NTNU

Norwegian University of
Science and Technology

Where Database Technology Meets Model-Driven Engineering

Rethinking Internal Data Representation in
Genus App Platform

Håkon Åmdal

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Svein Erik Bratsberg, IDI
Co-supervisor: Einar Bleikvin, Genus AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Research has recently shown a keen interest in database technology for analytical workloads that enables high-performance analysis and on-the-fly aggregation, where the main motivation is *Business Intelligence* and*Business Discovery* products. Such products store data in main memory as compressed columns to maximize memory utilization and CPU throughput. *Model-Driven Engineering*, a discipline that aims to increase developer productivity through the use of models on a higher level of abstraction, automates many of the complex programming tasks, like persistence and interoperability. One such product, *Genus App Platform*, has evolved over time and become a powerful and expressive tool for rapid application development. However, operations that process and analyze large amounts of data are slow, and the platform has a high memory footprint, mainly because no particular attention has been paid to storage format and structures in the source code. Based on the observation that *Genus App Platform* has many similarities with an in-memory database, we are motivated to investigate if the challenges in *Genus App Platform* can be overcome by applying techniques used in read-optimized databases.

In this research, we enhance data representation, implement column storage with dictionary encoding and bitpacking in *Genus App Platform* to reduce memory footprint and increase the platform's ability to handle and analyze large datasets. We identify core operations that can exploit the new storage format, like join and filter operations. We test our implementation using a benchmark for analytical workloads while monitoring that transactional performance is not negatively affected.

In *Genus App Platform*, column storage with dictionary encoding, bitpacking, and null pointer compression leads to a memory reduction of 67 % and a load time reduction of 36 % for the TPC-H inspired *Data Mart Load Benchmark*. Also, operations that are adjusted to utilize the column storage format sees a performance impact of one, two, and even three orders of magnitude compared to the original implementation. The new internal data representation in *Genus App Platform* does not significantly reduce transactional performance. Thus, by using *Genus App Platform* as a proof-of-concept, we have shown how techniques used by read-optimized databases increase *Model-Driven Engineering* versatility by enabling such tools to handle and analyze large datasets.

Sammendrag

Forskning har i den siste tiden vist stor interesse for databaseteknologi som muliggjør analyse av data med høy ytelse og umiddelbare beregninger, hvor hovedmotivasjonen for denne forskningen er produkter innen *Business Intelligence* og *Business Discovery*. Slik teknologi lagrer data i RAM på et komprimert kolonneformat for å få høy prosessorytelse og utnyttelse av minne. *Modelldreven Utvikling* er en disiplin som forsøker å øke utviklerproduktivitet ved hjelp av modeller på et høyere abstraksjonsnivå, samt å automatisere mange komplekse programmeringsoppgaver, som datapersistens og interopabilitet. Et produkt innen denne disiplinen, *Genus App Platform*, har over tid utviklet seg til å bli et kraftig verktøy for rask applikasjonsutvikling. Likevel tar operasjoner som analyserer og prosesserer store datamengder lang tid, og plattformen bruker mye minne, hovedsakelig fordi det aldri har blitt fokusert på intern datarepresentasjon. Siden *Genus App Platform* har svært mange likhetstrekk med en minne-basert database, er vi motivert til å utforske om utfordringene i *Genus App Platform* kan bli løst med teknologi benyttet i databaser optimalisert for leseytelse.

I denne oppgaven forbedrer vi datarepresentasjon, implementerer kolonnelagring med *dictionary encoding* og *bitpacking* i *Genus App Platform* for å redusere minneforbruk og forbedre plattformens egenskaper til å behandle og analysere store datamengder. I tillegg identifiserer vi kjerneoperasjoner som kan dra utnytte av det nye lagringsformatet, slik som join- og filtreringsoperasjoner. Vi tester implementasjonen vår med tester designet for analytisk arbeidslast mens vi i tillegg overvåker at skriveytelse ikke blir påvirket negativt.

For *Genus App Platform* ser vi at kolonnelagring med *dictionary encoding*, *bitpacking* og nullpekerkompresjon reduserer minneforbruket med 67 % i våre tester, i tillegg til at lastetiden reduseres med 36 %. Operasjoner i plattformen som har blitt omskrevet til å ta utnytte av det nye lagringsformatet ser en ytelsesforbedring på én, to, eller tre størrelsesordener. Den nye implementasjonen øker ikke skriveytelse signifikant. Vi har, ved å bruke *Genus App Platform* som et eksempel, vist hvordan teknikker benyttet av databaser optimalisert for leseytelse kan benyttes i *Modelldreven Utvikling* for å forbedre støtte for store datamengder; noe som igjen øker allsidigheten til rammeverk for *Modelldreven Utvikling*.

Preface

This Master's Thesis is the final deliverable of the Computer Science Program at the Department of Informatics, Norwegian University of Science and Technology. Master's research at this program consists of two parts: A pre-study, due December 2015, and the main thesis, due June 2016. The research is a collaboration between NTNU and Genus AS. The research is executed by Håkon Åmdal, supervised by Svein Erik Bratsberg at the Department of Informatics, NTNU, and Einar Bleikvin at Genus AS.

The pre-study, due December 2015, focused on *Business Intelligence* related performance improvements. The ultimate goal was to improve ad-hoc analytical abilities in *Genus App Platform* by applying state-of-the-art in-memory database technology. This research resulted in a literature review and a report enumerating the most important aspects when engineering such system.

This thesis builds on the pre-study, but with a broader perspective: It inspired us to reevaluate how *Genus App Platform* and other frameworks for *Model-Driven Engineering* should handle data internally, and see whether the storage formats and techniques from in-memory database technologies could be applied. The background theory in this report, as well as certain parts of the introduction, is, therefore, adapted from the pre-study.

Acknowledgments

First of all, I would like to thank *Genus AS* for proposing an interesting and challenging research project. It has been compelling working with storage representation and optimizations in *Genus App Platform*. Even more important to me, is that I have been working on a real business case and that my research has contributed towards a better product for *Genus AS*' customers.

It is my pleasure to express my gratitude to *Einar Bleikvin*, my co-supervisor at *Genus AS*. He has during the project period always shown interest in my work, been a great discussion partner, and contributed with his years of experience in programming, *Delphi*, and *Genus App Platform*. Most important is that he has trusted me in working on central components in *Genus App Platform*, and helped me fix numerous bugs and memory leaks.

Johnny Troset Andersen deserves recognition for his dedication to this research. Johnny has always provided be with the resources I need to execute the project, which includes access to key personnel within *Genus AS* and relevant literature. Also, Johnny has given valuable feedback on the project report. Last, but not least, he has always provided me with a desk at *Genus AS*' office and made me feel like home.

I am grateful for the support I have received from *Thomas Hedén*. Thomas has helped me establish benchmarks, access customer data, and maintain test environments. Ha has been a key contributor in report quality assurance, regarding language, structure, and content.

Other *Genus AS* employees that deserve recognition include *Bernt Almlid* for assistance with *Genus App Platform* multitenancy and architecture, *Ole Anders Bøe* for maintenance of benchmark environments, *Petter Bergfjord* for valuable feedback and clarifications on Chapter 4, *Martin Børke* for the input on the history of *Business Intelligence* and *Model-Driven Engineering*, and *Nina Holt* for feedback on report structure.

I would like to thank *Svein Erik Bratsberg* for being my supervisor on this project. Svein Erik has kept me on track for the project period and has always held his office door open for me. He has provided me with key literature for this project, and provided valuable feedback on the project report. Svein Erik has been an irreplaceable asset for this project, especially in the final weeks before the deadline.

Lastly, I am extremely grateful for *Marte Løge*'s support.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In this chapter we explain the background information and motivation of this research project, and based on this, we state our goals and research questions. We then proceed to explain how we plan to answer the research questions by stating our research methods, contributions, and deliverables.

1.1 Background and Motivation

There has been an increased interest in database systems that are tuned for ad-hoc analysis and *Business Intelligence* queries recently. These systems utilizes column storage, compression, and paralellization to maximize CPU and memory throughput, and normally builds on in-memory technologies. In general, because RAM is getting cheaper [27], and 64-bit CPUs has become more wide-spread, in-memory databases increasingly play a more significant role [26]. Systems tuned for *Business Intelligence* and capable of using main memory as primary storage include *Oracle Database* [39], *SAP HANA* [28], *Gorilla Time Series Database* [46], *QlikView* [50], *Tableau* [37], *MonetDB* [24], *Blink* [21], and *SAP NetWeaver* [42]. In-memory database systems are used where performance and low latency is a key design goal, and on systems that have no need for persistent storage [75].

Model-Driven Engineering, a discipline that aims to increase developer productivity by raising abstraction levels, has identified that there always will be a gap between the business problem and the implementation [31]. One of the main goals of *Model-Driven Development* research is to create technologies that shield software developers from complexities of the underlying platform. To cope with these complexities, and close the gap between the business problem and the implementation, a thorough understanding of the gap bridging process is required, understanding that is gained through experimentation and accumulation of experience. One of the major advantages of *Model-Driven Engineering* is that models are expressed in such way that they are closer to the problem domain and less bound to the underlying implementation [56]. At most times, *Model-Driven Development*-tools generate programs that are just as memory- and performanceefficient as hand-crafted programs, but there are occasional critical cases where *Model-Driven Engineering* are outperformed by tailored solutions.

Genus AS is one of the market players in *Model-Driven Engineering*, and, like most vendors in the field, aim to close the gap between business logic and implementation through abstractions and models. Their platform, *Genus App Platform*, uses generic software concepts on a higher level of abstraction than regular programming languages that are precise and non-ambigous [5]. These concepts, and the underlying implementation, have been refined and improved through years of trial and error on real customer use cases. However, since the main focus of the source code of the platform has been readability and maintainability, no particular attention has been paid to how data is represented internally. The result of this is that the platform has high memory usage and read-intense operations that process and analyze large amounts of data, such as join and data aggregations, are slow.

Our main motivation for this research is two-fold. First, we have identified a critical case in *Genus App Platform*, and likely *Model-Driven Engineering* in general, which is memory consumption and handling and analysis of large datasets. We are motivated to seek out how this problem can be solved in the context of *Model-Driven Development*. Second, we observe that *Genus App Platform* has many similarities with an in-memory database: Data is fetched from persistent data

sources, manipulated and analyzed in-memory, before being persisted in the data sources again. We are, therefore, motivated to see whether techniques used in state-of-the-art read-optimized can be applied to a *Model-Driven Engineering*-tool. Overall, our research is of interest because it increases the versatility of *Model-Driven Development*-tools and the number of problems where *Model-Driven Engineering* can be applied.

1.2 Research Goals

Based on our motivation, we define the following goals:

> **G1: Reduce memory consumption in *Genus App Platform* and increase the platform's ability to handle and analyze large datasets.**

We set **G1** not only to improve *Genus App Platform* and tackle its performance challenges, but also to see how this problem can be generalized and solved in the context of *Model-Driven Engineering*. By *handling large datasets*, we mean operations and analyses that work on thousands, even millions of elements at a time. Most of these operations, but not all, are associated with *Business Intelligence* related functionality. In addition to **G1**, we also set a broader, but yet intertwined goal:

> **G2: Introduce new evidence that *Model-Driven Engineering* can benefit from in-memory, read-optimized database technologies.**

As mentioned, *Genus App Platform*, and likely other *Model-Driven Engineering* supporting platforms, have many similarities with in-memory databases. We are curious to see whether techniques used in in-memory databases, mainly those optimized for analytical workloads, can be applied in the context of *Model-Driven Engineering*.

To reach **G1** and **G2**, we address the following research question:

> **RQ1: How can technology used by in-memory, read-optimized databases improve *Genus App Platform*'s ability to handle and analyze large datasets, and what can *Model-Driven Engineering*, in general, learn from database technology?**

By answering **RQ1**, we hope to address **G1** directly by making changes in *Genus App Platform* that increase the platform's ability to handle large datasets. However, by using our changes in *Genus App Platform* as a proof-of-concept, we plan to address **G2** by drawing general conclusions on the combination of *Model-Driven Engineering* and database technology.

We are aware that our goals and research question are prematurely presented in this section, as they are based on knowledge about *Model-Driven Engineering*, *Genus App Platform*, and read-optimized databases. However, this section becomes more apparent after reading Chapters 2-4. Section 4.7 will again present our goals and research question, but in the light of the discoveries made that far.

1.2.1 Research Scope

The scope of this research has changed during the project execution. The initial direction of the research aimed directly towards the *Business Intelligence* components in *Genus App Platform*, components which had the mentioned issues: High memory consumption and poor performance. Hence, the research started out with studying how these challenges could be overcome, with an emphasis on in-memory and read-only databases. However, as more insights were gained during the research, it became apparent that our findings could be applied at a wider scope, thus came the idea to use the discovered techniques in the entire platform. Not only would this likely help *Business Intelligence* and the analytical components of *Genus App Platform*, but also other parts of the system.

1.3 Research Methods

To reach our goals and to address our research question, we divide our research into two parts. The first part is a *literature review* on read-optimized, in-memory databases. This part also contains background theory on *Model-Driven Engineering*, *Business Intelligence*, and an analysis of *Genus App Platform*. The second part is a *design and experiment* type research, where we evaluate promising techniques from the literature review by applying them to *Genus App Platform*. The performance impact of the modifications are tested by benchmarks, and conclusions are drawn based on the results.

1.3.1 Related Work

The main goal of the related work research and literature review is to gain a deeper understanding of read-optimized, in-memory databases and find out which technologies that enable high performance and low memory usage. In Section 1.2.1, we saw that the original scope of the research was to improve analytical capabilities in *Genus App Platform*. Thus, this part emphasizes techniques used in online analytical processing (OLAP) databases. In this phase of the research, we also study *Model-Driven Engineering*, *Business Intelligence*, *Business Discovery*, *Delphi*, and provide an analysis of *Genus App Platform*.

Most of the literature review was performed using a method known as *Snowballing*, which is convenient if the scope of the project is uncertain [18]. The Snowballing method is the process whereby you start with a few number of authoritative papers and based on these you expand your list of readings by relevant work that the papers have cited. The identification of papers can also happen in the other direction, where you look for papers that have used the current one as a reference. Either way, this method is known to generate a large number of papers, so the researcher must be very strict and objective in which papers to read.

The initial papers, theses, and books used in this research were found in collaboration with department staff and regarded in-memory databases, columnar storage, and online analytical processing (OLAP) workloads. Both forward and backward searching were performed, and each paper was considered by reading the abstract, and conclusion and introduction if needed, to be put on the reading list. During the search process, we picked articles that could help us answer **RQ1**. When the field felt properly understood, we concluded the literature study. A similar process was used for the residual background theory, although this process was not as thorough as the study on in-memory databases.

To gain a deeper understanding of *Genus App Platform*, a combination of technical briefs, lessons from *Genus AS*' employees, and source code analysis was used. This part also included a study of *Delphi*, the programming language used to develop the *Genus App Platform* core.

The findings from this part of the study result in three chapters in this report: One chapter with background theory on *Model-Driven Engineering*, *Business Intelligence*, *Business Discovery*, and *Delphi*, one chapter that outlines techniques used by in-memory, analytical databases that enable high performance and low memory usage, and one chapter with the *Genus App Platform* analysis.

1.3.2 Evaluation

The second part of the research is a design and experiment type study, where the main goal is to answer **RQ1**. Here, we apply the most promising techniques used by in-memory, read-optimized databases to *Genus App Platform*, and evaluate if they increase the platform's ability to handle large datasets. Also, we see what *Model-Driven Engineering* in general can learn from database technology. According to France *et al.*, new techniques used in *Model-Driven Engineering* requires systematic accumulation of experience [31], and this step provides such experience.

More specifically, we change the internal data representation in *Genus App Platform*. We implement a column store, enhance data formats, and apply compression to reduce *Genus App Platform*'s memory usage. Then we make modifications to exploit the new storage format, which enables *Genus App Platform* to handle and analyze large datasets. We run benchmarks to measure the impact of our changes, both to assess the platform's capabilities to handle and analyze large datasets, but also to ensure that write and update operations have not been affected negatively.

We perform the research iteratively to gain a better understanding of the effects of each technique. For every iteration, relevant benchmarks are run, results discussed, and conclusions are drawn. Each iteration does not only identify methods which require further investigation and exploitation but also discards unpromising techniques or leaves them for future work. The iterations first seek to reduce memory footprint, and when the memory usage is reasonably low, efforts are put in to increase the performance of operations in *Genus App Platform* by utilizing the new

storage format. The iterations are *Genus App Platform* specific and aim to reach **G1**.

After the research implementation iterations, we discuss our results holistically and aim to address **G2**. In this part, we discuss the implications for *Genus App Platform*, *Model-Driven Engineering*, and traditional programming languages.

1.4 Contributions

The main deliverable from this research is this report which contains literature study, implementation details, tests results, and discussions of our findings. The report answers **RQ1** and address both goals stated in Section 1.2 and has an emphasis on the design and experiment part of the research. The source code is not a part of the deliverables for this research, although class diagrams and some listings are provided in the report.

This research contributes to an *improved computer-based product*, and it is mainly *Genus App Platform* which sees these improvements. However, our goal is to increase the versatility of *Model-Driven Engineering* supporting architectures, and we hope other *Model-Driven Development*-tools benefit from our findings.

Second, and more important, is that we *introduce new evidence* that *Model-Driven Engineering* can benefit from technologies used in read-optimized, in-memory databases. In other words, we *re-interpret* the database technology and apply it to the context of *Model-Driven Engineering* and re-evaluate how *Model-Driven Development*-tools should represent data internally. We have not found any research taking a similar approach.

1.5 Thesis Structure

We structure our thesis into three parts.

1.5.1 Related Work

The first three chapters of the thesis are associated with the first research phase, which is the related work and literature review.

- **Chapter 2** introduce relevant background material to this research. The chapter contains sections about *Model-Driven Engineering*, *Business Intelligence*, *Business Discovery*, and *Delphi* programming language.
- **Chapter 3** is a chapter on read-optimized databases. This chapter explores techniques used by these databases to achieve high performance and low memory usage, as well as how such systems are benchmarked.

- **Chapter 4** contains a top-down analysis *Genus App Platform*, containing platform architecture, application development, concepts, and source code. The last part of this chapter identifies challenges in *Genus App Platform*, which motivates the design and experiment part of the research.

1.5.2 Evaluation

The next part is the main research phase, where techniques learned from the literature review are implemented in *Genus App Platform* and evaluated. The first four chapters correspond to the four research iterations, where each chapter contains an introduction, an implementation, results, a discussion, and a conclusion.

- **Chapter 5** presents how column store is implemented in *Genus App Platform*.
- **Chapter 6** demonstrates how representing data as primitive data types in the columns reduce memory footprint.
- **Chapter 7** shows how compression techniques reduces *Genus App Platform* memory consumption.
- **Chapter 8** investigates operations that exploit the column format, like join and filter operations.

There is also one additional chapter in this part which does not correspond to any of the four research iterations.

- **Chapter 9** elaborates on techniques that were explored in this research and worthwhile to discuss, but not tested enough to draw any conclusions. Here, we examine the implications of UTF-8 string encoding and how database statistics can be used to select the correct storage format.

1.5.3 Discussion and Conclusion

The last part discuss and concludes the findings in this research.

- **Chapter 10** discuss our findings holistically and presents the implications for *Genus App Platform*, *Model-Driven Engineering*, and traditional programming languages. It also provides a section on research limitations and critics.
- **Chapter 11** concludes this research and points at interesting directions for future work.

Part I

Related Work

Chapter 2

Background Theory

This chapter introduces important background theory relevant to this research. First, we introduce *Model-Driven Engineering*. Then, to introduce a use-case where handling and analysis of large datasets is required, we give a brief introduction to *Business Intelligence* and *Business Discovery*. Last, we study *Delphi*, the programming language used in the *Genus App Platform* core.

2.1 Model-Driven Engineering

Since the introduction of computers, researchers have been working to raise the abstraction level at which software developers create programs [19]. One example of this is compilers and how programmers no longer need to know assembly language for the different processor architectures. Now, object-oriented programming languages enable software developers to create own abstractions and solve more complex problems than ever. *Model-Driven Engineering* is a continuation of this trend, where the goal is to automate many of the complex, but routine, programming tasks. Such tasks include interoperability, distribution, and persistence. One of the key technological foundations of *Model-Driven Engineering* is support for visual modeling.

Model-Driven Engineering research has identified that there is a gap between the problem domain and the software implementation domain [31]. Thus, most research focuses on bridging this gap. This process requires systematic experimentation and accumulation of experience, and new techniques should strive to reduce unnecessary complexities.

The ability to express models with concepts that are closer to the problem domain and less coupled to the underlying implementation is one of the main advantages with *Model-Driven Engineering* [56]. This makes programs easier to understand and maintain, as well as making program specification and requirements easier to communicate. In addition, according to Atkinson *et al.*, *Model-Driven Engineering* has two more major advantages [19]:

- In the short term, *Model-Driven Engineering* increase developer productivity and increase the amount of functionality an IT artifact can deliver.

- In the long term, *Model-Driven Development* supporting infrastructures are more maintainable, and it takes a longer time before the artifact becomes obsolete.

A common discussion topic in the field of *Model-Driven Engineering* is the efficiency of the modeled programs. Criticism has pointed out that machines cannot optimize code better than a creative human being with clever tricks [56]. However, it is widely known that modern compilers outperform most system developers, and it does so more reliably. The same observation can be done for most *Model-Driven Development* supporting infrastructures. Current tools generate programs that are within 5 to 15 percent effective as hand-crafted systems, both regarding memory consumption and performance. However, there are still critical cases where manually written code significantly outperforms *Model-Driven Engineering*, which has often been used as an excuse to discard a *Model-Driven Development* approach altogether.

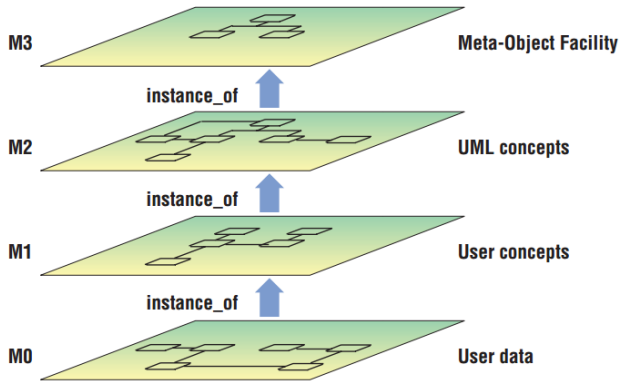


Figure 2.1: Traditional *Model-Driven Development* with Unified Modeling Language (UML) and Meta-Object Factory (MOF) model levels. (Adapted from [19])

2.1.1 Models and Model Levels

Central in *Model-Driven Engineering* is the use of models. According to Leppänen M, a model is a thing that is used to help the understanding of some other things [43]. A good model must appeal to our intuition, and it should reduce the amount of intellectual effort required to understand what the model represents [56]. In the field of *Model-Driven Development*, there are multiple model levels, where a model on a higher level describes/prescribes models on the next lower level.

Leppänen M distinguish between instance models, type models and meta models [43]. An *instance model* is a model which contains concepts that are instances of some other model, the *type model*. Analogously, the *type model* is an instance of the *meta model*. The latter is used to enable understanding, communication, analysis and design of models. *Model levels* are composed of models that comprise concepts on the same level. There are four model levels: instance level, type level, meta level, and meta-meta level.

