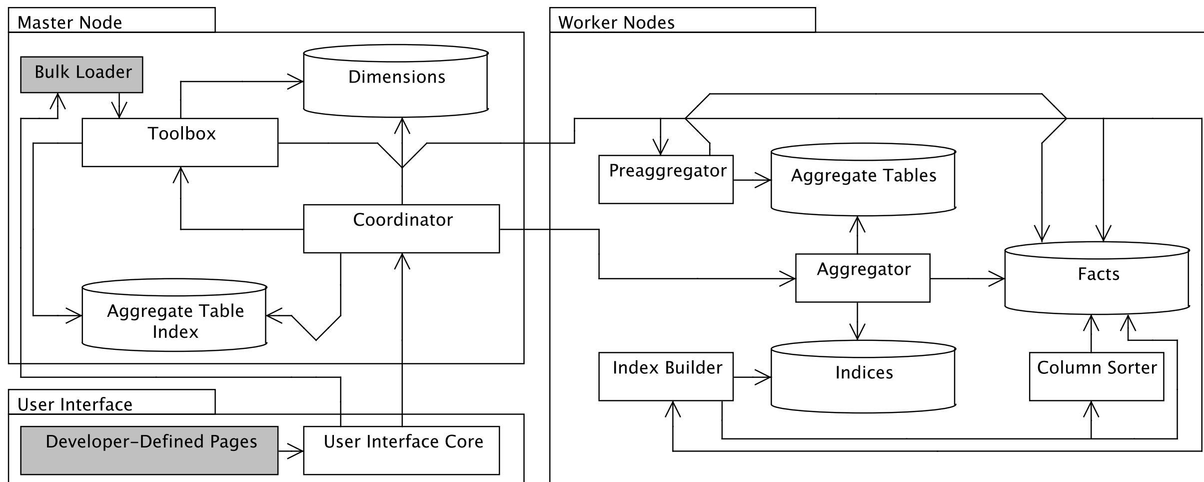


Figure 4.1: An illustration of how different components of Simian interact with each other. Grey parts are either completely or mainly developer-defined.

Figure 4.1 shows a diagram of the architecture. The components will be explained in detail in this chapter; arrows indicate that one component invokes functions on the component which is pointed to, or reads/modifies it. Simian is a framework, and as such for a particular application, additional code needs to be defined to take care of bulk loading and setting up the schema, and a developer is provided with tools to accomplish this task.

I will now give a brief overview of the system while explaining the diagram above. Dimensions represent a mapping from alphanumeric values to keys, and vice versa; they are only stored on the master nodes. Worker nodes primarily store the fact table data, which only consists of a table made up of those keys. Also, some dimensions and measures in the fact table can be derived using a feature called post-processing which will be explained later. Indices consist of compressed bitmaps and can speed up fact table scans for some query types; the column sorter can sort the fact table to improve the performance of those bitmap indices. Aggregate tables are pre-computed views on the fact table at a coarser level of detail; they similarly allow faster processing of queries and are stored on the worker nodes. The aggregate table index on the master node tracks the whereabouts of the aggregate tables. The coordinator is responsible for a diverse range of functions such as starting a bulk load, handling the aggregation process (receiving queries, passing them on to worker nodes, combining the results), and managing the dimension tables. The aggregator responds to aggregation requests and produces a partial result which is returned to the coordinator on the master node. A developer has to define a bulk loader which fills the system with data using the toolbox – it provides a wide range of functionality needed for this purpose, such as easy creation of requests to compute aggregate tables and indices. However, the toolbox is also used by other, built-in system components.

4.2 Data Organisation and Bulk Loading

This section will explain how data is stored in a Simian system and how it can be loaded with the data to be analysed.

Figure 4.2: An illustration of the conversion process, by which the original (alphanumeric) dimension attributes of the source data are converted into a purely numerical form to be stored on worker nodes.

4.2.1 Fact Data

Fact data is stored on the worker nodes in one large table, arranged in a column-oriented manner as explained in previous chapters. The way this works in Simian is that the original data is turned into a flat table by the bulk load process (if it was not in such a form to begin with), and then converted into a numerical form, for the reasons explained earlier in Chapter 3. This means that the (alphanumeric) dimension members referenced by each fact record are represented by some numerical identifier. This process is illustrated in Figure 4.2. There are several column types which can be configured, based on the size of one entry. Small columns can be represented with a `BYTE`-sized column. Columns whose identifiers cannot all fit into the range of a `BYTE` column can be stored in a `SHORT` column instead. For the largest columns, `INTEGER` columns can be used. This potentially saves a lot of space when there is no reason to use a larger column: for example, most companies divide their business into less than ten geographical areas, and the corresponding dimension data can be stored in a `BYTE` column. Additionally, Simian introduces the `DERIVED` type, representing a "virtual" column which is the result of post-processing. A `DERIVED` column can represent the same range as an `INTEGER` column. Apart from any additional objects which could be used to help with post-processing, this type of column does not take up any space.

Measures only have one possible type: a 64-bit integer representation, stored as the Java primitive type `long`. This can already encode most measures which are interesting to analyse, such as currency. This may seem peculiar – after all, one may argue, prices in many major currencies are generally fractional (e.g. 19.95 pounds) and therefore should be stored in floating point data types. However, this intuition is wrong, as floating point numbers would suffer from loss of precision, which is a problem when handling such data – aggregating sums of money is fundamentally a different class of problem than e.g. calculating the area of a circle. Therefore, in the Java realm, currency is generally stored either in scaled `long` variables or using the arbitrary precision `BigDecimal` floating point type. When using the `long` data type, the values will be stored without the point which e.g. separates pounds and pennies; when the value is about to be output, it will be decorated with the "missing point" to be more visually intuitive to the user.

4.2.2 Dimension Data

Each column of the fact table stored on worker nodes in a Simian system will consist of numerical values which have some kind of alphanumeric correspondent. For example, a column with year data may store the value 7, but this will actually correspond to an alphanumeric label of "2007".

Translating such keys to their alphanumeric correspondents and vice versa is the task of dimension tables. Unlike the dimension tables as used in star schemas, these are much simpler in Simian: there is no way for a dimension table to store values for multiple attributes, as is the case with traditional dimension tables which for example would represent a concept hierarchy. Each dimension table in Simian is a mapping from the key values used in exactly one column to their alphanumeric correspondents and vice versa. This is a consequence of the denormalised form in which data is stored in this research prototype. Dimension tables are stored centrally on the master node, and are not copied onto the worker nodes for space reasons.

Dimension members are assigned consecutively numbered key values using a counter, which is increased after each insertion into the dimension table. The dimension tables themselves are implemented using two maps: one from key values to their alphanumeric equivalents, and another map which maps the alphanumeric equivalents back to their respective key values.

4.2.3 The Bulk Load Process

A developer has to define a bulk load process – this is a class which implements a particular interface, and an object of this type is passed into the master node by the developer. Initially, the bulk load process has to inform the master node of several pieces of key information it needs to initialise the system: this includes the layout of the data (called the *schema*: dimension names and sizes, measure names), the addresses of worker nodes and the possible maximum size which should be reserved for the database. However, tools are provided to the developer to accomplish this task: there is a parser which automatically converts a configuration file into a schema of the master node, and another parser which can read a network configuration to be passed to the master node, again from a configuration file.

In the next step, the bulk load process has to load records into the system while also populating the dimension tables. Records which are sent to the master node are distributed across nodes in a round robin fashion, with a window size controlled by the bulk load process. This is generally the longest part of the procedure, as large amounts of data are loaded into the system.

After the fact and dimension data has been loaded into the system, the developer has to pass any post-processing dimensions or measures defined in the schema (which have to implement a particular interface) to the master node, along with any objects (such as lookup tables) they may require. The master node injects these into each worker node.

At this stage, the developer may specify which (if any) indices should be computed, along with general properties of the indices which will be explained later, such as whether the fact table should be sorted at all, which ordering is to be used while sorting, and some other parameters which will be used while computing query results. The developer then has to close the master node; this will seal the fact table, preventing any further insertions, and bitmap indices will be created as specified by the developer (after sorting, if requested). After this step has been completed, the developer can finally supply some aggregations which should be pre-computed in the form of aggregate tables. For both indices and aggregate tables, again standard parsers can be used to read in the necessary configuration files. Once the bulk load process has finished, the bulk loader object is dereferenced and the Java garbage collector is invoked to clear the often substantial amounts of intermediate objects which were created by the bulk loader. The system is now ready to accept queries.

4.3 Aggregation

In this section, I will first introduce the basic query language used to describe aggregation requests in Simian, and will then go deeper into what happens behind the scenes when results to such queries are computed.

4.3.1 Query Language

Listing 4.1: Example of a Simian query.

```
DIMENSIONS:
  OrderYear
  CustomerNation
  PartColor

CALCULATIONS:
  Revenue.SUM
  SupplyCost.SUM
  Profit.SUM

FILTERS:
  CustomerRegion += ["ASIA"]
  PartColor += ["navy" "firebrick"]
```

The simple query language provided by Simian splits the aggregated view the user wants to get on the data into three sections: the `DIMENSIONS` section specifies which dimensions should show up in the output to categorise each aggregated value. The `CALCULATIONS` part specifies combinations of measures and aggregation functions which should be calculated for each combination of dimension members. Four aggregation functions are supported: `SUM`, `MIN`, `MAX` and `AVG`. Finally, the `FILTERS` section contains allowed values for each specified dimension – whether it appears explicitly in the `DIMENSIONS` part or not.

Listing 4.1 shows an example query expressed in this language. It categorises aggregated revenue, supply cost and profit from a fact data set of ordered items by the dimensions `OrderYear`, `CustomerNation` and `PartColor`, while specifying that `CustomerRegion` must be "ASIA" and the `PartColor` has to be either "navy" or "firebrick". Apart from the expression `+=`, which allows the user to specify a set of values which are allowed for a dimension, the alternative expression `[-]` exists, which allows the user to specify values for members which correspond to numbers; for instance, `OrderYear [-] 2006 2009` is equivalent to `OrderYear += ["2006" "2007" "2008" "2009"]`.

The query language used in Simian allows users to create queries which are semantically equivalent to SQL `SELECT ... WHERE ... GROUP BY ...` queries, with the `DIMENSIONS` part representing columns to be projected, the `CALCULATIONS` part representing ones to be aggregated and specifying the aggregation function, while the `FILTERS` part represents the permitted values for columns in `WHERE`. A full summary of the query language is given in Appendix A.

4.3.2 The Aggregation Process

In order to perform aggregation, a query has to be submitted to the master node. This is typically done via the user interface; the user can either type in a query directly, or one of the developer-defined pages does this as part of the process of building the page. The description given here of the aggregation process does not include the special cases of using auxiliary data structures such as aggregate tables and indices, instead focusing on the basics.

Algorithm 1 The aggregation algorithm used by the worker nodes.

```
result ← new_map()
for i = 0 → size(fact_table) - 1 do
  passes_filtering ← true
  for all column ∈ keys(query.filters) do
    allowed_values ← get(query.filters, column)
    value ← get_value(fact_table, column, i)
    if !contains(allowed_values, value) then
      passes_filtering ← false
      break
    end if
  end for
  if !passes_filtering then
    next
  else
    for all (measure, function) ∈ query.calculations do
      dimension_values ← get_values(fact_table, query.dimensions, i)
      key ← append(dimension_values, [measure, function])
      value_at_record ← get_measure(fact_table, measure, i)
      if contains(result, key) then
        current_value ← get(result, key)
        new_value ← aggregate(function, current_value, value_at_record)
        put(result, key, new_value)
      else
        new_value ← initial_value(function, value_at_record)
        put(result, key, new_value)
      end if
    end for
  end if
end for
return result
```

The text form of the query is then compiled by the master node into a format suitable for the worker nodes. A query will contain references to alphanumeric names of dimensions, measures, and dimension members, all of which the worker nodes have no concept of, as the stored fact data solely consists of numerical values. Thus, the compiled form of a query contains indices of dimensions and measures, as opposed to their full, alphanumeric names. For filtering purposes, hash sets are included which contain allowed keys for dimensions the user wishes to filter. Filtering on a column thus consists of checking whether the value at the current record is contained in the set.

Algorithm 2 The algorithm used by the master node to combine results.

```
result ← result_a
for all key ∈ keys(result_b) do
  if contains(result_a, key) then
    function ← key[length(key)-1]
    value_a ← get(result_a, key)
    value_b ← get(result_b, key)
    put(result, key, combine(function, value_a, value_b))
  else
    value_b ← get(result_b, key)
    put(result, key, value_b)
  end if
end for
return result
```

Once the worker nodes receive the compiled query, they run Algorithm 1 (given here in pseudo-code form) to calculate a partial result. The arguments to this algorithm are *fact_table*, which just represents a reference to the fact table, and *query*, a structure representing the compiled query with three fields. The field *filters* is a mapping of the form $column \longmapsto allowed_values$, where *column* is the index of a column and *allowed_values* is a set of allowed values. The field *dimensions* represents the dimensions by which the result should be categorised. Finally, the field *calculations* consists of a list of tuples, all of which have two elements and are of the form $(measure, function)$, representing calculations which the user is interested in for the chosen dimensions. The algorithm returns *result*, which is a map from aggregation keys to aggregated values, of the form $key \longmapsto value$. An aggregration key is an array of integers: for an aggregation key of length N, the first $N-2$ entries contain the keys of the members of the dimensions specified in the query, excluding any dimensions the user only wants to filter through without displaying them in the final result. At each record, the algorithm consists of two parts: in the first part, it checks if the current record passes filtering constraints – if it does, the second part will update the result map to take this record into account. Once the worker node has iterated over all records of the entire data set, it will return the result to the master node. Note that this is a simplified version of the actual routine that was implemented: in practice, the worker nodes do a lot of switching between the four types of columns, each of which uses a different data type to be represented for space reasons. Also note that this does not include the actions needed to use indices during a scan. Multiple cores of a machine are utilised by running multiple virtual machines, each hosting a worker node, on that machine. The operating system will then take care of the scheduling, and usually one core will be used to run the aggregation algorithm of one worker node.

Once all results have been received by the master node, it will combine them by using Algorithm 2. This algorithm takes two inputs *result_a* and *result_b*, both of which are result maps as before, and returns *result*, which is the outcome of combining both input arguments. The implementation of this algorithm will be applied repeatedly until all results are reduced into one overall result.

At this stage, the master node has a complete result. However, it still contains only numerical index values for the dimensions, measures and functions. This obviously cannot be presented to the user in this form yet. Thus, the master node creates a so-called *tablet*: this is a list of string arrays. For each key in the result map, a row will be added, containing the dimension members together with the measure name, aggregation function applied and the result, as originally requested in the

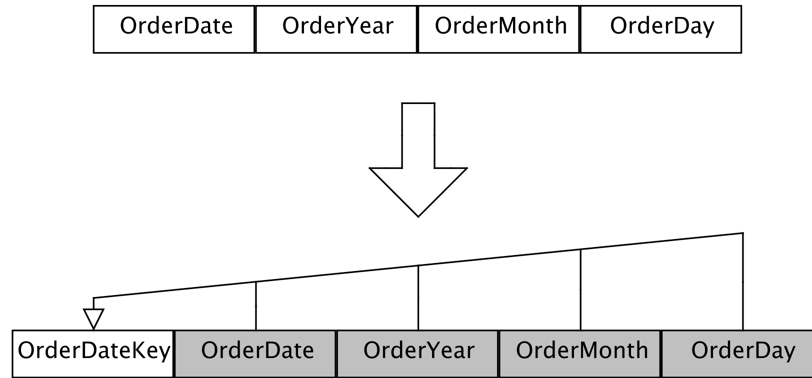


Figure 4.3: Post-processing: the columns at the top were all originally fully stored in the fact table. After post-processing has been introduced, they are derived from the column `OrderDateKey` and do not need to be stored.

query. This tablet is computed by integrating the result map with the alphanumeric equivalents of the dimension keys retrieved from the dimension tables as stored on the master node. This tablet is returned either to the user interface, to be rendered as a table, or to the handler for the developer-defined page which requested the aggregates for further processing.

4.4 Post-Processing

As mentioned before, post-processing is an important feature which can help us save large amounts of space. Simian has two types of post-processing: one for dimensions, and one for measures.

4.4.1 Derived Dimensions

If a dimension is of type `DERIVED`, Simian expects the bulk load process to provide a definition of this derivation, which is an object implementing an interface (called `PostProcessingDimension`) which allows the aggregation routine to interact with it. Three methods must be implemented: `getValue` receives an integer argument representing the row for which we want the value of this dimension to be computed, and a fact table reference which allows us to retrieve values of other dimensions for that row. The method `getSize` should return whether the results of the post-processing step can be stored in columns of size `BYTE`, `SHORT` or `INTEGER`. This is used later when aggregate tables are computed, for space efficiency reasons. Finally, `fetchPostProcessingObjects` retrieves any relevant objects, such as lookup tables, from the post-processing registry; this is called once on the worker node when it is closed. The post-processing registry stores any objects which are needed during post-processing, such as lookup tables, and is populated by the developer during the bulk load process. The reason for having the post-processing registry, as opposed to just making any auxiliary objects part of the dimension object is that some such objects could be used by multiple post-processing dimensions. When the aggregation routine then needs the value of a post-processing dimension for a particular row, it will just invoke `getValue` and use the result that was returned.

The reason to have post-processing dimensions is to reduce the space usage of dimensions which can be easily derived from others. For example, if we store a full date in a dimension, we can easily derive aspects of a date such as the year, month, day, or for example a combination of year and

month (e.g. `"Dec2010"`) by using this as a key for the lookup table, as shown in Figure 4.3. Many applications require a large amount of such time attributes, which can easily be derived from the full date; not storing all of them directly saves large amounts of space. We could alternatively view this as enabling us to use the Data Warehouse Striping partitioning scheme only where it is feasible: for relatively small dimensions. We can replicate small dimension tables with several columns (such as a listing of dates and their various derived attributes) onto the worker nodes to decrease space usage, while still denormalising other, large dimension tables. However, note that while the lookup tables themselves are not unlike dimension tables, they only store a mapping from a key, e.g. the `OrderDateKey` in Figure 4.3, to arrays of keys for the derived dimensions – this allows us to get the relevant keys for dimensions such as `OrderYear` and `OrderMonth` from the `OrderDateKey`, in this example. They do not contain the actual data, such as e.g. the name of a month. Dealing with this type of information is still the responsibility of the master node.

4.4.2 Derived Measures

We can also define a measure to be derived. The process is essentially the same as with derived dimensions: an object is injected into the worker nodes which implements the `PostProcessingMeasure` interface. This is similar to the `PostProcessingDimension` interface, except that the `getSize` method is dropped, as numerical aggregate values in aggregate tables are still stored using the `long` type, while more complex values such as averages follow their own format anyway (e.g. a tuple of sum and count for the case of the average function).

A common use of post-processing measures is to calculate measures such as profit from other measures – this can usually be done by subtracting a (production and/or supply) cost measure from a revenue measure. The default Simian `SubtractionMeasure` can be used for this purpose. Also, they are used to scale an existing measure by the value of some dimension – for example, we could calculate a promotional savings measure by multiplying the percentage discount with the nominal price of an item.

An additional post-processing measure included with Simian by default is the `Contributors` measure. Including `Contributors.SUM` in the list of measure/function combinations to be calculated will show how many records have contributed to a particular result. Thus, for example, if we want to aggregate only records where the order year is 2007, and there are 100,000,000 such records, then the value of `Contributors.SUM` will be 100,000,000.

4.5 Aggregate Tables

This section describes aggregate tables, a form of precomputation supported by Simian which can significantly improve query response times in certain cases.

Simian allows queries for certain combinations of dimensions to be precomputed. For example, we may be interested in precomputing a query which uses the dimensions `CustomerNation`, `SupplierNation` and `OrderYear`. It should not matter whether the dimensions are used in the `DIMENSIONS` or `FILTERS` part of the query – all queries using these dimensions in either of these parts should be responded to by using such an auxiliary data structure in the case that it has been set up.

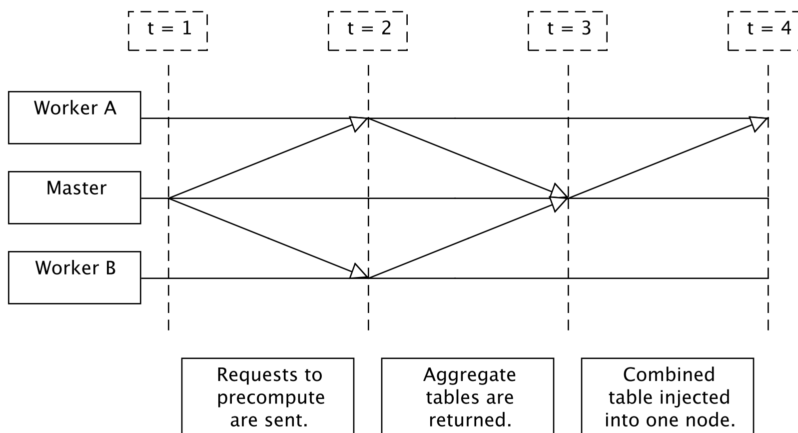


Figure 4.4: Message passing while computing a centralised aggregate table. The aggregate tables from both worker nodes are combined, then sent to be stored at worker A only. For local aggregate tables, the process stops after computing the tables at $t = 2$, and each worker node stores the table locally.

A developer can define a configuration file listing the combinations of dimensions to be precomputed. As with many other developer-defined aspects, a standard parser is available to read these into the master node. This happens after the fact and dimension data has been loaded into the system.

A Simian aggregate table is very much like a fact table. It contains columns corresponding to the combination of dimensions it represents, and instead of measures it stores all possible combinations of measures and functions in the original fact table – for each row aggregated by the combination of dimensions it represents. Answering a query from an aggregate table is thus not very different from answering it from a fact table, and the underlying algorithm is almost the same as Algorithm 1, except that the (already pre-aggregated) measures are handled differently – the appropriate measure aggregated with the requested function has to be selected among the columns.

Aggregate tables are always stored on worker nodes. However, there are two different ways in which aggregate tables can be stored. *Local* aggregate tables are the result of the master node sending a command to precompute a certain combination of dimensions to each worker node. After the precomputation process has finished, the worker node just stores the aggregate table locally. *Centralised* aggregate tables are different in that after precomputation has finished, the aggregate tables are sent back to the master node. The master node then combines these tables in a similar manner to the way partial results are combined during the aggregation process. Once the master node has integrated all aggregate tables, it finds the worker node with the lowest memory consumption, and injects the aggregate table into said worker node. Centralised aggregate tables have lower space requirements than local ones, as generally many combinations of dimension members are shared between the equivalent local aggregate tables on each node; thus, by integrating all of these into one aggregate table and storing it on only one node, usually a lot of space is saved. The entire process of computing a centralised aggregate table is illustrated in Figure 4.4.

When the master node receives a request to precompute a number of dimension combinations, it proceeds in descending order of the number of dimensions in each such combination; this means that the combinations with the largest number of dimensions are calculated first, with this ordering continuing until the smallest combinations are precomputed.

The use of aggregate tables to answer queries is centrally coordinated by the master node – for a

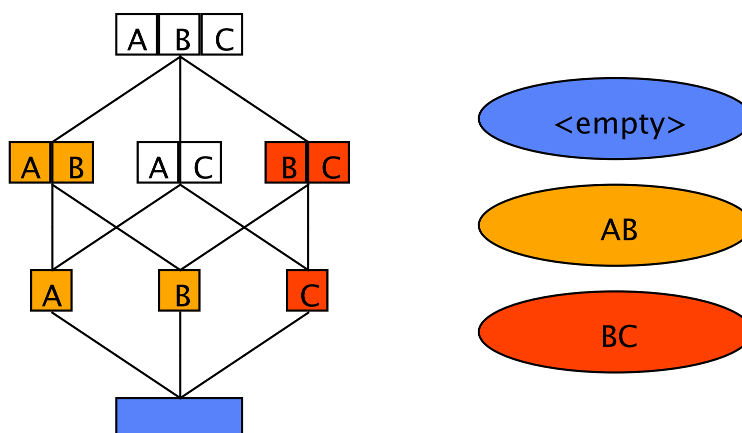


Figure 4.5: Simian will store a data structure like this to find the appropriate aggregate table to use. The legend on the right shows which colour in the lattice of aggregations corresponds to which pre-computed aggregation that is stored. Blue corresponds to the aggregation with an empty list of dimensions.

query for which a suitable aggregate table exists, the worker nodes are instructed to use it. The master node detects the presence of a suitable aggregate table by storing a fragment of the lattice of all possible aggregations as introduced in the background research chapter. Each point in the fragment is pointing to the identifier of an appropriate aggregate table to be used for this particular combination. To answer queries efficiently, each time a new aggregate table has been computed, the associated point in the fragment is made to point to this aggregate table, along with all other aggregations below this point in the hierarchy. This data structure is illustrated in Figure 4.5. As points lower in the hierarchy will result in a smaller aggregate tables, the scheme of computing the largest combinations first in combination with the maintenance mechanism of the data structure is a good heuristic for assigning smaller aggregate tables to simpler aggregations – resulting in faster query response times. The master node also stores an additional data structure which, for centralised aggregate tables, indicates the worker node that contains it. Thus, in this case it is sufficient to send an aggregation request to only this single node. If there is no information on which node stores the table, this is a local aggregate table; thus, requests are sent to all nodes, and the received partial results are combined. In the architecture diagram presented before, these two structures make up the aggregate table index.

4.6 Indices

Simian supports bitmap indexing. For this purpose, the JavaEWAH library is used, also utilised by the Apache Hive data warehousing system to implement the same feature. JavaEWAH provides a compressed bitmap data structure, which Simian uses as the base of the bitmap index. After all dimension and fact data has been loaded into the system, Simian can be instructed to compute bitmap indices for selected columns. The process can again be simplified by using a standard configuration file which can be automatically parsed.

Simian will then, on each worker node, create one bitmap for every value in each of the specified columns to be stored locally. So, for instance, if we have seven values in the `OrderYear` dimension, the index on that column will consist of seven bitmaps.

Since EWAH is a run-length encoding scheme, it can benefit from very large run-lengths, both to decrease total storage requirements and to improve the performance of a scan to find bit positions which are set. Thus, Simian offers a developer the opportunity to indicate whether sorting should occur prior to index creation and which columns should be sorted with which ordering in the index configuration file. The ordering which can be specified is lexicographic in nature, and the developer can express which columns should take precedence. Originally, I tried orderings based on cardinality – however, I decided that it is best to specify a custom ordering, to also reflect the relative "importance" of columns. For example, if lower cardinality columns take precedence, then it is possible that some low-cardinality column (for example region) will be assigned higher precedence than a dimension with a slightly higher cardinality expressing time, e.g. the year an order was made. However, it could be the case that the former is only sometimes used in queries, while most queries use the latter; thus, the time column should be given precedence in the ordering, to increase the run-length and potentially improve scan times when this column is used as a filter.

When a query involves a filter over one more columns which are indexed, a very basic greedy algorithm is used to decide whether to use the indices, and how to use them. The reason for this is that for certain queries, using bitmap indexing could actually harm performance, as the overhead in scanning the indices would be higher than the time savings gained. The greedy algorithm uses two constants to determine the indices to be used, both of which can also be configured using the configuration file:

`MAX_PROBABILITY`**.** This is the maximum probability the values specified in a filter must have for the bitmap index to be considered. The default setting is 50%. For instance, if we have a **uniformly distributed** dimension of cardinality ten, and two values are specified for inclusion in the filter, then the probability will be more or less 20%. However, if six values are specified for inclusion in the filter, the probability will rise to around 60%, and use of this bitmap index will be rejected for this query if using the default setting. Of course, since dimensions are not always uniformly distributed as in this example, the probability distributions are calculated by Simian before bitmap indices are created.

`MAX_OPERATIONS`**.** This specifies the maximum number of *operations* which can be performed between all bitmaps we may select for a particular query. The default setting is 5. For example, if a filter restricts one column to only one value, we get this bitmap index for free, as no additional operations have to be performed; in the case of two values, an `OR` operation has to be carried out. To combine such generated bitmaps for two columns, we have to carry out an `AND` operation. Clearly, if too many such operations are performed, this will result in a too large overhead to give us any advantages.

With these settings, the greedy algorithm now calculates the probability of the filter values for each respective column. It then initialises a counter to `MAX_OPERATIONS`, and tries to find the mimimum-probability column whose probability is also still below `MAX_PROBABILITY` and for which the required total number of operations is less than the counter value. If such a column is found, then it is added to the list of columns whose indices are used, and we try to find such a column again, excluding any columns we have already found in the search; otherwise, the search concludes. If a set of columns satisfying such criteria is found, a composite index is calculated, and then used by the aggregation routine to quickly skip non-matching columns.