Depending on the research field and what is being represented, the four model levels have different names and interpretation [43]. In information processing, the layers are:

- **Information System (IS) layer**, typically represents day-to-day information processing actions in an organization, like order processing or inventory control. This layer represents changes in an application.
- **Information System Development (ISD) layer**, which is the layer where information systems are analyzed, designed, implemented, and tested. This layer contains descriptions of a specific application.
- **Method Engineering (ME) layer** is where techniques and procedures for the ISD layer are developed, selected, configured and customized. In

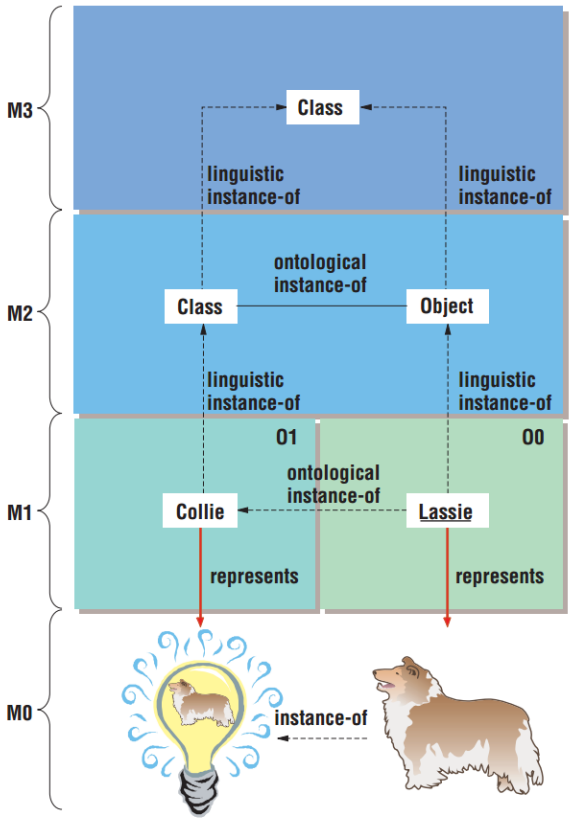


Figure 2.2: Linguistic metamodeling. Linguistic relationships span across model levels while ontological relationships are contained within the same model layer. (Adapted from [19])

this layer, new programming tools are created, for instance *Model-Driven Development*-architectures and programming languages.

- **Research work (RW) layer**, involves research that aims to produce better methods and concepts for the method engineering layer.

Unified Modeling Language (UML) and Meta Object Facility (MOF) is the first generation of *Model-Driven Development* infrastructure [19], and is illustrated in Figure 2.1. This generation has defined central components and terminology in *Model-Driven Engineering* research. However, this method has been criticized for lacking a clear description of how different layers relate to the real world. Also, it has a single-instance relationship between the model levels, which causes inconsistencies.

To overcome the challenges with UML and MOF, other frameworks have been

Figure 2.3: Class diagram in *Mendix*. (Adapted from [34])

proposed. One such framework is *linguistic metamodeling*, which separates linguistic relationships from ontological ones [19]. As seen in Figure 2.2, linguistic relationships span across model levels, while ontological relationships are typically on the same level.

2.1.2 Model-Driven Engineering in Practice: Mendix

Research on *Model-Driven Engineering* has resulted in several tools and methods. Some of these build on a sound, theoretical foundation, others take a more pragmatic and practical approach. Either way, the success of a tool taking a *Model-Driven Development* approach is heavily dependent on how application development in these tools is done in practice [34]. In this section, we study *Mendix Business Modeler*, or *Mendix*, to see how this product has approached *Model-Driven Engineering*.

Applications in *Mendix* are designed using three main models: An *information structure model*, which defines the main data objects in the application, a *microflow model*, which defines logic, and a *form model*, which defines system user interface. Applications are designed in a modeling tool and deployed to a model repository, where they become available for end users.

The *information structure model* defines the main data structure in an application, and is designed in an UML-like class diagram. As seen in Figure 2.3, this model does not only suport class definitions with properties, but also relations between objects. This model may also define validation rules and events which trigger on object creation, modification, and deletion.

In *Mendix*, the *microflow model* define complex logic in an application and allows the designer to extend the system with custom behavior. Microflows may change objects, control how and when forms are displayed, call custom Java code, or integrate with external web services. Microflows are designed in a visual editor which we see in Figure 2.4.

Forms and other graphical user elements are designed using a form editing tool, similar to those found in popular programming IDEs. The form editing tool is depicted in Figure 2.5. *Mendix* supports basic input fields, drop-down lists, tables, and more. A form can be associated with objects, other forms, and microflows.

When the application is deployed to the model repository, end users can access the

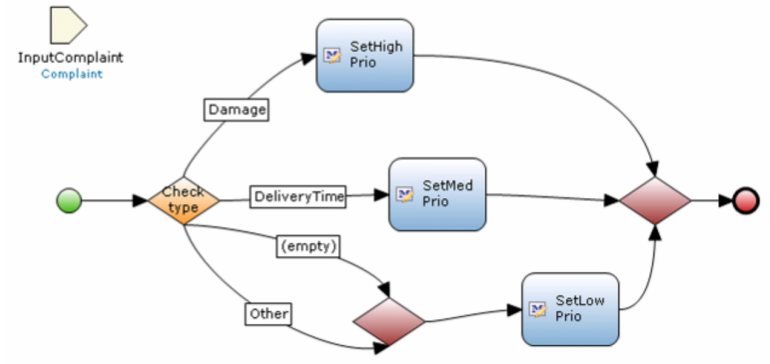


Figure 2.4: A microflow in *Mendix*. (Adapted from [34])

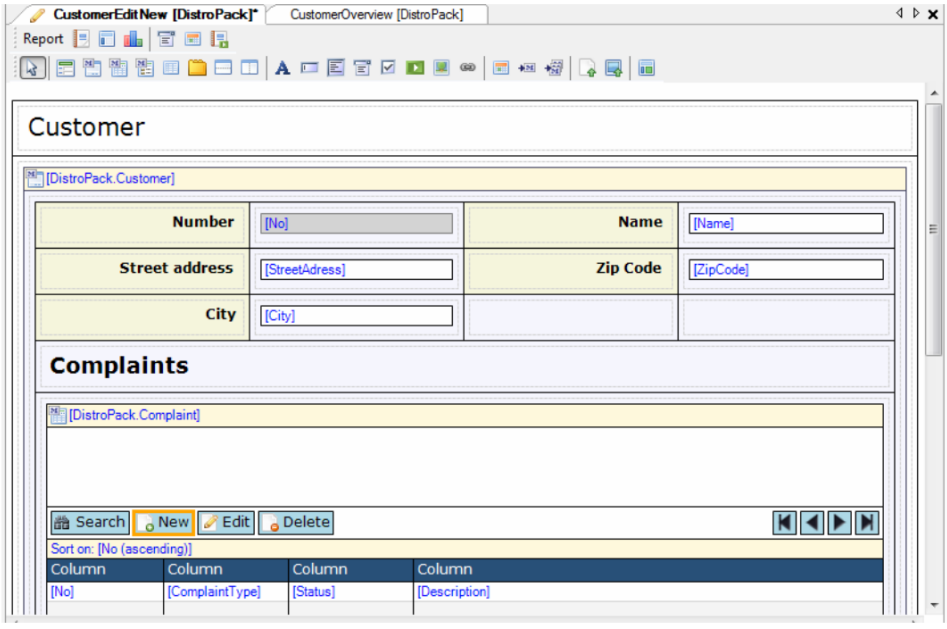


Figure 2.5: Form designer in *Mendix*. (Adapted from [34])

system using a web browser. When a user requests a web page from the server, the HTML and JavaScript required to show the page is created and sent to the browser. This interpreted approach is different from other *Model-Driven Engineering* tools, like *OptimalJ*, which relies heavily on code generation.

Genus App Platform has many similarities with *Mendix*. We study *Genus App Platform* in Chapter 4.

2.2 Business Intelligence

We introduce *Business Intelligence* in this thesis to explain a common use-case where the ability to handle and analyze large datasets is required. *Business Intelligence* reqirements in *Genus App Platform* is one of the key motivations to this research.

Business Intelligence is normally described as tools and techniques used to transform unstructured data into useful and meaningful information that can be used to support decisions [62]. The goal is to allow for easy interpretation and gain insight into the data, such that businesses end up with a competitive market advantage. *Business Intelligence* software can assist in making a wide range of business decisions, including strategic decisions as goals and product pricing or positioning, as well as operational decisions like priorities.

A challenge in *Business Intelligence* can be information overflow. To know which information is needed in a *Business Intelligence* application, decision makers must be aware of which types of decisions they should make, and have a model for each [17]. Since managers rarely fulfill the latter requirement, they add a safety factor and asks the IT department to provide everything. The result is information overload, and the much of the data is irrelevant.

2.2.1 Data Warehouses

Companies' need for *Business Intelligence* has traditionally been solved by data warehouses. These data warehouses extract, transform, and load data into structures that are suited for analytical queries and report generation.

The challenges with data warehouses are many, among them, the lack of flexibility. Reports are usually preconfigured and implemented by the IT department, which can result in lengthy reporting backlogs. Besides, the *QlikView* developers have pointed out some other drawbacks of traditional query-based *Business Intelligence* tools [49]:

- Only small subsets of the main dataset are extracted at a time. These subsets are divorced from the data that was not included in the query.

- Each query represents a single piece of information, and the information gathered from individual queries is hard to combine.
- Traditional systems do not maintain relationships between queries. A query can be hard to formulate, and it is not always easy to know what to look for. Traditional *Business Intelligence* applications do not let the user build queries step by step.

2.2.2 Star and Snowflake Schema

Many *Business Intelligence* applications and data warehouses usually access data that is organized in a *star or snowflake schema* [21]. Distinct for such schemas is that they have a huge fact table, which can have millions or billions of rows, and smaller dimension tables, each representing some aspect of the fact rows (e.g. category, region, time). The fact table is connected to the dimension tables using foreign keys. A snowflake schema is an extension of the star schema, where one or more dimension tables can have relationships that further describe a dimension.

Star and snowflake schemas are typically used in *Business Intelligence* applications because they are easier to optimize [40]. The query optimizer creates efficient query plans by filtering and applying joins on the most highly selective dimensions first. Secondly, queries on star and snowflake schemas are easier to anticipate, such that indexes, materialized views, and/or denormalization can be applied to improve query efficiency [21].

The disadvantage of using star and snowflake schemas for *Business Intelligence* is the lack of flexibility. There are certain situations where there are more than one large fact table and situations where there is no clear distinction between fact and dimension tables.

2.3 Business Discovery

To overcome the challenges with traditional *Business Intelligence* systems, a new type of products have emerged. We call these for *Business Discovery* products, a notion that was introduced by *Qlik* [51]. *Business Discovery* products normally build on in-memory technologies and are fast, elegant, and end user intuitive solutions to analyze business data. Examples of such products are *Microsoft PowerPivot*, *Tableau*, and *QlikView*.

Business Discovery products allow users to follow their ”information scent” or ”train of thought” when navigating through the data [37, 51]. As seen in Figure 2.6, there are no prespecified drill-down patterns, and users decide where to start and end. In these applications, grouping, joining, and calculations are performed on-the-fly. High-performance, in-memory technologies are used to enable such functionality.

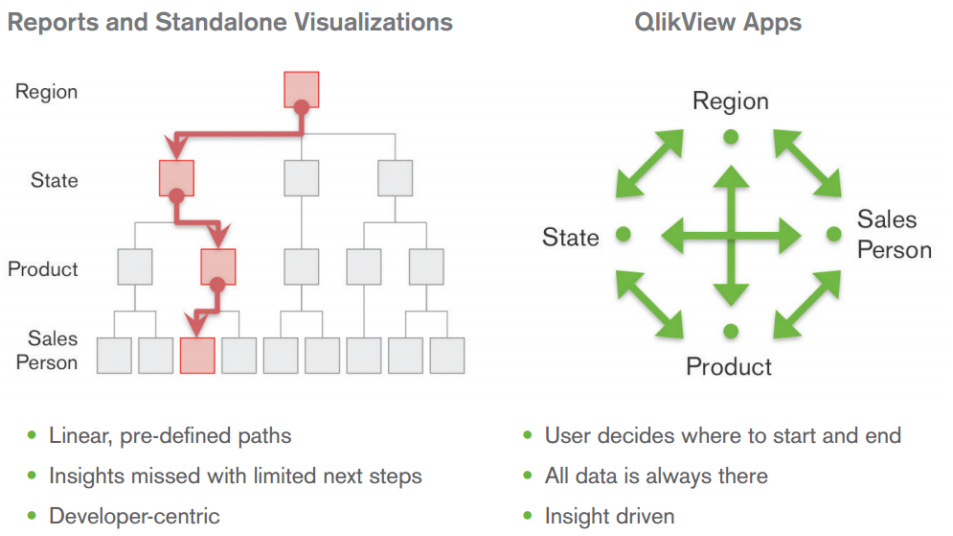


Figure 2.6: Comparison of a traditional reporting application and *QlikView*. Traditional *Business Intelligence* applications normally have predefined drill-down paths. *QlikView* allows the user decide where to start and end. (Adapted from [51])

In the next sections, we explain how a typical *Business Discovery* application works by using *QlikView* primarily as an example. *Tableau* and *Microsoft PowerPivot* work similarly.

2.3.1 Data Import

Data is loaded into a *QlikView* document using a data import script that connects the application with data sources like databases or files. The script uses an SQL-like syntax that lets the user specify names of fields and tables that are used in the data analysis. When the data import script is run, data is fetched from the sources and put into the *QlikView* in-memory engine such that data can be queried efficiently. No queries are proxied to the underlying data sources, *QlikView* manage all queries internally.

The data import script will regularly include more than one table. Multiple tables are *associated* if they have fields with the same name, as seen in Figure 2.7. Internally, there can be only one data connector between a pair of tables, so if multiple tables refer to a single table, a mechanism (either *synthetic keys* or *loose coupling*) must be applied to logically duplicate the table. This restriction ensures that there exists no more than one possible join path between any pairs of tables [14].

In the data import step, data might be preprocessed and transformed. First, data can be aggregated, for instance by summarizing or grouping, before being loaded

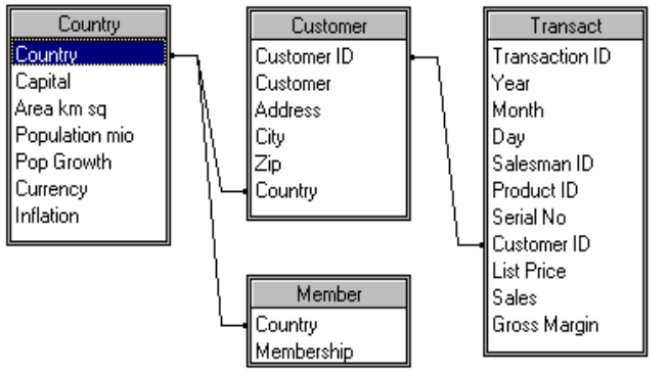


Figure 2.7: Four tables: Countries, customers, transactions, and memberships. The fields *Country* and *CustomerID* associate the tables. (Adapted from [52])

into *QlikView*. Second, data might be filtered. Both these techniques reduce data volumes. Lastly, data can be denormalized in the import script, i.e. pre-joining tables before import. Denormalizing can be done to lower the number of tables in the data extract and, by doing so, reduce application complexity.

2.3.2 User Interface

Users interact with the *Business Discovery* application through a reporting dashboard, as seen in Figure 2.8. Requirements for such panel are typically [51]:

- Clicking field values in list boxes.
- Lassoing data in charts, graphs, and maps.
- Manipulating sliders.
- Choosing dates in calendars.
- Cycling through different chart types.

Users navigate through the data by making selections in the user interface. When a selection is made, the item is made green, as seen in Figure 2.8. The current selection is also known as the *application state*. Upon selection, the rest of the elements in the panel are updated based on the new application state; aggregations are recalculated, and graphs and lists are updated. *QlikView* colors matched elements white and unmatched elements gray. Dashboards are typically available from different devices, including desktop computers, tablets, and mobile phones.

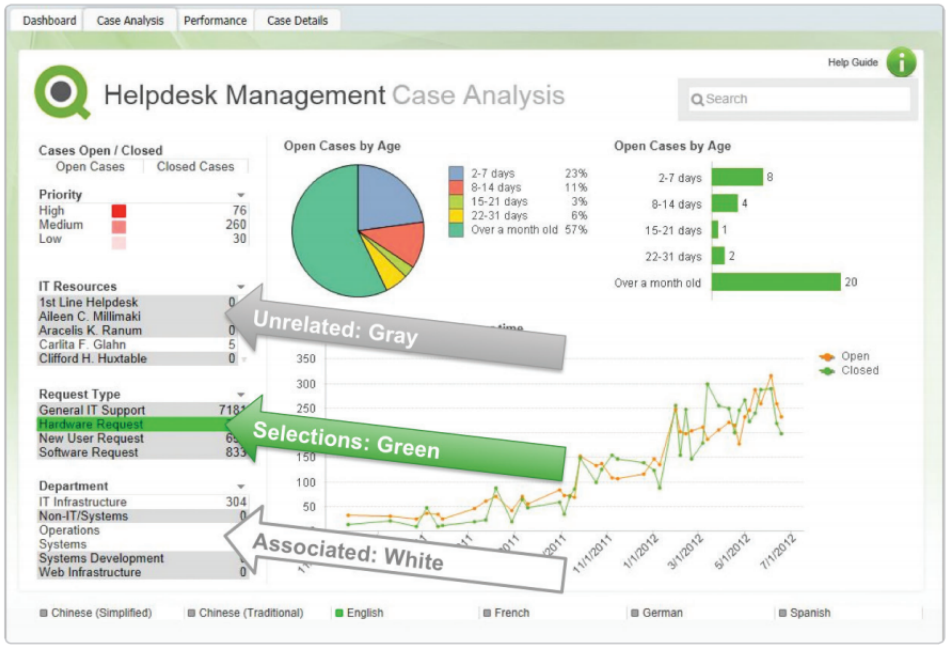


Figure 2.8: *QlikView* dashboard with various GUI elements, like lists and charts. Selections are green, matched data is white, and unrelated data is grey. (Adapted from [51])

2.3.3 Business Discovery and Queries

We see that *Business Discovery* products interact with the data using selections and filters in a reporting panel. This technique is different from database systems designed for OLAP workloads, which normally use a *query-based* interface, usually SQL. Still, *Business Discovery* products need to provide most functionality that SQL databases do, like row listing, filtering, joining, grouping and aggregation, and sorting.

Queries, or requests, in a *Business Discovery* application, have certain characteristics. First of all, they cannot be anticipated, more specifically, they are *ad-hoc*. Secondly, there is a limit to the number of returned rows. Users are interested in queries that can be analyzed quickly, so we do not expect a *Business Discovery* application to return thousands of rows as a result from a single table [30]. Also, the user interface has limitations in how much data that can be displayed at the same time.

2.4 Delphi Programming Language

The core of *Genus App Platform* is written in *Delphi*. It is, therefore, important to understand how this language works. *Delphi* is both a programming language and an integrated development environment [72], however, we will only consider the programming language in this section.

Delphi is a strongly typed, high-level, object-oriented programming language [4, 72]. The language is based on *Object Pascal*, a *Pascal* dialect. *Delphi* supports polymorphism and interfaces, Unicode, inline assembly, generic programming, pointer arithmetic, function overloading, and more. Simple preprocessing directives are allowed, but there is no support for macros. The *Delphi* compiler is efficient and compiles source code into native binaries for a wide variety of platforms, including *Windows* 32- and 64-bit architectures and *OSX* 32-bit architecture.

A program written in *Delphi* is composed of different source code units, with two mandatory parts: The *interface*, much like a header in *C*, which declares constants, types, variables, and function signatures, and the *implementation*, which contains the actual executable code [7]. Types, variables, and functions might also be declared in the implementation part. However, these will not be made accessible outside the unit. Each unit specifies its dependencies to other units, and the dependency graph is created automatically without using makefiles or similar mechanisms.

2.4.1 Delphi Types

Delphi supports a wide range of data types. In the language taxonomy, there is a total of seven different type categories. In this section, we elaborate on the three most relevant categories for this research.

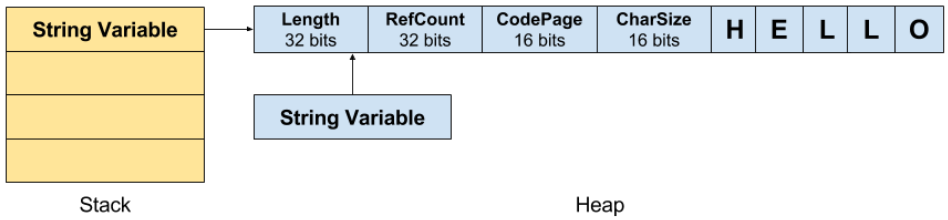


Figure 2.9: String structure in *Delphi*. All strings are heap allocated, although string pointers may exist both in the stack and on the heap.

Simple Types

Simple Types in *Delphi*, which are sometimes referred to as primitive data types in other programming languages, are divided into two categories: Ordinal and real [11]. Simple data types are byte aligned, which means no data type takes less than 1 byte.

Ordinal types, which define ordered sets of values, include integer, character, Boolean, enumerated, and subrange types. The different types have different sizes and ranges, for instance, integers can be represented by 32 or 64 bits.

A real data type defines a set of numbers that can be represented with a floating-point notation. Types include single (4 bytes) and double (8 bytes) precision floating point numbers, as well as a fixed-point data type for currencies.

String Types

Delphi has string support built into the language. Strings are reference counted and dynamically allocated on the heap by the compiler [72]. A string variable is a pointer to a structure that contains a 32-bit length indicator, a 32-bit reference count, a 16-bit data length indicating the number of bytes per character, and a 16-bit code page [12]. This is depicted in Figure 2.9. There is copy-on-write semantics if one variable that references the string changes the contents of the string.

There are two major string types in *Delphi*: `AnsiString` and `UnicodeString`. `UnicodeString`, which is the default, supports both UTF-8 and UTF-16 encodings, where UTF-16 is default on the *Windows* platform. UTF-16 stores each character with two or four bytes, while UTF-8 stores the 128 ASCII characters with only one byte. Conversions between string types and encodings happen implicitly.

Structured Types

For representing more complex data structures, *Delphi* provides a type category named *structured types*. This category include set, array, record, file, and class

types [13].

Delphi has two main array structures: Static and dynamic. Static arrays are allocated with a certain length, and will take up as much memory as the assigned length, even if values are not assigned. Dynamic arrays are more versatile; if a value that has not yet been assigned gets assigned, the array is reallocated. Dynamic arrays might also be reallocated using the `SetLength` function. Like strings, dynamic arrays are reference counted, but they do not have copy-on-write semantics.

The `record` type commonly represent a heterogeneous set of values. This type has value semantics, i.e. is not a boxed type, which means data is copied on assignment and passed by value as function arguments. Hence, records can both be allocated on the stack and the heap. Records are byte-aligned by default.

Classes in *Delphi* are dynamically allocated blocks of memory whose structure is determined by the class definition [2]. Class instances are allocated and deallocated by the memory manager by calling the constructor and destructor respectively. Values are stored in the same order which they are declared in the source code. In addition to the variables, the first 8 bytes of an object points to a virtual method table. Variables of class types are 64-bit pointers and can hold the value `nil`.

2.4.2 Memory Management

In *Delphi*, the memory manager is replaceable and dependent on which platform the code is compiled for [8]. Programs compiled for *Windows*, which is the case for *Genus App Platform*, use the *FastMM* memory manager. *FastMM* is optimized for programs that allocate small to medium size blocks, and it anticipates future block reallocations which reduce address space fragmentation. Block sizes are rounded up to the nearest 16 bytes.

2.4.3 Calling Conventions and Inlining

In *Delphi*, parameters are transferred to functions in registers, on the program stack, or both [9]. Which one is used is dependent on the calling convention of the function and the type of the parameters. Simple data types, unless the `var` keyword is specified, is passed by value while most other values are passed by reference.

The default calling convention is the *register* calling convention. This convention uses up to three registers to pass parameters to functions. The remaining parameters, if any, are passed on the program stack. This convention is the most efficient, as it usually avoids creating a new stack frame [9]. Access methods for class properties must use this convention for this reason.