4.7 Contrasting Aggregate Tables with Indices

Generally, it is feasible to pre-compute combinations of dimensions which will result in rather small aggregate tables. This is often useful for queries which will result in small result sets, but will have to consider every single row of the data set during computation. For example, a user might be interested in receiving a table of profit by region and year, without any restrictions on both attributes. Computing this by scanning the fact table will be time-consuming: while restrictions (even without indexing) can be used to quickly reject records not matching the description, it takes a comparatively long time to update the result map if a matching record is found. If now all records match the description due to the lack of filters, the map of aggregates will be updated at every single row in the fact table. However, pre-computing such a combination of dimensions reduces the number of rows to be considered dramatically, and is often feasible for such "high-level" queries: for example, assuming that there are ten regions and ten years, this would yield an aggregate table of one hundred rows. Bitmap indices cannot help in this case.

However, when the user starts drilling down to a finer level of detail, bitmap indices often start to be useful. For instance, the same user might decide they are interested in countries in East Asia only – thus, they will restrict the region to "East Asia" and add country to the list of dimensions. Having a bitmap index on region will allow us to quickly skip all records which do not have the region attribute set to "East Asia".

We can thus conclude that while aggregate tables help with high-level, coarse granularity queries which match many or all rows, bitmap indices in turn help with very specific, fine granularity queries which match only a subset of records. Which bitmap indices and aggregate tables to define is thus a design decision to be taken by the developer of a specific application: it does not only depend on the data set at hand, but also on the types of queries which are expected.

4.8 User Interface

The user interface was implemented using the Restlet framework. This allows a developer to create a simple servlet application, assigning paths from the root (e.g. `/hello_world`) to dynamic objects which serve the request. Simian defines several such paths by standard, including the root, and generally serves them by retrieving template pages, which it fills out with content. The template pages contain HTML tags and some JavaScript code – the jQuery library was used to implement the functionality of refreshing parts of a page without reloading it entirely (via AJAX).

The user interface consists of two parts. The first contains the standard functions which are included for any application which enables the user interface, and the second part exposes additional developer-defined components into the user interface. The standard part provides the following features:

Bulk Loading. This page allows a user to start a bulk load process. It also shows whether a bulk load process has already been started and whether it has finished.

Cluster Status. Here, a user can see the address, port and memory consumption of the master node, along with general information such as the number of records stored in the system as a whole. Each worker node is also listed, with address, port, memory consumption and the number of records it stores.

Star Schema Benchmark

Implemented using Simian.

Main Page
Bulk Loading
Cluster Status
Dimensions
Run Queries
Query Examples
Benchmarking

Query Result (edit query) (refresh)

Response time was 645 milliseconds.

CustomerName	CustomerAddress	CustomerCity	CustomerNation	CustomerPhone	MktSegment	Measure	Function	Value
Customer#000775555	mL1TqRr1	JAPAN 1	JAPAN	22-230-826-5285	FURNITURE	Contributors	SUM	473
Customer#000775555	mL1TqRr1	JAPAN 1	JAPAN	22-230-826-5285	FURNITURE	SupplyCost	AVG	88,992.144
Customer#000775555	mL1TqRr1	JAPAN 1	JAPAN	22-230-826-5285	FURNITURE	SupplyCost	SUM	42,093,284

Figure 4.6: The Simian user interface showing a query result.

Dimensions. This part of the user interface provides information about dimensions. The user can retrieve a list of dimensions along with their size (defined in terms of the sizes `INTEGER`, `SHORT`, `BYTE` and `DERIVED`) and cardinality. A user can also inspect the mappings between assigned keys and members for a dimension and the list can be filtered. Filtering uses wildcards of the form `UNITED *` – this expression, for example, would match both `UNITED STATES` and `UNITED KINGDOM`, but not `JAPAN`.

Query Runner. The query runner allows the user to submit an aggregation query, and renders the result as a table.

The developer can also define additional pages, which fulfill functions needed in a particular context. To define such an additional feature, a developer first needs to create a subclass of the class `Restlet`, used by the Restlet framework. An object of this class will serve the requests. They then need to define a path from the root where the page will be available. Also, they need to define a page title and whether or not the page should appear on the menu. If yes, the menu will now contain a link to the new page (using the page title as a description), and appropriate JavaScript handlers (using the jQuery framework) will automatically be added.

4.9 Implementation Details

In this section, I will discuss some very specific, low-level details about how Simian was implemented. Unlike the high-level, abstract view on the concepts employed in this project as provided in previous sections, this section deals specifically with programming tricks which were used and resulted in good performance, both in terms of time and space usage, and are more specific to the programming language that was used (Java).

4.9.1 Fact Table Structure

The implementation of the fact table represents columns as arrays. A multi-dimensional array has been created for each non-derived column type – each top-level array element represents a column, and consists of another array which stores the values of that column. So for example, columns of size `INTEGER` will be stored in a structure of type `int[][]`. I considered using some form of list for this type; while they were generally easier to deal with from the programmer perspective, they did not turn out to be as space efficient, as list data types in general incur a higher storage overhead than arrays. Measures are similarly stored in such an array structure. On the other hand, post-processing dimensions and measures consist only of the respective objects which derive the values; they are implemented as one-dimensional arrays of types `PostProcessingDimension[]` and `PostProcessingMeasure[]`, respectively, with each element representing a column. Because columns of each type are stored in data structures of dissimilar types, the actual implementation of Algorithm 1 contains a large amount of code which deals with these distinctions, unlike in the abstract representation. For instance, the loop which checks whether the current record passes the filtering constraints has actually been implemented as four loops: each such loop only checks the filtering constraints for columns of a single column type. Needless to say, the different column types do not only affect the implementation of the aggregation routine, but also need to be accounted for in code which provides other features of the system, such as aggregate tables and bitmap indices.

4.9.2 Trove

I have used the Trove library in many parts of the code. Trove is a freely available re-implementation of the Java collections library, which also provides *primitive* data structures. Generally, most standard implementations of data structures provided in `java.util` use generics, which is a Java programming language element to allow flexible parameterisation of classes to specify the types of objects they handle. For example, a reference of type `java.util.List<String>` points to an implementation of a list, with all list methods such as `get` or `add` handling the `String` type. A `java.util.List` can be similarly parameterised with any other Java class. This avoids the large amounts of casting which were a common sight in code written for versions of Java before 1.5, as such data structures generally only used the root `Object` type. However, generics are not directly capable of dealing with primitive types such as `int` or `byte`, as these are not classes. It is not possible to parameterise a class with a primitive type. The alternative here is to use standard wrapper classes – for example, an object of type `Integer` wraps around a primitive `int`. Thus, a `java.util.List<Integer>` can be defined. Java handles conversion between primitives and their wrapper classes automatically – this feature is called auto-boxing. However, converting back and forth between these forms incurs a rather large overhead.

Trove provides alternative, primitive implementations of standard data structures such as lists, maps and sets to the developer, and their methods take primitive arguments only, thus avoiding the need for autoboxing. For example, Trove provides the `TIntHashSet` class, which serves as a replacement for `java.util.HashSet<Integer>`. The number of such classes is large; for example, the hash map implementations cover most combinations of primitive types, and have appropriate names such as `TIntLongHashMap`. Using Trove instead of the Java standard implementations in such cases eliminates the need for auto-boxing.

An example use of Trove primitive collections can be found in the implementation of Algorithm 1. The hash sets which are transmitted to the worker node and contain permitted numerical values

for a column are primitive sets; for example, an `INTEGER` column will be filtered according to the contents of a `TIntHashSet` object. Generally, a lot of code which used to employ the standard Java collection classes in the initial phases of the implementation now uses Trove instead.

Chapter 5

Evaluation

In this chapter, I evaluate the research prototype described in Chapter 4. It contains two types of evaluation: quantitative evaluation uses a benchmark to gauge the performance which can be achieved with a complex data set, while qualitative evaluation discusses some other aspects which cannot be directly quantified, such as correctness.

Key questions to be covered in this chapter are:

General Performance. How fast or slow is the system at answering queries? Does this performance vary with respect to the type of a query? Are there any types of queries which yield particularly good or bad performance? How much hardware does the system require to run?

Scalability. Any distributed processing system will incur coordination and communication overheads, which vary depending on the number of nodes in the cluster. How large are these overheads for the system in question? Do they increase gradually when adding more nodes, or does system performance degrade rapidly?

Limitations. Are there any use cases for which the system could potentially fail to produce acceptable performance? If so, how could the system be changed to still provide good performance in such cases?

5.1 Benchmarking Methodology

In this section, I will describe the benchmark I have used for quantitative evaluation in this project, how the benchmark application was implemented and the design choices I have made in the process.

5.1.1 The Star Schema Benchmark

I have used the Star Schema Benchmark [44] (I will from now on refer to this as SSB) to evaluate this project. I have referenced it at various points earlier in the report, to support design decisions using observations I have made while using it as I was implementing the research prototype. However, in this chapter, I will use it for quantitative evaluation. It is based on the TPC-H decision support benchmark, published by the Transaction Processing Group (TPC). The SSB changes TPC-H into a more efficient star schema by simplifying the heavily normalised TPC-H schema. Additionally,

the SSB provides queries which seem to focus more on the core aggregation functionality provided by Simian – TPC-H seems to consist of general queries which are common in decision support systems, which may partially involve aggregation. I wanted to use a benchmark which is precisely about the kinds of workflows described in the background research section (drill-down, slicing, ...).

The schema consists of five tables, which are meant to simulate a data set which could be used in this form in a large trading company. The `CUSTOMER` dimension table contains information on customers such as their name, address, country of residence and region. The `SUPPLIER` table stores similar data about suppliers. The `PART` table stores information about the parts which have been traded; this includes attributes such as manufacturer, brand and colour. The `DATE` table contains over seven years of individual dates; each date has a large amount of attributes, ranging from the year, month and day it represents to properties such as which week of the year the date belongs to.

The fact table, `LINEORDER`, contains records which represent individual items which were ordered, and each record references the dimension tables. Some additional dimensions (for example, order priority) and measures such as the supply cost of the item are included in the fact table. The data set is randomly generated – a scale factor can be specified, which varies the amount of data which is generated. For instance, scale factor 10 yields around 60,000,000 fact table records, whereas scale factor 20 would yield about 120,000,000. The size of dimension tables also varies: for instance, the size of the supplier table is calculated from the scale factor by multiplying it with 10,000.

The authors of the original paper have included 13 SQL star join queries, all of which use the `GROUP BY` construct in conjunction with aggregate functions to compute results. This set of queries includes both relatively coarse granularity queries, as well as ones which yield a very fine granularity (one of them is described as a "needle-in-the-haystack" query). As the data set is randomly generated, it follows a uniform distribution; as such, the amount of columns which are relevant to a query can be calculated. SSB also includes a part in which the database is modified after running the queries; however, since this is a small part of the benchmark and Simian, being a research prototype, does not provide insertion/update/deletion functionality apart from insertion done during the initial bulk load, I decided not to implement this part of the benchmark.

5.1.2 The Benchmark Application

As expected, the benchmark application was implemented using the Simian framework, with queries translated from SQL into the Simian format. During the initial bulk load, the schema is further denormalised into a format suitable for Simian: for example, the region attribute of the `CUSTOMER` table becomes a column `CustomerRegion`. The application makes use of dimension post-processing – all date attributes (such as `OrderYear`) are derived from a date key (found in the columns `OrderDateKey` and `CommitDateKey`) using a lookup table which is injected into the worker nodes. Also, some measures are derived: for instance, the profit measure is calculated by subtracting supply cost from revenue. The column types used for non-derived attributes reflect the amount of values which have to be stored in that particular column; for instance, `SupplierNation`, which has only 25 distinct values, is represented as a `BYTE` column, while `CustomerName`, which can take hundreds of thousands and even millions of distinct values at higher scale factors is represented as an `INTEGER` column.

The application uses the standard Simian user interface with a few additional extensions: there now is a benchmark page, which allows us to gather data for a particular run. This includes general data such as bulk load duration, the memory usage of the master node and each worker node, and

also allows us to run the 13 benchmark queries in quick succession, either once or in sets of 20 such runs. Timings measured during these tests include (for each run of 13 queries) the total response time for each query and a total for all of them, as well as the raw time to compute a partial result at each worker. This allows sufficient analysis of which queries are fast and which are slow, at the same time giving an overall total and allowing us to estimate how large the overhead of managing a distributed system is. The original queries have been translated into the form Simian uses, and a full comparison along with some implementation specifics is given in Appendix B.

5.1.3 Configurations

Apart from just doing a basic scan (which is already rather fast) over the data to answer a query, Simian can additionally compute data structures to help with query processing. The two main features provided are indexing and aggregate tables. There are two different types of aggregate tables, as explained in Chapter 4: local and centralised. Since I wanted to compare the performance of the different approaches I have implemented to speed up query processing and also the basic case of iterating over the set, this yields four different types of configurations to be tried out:

Scan. This is the basic case where neither indices nor aggregate tables have been computed, and the system relies on scanning of columns only to respond to queries.

Local Aggregate Tables. Aggregate tables have been computed, and are stored on each worker node. When an aggregate table can be used to compute a query result, requests are sent to all worker nodes, and each of them uses this aggregate table instead.

Centralised Aggregate Tables. After aggregate tables have been computed, they are sent back to the master node, combined, and injected into one worker node. Queries to which this aggregate table is relevant will be computed by only sending a request to that worker node.

Indices. Bitmap indices have been defined on a few select columns to speed up scanning when filters are used. The table on each worker node will also have been sorted in order to improve the performance of the indices.

The basic case is simple. In the case of bitmap indexing, four bitmap indices have been defined on selected columns. The `MAX_PROBABILITY` and `MAX_OPERATIONS` properties are unchanged from the defaults; sorting is enabled, with the ordering as specified in the configuration file (a lower number after `SORT` means higher precedence) in listing 5.1.

Listing 5.1: The `indices.conf` configuration file.

```
SORT: YES
MAX_PROBABILITY: 0.5
MAX_OPERATIONS: 5

INDEX OrderYear SORT 1
INDEX CustomerRegion SORT 2
INDEX SupplierRegion SORT 3
INDEX Discount SORT 4
```

I have made the design decision to give the `OrderYear` column the highest precedence, as it seems to be used in most queries, unlike the other three columns. Due to this setting, the column will have the largest average run-length and therefore should be very fast to scan.

Listing 5.2: The `precompute.conf` configuration file. A backslash indicates that a line in the configuration file is continued in the next line of this listing.

```
C OrderYear Discount Quantity
C OrderYearMonthNum Discount Quantity

C OrderYear PartBrand PartCategory SupplierRegion
C OrderYear PartBrand SupplierRegion

C CustomerNation SupplierNation OrderYear \
    CustomerRegion SupplierRegion

C OrderYear CustomerNation PartMfgr \
    CustomerRegion SupplierRegion
C OrderYear SupplierNation PartCategory PartMfgr \
    CustomerRegion SupplierRegion

C <empty>
```

For the cases which use aggregate tables, I have defined a set of aggregations which are feasible to precompute – my definition of this is that an aggregate table will have less than 100,000 rows, and I could find such aggregate tables for 8 of the benchmark queries. The threshold of 100,000 is not a definite cut-off value; it could be possible to precompute larger aggregations, but it would take more time to create these data structures and also more space to store them, and the defined barrier seemed to work quite well in the context of this benchmark. Listing 5.2 shows the aggregations I came up with for the case of centralised aggregate tables, indicated by a `C` at the start of each line. In order to precompute local aggregate tables instead, the `C` has to be replaced with an `L`.

5.1.4 Hardware Resources and JVM Parameters

While I mainly used my own computer to develop the research prototype, I used the hardware equipment provided by the Department of Computing to carry out the evaluation. All of the machines used to host worker nodes were desktop PCs which are used by students during the day, while I would run my trials for the project mainly at night.

The worker nodes in all test cases of the qualitative part of the evaluation have been hosted on quad-core `Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz` machines (from `/proc/cpuinfo`). Each machine had 8 gigabytes of total main memory capacity.

In order to achieve parallelism and utilise all cores, four nodes were started on each machine. Each worker node instance was run with the JVM parameters `-server -Xms1700m -Xmx1700m`. The `-server` option may cause the system to start up slower, but is designed to maximise operating speed [45]. The other two options set enough minimum and maximum heap space to fit the data.

For the master node, the same machine configuration was used for experiments #1 and #2, while for experiment #3, a hexa-core `Intel(R) Xeon(R) CPU E5540 @ 2.53GHz` server (from `/proc/cpuinfo`) was used. The machine had 24 gigabytes of main memory, much more than the desktop PCs, but the choice was not necessary to fit the dimension data stored by Simian. It was required to give the master node more space during the denormalisation step of the bulk load process for speed

reasons (as the original dimension table files generated by the benchmark would be fully loaded into memory at this stage), and because the desktop machines could not fit the benchmark files on disk. The JVM parameters for the master node were `-server -Xms7500m -Xmx7500m` for experiments #1 and #2, and `-server -Xms15000m -Xmx15000m` for experiment #3. The operating system used in all cases was Linux with kernel version 2.6.38.2.

5.1.5 Cluster Management

I managed my cluster by using bash scripts, which would use SSH to connect to machines and start worker nodes. A bash file called `machines.sh` contained a list of all machines which would host worker nodes, and would be included by all other scripts. I wrote scripts to provide the following functionality needed to manage the cluster:

Network Configuration Generator. This script would generate a `network.conf` file on the basis of `machines.sh` which would be used by Simian, and automatically overwrite the old one. This file includes the addresses and ports of worker nodes; with the available hardware resources, I could fit four nodes onto each machine, which would then process requests in parallel.

Start/Shutdown Nodes. This script would connect to each machine and start the worker nodes, or shut down any worker nodes already running on the machines. The default setting is to start 4 nodes on each machine, with ports 5000, 6000, 7000 and 8000, respectively.

Cluster Status. This script would check whether the machines specified in `machines.sh` can be used for tests (not powered off or currently running the Windows operating system) by attempting to establish an SSH connection (with a low timeout) to each node to run the command `uname -a`. I am not relying on ping for this function, as Windows machines would still respond to pings, but will not allow me to run the worker node because they lack SSH access by default.

The scripts used the automatic login option to any of the lab computers enabled on one of the Department of Computing servers available to students – no password would be required to log into a machine and run commands. Cluster management thus was a very automated process.

5.2 Experiment #1: Scaling Up

This test case measured how much data we can fit onto each worker node, and how this affects performance. I generated an SSB data set with scale factor 100, yielding about 600 million fact table records – although the highest number loaded into the system in any trial was 135 million. Apart from the master node, two machines were used with four worker nodes each. For each configuration, I started by loading 110 million fact table records into the system, increasing this by a further 5 million records at each step. When the system failed to load the additional 5 million records at some point, I tried increasing the number of records by 2.5 million instead.

I have introduced the following definitions which will be used throughout this chapter. For each query, the *response time* is the time taken to submit a query in text form and then receive the result. The *total response time* is the sum of the individual response times of all 13 SSB queries as provided in the original paper, for one such consecutive run of these queries. The *mean total*

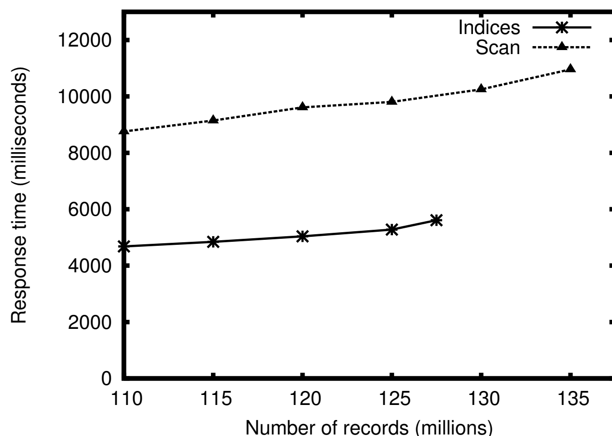


Figure 5.1: Mean total response times using scans with and without indices [#1].

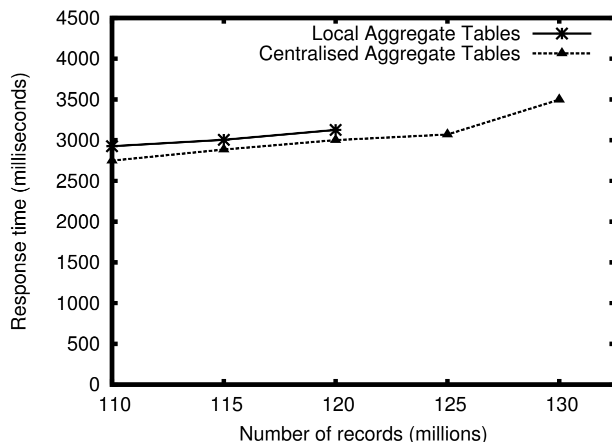


Figure 5.2: Mean total response times using local and centralised aggregate tables [#1].

response time is the total response time averaged over 100 runs. The results for the mean total response time measurements can be seen in Figure 5.2 and Figure 5.1. In terms of mean total response time, the configuration **Indices** was faster than **Scan**. For the configurations which precompute combinations of dimensions used by some queries, **Local Aggregate Tables** was slower than **Centralised Aggregate Tables**.

As we can see from the end points of the lines in Figure 5.2 and Figure 5.1, **Scan** could unsurprisingly fit the most records onto the worker nodes before reaching its scalability limits; it would still run fast with 135 million records in the system, but would fail to allocate the necessary space to store 137.5 million records. **Centralised Aggregate Tables** and **Indices** could fit the second and third most amount of records onto the worker nodes, at 130 and 127.5 million records, respectively. While **Centralised Aggregate Tables** would have a worker node throw a `java.lang.OutOfMemoryError` while trying to compute centralised aggregate tables for more records, **Indices** would finish computing the indices, but the worker nodes would frequently reach the garbage collector overhead limit while processing queries. The **Local Aggregate Tables** configuration, on the other hand, would take a very long time to compute the data structures, most likely because of very low availability of memory; I decided to stop the bulk load for this run after

eight hours, as even if it would finish eventually, the loading time would have been unacceptable. Thus, we can conclude that the system still exhibits acceptable response times for all configurations when increasing the number of records stored on each worker node. We have further identified the points at which each respective configuration cannot store the required data structures anymore due to lack of space.

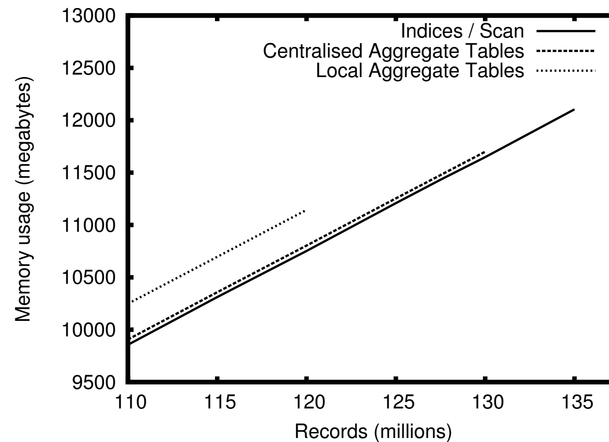


Figure 5.3: Sum of memory usage for all worker nodes. Line points have been omitted for clarity, but would be the same as in figures 5.2 and 5.1 [#1].

The space usage of the master node did not vary much; it was almost always in the range of 1.2-1.5 gigabytes, no matter which configuration was used; the usage estimates were taken from the Java system functions defined for this purpose. The combined space usage of all worker nodes has been plotted in Figure 5.3. Since the Java system functions to estimate memory usage would give results with too much variance for the worker nodes, I decided to instead write my own memory estimation function; it would count the memory usage from fact table data (by inspecting the size of the arrays), bitmap indices (using the function provided for this purpose by the library) and aggregate tables (by serialising them and counting the number of bytes). The **Indices** and **Scan** configurations both use more or less the same amount of memory, and in fact, the graph (which is using megabytes as units for the Y-axis) shows no difference; there is a small increase for **Indices** when looking at the same data in terms of kilobytes. This surprising result is due to the nature of the EWAH run-length encoding scheme when the bitmaps are generated on the basis of pre-sorted columns: the run-lengths are extremely large, and thus we will receive very good compression. The **Centralised Aggregate Tables** configuration does not use up much more space than these two: the total size of all aggregate tables together was only around 51 megabytes. Of course, since the **Local Aggregate Tables** configuration stores all of these aggregate tables on each worker node, the difference is much larger.

5.3 Experiment #2: Scaling Out

In experiment #1, we have seen how the system scales up – i.e., how much data we can pile onto a node, and how this affects performance. One of the key findings was that each configuration can fit 60 million records per machine, and we still receive acceptable response times. However, the experiment only used two machines for each trial; in order to assess scalability, we want to see how

Test Case	Scan	Local ATs	Centralised ATs	Indices
Query 1.1	1,122 ms	14 ms*	3 ms*	304 ms*
Query 1.2	814 ms	33 ms*	7 ms*	464 ms*
Query 1.3	754 ms	771 ms	767 ms	214 ms*
Query 2.1	820 ms	50 ms*	7 ms*	377 ms*
Query 2.2	794 ms	37 ms*	5 ms*	353 ms*
Query 2.3	745 ms	30 ms*	5 ms*	342 ms*
Query 3.1	935 ms	27 ms*	5 ms*	417 ms*
Query 3.2	549 ms	588 ms	554 ms	623 ms
Query 3.3	658 ms	631 ms	627 ms	789 ms
Query 3.4	646 ms	629 ms	613 ms	774 ms
Query 4.1	778 ms	23 ms*	4 ms*	347 ms*
Query 4.2	717 ms	30 ms*	5 ms*	323 ms*
Query 4.3	611 ms	648 ms	624 ms	290 ms*

Table 5.1: Average response times for each SSB query, as measured during experiment #2 in the case with 10 nodes and 600 million records, for each configuration. "ATs" means aggregate tables.

the system handles adding additional machines to the cluster of worker nodes. In experiment #2, I thus tried increasing both the number of nodes and the volume of data which is loaded into the system. I again used a scale factor 100 SSB data set. The number of machines used was increased from two to ten in several steps, with each step adding an additional two machines. The amount of data loaded into the system was proportional to the number of machines, and each machine would store 60 million records; thus, e.g. eight machines would store 480 million records in total.

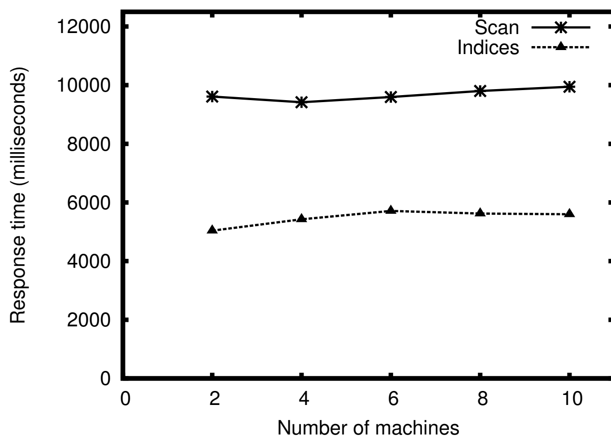


Figure 5.4: Mean total response times using scans with and without indices [#2].

The results for each query, in the case of 10 machines with 40 worker nodes storing 600 million records, can be seen in Table 5.1. Queries for which the auxiliary data structures defined in the respective configuration are accessed are marked with a star – i.e., the result is either computed from an aggregate table, or a bitmap index is used during the scan. We can see that when this happens, response time improves a lot.

As can be seen from Figure 5.5 and Figure 5.4, there do not seem to be any large spikes in terms of response time as the number of nodes increases, which is good, as it shows that the system does

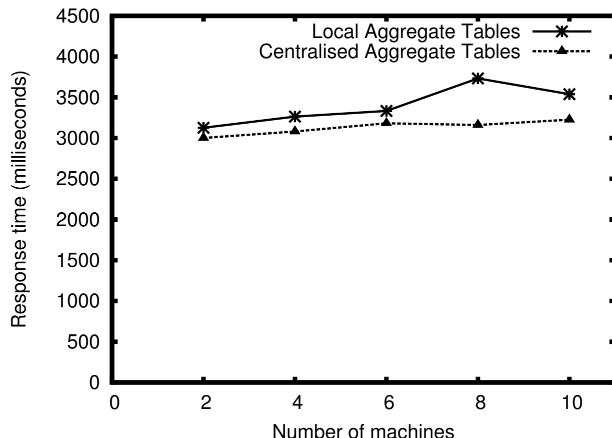


Figure 5.5: Mean total response times using local and centralised aggregate tables [#2].

not suddenly collapse when working with a larger number of nodes. However, there is a small but noticeable upwards trend for this metric, although it is not surprising. In general, allocating more computational resources to a problem will not cause an exactly proportional speed-up. What this means is that for example, if the computation of some result usually takes an amount of time t on one processor, parallelising the problem to work on two processors will not bring this down to exactly $\frac{1}{2}t$. In practice, coordination and communication overheads are introduced, which cause the parallel computation time to be higher than $\frac{1}{2}t$.