The *Delphi* compiler supports function inlining, which is a measure that may result in faster code at the expense of space [1]. By using the `inline` keyword in a method

or function definition, the programmer gives the compiler a hint that the function body should insert directly instead of generating a function call. The keyword is a suggestion, and there are several conditions where inlining cannot happen. For instance, the compiler will not inline virtual methods or functions containing assembly code.

2.4.4 Standard library

Delphi comes with an extensive standard library. This section lists some built-in classes which we have used in this research.

TArray

`TArray` is the generic implementation of dynamic arrays (`array of *`), which means it is referrence counted and dynamically allocated. The array is reallocated using the `SetLength` function. Its performance is studied in Appendix B.

TList and TObjectList

`TList` and `TObjectList` are wrappers for dynamic arrays with built-in memory handling. For instance, elements can be added to the end of the list without any explicit memory reallocation. `TObjectList` is like `TList`, but it assumes ownership over objects and automatically frees objects that are removed from the list. We compare `TList` performance with `TArray` in Appendix B.

TBits

`TBits` is a class that represents a bitmap. It has more or less the same interface as a list of booleans, except that this class is optimized for speed and memory using bitwise operations and assembly code. The `OpenBit` method finds the first open bit, or 0, in the bitmap.

Chapter 3

Read-Optimized Databases

This chapter forms the background theory on databases that are optimized for analytical workloads, where the main motivation is to investigate which technologies enable high-performance data processing to accommodate such workloads. The chapter elaborates on storage formats, compression, and testing, and also background theory on modern CPUs. We emphasize techniques used in in-memory databases, although most of our findings apply to disk-based databases as well.

The sources used to form this chapter were found using the *Snowballing* method. During this research, we identified several database systems and *Business Intelligence* solutions which apply the various techniques.

3.1 Database Terminology

This section presents terms and definitions central in the database field and important to understand for the contents of this chapter.

Online Analytical Processing (OLAP) We use the term Online Analytical Processing (OLAP) extensively in this chapter. By OLAP, we mean systems that enable users to analyze multidimensional data interactively from multiple perspectives [67]. OLAP is usually dominated by ad-hoc, complex queries that group, aggregate and summarize over large datasets [22]. OLAP systems can be both disk and memory-based. Column storage is considered to be an attractive solution for OLAP systems, a technique we study further in Section 3.2.

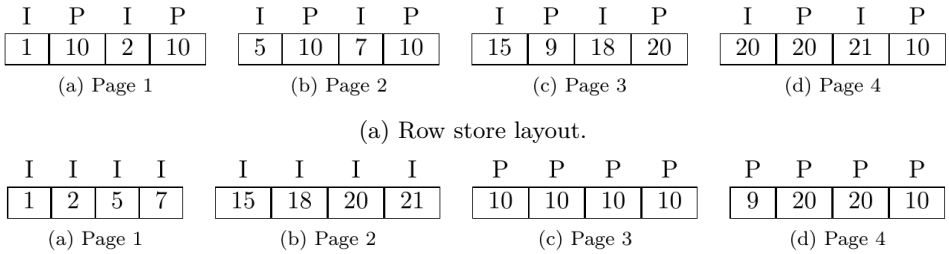
Online Transactional Processing (OLTP) Online Transactional Processing (OLTP) is a class of database systems that manage transaction-oriented applications [68]. Transactional workloads are typically referred to as insertion of new records, as well as updates and deletes of single records in the database. An OLTP system normally uses row storage for its data.

Database Management System (DBMS) A Database Management System (DBMS) is a computer software application for storage and analysis of data [64]. The most common way to interface with a database is through SQL, although other methods exist. Regarding performance, DBMSes can focus on analytical workloads (OLAP), transactional performance (OLTP), or both. In this literature review, we look at several systems, including *Oracle Database*, *IBM DB2 with BLU Acceleration*, *SAP HANA*, *SAP NetWeaver*, *Microsoft SQL Server*, *C-Store*, *Vertica*, *Blink*, *EXASOL EXASolution*, *Oracle Database*, *HyPer*, and *Hyrise*.

3.2 Column Storage

The most common storage format for OLTP systems is row storage, as we briefly mentioned in Section 3.1. Row storage enables easy fetching of values from the same tuple and is suited for updates, inserts, and deletes. However for OLAP workloads, columnar storage has turned out advantageous, mainly because of two reasons: First, aggregations are easier, since calculations are performed on data consecutive in memory. Second, column storage does not fetch more data than is needed for the query.

Our research has identified several systems using columnar storage. These include *MonetDB* [23, 24], *C-Store* [58], *SAP HANA* [28], and *Microsoft SQL Server* [3, 41], as well as the *Business Discovery* product *Tableau* [37].



(a) Row store layout.

(b) Column store layout.

Figure 3.1: Row and column oriented layouts for a table with two columns, I and P. In the row-oriented layout (a), records (I and P tuples) are stored next to each other within the pages. In the column-oriented layout (b), values from the I column and P column are stored separately on different pages. (Adapted from [22])

In a column store, each column in a table is stored separately in a continuous segment (unless data is horizontally partitioned, see Section 3.2.4), as opposed to row stores where attributes from a single row are stored together [22]. Figure 3.1 depicts both row and column oriented layouts for a table with two columns (I and P). The row storage alternates between I and P to store records next to each other on the pages while the column storage keeps I and P values separate.

The advantages of using column storage are many. The primary one is that no more data is accessed than strictly necessary for a query. In addition to this, columns are inherently more compressible [3]. Compression leads to higher performance due to better cache and memory utilization, and this effect is one of the reasons why *Microsoft SQL Server* use column storage. We elaborate on the performance benefits of compression in Section 3.3.

Column storage also comes with a more subtle advantage; columns have a *low degree of freedom* compared to row storage [23]. When operating on column values, only the local memory offset is required, not the global table layout. This removes some layers of indirection and query processing can be made more efficient. Boncz *et al.* claim that this is the main reason why column storage is advantageous.

One of the major disadvantages with column store is that it is not as easily updateable [22], especially if the columns are compressed or sorted. This challenge is typically overcome using a separate structure for writes and updates called a *delta store*. Using such structure, the main part of the database is stored column-wise in a static structure optimized for reads while the updates and inserts are accumulated in a smaller and more dynamic structure. We study delta stores in Section 3.14.1.

Another disadvantage with column storage is tuple materialization costs. Since the result of most DBMS queries should be returned as rows, columns must be stitched back together before returned to the client, an operation that can be expensive.

3.2.1 Sorting

Few systems sort values within a column, with the exception of *C-Store* [58] and *Vertica* [40]. Sorting values in a column store comes with some advantages. First, single value lookups are easily performed by a binary search. Second, and perhaps most important, is that sorted columns can be compressed aggressively by applying *Run-Length Encoding*. Run-length encoding is the main reason *C-Store* stores sorted data. We study this type of compression in Section 3.7.

Except from *C-Store* and *Vertica*, our research has shown little indication that sorting values in columns are common. For instance, *Microsoft SQL Server* [41], *Blink* [54] and *Oracle Database* [39] accept values in the order they appear.

3.2.2 Row Stores vs. Column Stores

Most research agrees that row stores are most suitable for OLTP workloads, and column stores are most appropriate for OLAP workloads. Abadi *et al.* set out to investigate whether there is a fundamental difference between row and column stores [16]. In their research, they used a row store with a vertically partitioned schema to mimic a column store. They also tried applying indexes to each column such that each column could be accessed independently. Their conclusion was that there *is something fundamental about column stores* that makes them perform so well, and that changes must be done to both storage layer and query executor to obtain the benefits of a column-oriented approach. The main reasons why column storage is better suited for OLAP workloads are:

- *Compression*, which we discuss in Section 3.3.
- *Vectorized execution*, which we discuss in Section 3.10.2.
- *Late materialization*, which we discuss in Section 3.10.3.

There are situations for OLAP databases where a row store performs better than a column store. A research executed by Holloway *et al.* shows that a row store can outperform a column store when processing time is the dominating constraint [35]. This is typically the case for low selectivity queries and queries with many predicates. To further improve row storage performance, tuples can be compressed. However, row storage will most likely never beat column stores for OLAP workloads, since bandwidth requirement for processing rows is higher than for columns.

We have also identified papers that claim OLTP databases also benefit from a columnar storage. The work of Farber *et al.* argues that columnar storage is suited for transactional workloads as well, mainly due to the compression [28]. Also, storing data in columns allows for dropping indexes, which is usually costly to maintain. Last, there are usually a lot more read operations than inserts, updates, and deletes in an OLTP database.

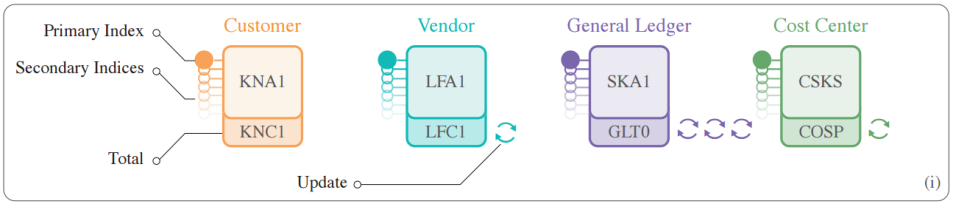


Figure 3.2: OLTP workloads will affect more than a couple of rows. Index structures must be maintained, and aggregations and materialized views must be updated. In the figure, an update that triggers a chain reaction is depicted. (Adapted from [47])

Plattner *et al.* claim that most OLTP queries request aggregates instead of single rows [47]. Also, updates to the database normally trigger a chain reaction of updates to indexes and materialized views, as seen in Figure 3.2. Their conclusion is that column storage is suited for OLTP databases due to efficient aggregation and absence of indexes. The absence of indexes also makes application development easier, since no performance layer must be specified by the application programmer.

3.2.3 Row Identifiers and Tuple Materialization

A row in a column store is identified by a unique identifier that is common to every value belonging to the same row in a table. Many systems store these IDs implicitly as virtual object IDs (`void`). A `void` for an object is calculated using a base ID and the offset from the first value in the column. For instance, the fourth value of a column with base ID 100 has an implicit ID of 103. `void` type identifiers are used in *MonetDB* [24], *C-Store* [58], *Vertica* [40], and *IBM DB2 with BLU Acceleration* [54]. Although stitching together rows in a column store is a trivial operation, it comes at a higher cost than in a row store.

If horizontal partitioning is used, a technique we discuss in the next section, the partition number must also be accounted for in row identification. For instance, *Microsoft SQL Server* identifies a row by a combination of row group ID and tuple ID [41].

3.2.4 Horizontal Partitioning

Several systems split columns horizontally. Partitioning data horizontally can be beneficial due to the following reasons:

- Storing metadata, like minimum and maximum values per block, exploits clustering in the data. A partition block can be skipped entirely if a predicate is outside the value range of a block.

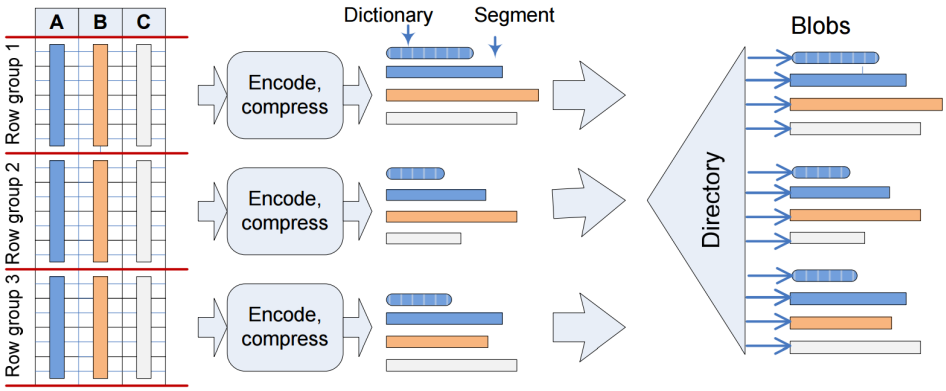


Figure 3.3: This figure illustrates how a column store index in *Microsoft SQL Server* is created and stored. The set of rows is divided into row groups that are converted to column segments and dictionaries. (Adapted from [41])

- If each partition has its dictionary, the dictionary can be scanned for the presence of a key. If the key is not there, the partition can be skipped.
- Smart partitioning of data based on the value frequencies handles data skew and allows for improved compression rates [53].
- Partitions provide a logical division of data that can be processed simultaneously. The division enables parallelization and balance [27].
- Horizontal partitions can be created one at a time, such that new insertions will not affect already existing partitions. New data can be accumulated in temporary structures, and when there are enough rows in these structures, read-only column partitions can be created and inserted into the database.
- The operating system might not be able to provide memory chunks large enough to contain an entire column. Partitioning the data horizontally helps overcome this limitation.

Instead of operating on entire columns at a time, *Microsoft SQL Server* divides the data into *row groups* that are groups of rows compressed into a columnar format [41]. Within the columns, data is not sorted. Each row group is encoded and compressed independently, and as we see in Figure 3.3, each partition has its own dictionary. We were unable to find out how many rows are contained by each row group, but the Larson *et al.* say that the number of rows in a row group must be small enough to benefit from in-memory operations and large enough to achieve high compression rates.

For *Oracle Database*, the column store is made up of multiple extents, called In-Memory Compression Units (IMCUs) [39]. Much like *Microsoft SQL Server*, data is loaded into the IMCUs without a sort order; they are stored the same way as they appear in the row format. Each partition consists of approximately half a million

rows, and each IMCU contains maximum and minimum values per column such that data can be pruned easily.

3.3 Compression

Historically, compression has been thought of a measure for reducing disk usage and memory footprint. However, in our case, compression of data also comes with the benefit of increased performance. A study conducted by Abadi *et al.* looked into database compression for in-memory databases, and concluded compression increases performance by a factor of two on average [16].

Among systems we have studied in this literature review, we have identified several that applies compression for performance reasons. Among these systems are *IBM DB2 with BLU Acceleration* [54], *C-Store* [58], *Vertica* [40], *Oracle Database* [45], *Gorilla Time Series Database* [46], and *EXASOL EXASolution* [27]. In addition to this, *Business Discovery* products *Tableau* and *QlikView* also use compression extensively to achieve good performance [37, 51].

Compression of data in a database is beneficial for performance due to:

- Cache locality is improved [27]. More values from a single column (or record) fit in the cache at the same time.
- Memory traffic is reduced. Compression can help turning a database from memory-bound to CPU bound [74].
- Compression reduces CPU cycles [58]. First of all, as stated above, it reduces memory latency and improves cache locality, such that more cycles can be used for calculations and not waiting for memory. Besides, compression enables working on multiple values in parallel using SIMD instructions, as we see in Section 3.10.4.

Still, even though compression is used to increase database performance, the fact that compression reduces memory usage is also important. Even though DRAM is cheap, it is rarely over-provisioned and unused [20]. Also, compressed data frees up space for other structures, like indexes and result caches. For instance, *Oracle Database* justifies their dual format by using the space freed up by compressing the columns [39, 40]. Compression can help an application avoid relying on slow, virtual memory.

3.3.1 Compression Types and Light-Weight Compression

A study performed by Westmann *et al.* investigates database compression, and concludes that the compression must be *light-weight* for maximum performance [61]. In other words, the real benefit of compression can only be leveraged if the decompression effort can be minimized [42]. Light-weight compression has been

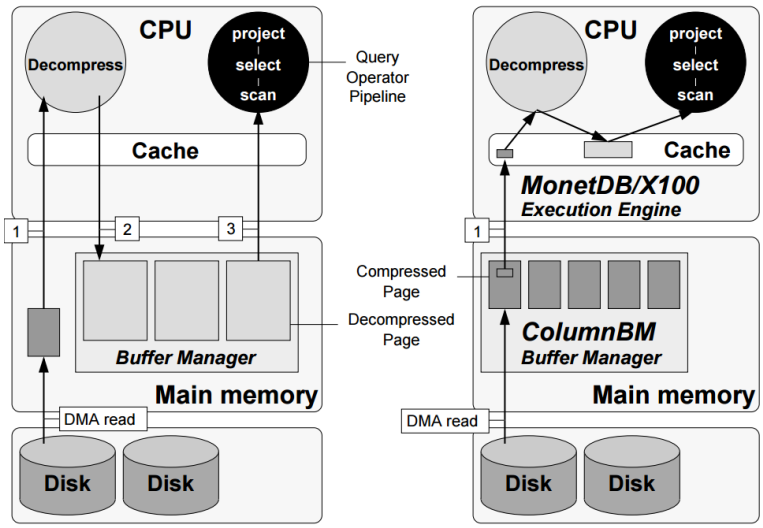


Figure 3.4: I/O-RAM vs RAM-CPU compression. In the left sub-figure, data is decompressed before it is put in the buffer manager (RAM). In the right sub-figure, data is kept compressed in the buffer manager and only decompressed when it is brought into the CPU cache. (Adapted from [76])

defined by Holloway *et al.* as bitpacking, dictionary encoding, delta encoding, and run-length encoding [35]. Holloway *et al.* also conclude that dictionary encoding and run-length encoding are the best compression schemes for column stores. These compression techniques are fast and fine-grained, which is essential for performance.

Zukowski *et al.* conclude that a compression algorithm should care about super-scalar processors. This implies that the algorithm should be able to pipeline loops, support out-of-order execution, and avoid if-then-else in inner loops [76]. We study modern CPUs and compilers in Section 3.9.

A database can use multiple compression schemes. First, most of the light-weight compression techniques can be combined, where the most common combination is dictionary encoding and bitpacking. We study this idea in Section 3.5. Second, different compression schemes can be used for different columns. In Section 3.2.1, we saw that *C-Store* and *Vertica* allow for multiple column projections (a subset of the columns), where each projection is sorted based on one of the columns in the subset.

3.3.2 Working Directly on Compressed Data

Earlier database systems with compression, data was decompressed when brought up to RAM. In 2006, Zukowski *et al.* suggested that data should not be decompressed

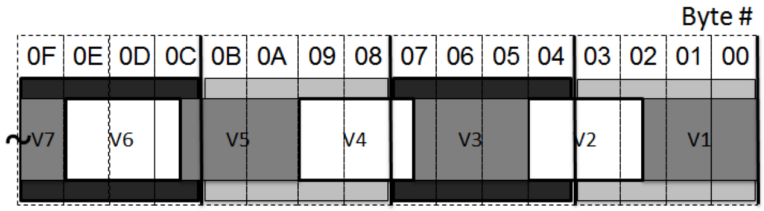


Figure 3.5: Bitpacked column values. Values are stored with no more bits than needed to represent the column, which results in values that are not aligned to machine word boundaries. Values may be spread across several machine words and share their machine word(s) with other codewords. (Adapted from [73])

when moved from disk to RAM, but when brought from RAM to cache [76], as depicted in Figure 3.4. *MonetDB/X100* is a system that decompresses when data is moved from RAM to cache [36].

However, the most performance benefit of compression is seen when the system works on the compressed data directly [42]. This implies that data should not be decompressed until it is materialized and sent to the user. This principle is backed by the creators of *Blink*, who says data should never be decompressed before absolutely needed [21]. *Oracle Database* claims one of the main performance benefits is to work directly on the compressed data [45]. Not decompressing data before it is needed relates to a technique known as *late materialization*, a technique we study in Section 3.10.3.

3.4 Bitpacking

Bitpacking is a form of compression where values are stored with no more bits than needed. In other words, if a column has no more than 32 distinct values, only 5 bits are required to represent a value. This way, in a 64-bit architecture, 100 values can be stored using $100 * 5 = 500$ bits, and not $100 * 64 = 6400$. Bitpacking has lower compression rates than algorithms that allow variable length for each value, but the main benefit is that values can be randomly accessed in constant time [53, 73]. Also, bitpacking enables SIMD processing, which we discuss in Section 3.10.4. Bitpacking is well suited for high cardinality, uniform distribution of values [35].

As seen in Figure 3.5, bitpacked values will not always align to word boundaries. For processing, values normally have to be moved to the word boundary, but this cost has been found to be negligible [35]. Aligning values can be done in an SIMD like fashion, a technique we study in Section 3.10.4.

In its simplest form, bitpacking works directly on the column data, but the compression scheme can be more powerful if combined with other compression types. We see in Section 3.5 that dictionary keys can be bitpacked, and in Section 3.6 that

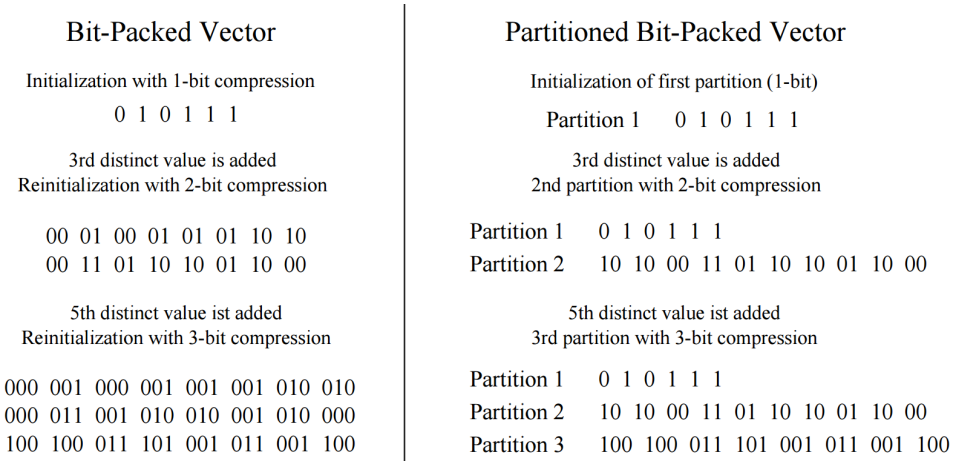


Figure 3.6: A normal bitpacked vector (left) and a partitioned bitpacked vector (right). Instead of rebuilding the entire vector on value overflow, the partitioned bitpacked vector has different partitions where each partition is compressed using an increasing number of bits. (Adapted from [29])

delta encoding benefits from bitpacking if the deltas between the values are small.

3.4.1 Issues with bitpacking

There are two major limitations with bitpacking [29]. The first is that if the bitpacking overflows, the full bitpacked vector must be rebuilt. What this means is that if all values are mapped, such that there are no available values with n bits, a new bit must be introduced, and the entire vector must be rebuilt where each value has $n + 1$ bits. This is depicted in the left part of Figure 3.6. To counter this effect, Faust *et al.* have suggested a partitioned bitpacked vector structure that creates a new partition with $n + 1$ bits on overflow, as seen in the right part of Figure 3.6. Although this technique improves performance on insert operations, read performance suffers due to the extra overhead of looking up a value.

The second limitation with bitpacking is that it does not account very well for data skew. In bitpacking, each distinct value in a vector contributes to the total number of bits required, completely disregarding the distribution of the values. Often in a database, there is a large number of distinct values, but they are not uniformly distributed. This problem can be solved with the partitioned vectors explained in the previous paragraph, by mapping the values that occur more frequently to the partitions with the fewest bit per value. Other algorithms map outliers to a separate structure, like *PForDelta* [22].

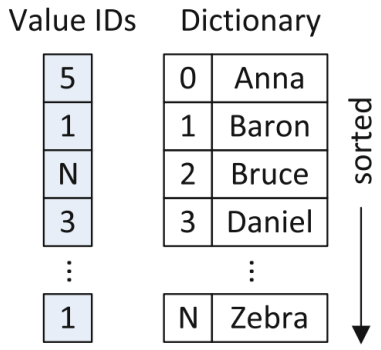


Figure 3.7: A sorted dictionary. Each distinct value is stored only once in the dictionary. A key is assigned to each entry in the dictionary, and those keys are stored in the columns instead of the actual values. (Adapted from [48])

3.5 Dictionary Encoding

Dictionary Encoding, or *Dictionary Compression*, is widely used within column store databases. Systems that use dictionary encoding include *Oracle Database* [39], *IBM DB2 with BLU Acceleration* [54], *SAP HANA* [28], *SAP NetWeaver* [42], *Blink* [36], *Microsoft SQL Server* [41], and more. *QlikView* stores each distinct data point only once [50]. *Tableau* does not mention anything about dictionary encoding in their whitepapers, but an official blog post claims that this compression technique is used [15].