5.3.1 Understanding the Overheads

I have decided to analyse the overheads involved with a finer level of detail. I performed the analysis on the measurements received with the **Scan** configuration, as there are no configuration-specific speed-ups which benefit some types of queries more than others; for example, the configurations which use aggregate tables would make certain types of overhead harder to measure and distort them, as scanning an aggregate table to compute a partial result takes far less time (usually below 20 milliseconds) than performing a full scan – why this is an issue will become clear after I explain the types of overheads I have identified.

To support this analysis, worker nodes were made to report the *processing time* for a query. This is the total time between starting and finishing the aggregation routine, and therefore purely represents the time taken to produce a partial result. For any query which has been submitted, I call the difference between the average processing time for all worker nodes and the maximum processing time which was reported by some worker node the *waiting overhead*. In an ideal world, each worker node would take the same amount of time to produce a partial result, and the waiting overhead would be zero. However, in practice there will be some degree of variation between the processing times of each worker node. Due to the way this distributed system works, all partial results have to be received before a complete result can be computed and returned. Thus, we will always end up waiting for the slowest worker node to finish its local computation in order to be able to compute a complete result. This is the effect that the waiting overhead seeks to capture. The other type of overhead I am trying to estimate is what I call the *remaining overhead*. This is the difference between the response time and the maximum processing time of a query. It seeks to capture both the network overhead and the time taken to combine partial results, absent in a single-node system.

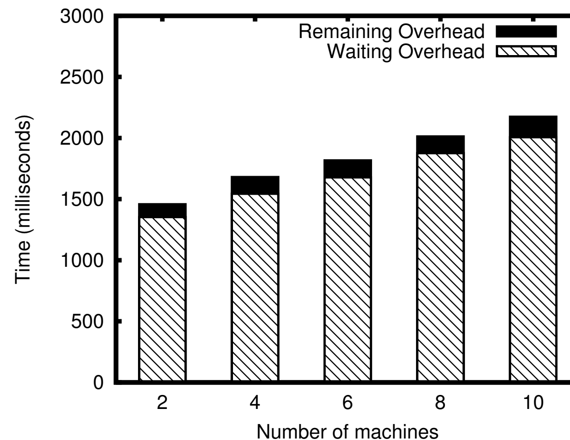


Figure 5.6: Average of the total waiting and remaining overheads for runs of all 13 SSB queries with the **Scan** configuration, by number of machines [#2].

Figure 5.6 shows the growth of both types of overhead as the number of machines is increased – the figures are an average taken from 100 runs of the 13 SSB queries. The number of worker nodes in the system is even higher, as each machine hosts four of them. As we can see, the major contributor to the overall overhead is the waiting overhead. It grows from 1353 milliseconds with two machines to 2007 milliseconds with ten. The remaining overhead is much smaller: it grows from 105 milliseconds to 168 milliseconds with two and ten machines, respectively.

The waiting overhead grows because the probability that *some* worker node will be slow increases as more are added to the cluster – and only one worker node which takes more time than usual to compute a result is already sufficient to slow the entire system down. The values for the processing time at each worker node can be viewed as following some probability distribution; thus, we can model this by sampling a single value from this distribution. Computation at N nodes can be seen as generating a sample of size N from the same distribution. Clearly, the expectation for the maximum value of such a sample grows as the sample size increases.

We can thus conclude that the single largest bottleneck in the system seems to be the part where the master node has to wait for partial results. The time spent waiting on the slowest node dwarfs all delays due to network communication or co-ordination such as merging of results.

5.4 Experiment #3: One Billion Records

On 30th of April 2011, a company called Metamarkets posted an entry on their blog about their own in-house OLAP solution called Druid [47]. It is a distributed in-memory aggregator like Simian, and is claimed to be able to aggregate a data set of one billion records in under a second. I will not be able to directly compare the performance of both systems: firstly, I do not have access to the system Metamarkets have built and secondly, the systems were probably built for different use cases. The research prototype presented as part of this project was evaluated using the Star Schema Benchmark, which uses a data set which could be sourced from the accounting data of a company, and since this benchmark was used early in the implementation phase, most optimisation approaches which were chosen will be geared towards this and similar data sets. On the other hand,

Test Case	Response Time
Query 1.1	1,144 ms
Query 1.2	812 ms
Query 1.3	751 ms
Query 2.1	834 ms
Query 2.2	668 ms
Query 2.3	733 ms
Query 3.1	840 ms
Query 3.2	576 ms
Query 3.3	664 ms
Query 3.4	659 ms
Query 4.1	789 ms
Query 4.2	720 ms
Query 4.3	633 ms

Table 5.2: Average response times for each SSB query, as measured during experiment #3.

Metamarkets use their system to analyse event data sourced from websites, and their data set will likely exhibit different patterns with respect to number of dimensions, cardinality of dimensions and the types of queries which will be commonly run. Metamarkets also honestly state that they have made a number of simplifying assumptions to fit their use case – which is alright, as after all, they want their system to show good performance for the types of data sets which it will analyse. However, the claim to have built a distributed system which is able to quickly analyse a data set of one billion records demonstrates the robustness of the system, and I decided to accept this challenge.

To test this assertion, I again used the Star Schema Benchmark and generated a data set with scale factor 170. This yielded about 101.47 gigabytes of fact table data and about 742.2 megabytes of dimension table data, and the data set includes 1,020,017,967 fact table rows which I distributed on 17 machines hosting four worker nodes each. This is slightly more than one billion records, but it gives us more or less the tried-and-tested figure of 60 million records per machine. Comparing the hardware requirements, Metamarkets has stated that they used 40 Amazon EC2 `m2.2xlarge` instances. Each such virtual machine instance has 34.2 gigabytes of memory and four fast virtual cores (with 3.25 *compute units* each – each such compute unit "provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor", according to Amazon). The cluster Metamarkets uses seems to have (altogether) more computational power than the machines I have used, but then again this could be a consequence of having different use cases. As explained earlier, I used a machine with more main memory available to act as a master node during this experiment. I used a **Scan** configuration – a claim of sub-second response time will generally exclude pre-computation, as generally, responding to a query from an aggregate table will take very small amounts of time for any such system, and does not constitute a real challenge. I thus do not believe that Metamarkets base their performance claims on precomputation. Also, the **Indices** configuration chosen for the other experiments results in very high average run-lengths for the columns (in the order of hundreds of thousands) and I do not believe it is a valid assumption that such high run-lengths will be available for all comparable data sets – this configuration was meant to show that bitmap indexing can produce better results, but constitutes a rather extreme case with the SSB.

The bulk load process took 6 hours, 52 minutes and 8 seconds to complete, and the reported combined memory usage on all worker nodes was 89.31 gigabytes, while the reported memory usage on the master node was 2.93 gigabytes.

Again, as for the other experiments, after gathering relevant statistics (memory usage, bulk load duration), I performed a benchmark using 100 runs of all 13 SSB queries. The results can be seen in Table 5.2. Apart from query 1.1, the average response times for each query were sub-second. Compared to the measurements made in experiment #2, there does not seem to have been an increase in response times – in experiment #3, the mean total response time was 9,823 milliseconds, while it was 9,945 milliseconds for the **Scan** configuration with 10 machines and 600 million records in experiment #2. This seems to indicate that the waiting overhead eventually converges to some level. In any case, it can be claimed that the system can successfully scale out to 17 machines, storing and aggregating more than one billion records while showing an acceptable amount of performance degradation.

In addition to using the queries for the Star Schema Benchmark, I also defined some new queries to test additional aspects of the system. I ran those queries after the runs of the SSB queries finished, and while the system started for this purpose was still running – thus, they were tested with 1,020,017,967 records loaded into the system. Since I did not have the time to create additional developer-defined pages to run these automatically, they were manually entered into the UI and each query was repeated 20 times to yield an average result for the response time.

5.4.1 Handling Large Dimensions

Listing 5.3: Query to test the performance of filtering on a large dimension.

```
DIMENSIONS:
  CustomerName
  CustomerAddress
  CustomerCity
  CustomerNation
  CustomerPhone
  MktSegment

CALCULATIONS:
  Contributors.SUM
  SupplyCost.SUM
  SupplyCost.AVG

FILTERS:
  CustomerName += ["Customer#000775555"]
```

One type of query is absent from the ones provided with the Star Schema Benchmark: none of the queries involve an attribute of very high cardinality, although the underlying data set does include such attributes – for example, an attribute such as `CustomerName` will have a unique value for each customer, and at the higher scale factors, there are millions of customers. Aggregate tables for such queries would be very hard to pre-compute: the result would have millions of rows. However, the actual queries of that type which will be submitted to the system will generally return small result sets, as usually filters will be provided to narrow down the search: computing very large

result sets (with millions of rows) would always take a long time, and it is not clear to me when a user would really need this – in fact, it is likely that such a query will time out with Simian. However, a business question such as “how much did all the items we have sold to this customer cost to supply?” is conceivable. Thus, I decided to test the performance of a corresponding query, shown in listing 5.3. I chose the value for the filter arbitrarily. When running this query, Simian will have to first translate the alphanumeric value the `CustomerName` dimension is restricted to into a key value. Once the result arrives, it will have to additionally be enriched with the values of two other large dimensions – `CustomerAddress` and `CustomerPhone`.

I submitted this query 20 times and the average response time was 612 milliseconds – thus, the research prototype handled this type of query well. I wanted to include a query like this as it would have otherwise been useless to load all the data associated with the large dimensions into the system – why load it if it is never accessed? I also wanted to see whether a query at such a fine level of detail would show any performance problems, however it didn't in this case – the lookups on large dimensions didn't seem to impact overall response time much, and also the high level of filtering here (very few records match this query – in this test run, only 473 out of over a billion) helped performance.

5.4.2 Scans Without Filtering

Filtering helps with performance – it decreases the number of records which match the restrictions (as there are no restrictions if there are no filters), allowing us to quickly reject them and to avoid the performance overhead of updating the result map. It can be seen from Table 5.1, Table 5.2 and the query definitions given in Appendix B that queries involving a higher level of detail tend to take less time to compute. However, and unlike in the Star Schema Benchmark, not every query will contain filters. In fact, a common starting point for a user exploring the data would be to aggregate some measure by e.g. `OrderYear` and `CustomerNation`. Although the necessary aggregate tables can be easily computed – the size, in rows, would be only 175 in this case, for 7 order years and 25 customer nations, respectively – this might not always be the case, and the system will have to update the result map at every record of the fact table, since the query does not contain any restrictions. Also, bitmap indexing does not help here due to lack of filters.

Listing 5.4: A query without filters.

```
DIMENSIONS:
  OrderYear

CALCULATIONS:
  Contributors.SUM

FILTERS:
```

I ran two queries to assess the performance in that case. The first query is shown in Listing 5.4. The second query is an extension of this which adds `CustomerNation` and `SupplierNation` to the `DIMENSIONS` part of the query. Each query was run 20 times, and an average of the response time was taken. The average response time for the first query was 3324 milliseconds, while for the second query it was 4963 milliseconds. The difference reflects the size of the output and the result map which is maintained by each worker node; for the first query, the size of the output will be 7, while for the second query, this will increase to 4375 (7 order years, multiplied by 25 nations each for

customer and supplier). Of course, as the result map is larger in the second case, there will be a larger overhead involved in maintaining it.

So while the system does not completely collapse when presented with such a query (while not having a matching aggregate table), the slow-down is noticeable. It may be worth to investigate how such a system would actually be used (i.e., determining the full workflow of a user), and whether pre-computing a certain depth of dimension combinations would help, with some rule limiting the size of the aggregate table. This would allow users to get results for popular queries such as an aggregation by nation and year quickly by sourcing them from an aggregate table. At finer levels of detail, filters will often be present, and in that case scanning becomes an efficient choice.

5.5 Qualitative Evaluation

In this section, I will discuss other properties of the system which I did not directly measure or quantify in some way. This includes aspects of the research prototype which have shown surprisingly good results, but also other areas which would need more work.

5.5.1 Overall Functionality

The research prototype allows users to type queries into a web-based interface, and to receive results which are rendered as a simple table. The table contains the dimensions specified in the `DIMENSIONS` part of the query as columns, and also has three additional columns: measure, function and value, representing the value that was aggregated and how this was accomplished. This basic representation of a query result is already sufficient and contains all information gained by the aggregation process; displaying the result in some other form is a representation issue. For instance, it is possible to manually copy a query result from the user interface into a spreadsheet application to create a pivot table from it; creating an automatic interface for this task would be an option. Other forms of visual representation are possible; another popular choice to make use of aggregation results are graphical methods such as bar charts or graphs. Although such complex representation of results is not supported by default, the framework allows a developer to define custom pages; the code which generates these pages can also send queries to the master node and has access to the aggregation results. This allows custom representation of results, as defined by the developer. Obviously, if an application called Simian II was ever to be written to be used in industry, it should allow advanced graphical representation of results. This could for example be achieved by adding support for a standard query language such as MDX – plenty of tools already exist to present output from MDX queries to a user in various ways.

In order to analyse the experimental results presented earlier in this chapter, I decided to reject the use of a spreadsheet program in favour of a tool fit for purpose: Simian. The Star Schema Benchmark application would generate CSV files with measurements taken during benchmarking, including timings and reported values for space usage. A separate application I wrote for this purpose (using Simian) would then load those CSV files and offer the ability to submit aggregation requests – a very useful feature for analysis. A custom page was written to support the analysis of the overheads, as this involved slightly trickier calculations – an aggregation result was received and further calculations were made to yield the required results. I then used the values computed by the application to create charts using another program. The basic application, including the bulk loader which reads in the data, only took about an hour to implement, and allowed me to

analyse the data much faster and at a finer level of detail than would have been possible with only a spreadsheet application. This has demonstrated that the framework is flexible – it can be easily adapted to deal with some new data set.

Standard features which are absent from the research prototype include data manipulation (insertion, deletion, update) and drill-through (retrieving records which contributed to an aggregated value). Especially the lack of data manipulation functions is due to the project time constraints; data manipulation is not a major focus of read-intensive systems, a description which fits most OLAP systems, and dealing with such concerns would have been both time-consuming and distracting from the core concepts to be included. On the other hand, adding drill-through functionality would not require many changes to the system architecture, since all the necessary data is already present in the fact table.

5.5.2 Range Check Filtering

The research prototype allows the user to define allowed ranges for an attribute in a query. There are two kinds of such restrictions: one considers the natural ordering of numbers when deciding whether a label is within the specified range, while the other uses lexicographic ordering of alphanumeric strings. There is a difference, since for lexicographic ordering, `"1" < "11" < "2"` holds, while the natural ordering of numbers does not result in such counter-intuitive range inclusion.

In more detail, ranges are implemented by scanning the dimension data structure and including all keys whose associated alphanumeric labels match the range into the set of allowed keys which worker nodes will use during aggregation. This approach works well for ranges on small attributes: iterating over all members and considering them for inclusion can be done in a short time, and the resulting set which is used for filtering is still quite small. It can also be used for larger dimensions such as e.g. dates when the resulting set of keys will be quite small – a 30-day interval will contain 30 values in the set.

However, performance issues may be encountered in other situations. If the dimension to be filtered is very large, then finding all matching keys may take a long time; additionally, the resulting set could be large, increasing the time taken to communicate it to the worker nodes and also increasing the check whether some key is contained in the set – implementations of set data structures such as hash sets tend to get less efficient as they get larger. Since a very common target for range check filtering are temporal dimensions like dates, changing the temporal granularity of such a column would affect results. This is, in a sense, the precision to which time is measured [49]. For example, the smallest unit of time used in the Star Schema Benchmark is a day. Each entry in the `DATES` dimension table identifies a single calendar date, and dates are represented numerically with values of the form `20101127`. However, we can imagine that this temporal granularity might not be sufficient in other contexts, and a finer level of detail would be needed. For instance, it might be necessary to store time information accurately to the second, usually done with timestamps which identify the number of seconds that have passed since some reference date. The 30-day interval will now most likely not be represented by 30 values anymore, but rather by every timestamp in that interval at which some event has occured. This is highly impractical to store in a set.

A possible solution to this problem would be to change the way in which keys are assigned, such that the keys reflect the ordering of the labels. In the current scheme, keys are assigned by using a counter; whenever a new dimension value is encountered, the current value of the counter is used as the new key and the counter is incremented. Particularly for temporal attributes, we could

change this to reflect the ordering: we could use timestamps as the keys instead. Then, in order to perform a range check, we take the keys of the lower and upper bounds specified and have a range checker object perform the check, instead of storing all these values in a set. This approach could also potentially result in a much more space-efficient way of dealing with translation from keys to values and vice versa, since a date can be easily inferred from a timestamp value. This also works the other way round: we can match some range of timestamps to a date.

5.5.3 Treatment of Dimensions

In this project, I have adopted the approach of denormalising a star schema to avoid joining the fact table with large dimension tables. This allows fact table scans to be performed faster and yields the same results for aggregation operations, with additional benefits in a distributed context.

However, dimension tables often carry additional information which is lost in the denormalisation process. For example, in the case of a dimension table storing customers, we can run a query to find out which country a certain customer is from. This relationship is preserved at the fact table level – a fact record will not list a customer as being from a country unless this information was stored in the original dimension table. However, we cannot easily run such a query anymore in this scheme, and denormalisation limits the scope of dimension browsing functionality.

Arguably, this type of operation is not a core concern of an OLAP system, which is meant to primarily handle aggregation. However, such a feature is obviously related, and it would be very useful to offer it to users. In order to realise this, we could for example link the OLAP system to another database which would store the necessary data. Most relational databases are generally fast for this type of lookup, since they implement special indexing structures such as B/B+-trees for this purpose – even using an on-disk database would be an option for such an ancilliary feature.

Another feature of the research prototype that is still rough around the edges and related to the treatment of dimensions is post-processing. A developer has to write their own code for post-processing of dimensions by implementing an interface (though some standard implementations for commonly encountered use cases exist); since this can become messy and is error-prone, it would be better for the framework to offer a cleaner, standard way to achieve this more easily.

5.5.4 Space Usage

For the case of the Star Schema Benchmark application, Simian stores the data in a way that could be argued to be space-efficient. The entire amount of data stored in memory by all nodes of the cluster together is not more than the size of the original input data files on disk. While it may be argued that the representation of the original input data is not the most space-efficient one, as even numerical keys are represented with alphanumeric strings, it does not seem like the research prototype increases the size of the source data excessively and unnecessarily when loading it into its own data structures.

However, we have to notice that the dimension data on master nodes tends to occupy rather large amounts of space for the higher scale factors. This is mainly because it is stored in a hash map based data structure to ensure fast lookup. The size is growing at increasing scale factors because of a subset of high-cardinality dimensions. For instance, low-cardinality dimensions such as `CustomerRegion` do not increase in cardinality, and even in a real-world data set rarely would

do so. However, other dimensions such as `CustomerName` grow, and would do so at a slow but steady rate in practice. At the same time, these large dimensions are likely to be some of the most rarely accessed ones. None of the Star Schema Benchmark queries access `CustomerName` or any other high-cardinality attribute; I had to create such a query myself (shown in listing 5.3), and it is thus likely to be a rather contrived example.

Although we did not encounter this case in this project, a single master node may not be able to store the dimension data if it is extremely large. It would be interesting to investigate schemes in which several machines store partitions of the dimension data. Also, if a large dimension is only rarely used, then keeping all of its mappings in memory may seem wasteful. It may thus be worth investigating whether mappings for very large dimensions could be stored on disk, while using some indexing scheme such as B/B+-trees to make lookups efficient. This could possibly be combined with some kind of caching scheme to speed up repeated accesses. However, dimension lookups are such a small part of the aggregation process that making some of the more rarely used ones operate from disk will likely not affect performance a lot, since the response time still mainly depends on the performance of the aggregation routine (performed on data in main memory). If lookups using on-disk structures turn out to be fast enough, it would even be possible to replicate the dimension mappings onto each worker node, i.e. to use an approach similar to Data Warehouse Striping/Selective Loading, but still with a denormalised schema. Since the large dimensions would not be stored in memory, this would not cause the types of problems discussed in Chapter 3. That being said, this is speculative and a possible area of future work.

Another possible extension in the area of space efficiency would be to see whether we can apply compression somewhere in the system to decrease the required amount of storage space. In particular, it may be possible to apply run-length encoding to the fact table data directly, and not only to bitmap indices. This is likely to not slow down scans, and could even potentially improve speed. However, many other options are likely to exist, and we would need to think about the trade-off between space savings and speed decreases (if any) when considering them.

5.5.5 Observations About the Benchmark

The benchmark used for the quantitative part of the evaluation presents us with a data set which is meant to simulate some real-world equivalent, and contains some common challenges OLAP systems have to address (e.g. high-cardinality dimensions). It also provides us with a set of queries whose response time should be measured. However, in my opinion, while these queries do represent typical OLAP operations, they do not do so exhaustively enough.

As stated earlier, I have created and tested a few additional queries for some aspects of the system. This included queries without filtering, and also queries which filter on a large dimension. Another area which has been neglected by the benchmark queries in my opinion are date range queries. While most queries include the year attribute, there is an absence of queries which would produce aggregation results for only a specific temporal range, such as e.g. a 30-day interval. These types of queries are more common in the real world than the benchmark seems to suggest, and it is essential that an OLAP system should handle them well. Another interesting approach that could have been explored, with enough example queries to measure performance, would have been partitioning the fact data according to a temporal dimension. For some data sets, most queries users will be interested in will involve the aforementioned date range queries at a very fine level of detail (e.g. 30-day intervals); distributing records according to the value of some date dimension would thus

allow the query planner to efficiently assign queries only to nodes which actually store records that match the specified date range.

A general point is that the use of a benchmark may introduce a tendency for the programmer to produce a system which is very good at this particular benchmark, but may not perform as well for other examples. For example, when I first implemented the benchmark, the results were quite poor; the benchmark queries would take in the order of seconds to complete, while some other queries I defined, such as the ones without filtering, would even take up to a minute to return results. I proceeded to profile the performance problems and implemented some counter-measures, such as denormalisation of large dimensions (with post-processing for some smaller dimensions), the use of primitive collection classes and generally more efficient programming methods. While the results have been generally good for the benchmark in question and also some additional queries I have defined, performance may ultimately be worse for some other contexts (i.e. data sets and queries) which I did not consider; it is thus possible that further refinements to the way Simian organises and aggregates its data may be necessary if we wanted to employ it in such contexts.

One may argue that I have denormalised the benchmark schema more aggressively than needed. This happened in reaction to the overheads I observed in dealing with large dimension tables, and because the core requirement for the aggregation process is that each attribute has its own unique set of keys, instead of being tied to e.g. the customer key in a dimension table. The approach initially greatly increased the storage requirements for attributes related to dates – since these could be stored together in a lookup table of an acceptable size (around 2,500 entries), I then introduced the concept of post-processing for dimensions. This still allowed me to have a separate set of keys for each dimension attribute, but also provided some degree of compression/storage efficiency on the worker nodes. Looking back at the schema, this approach may have been adopted for the other dimension tables as well. For example, it would have been possible to split out only the very specific, high-cardinality attributes of customers such as name and address. General attributes which are frequently repeated could have, however, still been kept in a separate lookup table and derived based on some shared key. As such, a lookup table for all combinations of the attributes of region, nation, city and market segment would only have a size of about 1,250 entries.

5.5.6 Data Storage Model

The Simian storage layer integrates many concepts typically found in non-relational databases. For instance, it does not provide many guarantees with respect to the consistency of the data – strictly speaking, it does not even have to, as the only transactions which change the state of the system are done during the initial bulk load phase. Also, it avoids join operations as is the case in many distributed database management systems. The only feature which could be seen as constituting a join is post-processing, whereby lookup tables can be used to derive some new measure or dimension. These lookup tables however are generally small and can be easily replicated onto each worker node. Simian eliminates large dimension tables via denormalisation, thus avoiding the necessity to manage them across the cluster of nodes. The other purpose of joins – enriching the aggregation results with the appropriate alphanumeric dimension data – is done via a centralised enrichment process at the master node. By using a simpler storage model, we gain performance and scalability in a distributed context while sacrificing certain consistency guarantees traditional relational databases give us – but these guarantees are not always needed for the correct operation of a system, and thus can be neglected in such cases.

5.5.7 Correctness

Of course, in order for a software system such as Simian to be useful, it should operate largely without errors. While programming mistakes can sometimes be discovered even in systems which have been thoroughly tested before deployment, the fastest program is of no use if its output is always wrong. Correctness is sometimes relative; for example, there are approximation algorithms for many problems, promising to solve them more quickly while incurring a small deviation from the *perfect* result. Indeed, approximate OLAP query answering has been studied [26]. However, this project aims to study a non-approximate approach, and as such, there is generally only one correct output; sometimes the underlying business rule behind a result may be questioned, but when using the same business rule, two correct systems should return the same result.

I wanted to see whether Simian produces correct results: otherwise, we could either have a deviation which could be explained somehow, or just completely wrong results which bear no resemblance to the correct result. To achieve this, I used a *reference implementation*: an already existing software system which I assume is correct, and should thus produce the same results as my research prototype if the latter is correct as well. Even in the case that that the reference implementation is incorrect, it is unlikely that for any particular result, both my research prototype and the reference implementation will have made exactly the same mistake to receive exactly the same error. For this purpose, I chose the H2 database system, which has already been discussed in some detail in Chapter 3. I chose to compare results for the individual queries of the Star Schema Benchmark – the schema was simple to create and data was loaded using the CSV import function of H2. Both the original SQL queries from the paper and their Simian equivalents were passed into the respective systems, and the results were saved into a spreadsheet file; the ordering of the results sometimes had to be altered such that each combination of dimension members in one would be on the same row as in the respective other result set. Then an additional row called `DELTA` was created in the spreadsheet. For each row of the result sets, it subtracts the aggregated values from each other. If they represent the same result, then the difference should be 0 for all result rows of some query. I loaded the first 200,000 fact records of a scale factor 1 data set into both systems; for Simian, apart from trying the basic case of a **Scan** configuration with a single worker node, I also tried two worker nodes each for all configurations as used previously during benchmarking. This would also cover the correctness of the bitmap indexing implementation and the computation of aggregate tables, together with the centralised case where the aggregate tables are combined into one single version. Running all of these test cases with only a single node may not capture all of the differences between them – for example, using the **Centralised Aggregate Tables** configuration on a single node is the same as using the **Local Aggregate Tables** configuration, as only one node stores the data structures and the routine to combine aggregate tables is never invoked.

The H2 case had equivalent results to all Simian cases for query flights 2, 3 and 4. However, the results for query flight 1 were initially slightly different. The query is meant to quantify how much revenue is sacrificed for the sake of promotional discounts. This measure is derived by multiplying one of the price measures with the discount dimension value for the given record. The calculation is implemented in Simian as a post-processing measure called `CustomerSavings`, and I have already changed the way this is calculated from the specification: since the discount is given as a percentage (e.g. a value of "10" corresponds to a 10% discount), the discount dimension must be scaled by 0.01 first in order to receive the true discount value, before multiplying this with the price measure. The SQL given does not do this correctly – the reason may be assumptions about the handling of percentages by the database which were not clearly stated. Additionally, I have also changed the

way the calculations are aggregated: when calculating the `CustomerSavings` measure for a record, we may end up with fractions of the smallest unit of currency; for instance, if we assume the price given is in pounds, then it would be possible for a customer to save 0.5 pennies; the SQL queries just aggregate these fractions as if a customer could actually be given such small discounts. In the real world, the value would probably be either truncated or rounded; I have chosen the latter option. After modifying the SQL queries for H2 of query flight 1 to use the same business rule, the results were the same, and this arguably more correct approach was used in all other runs of the Simian Star Schema Benchmark application we discussed previously.

Chapter 6

Conclusion

Overall, the implementation phase of the project has been successful in most of its original goals. The features provided by the research prototype (basic aggregation functionality, aggregate tables, bitmap indices) together enable fast computation of results for most types of queries. The data structures created are relatively space-efficient when comparing their size to the size of the original input data, and the amount of hardware resources required to operate a cluster of worker nodes is acceptable.

However, in the short time which was left to build a functional system after I had already carried out a solid amount of background research, some simplifying design decisions had to be taken in order to make the task more manageable. For example, after a bulk load has been completed, the system does not allow any changes to the data; operations such as insertions, deletions or updates are not possible. This is a fair assumption to make for a research prototype in a read-intensive context as is the case for this project; however, should a similar system be used in industry, data manipulation features may be required.