In a dictionary encoded column, each distinct value is stored once in a structure known as a *dictionary*. Keys are assigned to each entry in the dictionary, most commonly integers from zero and up. Columns store these keys and not the actual values, and data is compressed since each unique value is stored exactly once. Dictionary encoding using a sorted dictionary is illustrated in Figure 3.7. Dictionary encoding is particularly effective when a column in a column store has only a few distinct values in a large dataset [29].

Except from the compression, one of the major advantages of dictionary encoding, is that many database operations can be performed directly on the encoded values [29], which we have seen in Section 3.3.2 is important to achieve good performance. Integer comparisons are less expensive than comparing the actual values, especially for strings. Additionally, range and `LIKE` predicates can be turned into `IN`-list operations, since the dictionary can be scanned first to find the relevant integer keys [21].

If the columns are partitioned horizontally, which we have discussed in Section 3.2.4, it is common that each partition has a separate dictionary. This is the case for most database systems, like *Oracle Database* [39], *Blink* [21], and *Microsoft SQL Server* [41]. When a dictionary is stored per partition, it can be used for quick data

pruning; if a value is not present in the dictionary, the partition can be skipped. *Blink* and *MonetDB/X100* use this technique [21, 23].

When implementing dictionary encoding, special considerations should be taken. First, if the dictionary turns out bigger than the values it is replacing, dictionary encoding should not be used [35]. Secondly, dictionary encoding performs best if the dictionary fits inside the L2 cache of a processor.

Like bitpacking, dictionary encoding does not handle data skew very well, since each unique value needs an entry in the dictionary no matter how frequent that value appears in a column. Besides, for high cardinality columns, the compression is less efficient.

3.5.1 Sorted Dictionaries

Dictionaries can be either sorted or unsorted. Using a sorted dictionary enables easier value lookup using a binary search. However, more important for analytical performance, is that using a sorted dictionary can turn range scans into simple integer comparisons [29]. For instance, if we want to find all sales in 2010, we only need to look up the integer codes for January 1st, 2010 and January 1st, 2011 and find all integers within this range. Since integer comparisons are fast and effective, this technique will usually improve performance.

Most database systems today use sorted dictionaries. *SAP HANA* is an example of such system [28].

However, as briefly mentioned in Section 3.2.1, keeping a dictionary sorted implies a higher overhead on database inserts, updates, and deletes. To mitigate this problem, some systems divide their data into two stores: A read-optimized store, and a delta store (for updates and inserts). With this division, it is common to use a sorted dictionary for the read-optimized store and an unsorted dictionary for the delta store [47].

3.5.2 Dictionary Encoding and Bitpacking

Dictionary encoding is commonly used in conjunction with bitpacking. With this combination, the dictionary keys in the columns are stored with no more bits than necessary. Since dictionary keys normally are integers from zero to the number of entries, bitpacking enables high compression rates, especially for low-cardinality columns.

Systems using dictionary encoding and bitpacking include *IBM DB2 with BLU Acceleration* [54], *Blink* [21], *SAP NetWeaver* [74], and *SAP HANA* [48]. *QlikView* has also reported to compress data with only the number of bits required [51]. *Microsoft SQL Server* does not apply bitpacking on dictionary keys in columns, and instead store them as 32-bit integers [41].

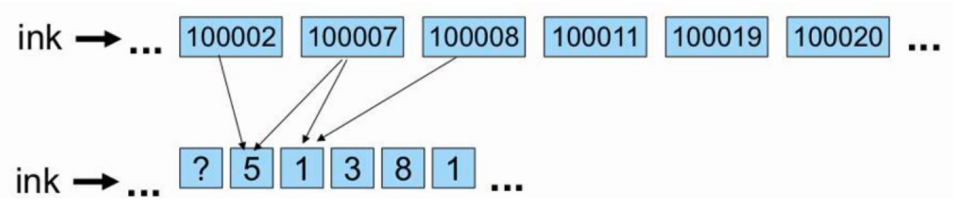


Figure 3.8: *Delta Encoding*. The difference between the current and the previous value is stored instead of the actual value. Since the difference between values usually is smaller than the actual values, fewer bits can be used to store the data. (Adapted from [60])

The same advantages and disadvantages of bitpacking apply to dictionary encoding with bitpacked columns. For instance, values can be looked up in constant time and queries can be processed in an SIMD-like fashion. However, insertions to the dictionary might lead to an overflow, which requires the entire column to be rebuilt.

3.6 Delta Encoding

Delta Encoding, or *Delta Compression*, is a compression method where the difference between the previous and the current value in a column is stored instead of the actual value [65]. This is illustrated in Figure 3.8. Systems using this form for encoding include *Vertica* [40], *C-Store* [58], and *Blink* [53].

Delta encoding is particularly effective when there are small differences between consecutive values in a column. In *C-Store* and *Vertica*, *Delta Encoding* is the compression of choice for sorted columns with high cardinality, since sorted columns minimize the deltas. Delta encoding can also be effective on sorted, low-cardinality columns, but is in general outperformed by run-length encoding in this case.

One of the major drawbacks of delta encoding is that values cannot be accessed in constant time. To find a particular value at index i, all the values from 0 to $i - 1$ must be decoded and accumulated. Also, delta compressed columns are hard to work on directly without decompression.

Bitpacking can be used in conjunction with delta encoding. If the deltas are small, which they typically are if applied to a sorted column, bitpacking can compress a column significantly. The column can be compressed even further by using dictionary encoding, delta encoding, and bitpacking at the same time because the variance in the column values will be reduced when replaced with dictionary keys. However, applying delta encoding to bitpacked, dictionary encoded columns comes at a cost; Queries can no longer work directly on the compressed data using simple integer operations. In other words, we improve compression, but at the expense of performance.

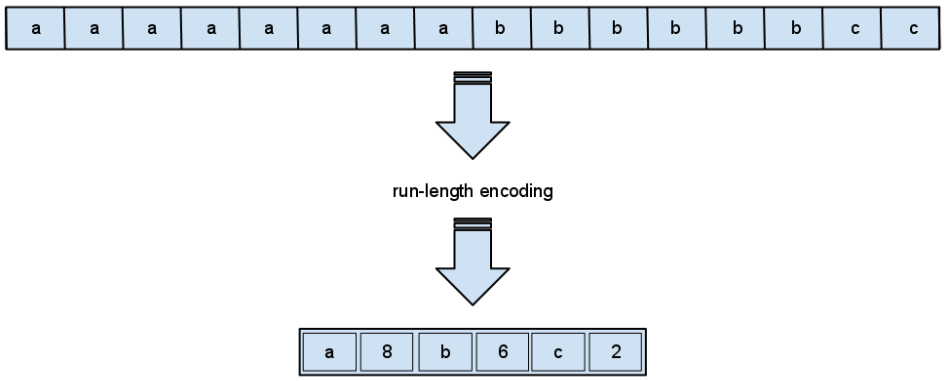


Figure 3.9: *Run-Length Encoding* on a sorted value range. Repeated values are replaced with one instance of the value and an integer indicating how many times that value occurs in the sequence. (Adapted from [57])

3.7 Run-Length Encoding

Run-Length Encoding is a lossless data compression algorithm that replaces repeating data values with only one instance of the data and the number of how many times that value appears in the sequence [57], like illustrated in Figure 3.9. Although it can be used for any sequence, run-length encoding works best on sorted data [22, 35]. In this case, each unique value in the sequence is represented exactly once. Run-length encoding performs best for low-cardinality columns. The method can also be applied to compress sparse bitmaps [58].

Run-length encoding is used in *C-Store* [58], *Vertica* [40], *Oracle Database* [45], and *SAP NetWeaver* [42], but only if the values are sorted.

Run-length encoding enables the query operators to work directly on the compressed data. For instance, queries can exploit run-length encoding on sorted columns when performing grouping and aggregation, since the values are already stored as groups.

3.8 Bitmaps and Bitmap Indexes

C-Store and *Vertica* compress certain low cardinality columns using bitmaps. With this technique, each unique value in a column is represented as a bitmap where the set bits, or 1s, indicate which rows that have that value. If there are few distinct values in a column, the size of the bitmaps will be smaller than storing the actual values, and compression is achieved. Since these bitmaps are typically sparse, they can be compressed further with techniques like run-length encoding.

A *Bitmap Index* is a particular index structure where a bitmap represents each

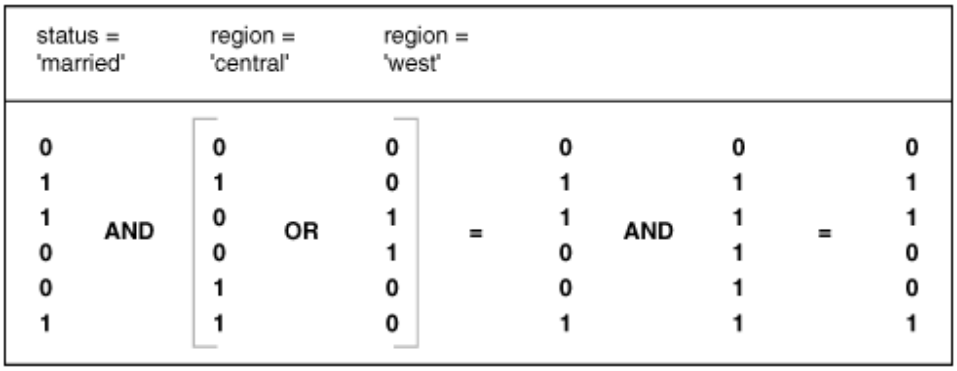


Figure 3.10: Executing a query using bitmap indexes. Queries are efficiently processed using bitwise operations like `AND` and `OR`. Adapted from [6])

distinct value in a column, and all rows containing that value is set to 1. It can be used to aid predicate evaluation. A bitmap index is most efficient on queries that contain multiple `WHERE` clauses since many candidate rows can be excluded using bitwise `AND` and `OR` operations, as seen in Figure 3.10 [6]. Since bitmap indexes combine so easily, composite indexes are extraneous.

Business Discovery product *QlikView* reports that it uses binary indexes for each field [50], which we believe are the same as bitmap indexes. Some database management systems also support bitmap indexes, including *Oracle Database* [6] and *IBM DB2 with BLU Acceleration* [54].

In general, low-cardinality columns, which is columns with few distinct values, are better suited for bitmap indexes than high-cardinality columns [6]. The reason for this is because a bitmap must be created and maintained for each unique value in the column. Besides, since these indexes are hard to maintain, they are not suited for inserts, updates, and deletes. When deciding whether to add a bitmap index to a column, *Oracle Database* recommends at least 100 rows per distinct value.

3.9 Modern CPUs and Compilers

Modern processors are capable of performing an enormous amount of calculations per second, but that depends on the amount of available and independent work. The instructions-per-second (IPC) difference between minimal and maximal CPU utilization can easily be one order of magnitude [23]. Hence, database software must be implemented such that it fully exploits the processing power made available by the CPU.

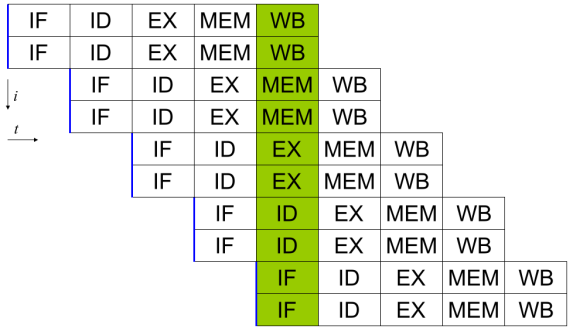


Figure 3.11: A simple superscalar pipeline. Multiple execution units allow for processing multiple instructions in parallel. (Adapted from [71])

3.9.1 Pipelining, Superscalar Processing, and Independent Instructions

Modern processors improve clock rate and IPC by using a technique known as *pipelining* [23]. By dividing an instruction execution into multiple steps, there is less work per stage, and the CPU frequency can be increased. Figure 3.11 depicts an example pipeline with five stages. However, pipelining also introduce two dangers; *instruction dependencies* and *branch misprediction*.

In a pipeline, *dependencies between instructions* impose a problem. If an instruction is dependent on another, it must wait for the other instruction to complete before it enters the pipeline. Dependent instructions can severely hurt performance, especially if the pipeline is long.

Conditional branches are also affected by dependencies between instructions. When executing a conditional branch instruction, the decision whether to take a branch is usually dependent on the result of a preceding instruction [23]. To avoid stalling the pipeline when waiting for the expression to evaluate, modern CPUs use a technique known as branch prediction where the processor immaturely starts executing the branch that is most likely to be taken. The performance penalty occurs if a branch is *mispredicted*, where the instructions in the pipeline must be invalidated (pipeline flushing).

Another way that performance is increased in a processor is by having multiple execution units. We refer to such processors as *superscalar*. As seen in Figure 3.11, a superscalar processor can have multiple instructions in the same stage of the pipeline, which allows IPC (instructions per cycle) > 1. However, for this functionality to be fully utilized, independent work is required.

Hence, to reach maximum performance for a pipelined, superscalar processor, we must find independent work. Since most programming languages do not let the programmer specify which instructions are independent of each other, compiler

```
for (i = 0; i < 100; i += 1) {
        a[i] = b[i] + c[i];
}
```

(a) Original loop

```
for (i = 0; i < 100; i += 5) {
        a[i] = b[i] + c[i];
        a[i + 1] = b[i + 1] + c[i + 1];
        a[i + 2] = b[i + 2] + c[i + 2];
        a[i + 3] = b[i + 3] + c[i + 3];
        a[i + 4] = b[i + 4] + c[i + 4];
}
```

(b) Unrolled loop

Figure 3.12: By unrolling loops, instructions are made independent, and the number of branches is reduced.

optimizations play a critical role in CPU utilization [23]. The most widely used technique used by the compilers to address this challenge is *loop unrolling*, which is used to reduce the number of branches and increase independence between instructions [66]. As seen in Figure 3.12, loop unrolling reduces the number of iterations in a loop (reduction of branches) and replaces it with multiple instances of the same instruction. If the instructions are independent, they can be processed in parallel.

3.9.2 CPU Caches

Since transferring data from main memory to CPU can take around 200 cycles, modern CPUs utilize multiple layers of on- and off-chip caches to reduce this latency. Efficient usage of caches is paramount for CPU throughput since roughly 30% of all instructions in a program are memory loads or stores [23]. We know that IPC for DBMSes is strongly impaired by cache misses, and cache utilization is an important topic for in-memory databases [27].

The best way to tackle this challenge is to design algorithms and data structures that are *cache aware* [28]. Designing such programs is out of the scope of this report, but it briefly boils down to two things:

- *Coordinate temporal and spatial locality.* Data processed together should be stored at consecutive memory addresses. Code locality is also important [44].
- *Avoid false sharing of cache lines.* Multiple cores in a processor should not write to data belonging to the same cache entry at the same time to avoid unnecessary invalidations.

Compression, which we described in Section 3.3, and *vectorized execution*, which we discuss in Section 3.10.2, are two techniques used to improve cache performance [41, 42].

Prefetching is another method used to increase cache utilization. Prefetching proactively loads data into caches such that the data is available when an instruction needs it.

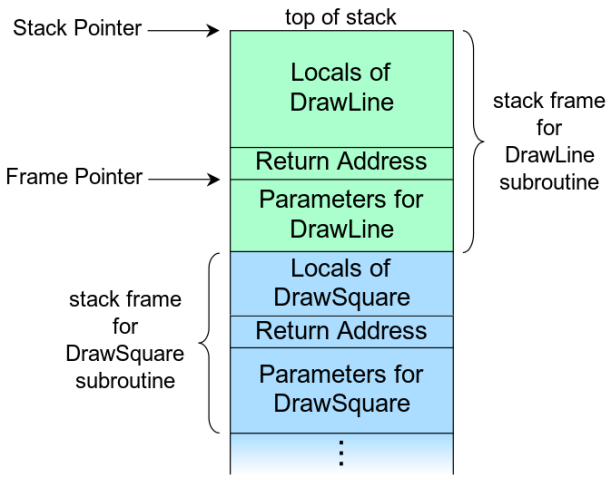


Figure 3.13: A call stack for a program in execution. Each stack frame contains input parameters, function return address, and local variables for a subroutine invocation. (Adapted from [63])

Our research has identified several systems that are designed to be cache-aware, including *MonetDB* [24], *Microsoft SQL Server* [39], and *IBM DB2 with BLU Acceleration* [54]. The developers of *EXASOL EXASolution*, the top performing database system in the TPC-H benchmark, claim that high level of data locality is one of the key factors to performance.

3.9.3 Call Stack and Subroutine Invocations

A *call stack* is commonly used in a computer program to store information and state about active subroutines [63]. Each time a subroutine is called, a *stack frame* is added to the call stack that stores the input arguments, return address, and variables local to the subroutine. See Figure 3.13. The call stack can be implemented in both hardware and software, and the implementation varies between different systems. This stack-based technique implies that calling a subroutine comes at a cost; registers must be stored on the stack, and a new stack frame must be added.

Trading off time with space is usually done to address the above challenge; adding more code to improve program efficiency. *Function inlining* is a technique used by compilers where the subroutine code is copied into the caller's body. This way, no new stack frame is created for the subroutine, avoiding the overhead associated with a function invocation.

Macro expansion is another form of code generation. Macros are usually specified by the application programmer, and can be used for programmer-controlled inlining of

functions or constant values. Macros can also be used to generate multiple versions of function or class definitions (templating), a technique commonly used to let a single implementation support different data types.

3.10 Hardware Utilization

In this section, we enumerate several topics, techniques, and concepts that are used to utilize the available hardware and maximize CPU throughput.

3.10.1 Loop Pipelining

The absence of loop pipelining can have dramatic effects on query performance [23]. Boncz *et al.* show that *MySQL database* uses 49 cycles per tuple because loops are not unrolled. In this system, tuples are processed one at a time with 1-2 function calls to extract the needed data from a tuple per iteration [16]. Besides, evaluating a predicate is usually a small operation compared to the overhead associated with calling subroutines. In other words, most of the 49 instructions are spent managing the call stack or waiting for instruction dependencies.

To ensure proper loop pipeline behavior, compilers must be aware that pointers do not overlap, such that loop unrolling can be used [23]. Operations in *MonetDB/X100* are compiled with compiler hints that tell the compiler that processing a tuple is independent of the others. In standard *C* compilers, this can be done by using the `__restrict__` pointer type.

3.10.2 Vectorized Execution

To avoid unnecessary subroutine invocation overhead and help the compiler identify which instructions are independent, *vectorized execution* is normally used. Vectorized execution, or block iteration, is the technique where multiple rows are processed at the same time to avoid the overhead associated with tuple-at-a-time processing [16]. Vectorized execution enables loop unrolling and memory prefetching which minimizes cache misses [41]. Research performed by Abadi *et al.* shows that vectorized execution in columns stores improves performance by 50% on average.

Several systems studied our research use vectorized execution. *IBM DB2 with BLU Acceleration* and *Microsoft SQL Server* work on batches of thousands of row at a time [41, 54]. *MonetDB* and *MonetDB/X100* use vectors instead of single values as their primary structure for storing data [23, 24]. Vectorized execution is also used by *C-Store* and *Blink* [36, 58].

In vectorized execution, blocks of values from the same column are sent to an operator for evaluation [76]. Query operators in *Microsoft SQL Server* work on row batches, batches that contain thousands of rows stored in a column format.

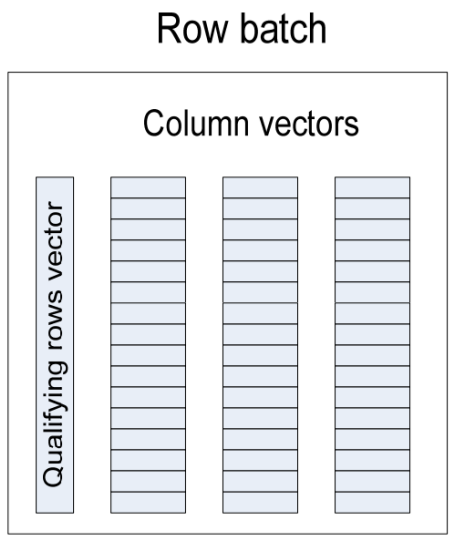


Figure 3.14: *Microsoft SQL Server* operators work on row batches. Each row batch contains thousands of rows stored as column vectors. Also, a bit vector indicates which rows that qualifies for a query. (Adapted from [41])

In addition to the column vectors, an additional *qualifying rows vector* is used to indicate which rows that have been logically purged from the batch when executing a query. Figure 3.14 shows this structure.

Regarding the size of the vectors in vectorized execution, we have found that vectors should not be too small due to increased overhead and less parallelism, nor too big, as it should fit in CPU cache [23].

Although vectorized execution for column stores normally outperforms tuple-at-a-time query processing, there are some disadvantages by using this model. Neumann *et al.* claim that vectorized execution eliminates a major strength of the iterator model, namely *pipelining* [44]. In this context, pipelining means the ability for an operator to pass tuples to its parent operator without copying the value. When vectorized execution is used, intermediate results have to be stored somewhere (materialized), which consumes memory bandwidth.

3.10.3 Late Materialization

Late materialization is the principle of not stitching together tuples before necessary [16]. Systems devoted to *late materialization* work on columns for as long as possible. According to a research by Abadi *et al. late materialization* can increase performance in columns stores by 5%-50% depending on the query [16].

Late materialization is advantegous because [16]:

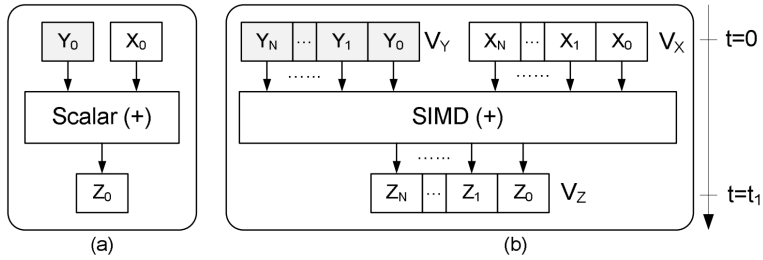


Figure 3.15: SIMD execution model: In scalar mode (a): one operation produces one result. In SIMD mode (b): one operation produces multiple results. (Adapted from [74])

- Compressed columns must be decompressed before materialization, which eliminates the benefits of working directly on compressed data.
- Cache performance is better for columns than for rows, which means processing columns are more efficient.
- Vectorized execution can be used on columns only.
- Early materialization might construct tuples that are discarded later in the query execution.

The *late materialization* principle is used by several database systems, including *IBM DB2 with BLU Acceleration* [54] and *MonetDB* [24].

3.10.4 SIMD

Within a single execution context, instructions that work on multiple elements at a time can be used to increase query performance. We refer to these instructions as single input, multiple data (SIMD) instructions [70]. As seen in Figure 3.15, an SIMD instruction applies the same operation to multiple operands simultaneously. SIMD processing in a database context is particularly effective if we can keep an entire processing block in the CPU registers [44], and Willhalm *et al.* show that SIMD processing using a vectorized model can be up to 1.5 times faster than databases optimized for scalar execution and instruction-level parallelism [74].

Our research shows that several database systems use SIMD parallelization. Systems in this category include *Oracle Database* [39], *Blink* [21], and *IBM DB2 with BLU Acceleration* [54]. A whitepaper from the developers of *EXASOL EXASolution* says that SIMD features of modern processors must be used to reach the highest level of performance [27].