To sum up, this project has demonstrated that building a distributed in-memory aggregator is a feasible task, and that techniques such as column-oriented storage, denormalisation, materialised views (with central storage) and run-length encoded bitmap indices can help with this goal. It has also highlighted how certain techniques which would work well for an on-disk system are not directly transferrable to the in-memory realm – for example, storing large amounts of alphanumeric dimension data is unlikely to cause problems when using a hard disk. However, if the dimension data is to be stored in main memory together with fact table data, the former can take up a lot of memory which could be used for the latter instead.

The techniques researched and implemented as part of this project could be used to create a system which would run on a cloud computing platform such as Amazon EC2. Such platforms are based on the utility computing model: only computational resources which are used have to be paid for. In combination with a scalable distributed system, this is an attractive choice to manage data growth: when the currently used computational resources are not powerful enough anymore, an additional node running the system can be connected to handle the data volume. The research prototype has been demonstrated to scale well in such cases. Such a system can be used to handle increasing data volumes in the foreseeable future, even in the absence of substantial hardware improvements.

6.1 Possible Extensions

In this section, I will list some additional work which could have been carried out as part of this project given enough time. This includes standard features which are available in most comparable systems used in industry, but left out in this research prototype; additionally, I cover some interesting new ideas which could be implemented and then studied.

Data Manipulation Features. As mentioned previously, Simian becomes read-only after the initial bulk load phase and does not currently offer any data manipulation functionality, with operations such as insertions, deletions or updates. The current usage pattern the system would support is overnight loading of daily snapshots of the data set to be analysed. While OLAP systems are generally read-intensive, in some cases fresher data will be required, with the data set being updated several times a day.

Efficient View Maintenance. Especially if changes to the data are very frequent, the required modifications will have to be carried out quickly. This does not only include updates to the structure storing the base data: associated data structures such as indices and aggregate tables will have to provide a fresh view on the data as well. As we have seen in the background research, some efficient view maintenance techniques have been proposed, and investigating these and typical usage scenarios would be interesting.

Time Dimensions. In this project, I did not assign special importance to dimensions representing time. However, queries to OLAP systems very frequently involve time; for example, an analyst is likely to be interested in the profit a company has made in the last quarter, as opposed to a figure calculated for the entire history of the company. Giving special treatment to time dimensions and potentially some better query processing techniques which could result from this are another promising area of study.

Standards Compliance. In many fields, there are established protocols which are used by most applications to provide uniform interfaces. For OLAP, these are MDX and XMLA. By being able to deal with these standards, we gain the ability to use many tools in conjunction with this system, e.g. spreadsheet plugins. This would make the system more useful for industrial usage.

More Data Sets. As discussed in the evaluation chapter, although Simian has shown good query performance with the data sets and queries I have tried, others may exist for which performance may be worse. I also suggested that data sets with an important (frequently used in queries), fine-grained temporal dimension may yield slower response times. Thus, more data sets would need to be investigated, in order to test the limitations of the approach tried in this project and to possibly find modifications which would still ensure good performance.

Persistence. Bulk loading a cluster of worker nodes is slow. Loading the scale factor 170 data set with more than one billion records into the cluster took almost seven hours. It does not help that if any of the machines has to be rebooted or another event such as a power failure happens, the entire process has to repeated. A possible solution to this would be for the worker nodes to keep a copy of their data on disk. It is easier for the worker node to read in such data in its own format, rather than having to rely on the master node to transform the source data into records which will be sent to it for insertion. Given that data manipulation functionality is available, it would become possible to start worker nodes from their persisted state, and apply any changes to the data held by a worker node back to that persisted state.

Of course, we could use the same scheme with data held by the master node. This way, having to wait for a full bulk load to complete would become a rare occurence.

Fault Tolerance and Load Balancing. In this project, I did not implement any kind of fault tolerance. Both the individual machines and the network infrastructure were assumed to be perfect and never fail. However, if a node is disconnected from the network or the operating system crashes, the entire system becomes non-operational. For fault tolerance, a simple replication scheme could be implemented. For example, if the fact table records we wish to analyse can be fit onto 3 machines, we could additionally store them on another 3 machines. Thus, if a machine stops working for some reason it is highly likely that the data it stored is still present on another machine in the cluster. This could also provide load balancing, as two concurrent queries could be processed by a separate set of worker nodes. Additionally, persistence could be used to provider quicker recovery: if the state of a worker node is persisted onto stable storage, we could initialise a worker node from the persisted state once the machine recovers. Also, write-ahead logging and log-based recovery could be used to ensure the consistency of the persisted state, even in the presence of failures.

Appendix A

The Query Language

Listing A.1: An example Simian query.

```
DIMENSIONS:
  OrderYear

CALCULATIONS:
  Revenue.SUM
  Contributors.SUM

FILTERS:
  OrderYear += ["2007" "2008"]
```

OrderYear	Measure	Function	Value
2007	Contributors	SUM	12,345
2007	Revenue	SUM	12,345,678
2008	Contributors	SUM	23,456
2008	Revenue	SUM	23,456,789

Table A.1: A possible result of the query in Listing A.1.

Listing A.1 shows an example query, and Table A.1 shows a possible result for that query. All Simian queries consist of the following parts:

DIMENSIONS. This part specifies the dimensions according to which the output will be classified. Records with identical values for these dimensions will be aggregated into one result for each of the calculations specified in the query. For example, the above query will be grouped by the `OrderYear` dimension.

CALCULATIONS. This part specifies the calculations which should be carried out for each combination of dimension members, consisting of measures and aggregation functions which should be applied to them. For instance, in the above query, `Revenue` will be aggregated using the `SUM` function.

FILTERS. This part specifies any filtering criteria which should be applied to records. Records which do not match these criteria will not be reflected in the result. The basic filter operator

`+=` used in the above query adds a list of values to the allowed ones for some dimension. If no values are specified for a dimension, any value will be accepted. Filters for the same dimension can be combined, so e.g. `OrderYear += ["2007" "2008"]` is equivalent to two separate lines `OrderYear += ["2007"]` and `OrderYear += ["2008"]`.

This query language is very simple, but efficiently supports the standard workflow of a user with respect to the abstract model of the hypercube (roll-up, drill-down, slice-and-dice).

The operator `+=` is not the only one supported. Ranges for numerical values can be specified using `[-]` and lower/upper bounds. `[<]` and `[>]` specifies that the numerical values of the dimension should be lower or higher than the supplied value. These operators only consider numerical values, and ignore any dimension member which contains characters other than numbers. However, lexicographic ordering is supported for general alphanumeric strings, via the operators `[...]`, which can be used to specify a range for alphanumeric strings, and `[<<]`, `[>>]` can be used to specify values which are smaller/larger than the argument with respect to lexicographic ordering. Example usages are `OrderYear [...] ["2007" "2008"]`, `OrderYear [<<] "2007"` and `OrderYear [>>] "2007"`. In many cases with numerical values, the first expression is equivalent to `[-]`, however it theoretically could include the value `"20077"`, whereas `[-]` wouldn't. More query examples can be found in Appendix B, which details the queries used for the Star Schema Benchmark.

Appendix B

The Star Schema Benchmark

This appendix lists the queries for the Star Schema Benchmark which were used to for the quantitative part of the evaluation of Chapter 5, and gives a general overview of how the benchmark was implemented.

B.1 General Comments

The system was loaded from the same data files as produced by the data set generator supplied with the benchmark. A detailed explanation of the schema, along with information about aspects such as dimension cardinality, can be found in [44]. However, the schema of the data was changed to conform to a flat table. The conversion process gives attributes appropriate new names while it flattens the schema; for example, the dimension `CustomerNation` is assigned for each record by finding the `C_CUSTKEY` in the `CUSTOMER` dimension table matching `LO_CUSTKEY` from the `LINEORDER` fact table, and then taking `C_NATION` from the dimension table as the value. Date attributes (e.g. the year an order was made) are derived from date keys via post-processing and use a lookup table.

The benchmark queries are organised into four *flights*, with each containing three or four queries. All SQL queries given in the original paper were translated into Simian counterparts. Some measures used by the queries rely on calculations such as subtractions, e.g. in query flight 4; appropriate post-processing measures were introduced to handle these calculations, e.g. `Profit`. Note that the name of the measure used in query flight 1 has been changed to `CustomerSavings`, as according to the original paper, this is a "what-if" query designed to show the amounts of money sacrificed for the sake of company-wide discounts given to customers, and *potential* revenue which could be gained by eliminating these; I decided that this new name is more intuitive than just "revenue".

The queries, in exactly the form as they were used for the benchmark, are given starting from the next page and were demonstrated to be semantically equivalent in the correctness test described in the evaluation section. Note that for query 2.2, I could have used the expression `PartBrand [...] ["MFGR#2221" "MFGR#2228"]` instead, however when I started gathering benchmark results the operator `[...]` was not implemented yet.

Simian does not have an equivalent to the SQL `ORDER BY` statement. Instead, output presented to the user is ordered lexicographically by columns, from left to right. Thus, the ordering properties specified by the original queries are omitted.

B.2 Query Flight 1

Figure B.1: Query 1.1 (Simian).

```
DIMENSIONS:

CALCULATIONS:
  CustomerSavings.SUM

FILTERS:
  OrderYear += ["1993"]
  Discount [-] 1 3
  Quantity [<] 25
```

Figure B.2: Query 1.1 (SQL).

```
SELECT
  SUM(lo_extendedprice*lo_discount)
    AS revenue
FROM
  lineorder, date
WHERE
  lo_orderdate = d_datekey AND
  d_year = 1993 AND
  lo_discount between 1 and 3 AND
  lo_quantity < 25
```

Figure B.3: Query 1.2 (Simian).

```
DIMENSIONS:

CALCULATIONS:
  CustomerSavings.SUM

FILTERS:
  OrderYearMonthNum += \
    ["199401"]
  Discount [-] 4 6
  Quantity [-] 26 35
```

Figure B.4: Query 1.2 (SQL).

```
SELECT
  SUM(lo_extendedprice*lo_discount)
    AS revenue
FROM
  lineorder, date
WHERE
  lo_orderdate = d_datekey AND
  d_yearmonthnum = 199401 AND
  lo_discount between 4 and 6 AND
  lo_quantity between 26 and 35
```

Figure B.5: Query 1.3 (Simian).

```
DIMENSIONS:

CALCULATIONS:
  CustomerSavings.SUM

FILTERS:
  OrderYear += ["1994"]
  OrderWeekNumInYear += ["6"]
  Discount [-] 5 7
  Quantity [-] 26 35
```

Figure B.6: Query 1.3 (SQL).

```
SELECT
  SUM(lo_extendedprice*lo_discount)
    AS revenue
FROM
  lineorder, date
WHERE
  lo_orderdate = d_datekey AND
  d_weeknuminyear = 6 AND
  d_year = 1994 AND
  lo_discount between 5 and 7 AND
  lo_quantity between 26 and 35
```

B.3 Query Flight 2

Figure B.7: Query 2.1 (Simian).

```
DIMENSIONS:
  OrderYear
  PartBrand

CALCULATIONS:
  Revenue.SUM

FILTERS:
  PartCategory += ["MFGR#12"]
  SupplierRegion += ["AMERICA"]
```

Figure B.8: Query 2.1 (SQL).

```
SELECT
  SUM(lo_revenue),
  d_year,
  p_brand1
FROM
  lineorder, date, part, supplier
WHERE
  lo_orderdate = d_datekey AND
  lo_partkey = p_partkey AND
  lo_suppkey = s_suppkey AND
  p_category = 'MFGR#12' AND
  s_region = 'AMERICA'
GROUP BY
  d_year, p_brand1
ORDER BY
  d_year, p_brand1
```

Figure B.9: Query 2.2 (Simian).

```
DIMENSIONS:
  OrderYear
  PartBrand

CALCULATIONS:
  Revenue.SUM

FILTERS:
  PartBrand += \
  ["MFGR#2221" "MFGR#2222" \
   "MFGR#2223" "MFGR#2224" \
   "MFGR#2225" "MFGR#2226" \
   "MFGR#2227" "MFGR#2228"]
  SupplierRegion += ["ASIA"]
```

Figure B.10: Query 2.2 (SQL).

```
SELECT
  SUM(lo_revenue),
  d_year,
  p_brand1
FROM
  lineorder, date, part, supplier
WHERE
  lo_orderdate = d_datekey AND
  lo_partkey = p_partkey AND
  lo_suppkey = s_suppkey AND
  p_brand1 between 'MFGR#2221'
    and 'MFGR#2228' AND
  s_region = 'ASIA'
GROUP BY
  d_year, p_brand1
ORDER BY
  d_year, p_brand1
```

Figure B.11: Query 2.3 (Simian).

```
DIMENSIONS:
  OrderYear
  PartBrand

CALCULATIONS:
  Revenue.SUM

FILTERS:
  PartBrand += ["MFGR#2221"]
  SupplierRegion += ["EUROPE"]
```

Figure B.12: Query 2.3 (SQL).

```
SELECT
  SUM(lo_revenue),
  d_year,
  p_brand1
FROM
  lineorder, date, part, supplier
WHERE
  lo_orderdate = d_datekey AND
  lo_partkey = p_partkey AND
  lo_suppkey = s_suppkey AND
  p_brand1 = 'MFGR#2221' AND
  s_region = 'EUROPE'
GROUP BY
  d_year, p_brand1
ORDER BY
  d_year, p_brand1
```

B.4 Query Flight 3

Figure B.13: Query 3.1 (Simian).

```
DIMENSIONS:
  CustomerNation
  SupplierNation
  OrderYear

CALCULATIONS:
  Revenue.SUM

FILTERS:
  OrderYear [-] 1992 1997
  SupplierRegion += ["ASIA"]
  CustomerRegion += ["ASIA"]
```

Figure B.14: Query 3.1 (SQL).

```
SELECT
  c_nation,
  s_nation,
  d_year,
  SUM(lo_revenue) AS revenue
FROM
  customer, lineorder, supplier, date
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_orderdate = d_datekey AND
  c_region = 'ASIA' AND
  s_region = 'ASIA' AND
  d_year >= 1992 AND
  d_year <= 1997
GROUP BY
  c_nation, s_nation, d_year
ORDER BY
  d_year ASC, revenue DESC
```

Figure B.15: Query 3.2 (Simian).

```
DIMENSIONS:
  CustomerCity
  SupplierCity
  OrderYear

CALCULATIONS:
  Revenue.SUM

FILTERS:
  OrderYear [-] 1992 1997
  CustomerNation += \
    ["UNITED STATES"]
  SupplierNation += \
    ["UNITED STATES"]
```

Figure B.16: Query 3.2 (SQL).

```
SELECT
  c_city,
  s_city,
  d_year,
  SUM(lo_revenue) AS revenue
FROM
  customer, lineorder, supplier, date
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_orderdate = d_datekey AND
  c_nation = 'UNITED STATES' AND
  s_nation = 'UNITED STATES' AND
  d_year >= 1992 AND
  d_year <= 1997
GROUP BY
  c_city, s_city, d_year
ORDER BY
  d_year ASC, revenue DESC
```

Figure B.17: Query 3.3 (Simian).

```
DIMENSIONS:
  CustomerCity
  SupplierCity
  OrderYear

CALCULATIONS:
  Revenue.SUM

FILTERS:
  OrderYear [-] 1992 1997
  CustomerCity += \
    ["UNITED KI1" "UNITED KI5"]
  SupplierCity += \
    ["UNITED KI1" "UNITED KI5"]
```

Figure B.18: Query 3.3 (SQL).

```
SELECT
  c_city,
  s_city,
  d_year,
  SUM(lo_revenue) AS revenue
FROM
  customer, lineorder, supplier, date
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_orderdate = d_datekey AND
  (c_city = 'UNITED KI1' OR
   c_city = 'UNITED KI5') AND
  (s_city = 'UNITED KI1' OR
   s_city = 'UNITED KI5') AND
  d_year >= 1992 AND
  d_year <= 1997
GROUP BY
  c_city, s_city, d_year
ORDER BY
  d_year ASC, revenue DESC
```

Figure B.19: Query 3.4 (Simian).

```
DIMENSIONS:
  CustomerCity
  SupplierCity
  OrderYear

CALCULATIONS:
  Revenue.SUM

FILTERS:
  OrderYearMonth += ["Dec1997"]
  CustomerCity += \
    ["UNITED KI1" "UNITED KI5"]
  SupplierCity += \
    ["UNITED KI1" "UNITED KI5"]
```

Figure B.20: Query 3.4 (SQL).

```
SELECT
  c_city,
  s_city,
  d_year,
  SUM(lo_revenue) AS revenue
FROM
  customer, lineorder, supplier, date
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_orderdate = d_datekey AND
  (c_city='UNITED KI1' OR
   c_city='UNITED KI5') AND
  (s_city='UNITED KI1' OR
   s_city='UNITED KI5') AND
  d_yearmonth = 'Dec1997'
GROUP BY
  c_city, s_city, d_year
ORDER BY
  d_year ASC, revenue DESC
```

B.5 Query Flight 4

Figure B.21: Query 4.1 (Simian).

```
DIMENSIONS:
  OrderYear
  CustomerNation

CALCULATIONS:
  Profit.SUM

FILTERS:
  PartMfgr += \
    ["MFGR#1" "MFGR#2"]
  CustomerRegion += ["AMERICA"]
  SupplierRegion += ["AMERICA"]
```

Figure B.22: Query 4.1 (SQL).

```
SELECT
  d_year, c_nation,
  SUM(lo_revenue-lo_supplycost)
    AS profit
FROM
  date, customer, supplier, part, lineorder
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_partkey = p_partkey AND
  lo_orderdate = d_datekey AND
  c_region = 'AMERICA' AND
  s_region = 'AMERICA' AND
  (p_mfgr = 'MFGR#1' OR
   p_mfgr = 'MFGR#2')
GROUP BY
  d_year, c_nation
ORDER BY
  d_year, c_nation
```

Figure B.23: Query 4.2 (Simian).

```
DIMENSIONS:
  OrderYear
  SupplierNation
  PartCategory

CALCULATIONS:
  Profit.SUM

FILTERS:
  OrderYear += ["1997" "1998"]
  PartMfgr += \
    ["MFGR#1" "MFGR#2"]
  CustomerRegion += ["AMERICA"]
  SupplierRegion += ["AMERICA"]
```

Figure B.24: Query 4.2 (SQL).

```
SELECT
  d_year, s_nation, p_category,
  SUM(lo_revenue-lo_supplycost)
    AS profit
FROM
  date, customer, supplier, part, lineorder
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_partkey = p_partkey AND
  lo_orderdate = d_datekey AND
  c_region = 'AMERICA' AND
  s_region = 'AMERICA' AND
  (d_year = 1997 OR d_year = 1998) AND
  (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
GROUP BY
  d_year, s_nation, p_category
ORDER BY
  d_year, s_nation, p_category
```

Figure B.25: Query 4.3 (Simian).

```
DIMENSIONS:
  OrderYear
  SupplierCity
  PartBrand

CALCULATIONS:
  Profit.SUM

FILTERS:
  OrderYear += ["1997" "1998"]
  PartCategory += ["MFGR#14"]
  CustomerRegion += ["AMERICA"]
  SupplierNation += \
    ["UNITED STATES"]
```

Figure B.26: Query 4.3 (SQL).

```
SELECT
  d_year, s_city, p_brand1,
  SUM(lo_revenue-lo_supplycost)
    AS profit
FROM
  date, customer, supplier, part, lineorder
WHERE
  lo_custkey = c_custkey AND
  lo_suppkey = s_suppkey AND
  lo_partkey = p_partkey AND
  lo_orderdate = d_datekey AND
  c_region = 'AMERICA' AND
  s_nation = 'UNITED STATES' AND
  (d_year = 1997 OR
   d_year = 1998) AND
  p_category = 'MFGR#14'
GROUP BY
  d_year, s_city, p_brand1
ORDER BY
  d_year, s_city, p_brand1
```

Bibliography

[1] *UCAS Statistical Enquiry Tool*. Accessed in November 2010. URL: http://search1.ucas.co.uk/fandf00/index.html

[2] Aberdeen Group (2009). *Data Management for BI: Strategies for Leveraging the Complexity and Growth of Business Data*. Page 8.

[3] Dehne, F., Eavis, T., Rau-Chaplin, A.: *A Cluster Architecture for Parallel Data Warehousing*. Proceedings, First IEEE/ACM International Symposium on Cluster Computing and The Grid (2001). Pages 161-168.

[4] Han, J., Kamber, M.: *Data Mining: Concepts and Techniques*. 1st Edition (2000), Morgan Kaufmann. ISBN-13: 978-1558604896. Pages 121-126.

[5] Marco Costa and Henrique Madeira. 2004. *Handling big dimensions in distributed data warehouses using the DWS technique*. In Proceedings of the 7th ACM international workshop on Data warehousing and OLAP (DOLAP '04). ACM, New York, NY, USA, 31-37. DOI=10.1145/1031763.1031770

[6] Surajit Chaudhuri and Umeshwar Dayal. 1997. *An overview of data warehousing and OLAP technology*. SIGMOD Rec. 26, 1 (March 1997), 65-74. DOI=10.1145/248603.248616 http://doi.acm.org/10.1145/248603.248616

[7] Narasimhaiah Gorla. 2003. *Features to consider in a data warehousing system*. Commun. ACM 46, 11 (November 2003), 111-115. DOI=10.1145/948383.948389 http://doi.acm.org/10.1145/948383.948389

[8] Mark Levene, George Loizou. *Why is the snowflake schema a good data warehouse design?*. Information Systems, Volume 28, Issue 3, May 2003, Pages 225-240.

[9] M. Golfarelli, D. Maio, and S. Rizzi. *Conceptual design of data warehouses from E/R schemes*. In Proceedings of the Hawaii International Conference on System Sciences, pages 334343, Hawaii, 1998.

[10] Raquel Almeida, Jorge Vieira, Marco Vieira, Henrique Madeira and Jorge Bernardino. *Efficient Data Distribution for DWS*. Lecture Notes in Computer Science, 2008, Volume 5182/2008, pages 75-86.

[11] *What Hive is NOT*. Accessed 4th of January, 2011. URL: http://wiki.apache.org/hadoop/Hive

[12] E. F. Codd. 1970. *A relational model of data for large shared data banks*. Commun. ACM 13, 6 (June 1970), 377-387. DOI=10.1145/362384.362685 http://doi.acm.org/10.1145/362384.362685

[13] George P. Copeland and Setrag N. Khoshafian. 1985. *A decomposition storage model*. SIGMOD Rec. 14, 4 (May 1985), 268-279. DOI=10.1145/971699.318923 http://doi.acm.org/10.1145/971699.318923

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. *MapReduce: simplified data processing on large clusters*. Commun. ACM 51, 1 (January 2008), 107-113. DOI=10.1145/1327452.1327492 http://doi.acm.org/10.1145/1327452.1327492

[15] Leavitt, N. (2010). *Will NoSQL Databases Live Up to Their Promise?* Computer, 43:1214.

[16] Jim Gray. 1981. *The transaction concept: virtues and limitations (invited paper)*. In Proceedings of the seventh international conference on Very Large Data Bases - Volume 7 (VLDB '1981), Vol. 7. VLDB Endowment 144-154.

[17] Thomas Kyte (2005). *Expert Oracle*. ISBN10: 1-59059-525-4. Page 135.

[18] Agrawal, R.; Gupta, A.; Sarawagi, S. IBM Almaden Res. Center, San Jose, CA. *Modeling multidimensional databases*. Data Engineering, 1997. Issue 7-11 Apr 1997, Pages 232-243.

[19] M. Gyssens, L.V.S. Lakshmanan. *A Foundation for Multi-Dimensional Databases*. In 23rd VLDB Conference, Athens, August 1997.

[20] Carl Nolan.*Manipulate and Query OLAP Data Using ADOMD and Multidimensional Expressions*. Microsoft. Accessed 18th of June, 2011. URL: http://www.microsoft.com/msj/0899/mdx/mdx.aspx

[21] Microsoft Corporation, Hyperion Solutions Corporation. Accessed 18th of June, 2011. *XML for Analysis Specification*. Version 1.1, 11/20/2002. URL: http://www.xmlforanalysis.com/xmla1.1.doc

[22] Jim Gray et al. *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*. Data Mining and Knowledge Discovery. Volume 1, Number 1, 29-53.

[23] Michael Stonebraker (1986). *The Case for Shared Nothing*. Database Engineering, Volume 9, Number 1.

[24] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. *Integrating compression and execution in column-oriented database systems*. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD '06). ACM, New York, NY, USA, 671-682. DOI=10.1145/1142473.1142548 http://doi.acm.org/10.1145/1142473.1142548

[25] Henry, S.; Hoon, S.; Hwang, M.; Lee, D.; DeVore, M.D.; Dept. of Syst. & Inf. Eng., Virginia Univ., Charlottesville, VA, USA. Systems and Information Engineering Design Symposium, 2005 IEEE. *Engineering trade study: extract, transform, load tools for data migration*. Pages 1 - 8.

[26] Alfredo Cuzzocrea. *s-OLAP: Approximate OLAP Query Evaluation on Very Large Data Warehouses via Dimensionality Reduction and Probabilistic Synopses*. In Proceedings of ICEIS'2009. Pages 248-262.

[27] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. *Implementing data cubes efficiently*. In Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD '96), Jennifer Widom (Ed.). ACM, New York, NY, USA, 205-216.

[28] Wen-Yang Lin and I-Chung Kuo. *A Genetic Selection Algorithm for OLAP Data Cubes.* Knowledge and Information Systems, Volume 6, Number 1, 83-102.

[29] *GQL Reference.* Accessed 24th of May, 2011. URL: http://code.google.com/appengine/docs/python/datastore/gqlreference.html

[30] Daniel Lemire, Owen Kaser, Kamel Aouiche. *Sorting improves word-aligned bitmap indexes.* Data and Knowledge Engineering 69 (1), pages 3-28, 2010.

[31] N. Koudas (2000). *Space efficient bitmap indexing.* Proceedings of the ninth international conference on Information and knowledge management (CIKM '00). New York, NY, USA: ACM. pp. 194201.

[32] *JPivot.* Accessed 18th of June, 2011. URL: http://jpivot.sourceforge.net/index.html

[33] *Mondrian.* Accessed 18th of June, 2011. URL: http://mondrian.pentaho.com/

[34] *Palo.* Accessed 18th of June, 2011. URL: http://www.palo.net/en/

[35] *"Auch Palo ist eine MOLAP-Datenbank."* means *"Palo is also an MOLAP database."* in English. Accessed 18th of June, 2011. URL: http://www.jedox.com/wikipalo/de/index.php/Kategorie:OLAP (German)

[36] Adapted from http://download.oracle.com/docs/cd/B14099_19/core.1012/b13994/disco.htm

[37] Adapted from http://jpivot.sourceforge.net/temp-N101F1.html

[38] *Tuning for Oracle Database on UNIX.* Accessed 29th of May, 2011. URL: http://download.oracle.com/docs/html/B10812_06/chapter8.htm

[39] *An Overview of RMI Applications.* Accessed 30th of May, 2011. URL: http://download.oracle.com/javase/tutorial/rmi/overview.html

[40] *LucidDB.* "It is based on architectural cornerstones such as column-store, [...]". URL: http://www.luciddb.org/

[41] *Advanced Serialization.* Accessed 30th of May, 2011. URL: http://java.sun.com/developer/technicalArticles/ALT/serialization/

[42] K. Wu, E. J. Otoo, A. Shoshani, H. Nordberg. *Notes on design and implementation of compressed bit vectors.* Tech. Rep. LBNL/PUB-3161, Lawrence Berkeley National Laboratory, available from http://crd.lbl.gov/~kewu/ps/PUB-3161.html (2001)

[43] *JavaEWAH.* Accessed 1st of June, 2011. URL: http://code.google.com/p/javaewah/

[44] Pat O'Neil, Betty O'Neil, Xuedong Chen. *Star Schema Benchmark – Revision 3, June 5, 2009.* Accessed 8th of June, 2011. URL: www.cs.umb.edu/~poneil/StarSchemaB.PDF

[45] *Frequently Asked Questions About the Java HotSpot VM.* Accessed 8th of June, 2011. URL: http://www.oracle.com/technetwork/java/hotspotfaq-138619.html

[46] Yanif Ahmad and Christoph Koch. *DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases.* Proceedings of the VLDB Endowment, August 2009.

[47] *Introducing Druid: Real-Time Analytics at a Billion Rows Per Second.* Accessed 18th of June, 2011. URL: http://metamarketsgroup.com/blog/druid-part-i-real-time-analytics-at-a-billion-rows-per-second/

[48] Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*. The MIT Press (2001). ISBN 978-0-262-03293-3.

[49] C. Bettini, C. Dyreson, W. Evans, R. Snodgrass, and X. Sean Wang. *A Glossary of Time Granularity Concepts*. In Temporal Databases: Research and Practice, Lecture Notes in Computer Science 1399, O. Etzion, S. Jajodia, and S. Sripada, editors, Springer-Verlag, pp. 406-411.