Oracle Database performs scans against the columns using instructions that work on multiple operands simultaneously [39]. As seen in Figure 3.16, a filter operation

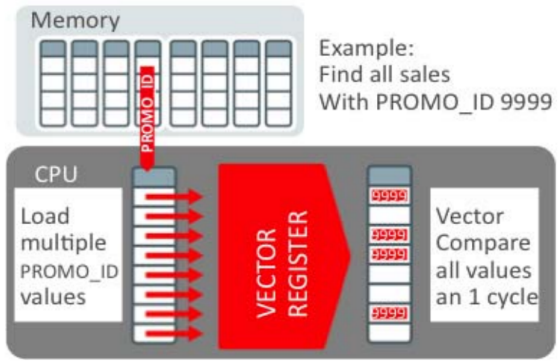


Figure 3.16: Filter operation in *Oracle Database* using SIMD vector processing. (Adapted from [39])

benefits from SIMD as multiple values are compared in parallel. *IBM DB2 with BLU Acceleration* and later versions of *Blink* work similarly [21, 54].

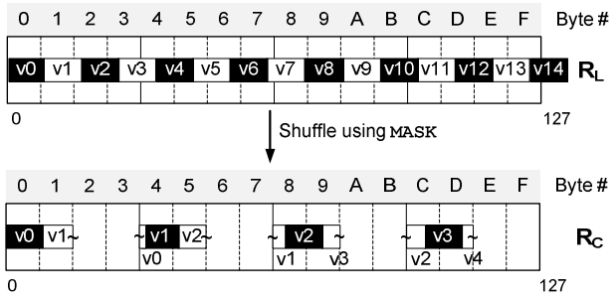
Bitpacking of column values further increase SIMD performance, because more values fit within a single data word. However, since most SIMD instructions require operands to be aligned, some preprocessing steps must be applied to the column values [74]. As seen in Figure 3.17, a bitpacked vector can be prepared for an SIMD operation using mask and bitshift operations. Vectors that are already aligned with data words can be queried more efficiently since the preprocessing can be skipped.

Most literature refers to SIMD parallelization as utilizing special processor and instruction set extensions, like the Intel SSE and Intel AVX2 extension [73, 74]. However, general CPU instructions can also be used to evaluate predicates in a *SIMD-like fashion*. In *Blink*, ordinary CPU instructions work directly on multiple values in a bitpacked column by applying a suitable mask and compare the result with a second mask containing the expected values [36]. This technique is very efficient since masks for a column are calculated once per query.

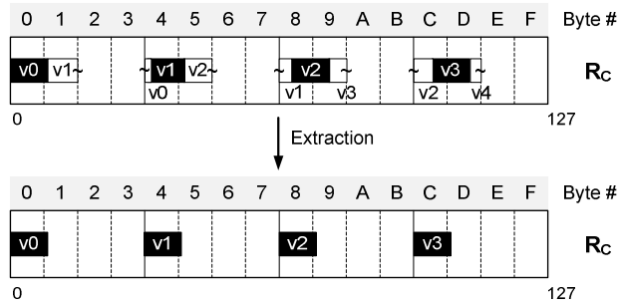
3.10.5 Branch Avoidance

We saw in Section 3.9 that branches should be avoided due to the penalties of branch misprediction. Besides, branches also cause dependencies between instructions.

The consequences of inaccurate branch prediction are studied by Neumann *et al.* [44]. In this research, the performance of queries with various selectivities was tested. As we can see in Figure 3.18, queries with 40%-60% selectivity executed on an AthlonMP processor are roughly 2-3 times slower than queries with selectivities close to 0% or 100%. Hence, selectivity can severely affect the query performance. The Itanium2 processor does not have the same characteristic, as the Itanium architecture allows for both *not taken* and *taken* branches to be executed simultaneously.



(a) Using a `MASK` operation to align values to data words.



(b) Extracting values by bitshifting followed by masking out irrelevant bits.

Figure 3.17: Aligning a bitpacked vector for SIMD execution. (Adapted from [74])

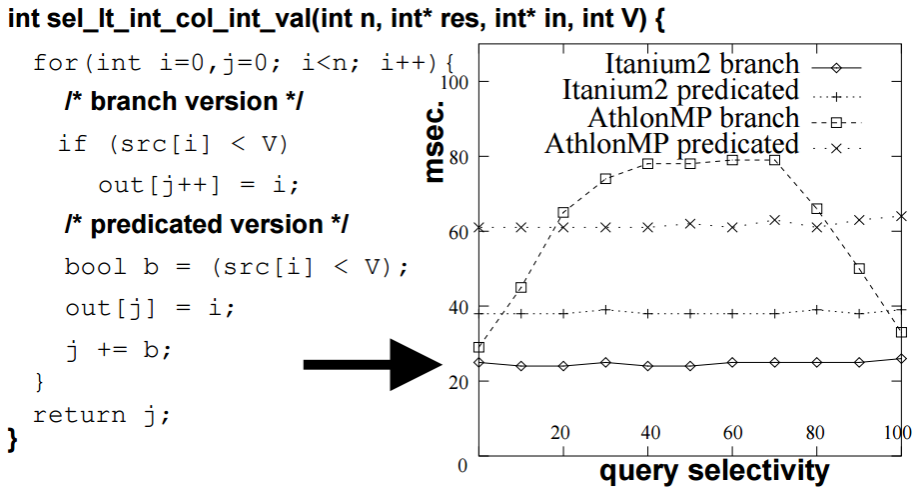


Figure 3.18: Predicate evaluation performance for queries with different query selectivities. A *branch version* and a *predicated version* are tested. For the AthlonMP processor, the branch version are 2-3 times slower on queries with 40%-60% selectivity, while the Itanium2 processor has constant processing time. The predicated version offers constant processing time for both processors. (Adapted from [23])

Neumann *et al.* also developed a branchless version (predicated version) to evaluate predicates in the queries. The branchless variant is denoted as `predicated version` in Figure 3.18. For both AthlonMP and Itanium2 processors, this implementation offers constant performance for all selectivities, but is, in general, more expensive.

Branch avoidance is also important in other parts of the system, for instance when decompressing. Zukowski *et al.* present a decompression algorithm that is free for *if-then-else* statements [76]. By running the algorithm in two tight loops instead of one, branch misprediction is reduced, and the loops can be pipelined by a compiler.

3.10.6 Macro Expansions

MonetDB uses macro expansion to reduce layers of indirection and to optimize query execution performance [24]. Since operators normally are type-generic, *MonetDB* has for each algorithm multiple implementations that are specific to a certain type. The implementations are generated automatically using macros, which is why they are called *macro expansions*. Figure 3.19 shows how the `select` operator is expanded into 173 implementations, depending on which algorithm and data types are queried.

The vector data structure in *IBM DB2 with BLU Acceleration* is implemented using C++ templates to support multiple data types.

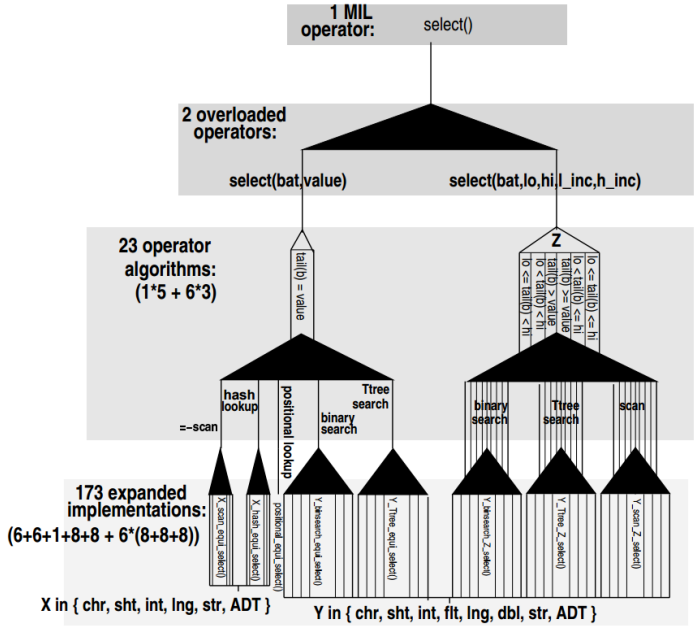
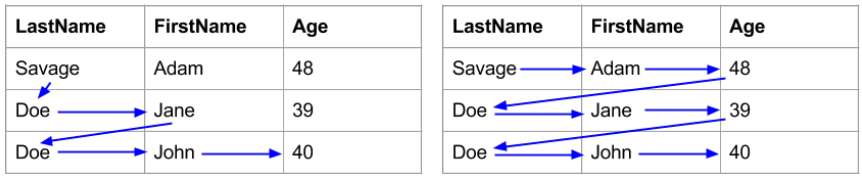


Figure 3.19: Macro expansions in *MonetDB*. For different algorithms and data types, the `select` operator has a total of 173 implementations. (Adapted from [24])



(a) Short-circuiting (b) No short-circuiting

Figure 3.20: Predicate evaluation for query `WHERE LastName='Doe' AND FirstName='John' AND Age>21`. (a) skips evaluating rest of the predicates if one predicate is false, while (b) evaluates all predicates regardless of previous results.

3.10.7 Short-Circuiting

Short-circuiting is referred to a special case of Boolean operator evaluation in which the next argument is not evaluated if the current argument is sufficient to determine the value of the expression [69]. Figure 3.20 illustrates the difference between short-circuit and non-short-circuit predicate evaluation. In short-circuiting, the scan proceeds to the next tuple as soon as one predicate is false, as opposed to the verision without short-circuiting that evaluates all predicates regardless of previous results.

Since short-circuit boolean operators are control structures and not simple arithmetic operators, there is a chance of branch misprediction. That is why *Blink* does not short-circuit between tuples [53, 36]. If a block is selected for scanning, all fields in the records are checked. According to Raman *et al.*, short-circuiting only improves performance on low selectivity queries [53].

3.11 Joining

Joining is a common database operation that combines records from two or more tables. In our research, we have studied several DBMSes, and seen how their join algorithms work. For joining two tables, three main methods exist [25]:

- *Partition-based approach*, where records of both tables are split into groups based on the hash value of the keys. This technique is most effective on data volumes that are too large to fit in main memory.
- *Sort-merge approach*, where both tables are sorted and then merge results by concatenating records with equal key values. This approach is most effective when one or both operands are sorted in advance.
- *Nested loop approach*, which compares all rows in both tables.

The *nested loop* approach is the most popular joining algorithm for in-memory databases [24]. We therefore only discuss this method.

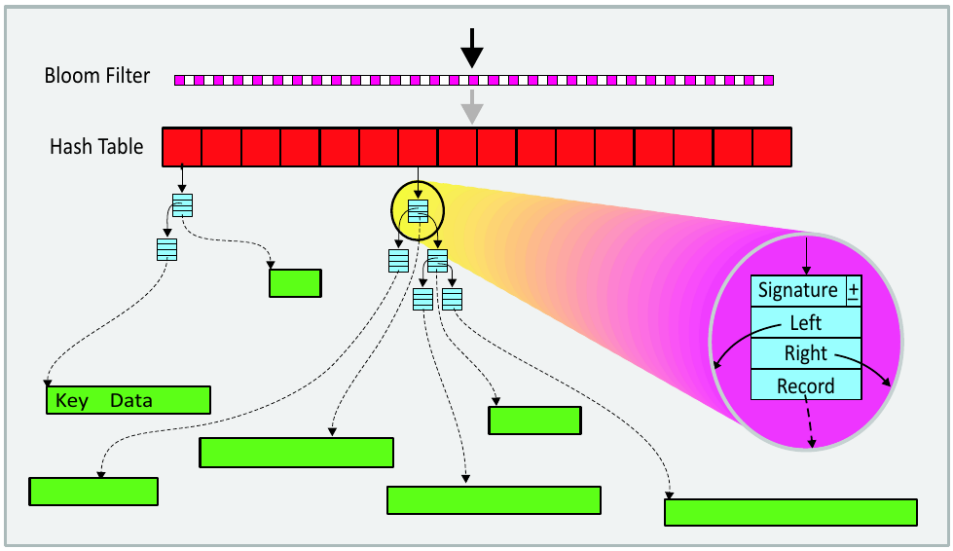


Figure 3.21: An example nested loop join structure. Records are first tested against a Bloom filter. If found in the filter, the join key is searched in the join structure. Records are first hashed, and then each entry in the hash table is the root of a binary tree. (Adapted from [25])

3.11.1 Nested Loop Algorithms

In its simplest form, the *nested loop* method compares the join key in all rows directly using a double loop. This simple algorithm has a runtime of $O(n * m)$ where n and m is the size of tables A and B respectively. However, to improve performance in a *nested loop* algorithm, hashmaps are commonly used. Usually, the join is performed by hashing the smaller (inner) relation first, then probe the hashmap by scanning the larger (outer) relation.

Kjell Bratbergsengen shows how a combination of hash tables, bloom filters, and binary trees can be used in a *nested join* algorithm. In the probe phase, keys are first checked towards a Bloom filter. Bloom filters never return false negatives and is an efficient way of reducing the numbers of keys entering the join. If the key is found in the Bloom filter lookup, it is hashed and checked up against a hybrid hashmap/binary-tree structure. In this structure, each entry in the hashmap is the root node of a binary tree which are used to look up values efficiently. The join algorithm is illustrated in Figure 3.21.

In the nested loop approach, one of the operands might be partitioned [25]. For example, a join might be partitioned by only hashing a subset of the inner relation at a time. The entire join algorithm will then include several probe passes over the outer relation, one for each subset. Historically, this technique has been applied to ensure the whole hash structure fits in RAM. We conclude that a similar concept

can be applied to CPU caches and that the join algorithm might benefit from partitioning the inner relation such that each subset fits in the CPU cache.

We see that several DBMSes in this research use a *nested loop* join variant with Bloom filters, including *Oracle Database* [39], *IBM DB2 with BLU Acceleration* [54] and *Blink* [53]. Barber *et al.* explains how Bloom filters are effective in eliminating non-matching join outers before they enter the join [20].

Our research has shown that one of the key design goals for efficient joining using the *nested loop* approach with hashmaps, is to keep the hash tables collision free [53, 54]. One way to ensure this is to use the dictionary keys in *Dictionary Encoding* as a perfect hashing function. If a table is joined on several keys, a minimal perfect hash can be calculated.

Regarding implementation, it is important that the algorithm and the hash table are *cache-aware*. One way to improve cache performance in a *nested loop* join, is to use linear probing instead of open-chain addressing for the hashmap [53]. Open-chain addressing should only be used for overflow buckets.

3.12 Database Statistics

Database Statistics are commonly used by a query optimizer to make better decisions about creating efficient execution plans. These statistics may the include number of records, selectivity, column cardinality, value distribution, and more. In our case, *Database Statistics* can be used to prune horizontal partions based on the column minimum and maximum values. This technique exploits clustering in the columns, especially when the columns are sorted, or partially sorted, like timestamps. *Oracle Database* [39], *IBM DB2 with BLU Acceleration* [54], *Vertica* [40], *MonetDB/X100* [23], *Microsoft SQL Server* [41], and *EXASOL EXASolution* [27] store partition metadata for quick data pruning.

Most database systems keep the statistics stored together with the table. However, other schemes exist. *IBM DB2 with BLU Acceleration* uses a synopsis table to keep track of all column pages (partitions), including minimum and maximum values. This way, irrelevant pages can easily be skipped [54].

It can sometimes be useful to know a column's value distribution. For instance, the frequency partitioning in *Blink* and *IBM DB2 with BLU Acceleration* uses the columns' value distributions when determining how to partition the data [53, 54]. The query optimizer in *Microsoft SQL Server* also uses the value distributions when creating execution plans [41]. Value distributions are usually determined by creating histograms, and to make these histograms, random sampling can be used. In *Microsoft SQL Server*, two techniques are used: One is truly random, where values are picked across the whole column, and one is a grouped version, where a random sample range is picked.

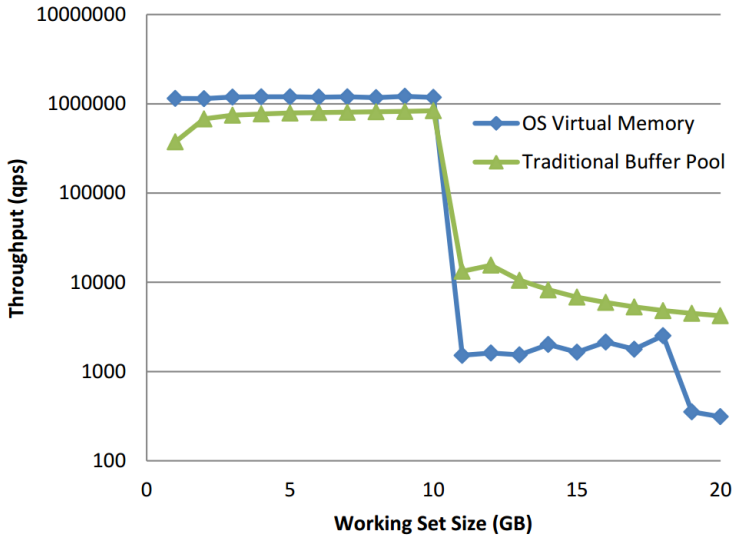


Figure 3.22: Database performance for systems with and without a buffer manager. This figure shows two things. First, when a database system starts spilling pages to disk, the throughput is drastically reduced. Second, databases without buffer managers perform better when data is in memory, but when the OS starts swapping pages to disk, they are much slower than systems with a buffer manager. (Adapted from [33])

3.13 Disk Support

To support datasets larger than provisioned RAM, the most trivial way is to use the OS' virtual memory. This is done by *MonetDB* [24], *Blink* [20] and *QlikView* [50]. However, the page replacement algorithms for virtual memory do not work very well on database workloads. The work of Graefe *et al.* shows that main-memory databases suffer a sudden performance drop when virtual memory mechanism starts swapping pages to disk [33], as seen in Figure 3.22. *QlikView* has reported a significant loss in performance once the OS starts paging.

Falling back to virtual memory are for some systems unacceptable for performance and correctness reasons [33]. To overcome these challenges, major database vendors like *Oracle Database* and *Microsoft SQL Server* use buffer managers for all their database operations, which we see in Figure 3.22 improves query performance when the working set is larger than the provisioned RAM. However, the same figure also reveals a drawback of using a buffer manager; the extra layer of indirection comes at a performance cost for working sets that fit in memory. The column store engine in *Microsoft SQL Server* performs worse than the in-memory *Microsoft VertiPaq* on workloads that fit in memory because the latter does not have a buffer manager [30]

3.14 Mixed Workloads

In this chapter, we have mainly studied read-optimized systems. Still, most of these systems allow for inserts, updates, and deletes, although these operations are not as efficient as for systems optimized for transactional workloads. However, some systems have a design goal to support both transactional and analytical workloads (mixed workloads) equally well. Examples of such systems are *Hyrise* and *HyPer*, and can be referred to as OLXP systems [47]. *Oracle Database* and *SAP HANA* also support mixed workloads, and they both strive to keep query transparency; applications using the database should not need to rewrite any queries to benefit from the underlying optimizations [28, 39].

Database consistency, correctness, and data freshness can be sacrificed for the benefit of better performance. For instance, updates can periodically be merged into the database by for example replacing one immutable structure with a more up-to-date version. *HyPer* sacrifices correctness and data freshness by using memory snapshots [38]. In this system, there are multiple read-only processes, but only one process for writes. The read-only processes periodically fork the main write process to obtain a memory snapshot of the current state of the database. Hardware and OS assisted replication mechanisms make sure the snapshots are created efficiently and consistent with the transactional data.

3.14.1 Delta Store

A delta store is commonly used in OLAP systems to accommodate inserts, updates, and deletes [54, 58]. Other names for delta store include *insert buffer* or *write-optimized store*. Such store is used because structures optimized for read performance are typically immutable or hard to update. For instance, inserting a new key into a sorted dictionary requires most key/value pairs to be reassigned. Besides, a newly inserted value might cause overflow in a bitpacked column.

Delta stores are periodically merged into the main storage, which is often called the read-optimized store. Such operation can either be triggered when the size of the delta store exceeds a certain threshold or through a periodical trickle operation [28, 39].

Many systems disallow updates and instead replace an update operation with one deletion and one insert. Deletes are usually implemented as invalidation bit vectors [40, 54], and updates are appended to a suitable structure, like an unsorted B+-tree [48], or an uncompressed column [28]. *Delete-insert* updates has the benefit of allowing time-travel queries [47, 55].

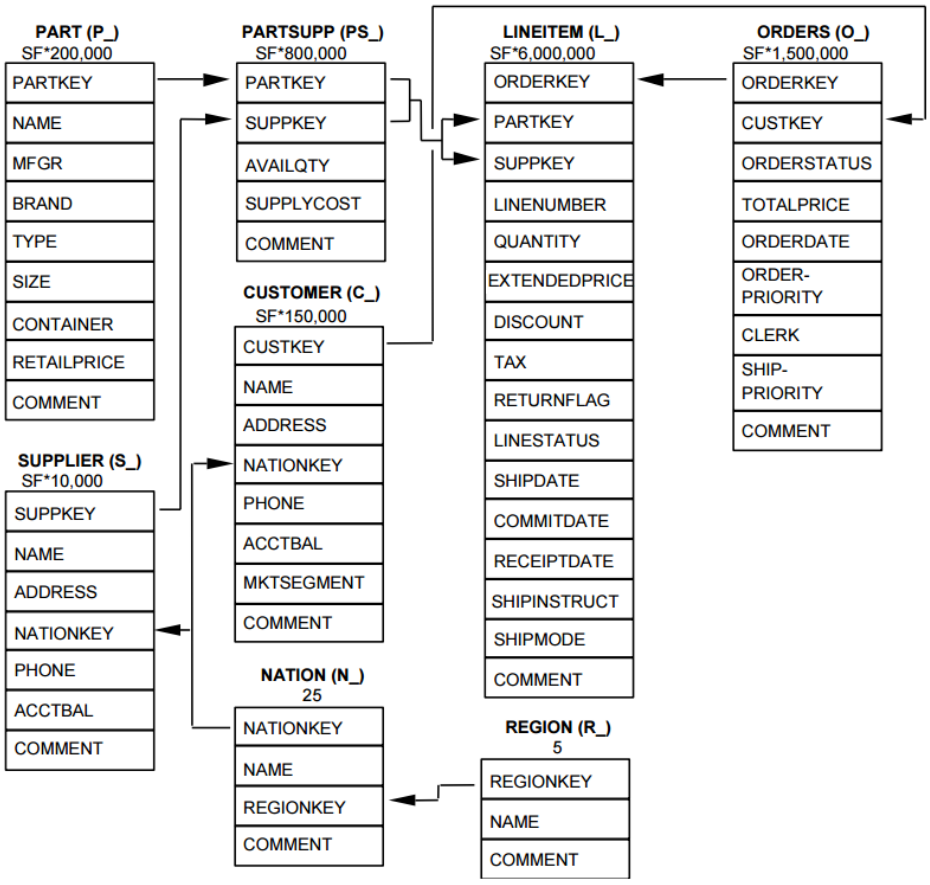


Figure 3.23: The TPC-H schema. (Adapted from [59])

3.15 Testing OLAP Databases

The TPC-H benchmark is commonly used to test analytical workloads. The benchmark is made for decision support workloads and consists of a suite of business oriented ad-hoc queries [59]. The benchmark has 22 complex read-only queries, which are both memory and CPU bound [23], and two update queries for data refresh. The TPC-H specification says the benchmark illustrates a decision support system that:

- Examine large volumes of data.
- Execute queries with a high degree of complexity.
- Give answers to critical business questions.

The schema for the TPC-H benchmark consists of eight separate tables, as seen in

Table Name	Cardinality (in rows)	Length (in bytes) of Typical[2] Row	Typical[2] Table Size (in MB)
SUPPLIER	10,000	159	2
PART	200,000	155	30
PARTSUPP	800,000	144	110
CUSTOMER	150,000	179	26
ORDERS	1,500,000	104	149
LINEITEM[3]	6,001,215	112	641
NATION[1]	25	128	< 1
REGION[1]	5	124	< 1
Total	8,661,245		956

Figure 3.24: TPC-H database size and cardinalities. (Adapted from [59])

Figure 3.23. The table columns have a variety of different data types, including integers, floating points, variable and fixed width strings, identifiers, and booleans.

Figure 3.24 shows the minimum population for the TPC-H benchmark, which is a database of 10,000 suppliers and roughly 6 million line-items (items per order). The minimum population corresponds to approximately 1 GB. To test larger data sizes, a scaling factor is commonly applied to increase the size of the dataset. According to the specification, data in the tables should be uniformly distributed.

The TPC-H benchmark tests uniformly distributed data, but, in reality, it is quite common that data is skewed. For instance, a retailer might expect that 99% of the sales are performed on weekdays, and around 40% of the total sales for a year is done around Christmas [53]. Tests on data with non-uniform distributions should be carried out to see how the algorithms and data structures in the database handle outliers. Data skew can be modelled with a *Zipfian* distribution [35].

Chapter 4

Genus App Platform

This chapter contains a top-down analysis of *Genus App Platform*, with platform architecture, how applications are developed using the platform, concepts, and source code. The last part of this chapter identifies challenges in Genus App Platform, which motivates the second and primary part of this thesis, which is the design and experiment research.

We obtained information for this chapter by reading technical briefs, attending platform courses, engaging discussions with *Genus AS*' employees, and analyzing source code.

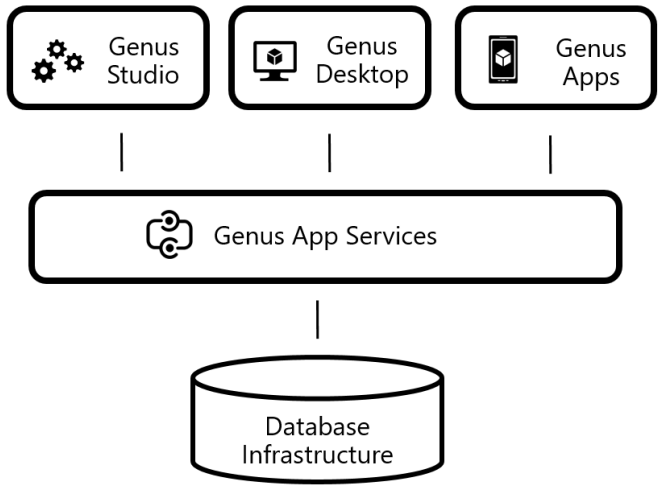


Figure 4.1: Components in *Genus App Platform*. *Genus App Services* connects end user and modeling clients with the underlying database infrastructure, and provides authentication and session management. (Adapted from [32])

4.1 Introduction

Genus App Platform is a platform for software development that aims to increase developer productivity and simplify change management through model-driven development [32]. *Genus App Platform* is a "no code" tool, which means end user applications are modeled without writing a single line of code. This approach enhances collaboration between business and IT, reduces life cycle costs, and lets the application developers focus on business logic instead of implementation details.

Genus App Platform uses generic software concepts on a higher level of abstraction than regular programming languages that are precise and non-ambiguous [5]. These concepts, and how they are implemented in the underlying platform, have been refined and improved through years of trial and error on real customer use cases.

If we relate *Genus App Platform* to the model layers defined in Section 2.1.1, we see that it spans over three layers: *Genus App Platform* platform developers work in the *method engineering (ME) layer*, while expert users using the modeling tool to create end user applications work on the *information system development (ISD) layer*. End users are in the *information system (IS) layer*.

4.2 Components and Architecture

Genus App Platform consists of five major components, as seen in Figure 4.1. These include two backend components, *database infrastructure* and *Genus App Services*, and one component for application modeling, *Genus Studio*. *Genus Desktop* and *Genus Apps* are end user clients for *Windows* and mobile devices respectively.

4.2.1 Database Infrastructure

The database infrastructure stores the application data, as well as the application itself (application model). *Genus App Platform* connects to a wide variety of data providers, often traditional relational databases such as *Oracle Database*, *Microsoft SQL Server*, and *MySQL*. However, there exists several adapters, or object-relational mappers, for other data sources, including SOAP-based web services. More integrations are planned in the future, which include adapters for NoSQL databases and RESTful web APIs.

Applications developed in *Genus App Platform* may build on already existing data sources or define a new, or partially new, data source to store application data. Since *Genus App Platform* handles all data integrity with validations, triggers, duplicate handling, and foreign keys, such constraints do not need to be specified and defined in the database infrastructure.

4.2.2 Genus App Services

Genus App Services is the main backend component in *Genus App Platform*. It facilitates communication between the clients and the database infrastructure, handles authentication, and manages sessions. *Genus App Services* is made up of several nodes and node groups, where each node in a node group is configured to run one or more services that are required by the application. Requests to *Genus App Services* are sent to the right node group based on application model and application dataset, and to the correct node based on which service is required.

All nodes in *Genus App Services* are stateless. This principle simplifies horizontal scaling since more nodes may be added if one particular service becomes a bottleneck. Nodes may still use caching mechanisms for performance reasons.

4.2.3 Genus Studio

Genus Studio is the software modeling tool in *Genus App Platform*. In this tool, expert users use models close to the business problem and generic software concepts to create end user applications. In *Model-Driven Engineering* terminology, this tool belongs to the information system development layer. We study application development in *Genus Studio* in Section 4.3.

Genus Studio runs on the *Windows* platform.

4.2.4 Genus Desktop and Genus Apps

Genus Desktop and *Genus Apps* are the end user clients for *Windows* and mobile platforms respectively. End users access their applications and their respective data using these clients. Instances of these clients point to an application model and a corresponding dataset, and they communicate with *Genus App Services* to receive sufficient model information and data to run the application.

In *Model-Driven Development* terminology, *Genus Desktop* and *Genus Apps* correspond to the *Information System Layer*.

4.3 Application Development

Genus Studio is the software modeling tool in *Genus App Platform* and is used by expert users to design and develop end user applications. Applications are defined by specifying three different layers: The *data layer*, *logic layer*, and *user interface layer*. These layers are analogous to the information structure, microflow, and form models used by *Mendix*.

4.3.1 Data Layer

One of the layers in *Genus App Platform* application development is the *data layer*, which is seen in Figure 4.2. This layer contains object class definitions with properties, as well as hierarchies and connections to explain how object classes relate to each other. The data layer specifies how data is fetched and stored in the underlying database infrastructure, for instance by specifying database connection strings and table names. Schemas in the data layer represents how the object model is represented when exposed through an API, which is primarily used to create and consume SOAP-based web services.

The data layer provides security functionality. This feature limits access to certain objects or certain properties. Other core functionality specified in the data layer include integrity and validity checks, formulas, triggers, and events.

A class diagram is used to model the data layer, which we will discuss in greater detail in Section 4.4.

4.3.2 Logic Layer

The *logic layer* in *Genus App Platform* lets the modeler extend the application with custom logic. The main components in this layer are *tasks and effects*, which is

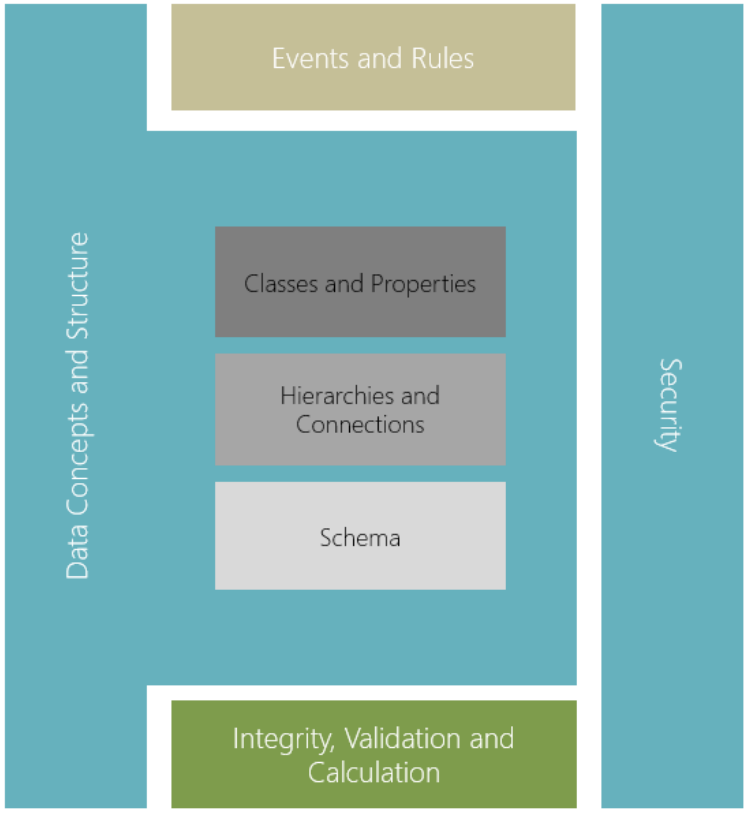


Figure 4.2: The data layer in *Genus App Platform*. This layer defines object classes, object relationships, data integrity and calculation, and security. (Adapted from [32])

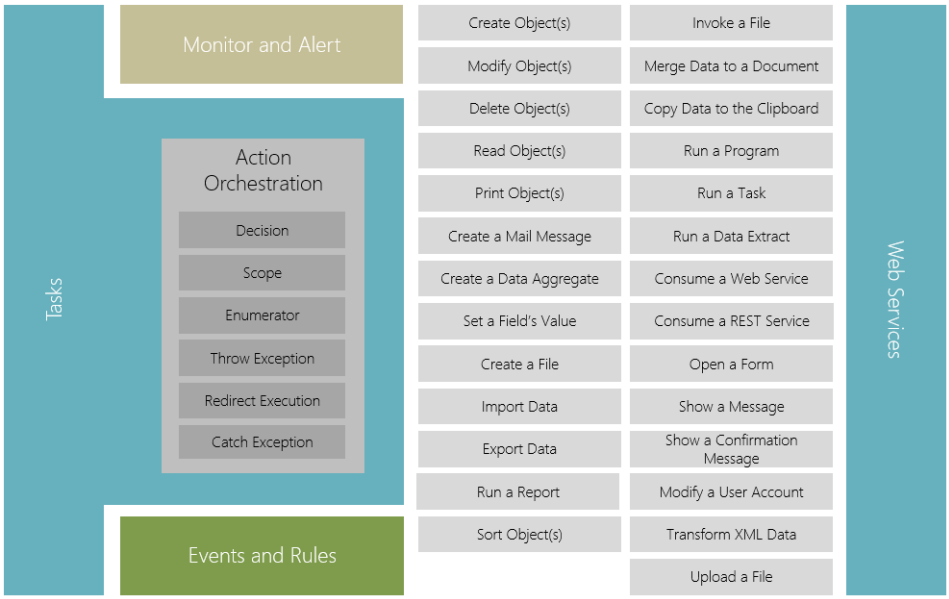


Figure 4.3: The logic layer in *Genus App Platform*. Here, tasks and effects are specified through action orchestration. Tasks contain programming-like constructs and is composed of a wide variety of actions. (Adapted from [32])

analogous to the microflows in *Mendix*. The logic layer is shown in 4.3.

Effects and tasks are created through *Action Orchestration*, which lets the system developer specify logic through programming language-like constructs, such as loops, conditional statements, and exceptions. There is a large number of different actions that might be included in an orchestration, which include object creation, modification, and delete, file handling, and consuming web services. There are several ways to run an effect; for instance through clicks and interaction in the user interface, recurring events triggered by a timer, or triggers caused by changes in the data.

4.3.3 User Interface Layer

The GUI made available to the end users is modeled in the *user interface layer*. The main components in this layer are the table and form views, where the latter can be designed for either *Genus Desktop* or *Genus Apps*. Form views allow the modeler to specify custom layouts and include a wide variety of different GUI elements. Supported elements, which is seen in Figure 4.4, include text boxes, buttons, calendars, maps, and more.

Components in the user interface layer fetch data from a data source, which specifies

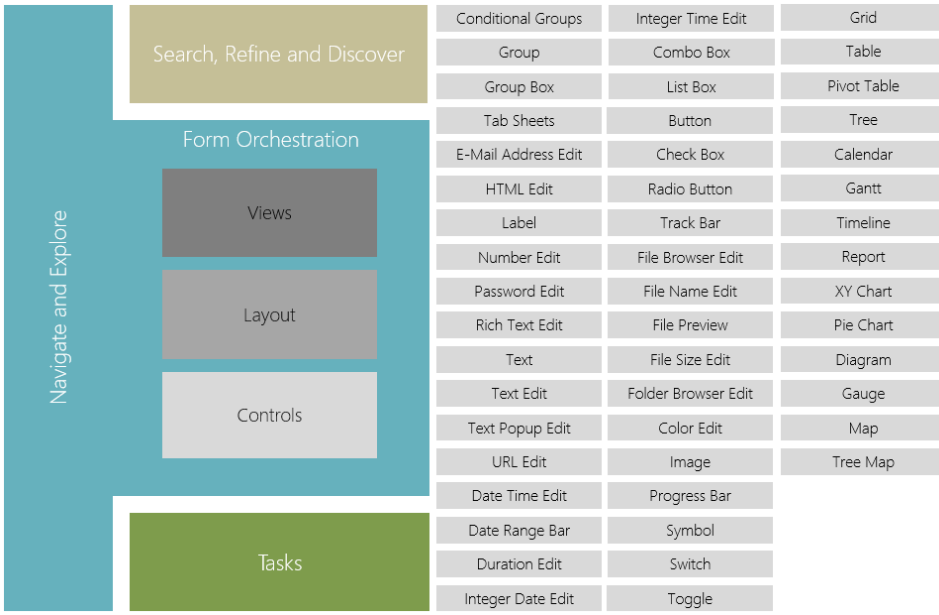


Figure 4.4: The user interface layer in *Genus App Platform*. In this layer, forms and tables are specified, with a wide variety of available GUI elements. (Adapted from [32])

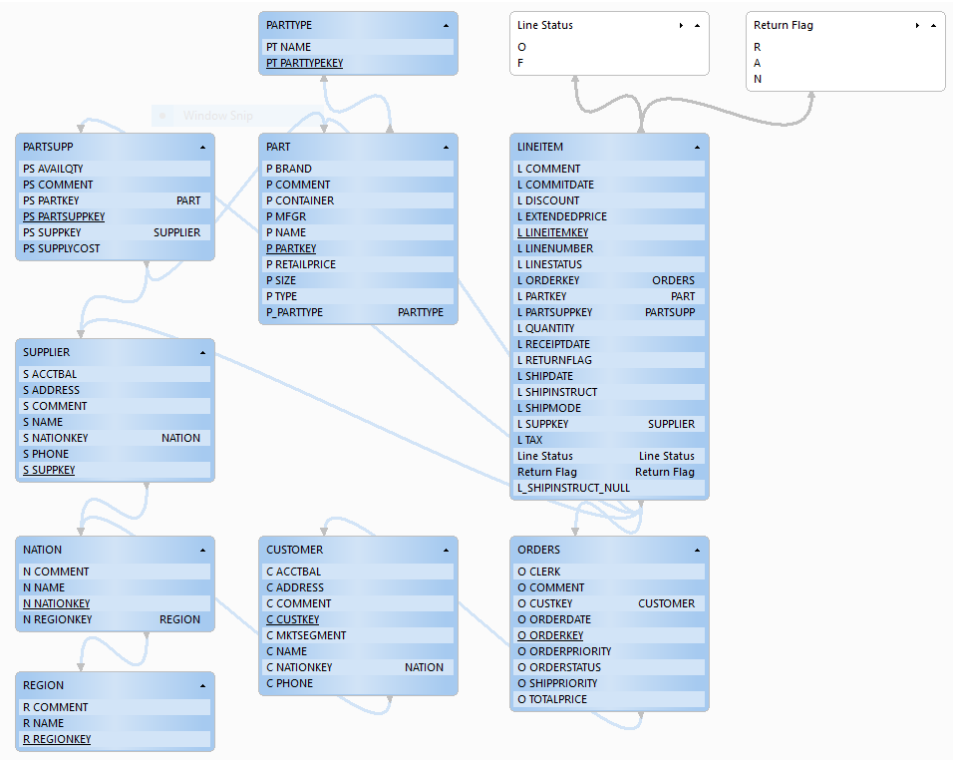


Figure 4.5: Class diagram from *Genus Studio* for the *TPC-H Benchmark*. Blue units represent object domain compositions which mean they are mapped to the underlying database infrastructure. White units represent code domain compositions where values are ”hard coded” and fetched from the application model itself.

which data that should be made available to the users. The user interface may interact with the logic layer through user interaction, such as button clicks.

4.4 Concepts

To better understand how the three application development layers relate to *Genus App Platform* source code, we elaborate on some important concepts used in the platform.

4.4.1 Object Classes

An object class in *Genus App Platform* is a hybrid between a table definition in a relational database and a class in a programming language. An object class,

sometimes referred to a composition, specifies several class properties (Section 4.4.2), security settings, integrity constraints, formulas, and other display options. An object class can either be in the *object domain* or the *code domain*. Compositions belonging to the object domain must specify how it relates to the underlying database infrastructure, for instance by providing a database connection string and table name, or an XML schema if the data is fetched using a SOAP-based web service. Object classes in the code domain do not have this ability, since these values are ”hard coded” into the application model, and not fetched from any data source.

Object classes are defined in the application data layer and specified in a class diagram, which is depicted in Figure 4.5.

4.4.2 Object Class Properties

An object class is composed of one or more properties. Such properties can be simple data types, like integers and strings, or a function type, which calculates their value based on other properties in the class. In addition to specifying the basic data type that holds the value, properties also define an interpretation of the data. There exists many such interpretations in *Genus App Platform*, including *file*, *password*, *date format*, and *color*. The data interpretation may also be another object class, which is used on foreign key fields. This creates relationships between object classes in the data layer.

An object class property has its own set of security and validation rules and includes display options like formatting and screen tip.

4.4.3 Data Source

A *data source* in *Genus App Platform* is used by effects, tables, and forms to retrieve, create, modify, and delete data. One particular data source is associated with an object class, a filter which defines which subset of the objects that should be available, and cardinality; whether there may be one or multiple elements in the data source. A data source stores the composition objects in main memory of the clients.

Which data subset that is loaded into a data source is specified using a filter. This filter is translated to SQL or another relevant query language and then sent to the database infrastructure. Thus, at the initial data load, *Genus App Platform* utilizes efficient database servers and reduce network traffic as data is filtered in the backed. However, a data source does not need to load any data at initialization. Empty data sources are, for instance, used for temporary in-memory processing.

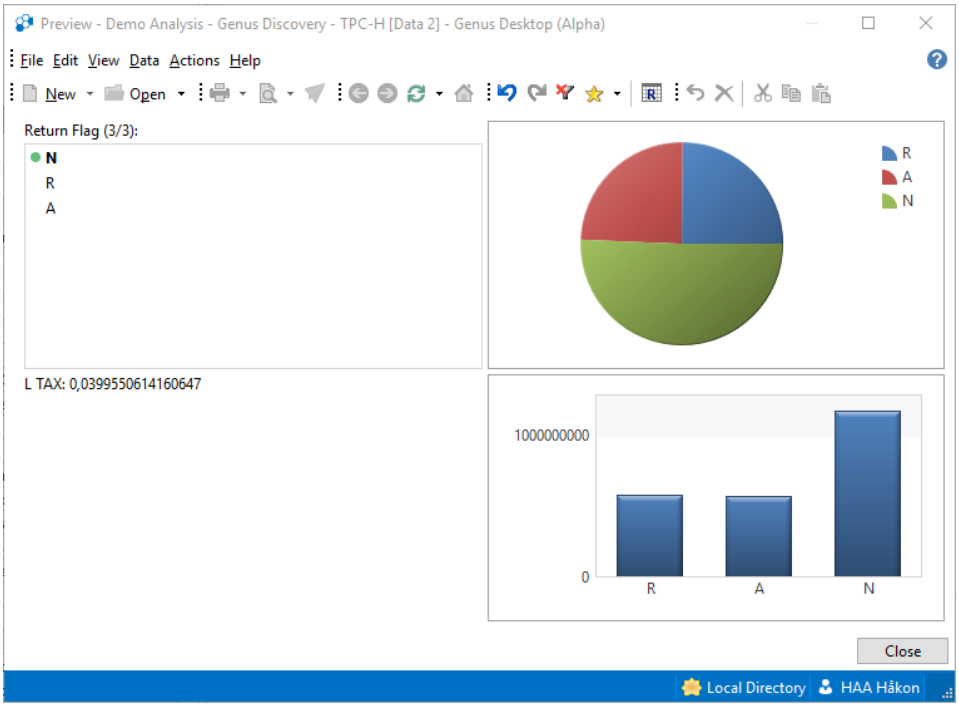


Figure 4.6: *Genus App Platform* analysis, or *Genus Discovery*, user interface.

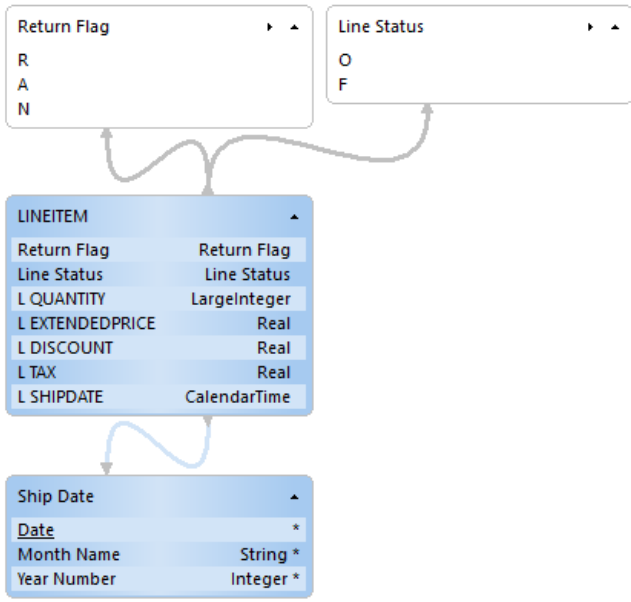


Figure 4.7: Data mart for the *Data Mart Load Benchmark*, which contains only the object classes and fields needed to answer that particular query. The mart conforms to a snowfalke schema.

4.4.4 Analysis

The *analysis* component, sometimes refered to as *Genus Discovery*, is the *Business Discovery* functionality in *Genus App Platform*. *Genus Discovery* is is very similar to *QlikView*, which we studied in 2.3, but since object classes and relations are already specified in the data layer, no data import script is needed. Like *QlikView*, the analysis component allows end users to follow their "information scent" and click their way through the data using an intuitive user interface. An example user interface is seen in Figure 4.6.

One particular dashboard is referred to as *an analysis*. Each analysis links to a specific data extract, or *data mart*, which we study in the next section.

4.4.5 Data Mart

A data mart defines a data extract that is used by the analysis component in *Genus App Platform*. A data mart is specified by a set of data sources, and for each data source, a list of published fields. All data sources in a mart might specify a filter, such that the analyses only access a particular subset of the data.

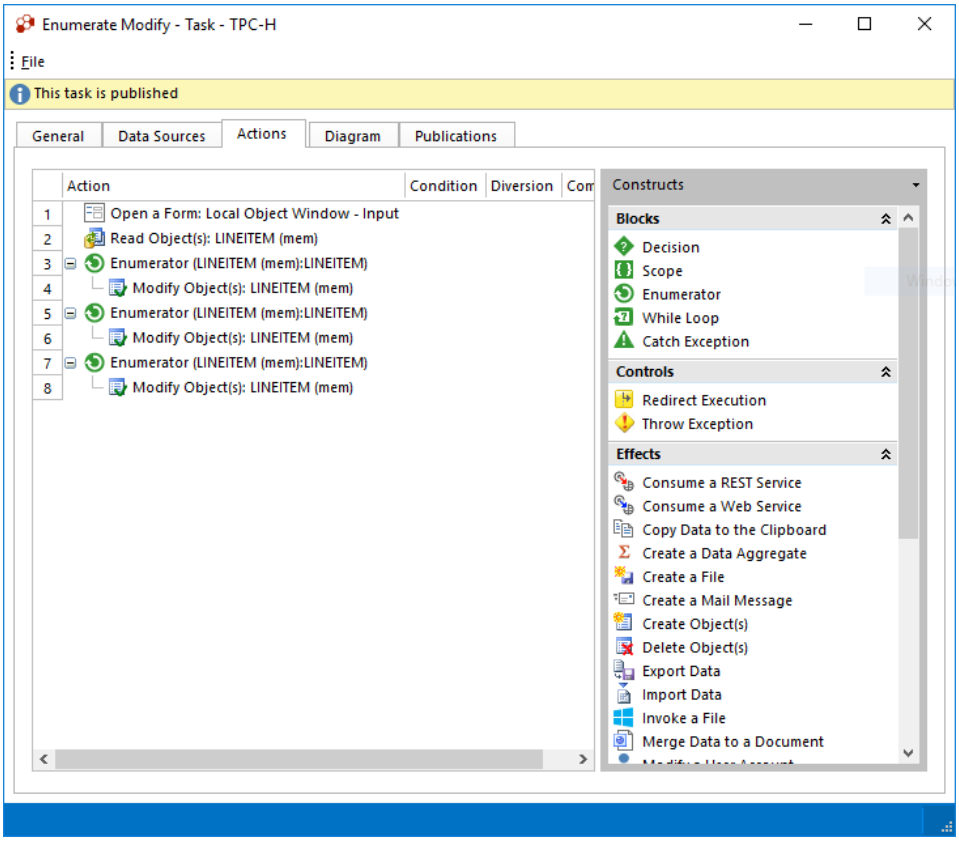


Figure 4.8: A task definition in *Genus App Platform*. A task is composed of actions and effects. This way, a task may read from and to data sources, enumerate items using for or while loops, consume services, import, export, and more.

The process to define a data mart, including the user interface, is similar to defining object classes in an object diagram, like seen in Figure 4.7. However, there is one important distinction: A data mart must conform to a snowflake schema, which means there can only exist one path through the class diagram. For instance, in Figure 4.5, both the customer and supplier classes refer to the nation class. In a data mart, there would be two nation entities; one entity for customer nationality and one for supplier nationality.

4.4.6 Tasks

Tasks in *Genus App Platform* are similar to a procedure or a function in a programming language, and is defined at the logic layer. A custom user interface, depicted in Figure 4.8, enables modelers to combine actions and effects into logic, a process

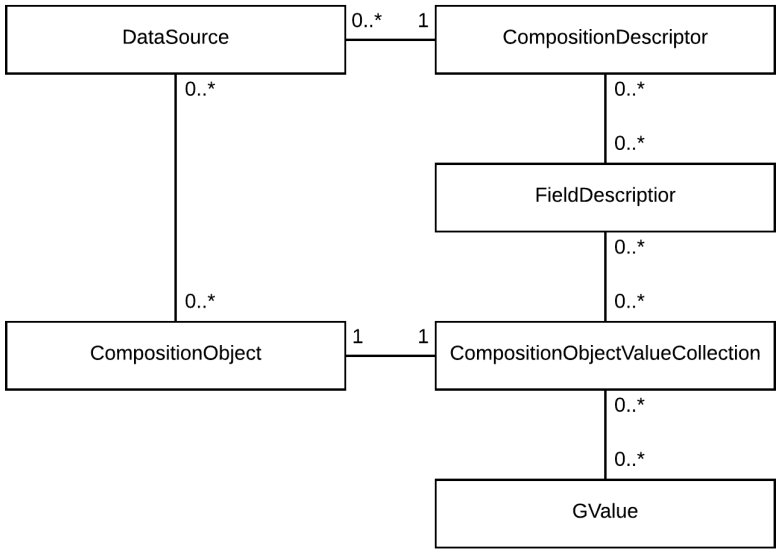


Figure 4.9: A class diagram with the most important classes in *Genus App Platform* data representation.

denoted as *action orchestration*. Tasks use data sources, which we saw in Section 4.4.3, to retrieve, create, modify, or delete data. Tasks might be triggered manually, by user interface interaction, or periodically by *Genus App Platform* agents.

4.5 Source Code and Classes

This section aims to give an overview over the classes used to represent data in *Genus App Platform*, which includes the representation of object classes, object class properties, object instances, and data elements. Figure 4.9 displays a class diagram for the classes we discuss in this section.

The source code of *Genus App Platform* core is written in *Delphi*, which we studied in Section 2.4.

4.5.1 GValue

The `GValue` class, short for *Genus Value*, is a base class that can contain any value of any type that is supported by the platform. These types include integers, floating point numbers, strings, object handles, dates, and more. The `GValue` class is a

necessity in *Genus App Platform* to omit the rigid type system in *Delphi*, and to enable that all values, irrespective of type, are handled the same way.

For all value types supported in *Genus App Platform*, there are get and set methods for that specific value type, which by default throw exceptions indicating that they are not implemented. Then, for each type, a subclass exists which stores the data and overrides the corresponding get and set methods. For instance, there exists an `IntegerValue` subclass with an integer property that extends `GValue` and overrides the `GetAsInteger` and `SetAsInteger` methods.

A `GValue` is a class type and not a record type, hence is allocated by the memory manager and put on the heap. Thus, a `GValue` variable is a pointer. Like strings and dynamic arrays, `GValue` has built-in reference counting to enable cloning and data re-use. Still, the class is a true value type, which implies that it is immutable.

4.5.2 CompositionDescriptor

The object class, or composition, which we discussed in Section 4.4.1, is represented by the `CompositionDescriptor` class. Conceptually, this class is the same as a class in a programming language. A `CompositionDescriptor` contains all information needed to describe an object class, for instance which properties, or `FieldDescriptor`s, that belongs to to the composition.

Since `CompositionDescriptor` instances are a part of the application model, they belong to the ISD layer, which we discussed in Section 2.1.1. In the linguistic meta modeling framework, this construct makes up the left part of Figure 2.2, which is the ”Class” and ”Collie” entities.

4.5.3 CompositionObject

Whereas the `CompositionDescriptor` class represents an object class definition, the `CompositionObject` class represents particular instances of that object class. Conceptually, this class is the same as an object, or class instance, in a standard object oriented programming language. A `CompositionObject` stores all data belonging to the particular instance in a `CompositionObjectValueCollection`, as well as some other object state variables.

Instances of `CompositionObject`s make up the data in the applicaiton, and is, therefore, associated with the IS layer in *Model-Driven Engineering*. In the linguistic meta modeling framework, this class is depicted on the right part of Figure 2.2. `CompositionObject` has an ontological *instance-of* relationship with `CompositionDescriptor`.

4.5.4 FieldDescriptor

The `FieldDescriptor` class describes an attribute in a composition, which is the object class property we studied in Section 4.4.2. It holds all relevant data about a property, mainly type and representation, but also constraints and formatting rules. Like `CompositionDescriptor`, instances of the `FieldDescriptor` class exist in the ISD layer.

4.5.5 CompositionObjectValueCollection

The `CompositionObjectValueCollection` class holds all data for a composition object, and has a one-to-one relationship with the `CompositionObject` class. Conceptually, this class represents all the properties with corresponding data stored in a class. Like a `CompositionObject`, instatiations of this class belongs in the IS layer.

A `CompositionObjectValueCollection` contains two lists, one with object class properties, and one with `GValues`. To look up a particular property, a linear search through the property list is performed, and when the particular property is found, the function returns the corresponding `GValue`. One way to see the `CompositionObjectValueCollection` class, is as a database table with one row.

4.6 Challenges in Genus App Platform

There are several challenges in the current *Genus App Platform* design. For years, the main area of focus has been to keep the source code readable and maintainable. However, this approach does not normally lead to an implementation that is optimal performance-wise. In this section, we discuss some parts of the *Genus App Platform* source code that is not optimal and has resulted in challenges related to high memory usage and poor memory access patterns.

4.6.1 Excessive Amount of Pointers

One of the main challenges in *Genus App Platform* is that a significant amount of pointers is needed to store object data. As we saw in Section 4.5.5, each `CompositionObjectValueCollection` not only stores `GValue`s containing the data, but also pointers to all data descriptors. Although this implementation is flexible because every value collection is self-contained and objects can have a variable number of fields loaded, the overhead for storing all these pointers is substantial, especially on a 64-bit architecture where all pointers are 8 bytes. Figure 4.10 illustrates all pointers in play when a data source is filled with data.

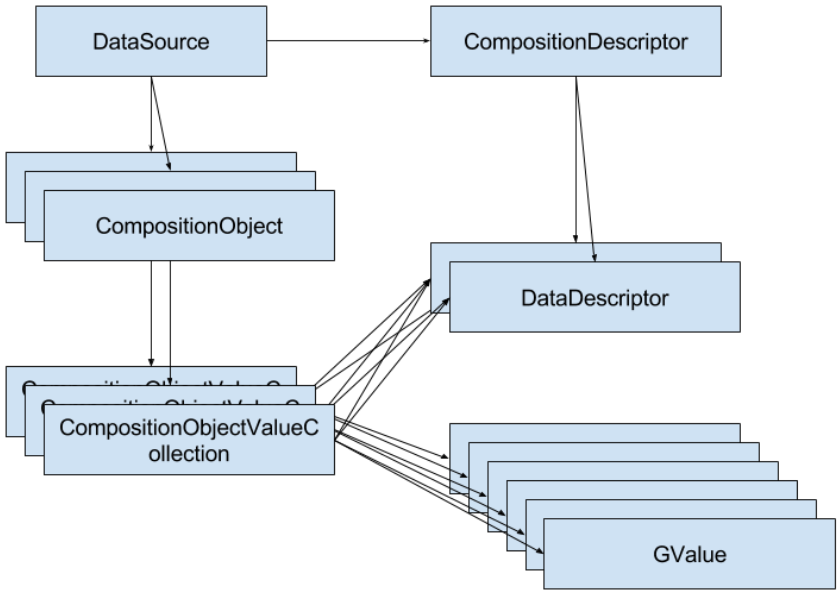


Figure 4.10: A data source with three composition objects, where each object has two properties. For each object, there are pointers to both data values and field descriptors.

We study the storage overhead caused by pointers using a simple example. In our example, we use an object class with 15 object class properties. In the data source, there are 1,000,000 elements. The pointers needed in this setup are:

- **1 pointer**, for the data source.
- **2 pointers**, for the composition descriptor and the pointer to the list of data descriptors.
- **15 pointers**, for the composition data descriptors
- Then, for every object:
  - **1 pointer**, for the composition object.
  - **1 pointer**, for the value collection pointer.
  - **15 pointers**, for all data descriptors in the value collection.
  - **15 pointers**, for all values in the value collection.

The example results in 32,000,018 pointers, which is roughly 244 MB on a 64-bit architecture. Moreover, this is only the overhead caused by pointers, not the actual data. If all values are 32-bit integers, the memory needed store these 15 million numbers is 57 MB, which in this example is 20 % of the space used by pointers. For 1 byte booleans, the percentage is even less.

4.6.2 Inefficient Data Access and Poor Memory Locality

Several operations in *Genus App Platform* access many values in tight loops, and it is not uncommon they read all values for a given data descriptor in a data source. Examples of such operations are filter and join operations. However, accessing values in the current implementation comes with a substantial overhead. The following steps are performed for every data access:

- The `CompositionObjectValueCollection` is scanned linearly for the correct data descriptor.
- The corresponding `GValue` is found, and the accessor method (i.e. `GetAsInteger`) is looked up in the dispatch table.
- A new stack frame is put on the stack to run the `GetAsInteger` method.

The compiler might be able to optimize some of the steps above. For instance, if the *register* calling convention is used, values may be accessed without creating a new stack frame. However, the process of getting a value is far from ideal. As we saw in Section 3.10, the difference between minimal and maximal CPU utilization can easily be one order of magnitude, where branches and new stack frames may severely hurt performance. Last, the linear search through data descriptors hurt throughput.

Not only are there many steps required to access data in *Genus App Platform*, but there is also poor memory locality on the data elements. Since `GValue` is a class type, thus allocated on the heap by the memory manager, we no longer have control over where values are located in main memory. We must assume the location is arbitrary, and that `GValue`s with temporal locality are not located next to each other in memory. We read in Chapter 3 how important cache locality is for performance.

4.6.3 Storage Overhead

`GValue`s have much storage overhead. First, since they are reference counted, each value contains a 4-byte integer. Second, since a `GValue` is a class type, the first 8 bytes in an object is a pointer to a virtual method table. Last, a `GValue` variable is a pointer to the structure containing the data, a pointer which is an additional 8 bytes. Hence, for a 32-bit integer, only 4 of 24 bytes are used to store the actual data.

Secondly, *Genus App Platform* applies no form of compression. Using the same reasoning as above, we see that each boolean value will require 21 bytes to store. Theoretically, only 1 bit is needed to store such value, and we studied in the section about compression, Section 3.3, how to achieve this effect. Reducing the memory per value from 21 bytes to 1 bit is a memory reduction of almost 200 times.

We saw in Chapter 3 that small storage overhead and compression are paramount for read-optimized databases. Efficient data storage may turn a process from being I/O bound to CPU bound.

4.7 Research Motivation

This chapter concludes the related work part of our research. We have studied background theory on *Model-Driven Engineering* and *Business Intelligence*, elaborated on techniques used by read-optimized databases, and provided an analysis of *Genus App Platform*. The literature review has given us sufficient insight to proceed with the evaluation part of this research.

We observe that *Genus App Platform* is in fact a sophisticated in-memory database. Data sources used in various parts of the platform query the underlying database infrastructure and loads the data into main memory. We also see *Genus App Platform* source code has several challenges with pointer and storage overhead and data locality. We repeat our research goals:

> **G1: Reduce memory consumption in *Genus App Platform* and increase the platform's ability to handle and analyze large datasets.**
>
> **G2: Introduce new evidence that *Model-Driven Engineering* can benefit from in-memory database technologies.**

We are motivated to apply techniques used in read-optimized databases to *Genus App Platform*. We believe techniques like column storage and compression can mitigate several problems in *Genus App Platform* source code, such that memory usage can be reduced, and the platform gains the ability to handle and analyze large datasets. We also believe this approach serves an example of how such problems are solved in the context of *Model-Driven Engineering*. As a result of this, we form our research question:

> **RQ1: How can technology used by in-memory, read-optimized databases improve *Genus App Platform*'s ability to handle and analyze large datasets, and what can *Model-Driven Engineering*, in general, learn from database technology?**

By answering **RQ1**, we hope to address **G1** directly by making changes in *Genus App Platform* that increase the platform's ability to handle large datasets. However, by using our changes in *Genus App Platform* as a proof-of-concept, we plan to address **G2** by drawing general conclusions on the combination of *Model-Driven Engineering* and database technology.

We plan to optimize *Genus App Platform* in the method engineering layer, such that the information system development layer is affected as little as possible. Much like a compiler is better at optimizing code than a human programmer, we believe *Genus App Platform* has the potential to optimize storage format without modeler

intervention. This way, *Genus App Platform* expert users may focus on the business problem and not on the underlying implementation.

Part II

Evaluation

Chapter 5

Iteration I: Column Store

In this chapter, we modify how *Genus App Platform* represents data internally by replacing the original row storage format with a column store. We see how it successfully reduces memory consumption and load times for the *Data Mart Load Benchmark*. The work undertaken in this chapter lays some important groundwork for further optimization.

This chapter makes up the first out of four main iterations in the design and experiment part of this research. As mentioned in the introduction, we aim to reduce memory consumption in the first iterations of the research.

5.1 Introduction

Starting our main part of the research, we aim to reduce memory footprint in *Genus App Platform*. As we saw in Section 3.3, lower memory usage are likely to increase performance because most database processes are memory-bound. Moreover, we hypothesize that memory management is expensive and that the number of allocations, as well as the size of the allocated memory chunks, should be reduced. Inspired by the techniques used by read-optimized databases in Chapter 3 and motivated by the challenges in *Genus App Platform*, we change how data is represented internally within the platform.

We choose to implement a column store, which we read about in Section 3.2. The reason for this is two-fold. First, as we saw in the literature study, columns are inherently more compressible. Hence, a column store is better suited to reduce memory consumption and to test our hypothesis about the costs of memory management. Second, the cases where *Genus AS* has experienced performance issues are in situations where a column store is better suited than a row store, for example the read-intense join and filter operations.

We believe that conventional transactional processing in *Genus App Platform*, where a row store might be better suited, will not suffer from using a column store. In *Genus App Platform*, there is much overhead on object create, update, and delete, like constraint checks, data validation, and memory allocations. Thus, we hypothesize that tuple materialization costs and the effects of decreased memory locality are negligible. Also, a row store might require index structures to accommodate certain operations; indexes which, as we have seen in Section 3.2.2, are costly to maintain.

5.2 Implementation

In this section, we explain how a column store is implemented in *Genus App Platform*. We change the original row store by introducing data source indexes to `CompositionObject`s and replace `CompositionObjectValueCollection` with a new class we denote as `CompositionValueCollection`.

5.2.1 CompositionValueCollection

As we saw in Section 4.6, one of the challenges in *Genus App Platform* is the `CompositionObjectValueCollection` class that has an excessive amount of pointers. Not only does the class hold references to all data for an object, but also pointers to field descriptors. Although this class is self-contained and flexible, there is no need to store the field descriptor references in every row.

To reduce the number of pointers, and to create a column store, we replace instances of `CompositionObjectValueCollection` with a new class which we denote as

`CompositionValueCollection`. This class represents the data storage container for objects in a data source. Also, we extend the `CompositionObject` class with a new integer attribute, `DatasourceIndex`, to identify which objects belong to which indexes in the data buffers. The data source index can be thought of a row identifier which we saw in Section 3.2.3.

Our modifications imply that composition objects in a data source no longer queries its own data collection, or `CompositionObjectValueCollection`, for values, but instead request values from one shared `CompositionValueCollection`. To access values in the column store, objects must pass their data source index and a field descriptor to indicate row and column respectively. A comparison between the original and the new implementation is seen in Figure 5.1. Even though the original `CompositionObjectValueCollection` is discarded as the main data source storage structure, the structure may still be created by the object. *Genus App Platform* uses `CompositionObjectValueCollection` extensively in various parts of the application, for instance for object cloning and data transfer.

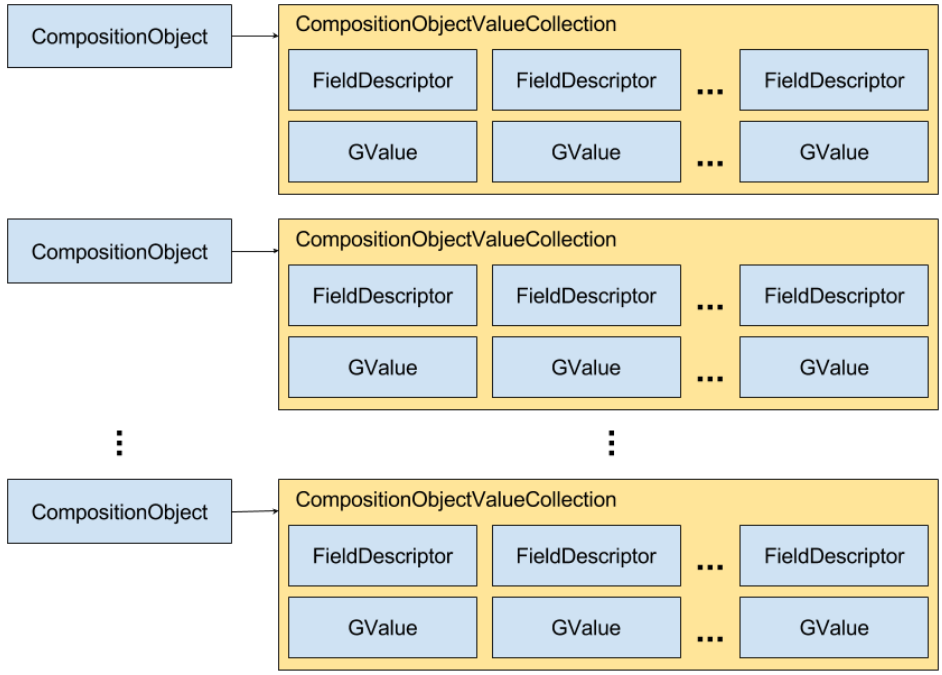
The class diagram for `CompositionValueCollection` is seen in Figure 5.2. The main methods in this class are `GetValue` and `SetValue`. In these methods, a dictionary lookup finds the correct column, or `FieldValueCollection`, and requests the value for the given data source index. `IsAssigned` and `UnassignValue` work similarly. In addition to the access methods for single data elements, row- or column-wise operations like `UnassignAllValues` exist.

Data source indexes are assigned by passing composition objects to the `Register-Object` method. A private variable, `maxDatasourceIndex` is used to keep track of the maximum index assigned so far. When an object is removed from a data source, `RemoveObject` is called, and this method keeps track of removed objects in the `assignedIndexes` bitmap.

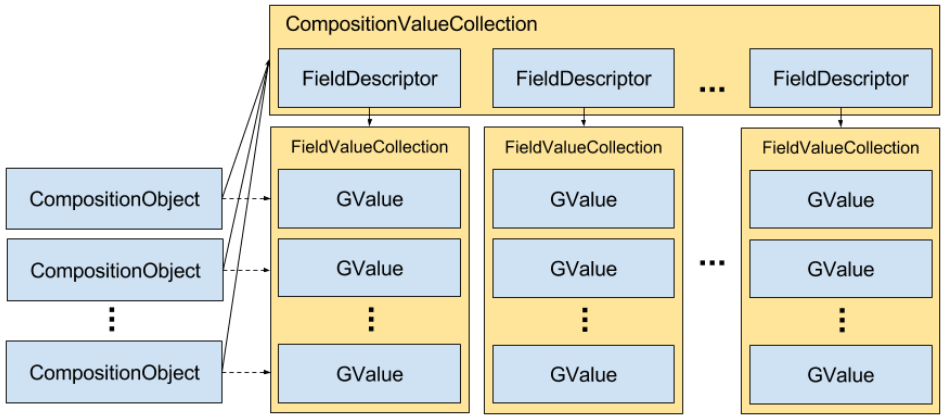
5.2.2 FieldValueCollection

We implement columns as a class we denote as `FieldValueCollection`. This class represents an index based list structure that automatically handles memory allocation. Internally, values are stored in a `TArray<GValue>` type. We chose that structure based on the performance benchmark in Appendix B. A class diagram is found in Figure 5.3.

Besides from the value array and corresponding `GetValue` and `SetValue` methods, `FieldValueCollection` contains bitmaps for indicating null values and unassigned values. Keeping track of null values is strictly not needed, since `GValue`s are boxed and can be set to `nil`, but it becomes apparent why we need this bookkeeping in Chapter 6. Unassigned values have a special semantic in *Genus App Platform*: It means that a value exists, but has been unassigned for performance reasons.



(a) Original implementation with `CompositionObjectValueCollection`. Each object has its own value collection, and each value collection contains references to both data descriptors and the data itself.



(b) Column store implementation with `CompositionValueCollection` and `FieldValueCollection`. All objects in a data source point to the same value collection, and access data using data source indexes and field descriptors to indicate row and column respectively.

Figure 5.1: Comparison of the original row store implementation in *Genus App Platform* and the new column store implementation.

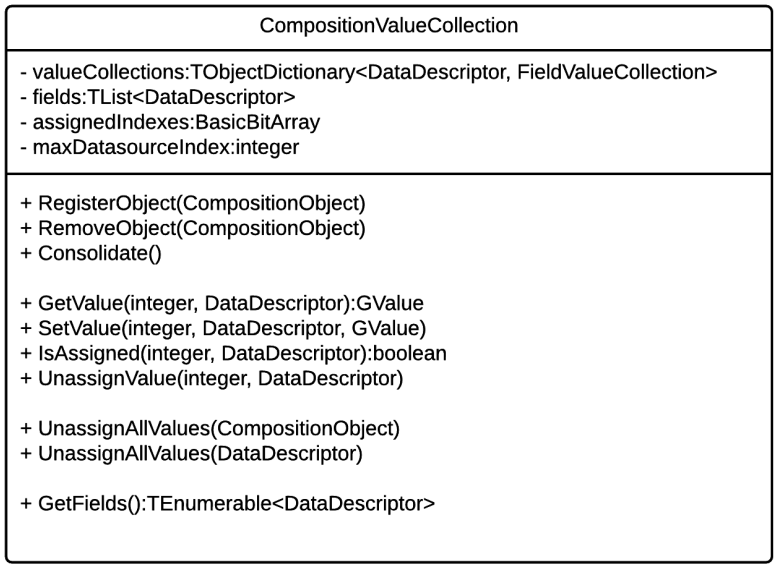


Figure 5.2: `CompositionValueCollection` class diagram.

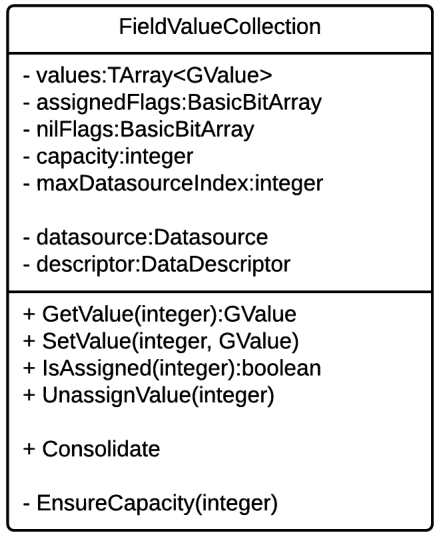


Figure 5.3: `FieldValueCollection` class diagram.

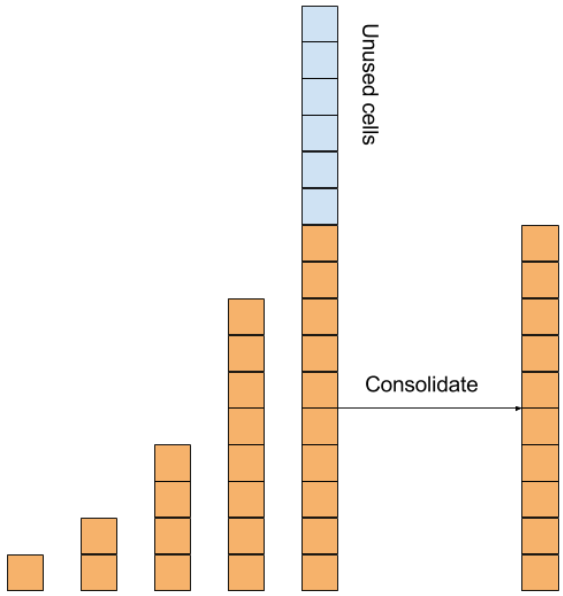


Figure 5.4: Growth strategy for value buffers and bitmaps. In the load phase, the buffers double every time more space is needed. After the load phase, a consolidation is performed which reduces buffer size to the exact size of the data source.

5.2.3 Growth Strategy

In `FieldValueCollection` we use *Delphi* dynamic arrays for data and are, therefore, in control over array allocation size. Since objects arrive in a data stream, we are not sure how large our buffers should be until we receive all objects. Hence, we must find an efficient growth strategy for the column store.

For every array resize, we run the risk that all data in the array must be copied from one smaller memory chunk to a larger one. Since this is a costly operation, one should be generous when resizing arrays. Commonly used in such problems is a doubling strategy, where array buffer size double each time an index outside of array bounds appear. We implement this behaviour in `FieldValueCollection` too, in the `EnsureCapacity` method. This function is called to avoid index-out-of-range exceptions by all methods that access the data array or bitmaps.

Using the doubling strategy for column growth, we run the risk of allocating twice as much memory needed for a data source. We, therefore, implement a `Consolidate` method which resizes all value buffers and bitmaps to the exact size of the data source. The full growth strategy in depicted in Figure 5.4

In `CompositionValueCollection`, columns are created on-demand as composition objects set values. If an object requests a value from a column that does not exist,

`nil` is returned.

5.2.4 CompositionObject Modification

To accommodate the new `CompositionValueCollection` class, we perform some modifications to the `CompositionObject` class. First and foremost, we replace the `CompositionObjectValueCollection` member variable with a pointer to the column store. At the same time, we remove all code that accesses the original row structure and replace it with calls to the `CompositionValueCollection` instance. For all interaction with this class, the data source index is passed to indicate which row the composition object belongs. Second, we call `RegisterObject` within the constructor of the `CompositionObject` class, such that a data source index is assigned. Analogously, `RemoveObject` is called in the destructor.

Some methods in `CompositionObject` require a new implementation as a result of the column store. These methods include `CloneValueCollectcion` and `Adopt-ValueCollection`. Earlier, these methods were trivial, as they only cloned and replaced `CompositionObjectValueCollection` instances. However, since composition objects no longer have ownership in such structures, they must be cloned or adapted by iterating field descriptors and reading or writing data in the column store.

5.3 Results

We test the changes done in this iteration by using Benchmark A.1, the *Data Mart Load Benchmark*. We use this benchmark to see whether memory footprint has been reduced. We are also curious to see the performance impact on data load time, lookup index generation (join), and source measure lookup. To see whether our changes has caused negative effects on write performance, we also run Benchmark A.2, the *Write Benchmark*. Full benchmark details are found in Appendix A.

In this iteration, we test two different configurations: The original *Genus App Platform* implementation and the new column store. Due to time constraints, we run our benchmarks only three times and report the mean. All measurements had low variance, within 15 % of the average measurement.

5.3.1 Data Mart Load Benchmark

In the *Data Mart Load Benchmark*, we found a significant memory reduction in the column store implementation. As we see in Figure 5.5, for SF1 the memory used per `LINEITEM` is reduced with 38 %, and for SF10 the reduction is 30 %. The total application memory footprint for the analysis is reduced from 2685 MB to 1777 MB.

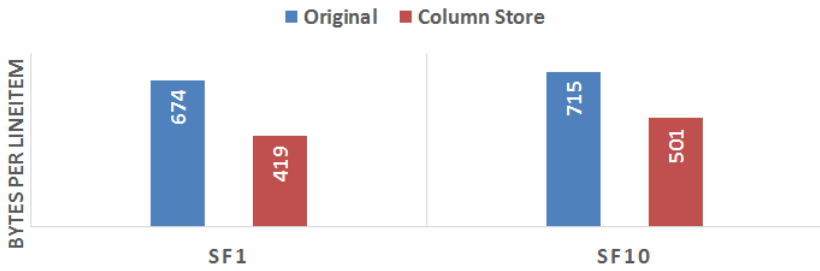
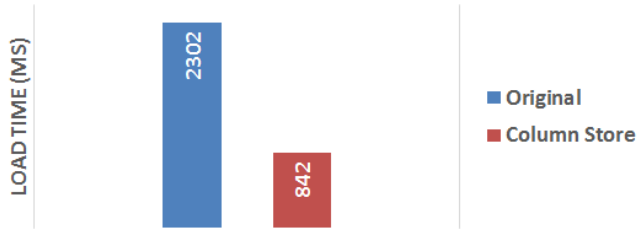
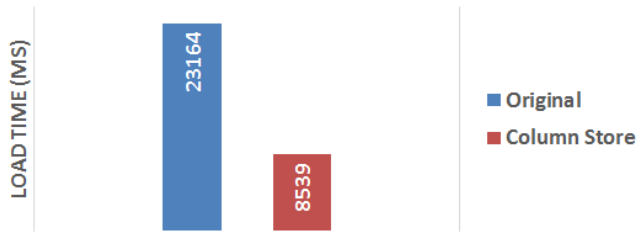


Figure 5.5: Bytes per `LINEITEM` used by the original and column store impelementations, Benchmark A.1 with scaling factors SF1 and SF10.



(a) SF1



(b) SF10

Figure 5.6: Data source load time for for original and column store implementations, Benchmark A.1 with scaling factors SF1 and SF10.

In Figure 5.6, we see that the load time also is significantly reduced using the new `CompositionValueCollection` class. For both scaling factors, the time it takes to populate the data source has been reduced by 63 %.

We observe that lookup index generation, or the join operation, is slightly hurt performance-wise by the `CompositionValueCollection` implementation. For SF1, the time it takes to perform a join between `RETURNFLAG` and `LINEITEM` tables is increased by approximately 30 %. We measure similar changes for SF10, but the percentage difference is smaller.

We observe a larger reduction in performance looking at the source measure lookup operation in Figure 5.8. Here, the time it takes to extract all values in a column into a floating point number array is doubled. We measure the same effect for both scaling factors.

5.3.2 Write Benchmark

To see how the write speed is affected by our new implementation, we run Benchmark A.2, the *Write Benchmark* on the original implementation and the column store implementation. Each run modified 1000 elements, which is the default for this benchmark. The results are presented in Figure 5.9. There are no significant differences between the two implementations.

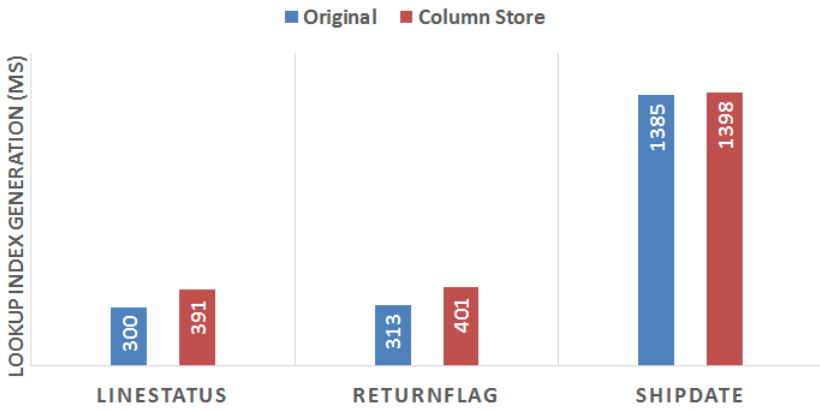
5.4 Discussion

In Section 4.6.1, we saw how a data source with objects with 15 properties each needed 32 pointers, or 256 bytes, per objects just for data and data access. Now, this number is reduced to 17 pointers and an integer, which is 140 bytes. Hence, we have the potential to save 45% memory by using our new system.

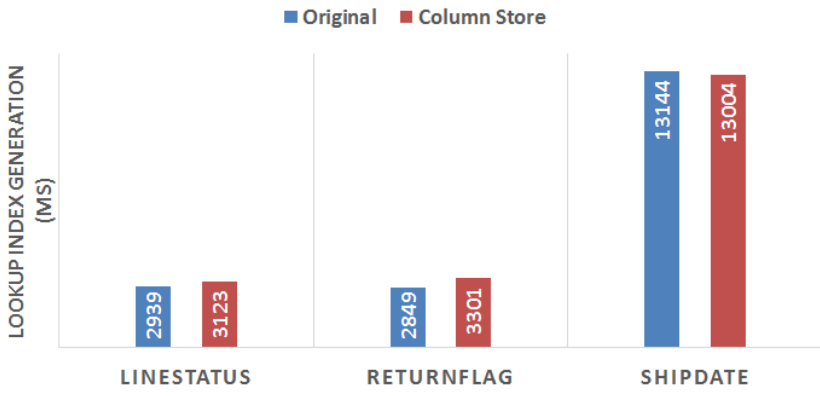
We see that the memory used per object in the *Data Mart Load Benchmark* is reduced by 38 % and 30 % for scaling factors SF1 and SF10 respectively. We attribute this observation to the column store, and how it reduces the number of pointers needed to represent data in a data source.

We are observe that the data source load time is reduced as a consequence of our new `CompositionValueCollection` and `FieldValueCollection` classes. This observation strengthens our hypothesis that memory management is expensive: The original implementation allocated one row-structure per object, whereas the doubling strategy in our column store keeps the number of allocations at a minimum. Moreover, the reduced load time in Benchmark A.1 might also imply that reduced memory usage is directly correlated with performance and that our new implementation has better memory locality and cache hit rate.

Both read operations, lookup index generation and source measure lookup, suffer from the new column store implementation. On these operations, the time it takes

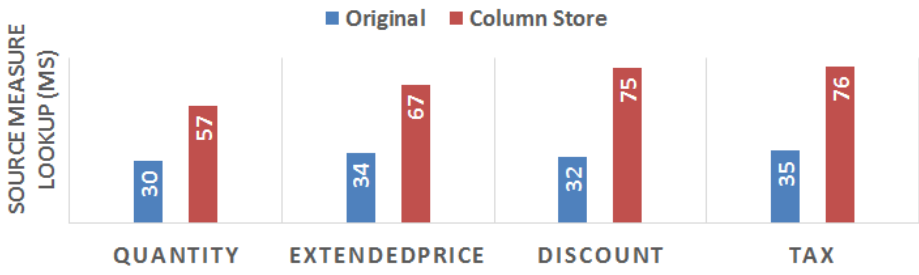


(a) SF1

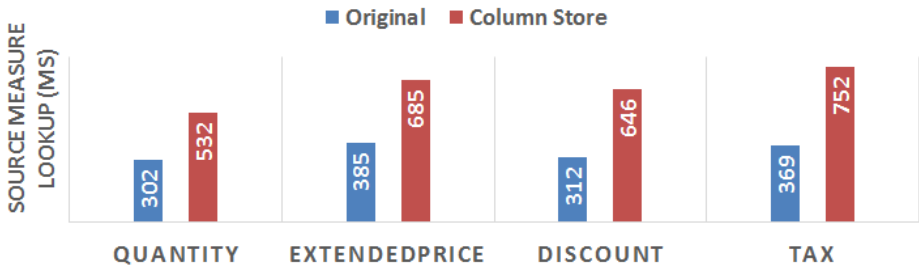


(b) SF10

Figure 5.7: Lookup index generation performance comparison for `LINESTATUS`, `RETURNFLAG`, and `SHIPDATE`, Benchmark A.1 with scaling factors SF1 and SF10.



(a) SF1



(b) SF10

Figure 5.8: Source measure lookup operation performance comparison for `QUANTITY`, `EXTENDEDPRICE`, `DISCOUNT`, and `TAX`, Benchmark A.1 with scaling factors SF1 and SF10.

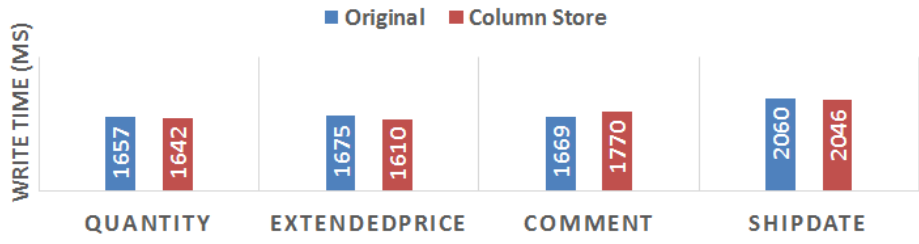


Figure 5.9: Write performance results for Benchmark A.2, 1000 elements.

to perform the operation has been increased by 30 % and 100 % respectively. Since both operations read values intensively from composition objects in tight loops, we believe the test result implies that the `GetValue` function in the column store is not as efficient as the original. This can be caused by the dictionary lookup to find the correct column, or by the extra checks to ensure capacity or for `nil` values.

The fact that Benchmark A.2, the *Write Benchmark*, shows that write performance is not affected by the column store, strengthens our hypothesis that the column overhead is negligible in such situations. Data modification is expensive in *Genus App Platform*. Modifying 1000 elements in the write benchmark takes roughly five times longer than generating a lookup index for 60,000 rows, which indicates that there is a severe overhead in object iteration and value modification unrelated to the data storage layout.

We are unsure why different scaling factors yielded a different number of bytes per `LINEITEM`. It might be caused by the caching and value reuse mechanisms used by the data load module in *Genus App Platform*. Data in SF10 might have a different data distribution than SF1. Either way, both the original and column store implementations are affected by this phenomenon, which makes it likely that it is unrelated to our work. The other measurements in *Data Mart Load Benchmark* increase linearly; measurements from SF10 are roughly ten times higher than for SF1.

5.5 Iteration Conclusion

We conclude this iteration by stating that there is a large potential in column storage regarding memory footprint and load times. Memory used per `LINEITEM` in the *Data Mart Load Benchmark* has been reduced by 30 - 38 %, and the time it takes to load all data in this benchmark by 67 %. Write performance, tested with Benchmark A.2, has not been affected by the new implementation.

Still, both read-intense operations tested in Benchmark A.1 are slowed down, especially the source measure lookup operation. Here, the time it takes to generate an array of double precision numbers for a data descriptor is doubled with the column store. We believe these changes are caused by a `GetValue` function less efficient than the original.

This iteration lays the groundwork for further investigation and to check whether database technologies and column store can help *Model-Driven Development*-tools' ability to handle and analyze large datasets. We believe even more memory can be saved by optimizing storage formats, applying compression, and removing more pointers. We do this in the next iterations, Chapter 6 and Chapter 7.

Column storage comes with several benefits, like vectorized execution, late materialization, and the ability to be pipelined by a modern CPU. We have not yet utilized the full potential of the `CompositionValueCollection` and `FieldValueCollec-`

`tion` classes, so far we have only used them to reduce the number of pointers in *Genus App Platform*. We believe the mentioned techniques can be applied to the read-intense operations of Benchmark A.1. We study this in Chapter 8.

Since the measurements for different scaling factors in the *Data Mart Load Benchmark* increase linearly, that is, measurements from SF10 are roughly ten times higher than for SF1; we will focus on SF10 from now. For the next iterations, we run tests with both SF1 and SF10, but we generally only report and discuss the results from SF10. An exception to this is the number of bytes per `LINEITEM`, where we continue to report on both scaling factors.

5.5.1 Future Work

With our current solution, new data source indexes, or row identifiers, are assigned by incrementing a counter in `CompositionValueCollection`. This implies that if objects are removed from a data source, no new objects will be assigned the data source indexes that has become available. As seen in Figure 5.2, we propose a *bitmap for assigned indexes*, `assignedFlags`. Future work might investigate the effects of using this bitmap, and the `OpenBit` utility function, to assign indexes to a data source. Since `OpenBit` is a linear-time operation, one might consider an object loading state: Indexes are assigned by using a counter if the data source is fed with data from the database, and once this operation has completed, the `assignedFlags` bitmap is used.

In Section 3.2.4, we saw that horizontal partitioning of columns may be beneficial for a column store for several reasons, like metadata pruning, handling of data skew, and parallelism. By introducing a `PartitionIndex` variable in addition to `DatasourceIndex`, and change the column store interface to accommodate the new index, horizontal partitioning can be made possible in *Genus App Platform*. Future work should elaborate on the changes needed in `CompositionValueCollection` and `FieldValueCollection` to enable horizontal partitioning, and investigate the effects of this technique.

Chapter 6

Iteration II: Storage Format Enhancement

In the previous chapter, we concluded that there was potential in column storage in *Genus App Platform*, but that more memory could be saved by enhancing the data storage format. In this chapter, we show how memory usage is reduced and memory locality improved by replacing `GValue`s with primitive data types. We also study the effect of loading raw string values from XML directly into the columns.

This chapter forms the second iteration of our design and experiment part of the research. In this iteration, we are mainly pursuing reduced memory footprint, but the modifications from the storage format enhancements serve as an essential foundation for column operations, which we discuss in Chapter 8.

6.1 Introduction

In Section 4.6, we saw that two of the challenges in the original *Genus App Platform* data representation are poor memory locality and inefficient storage usage. Both challenges are caused by the `GValue` class. In this structure, data is stored inefficiently due to the overhead of pointers and reference counting. For example, it takes 24 bytes to store a 4-byte integer. Secondly, since `GValue` is a class type, it is heap allocated by the memory manager, and we have no control over where the values reside in memory. In Chapter 3, we saw that spatial and temporal locality is paramount for performance.

One might think that memory locality and cache hit rate have been improved by the introduction of `FieldValueCollection` in the previous chapter. This is only partially true; we store value pointers consecutively in memory with the `TArray` type, but the structures containing the data itself reside in arbitrary memory locations.

Section 2.4.1 explained that *Delphi* supports a broad range of simple, or primitive, data types. These include integer, character, Boolean, real, and more. Common for these data types is that they are value types and a variable with one such type stores the data directly, and not as a pointer to the heap. This is also the case for the *Delphi* `record` type.

We observe that if we change the value buffers in `FieldValueCollection` from holding `GValue` references to storing primitive value types directly, we overcome the above challenges. First, we reduce memory consumption by transitioning to primitive data types, since these values have no pointer or reference counting overhead. Second, since we plan to store the values directly in *Delphi* array structures, we improve memory locality.

The data source loading mechanism in *Genus App Platform* operates with `GValue`s, but since we plan to use this class no longer to represent data in a data source, we are curious to see whether `GValue`s can be eliminated from the data load process. We believe data load time can be reduced by loading the database values directly from XML instead of using `GValue`s to load values into columns.

Motivated by the memory reduction and the ability to regain control over memory, we create a new structure based on the `FieldValueCollection` interface, which we denote as `PrimitiveFieldValueCollection`. We also circumvent `GValue` creation in the *Genus App Platform* load module and pass raw XML values directly to the columns. We hypothesize that these changes will reduce memory consumption and decrease load time.

6.2 Implementation

In this section, we start by explaining the `PrimitiveFieldValueCollection` column class and how *Genus App Platform* selects the correct primitive column type.

Listing 6.1: `GetValue` function in `PrimitiveFieldValueCollection`.

```
function PrimitiveFieldValueCollectionBase<TType>.GetValue
( index : integer )
: CGValue;
begin
  EnsureCapacity(index);
  if not nilFlags[index] then
    Result := valueHelper.CreateCGValue(values[index])
  else
    Result := nil;
end;
```

Then we elaborate on how we circumvent `GValue` creation in the data load module.

6.2.1 PrimitiveFieldValueCollection

To add primitive data type support to our column store, we extract core column functionality to an abstract class `FieldValueCollectionBase`. Then, we introduce a new class that extends from this base class, `PrimitiveFieldValueCollection`. This class is a generic class with a subclass for every supported data type. The hierarchy is shown in Figure 6.1.

`PrimitiveFieldValueCollection` holds all data in a `TArray` of the primitive data type of choice. However, `GetValue` and `SetValue` methods still use `GValue` as return and input types respectively. To help isolate all code related to value conversion and avoid code duplication, we create a value helper class which we instantiate in all primitive column subclasses. This class contains methods for extracting primitive data values from a `GValue`, creating `GValue`s based using primitive data types as input, as well as some simple comparison operators. We have provided an example of how it is used in Listing 6.1. Like `PrimitiveFieldValueCollection`, the class is generic, with subclasses for every supported data type.

In the last iteration, we introduced the `nilFlags` bitmap. Whereas `FieldValueCollection` could store `nil` in the value buffers instead of pointers to `GValue`s, we no longer have this opportunity, as simple value types in *Delphi* are not nullable. Hence, `nilFlags` bitmap is used to indicate which values are null. As seen in Listing 6.1, the flags are checked before creating a value.

Even though they are not considered as simple data types in *Delphi*, we apply the same techniques for strings and records. Record types, like `TGuid`, have value semantics and work similarly as simple data types, which means they are allocated consecutively in memory using the `TArray` type. Strings, however, are pointer types, which means pointers are stored consecutively in the columns, but not the data itself. Hence, we do not get the benefit from explicit memory control on strings, but

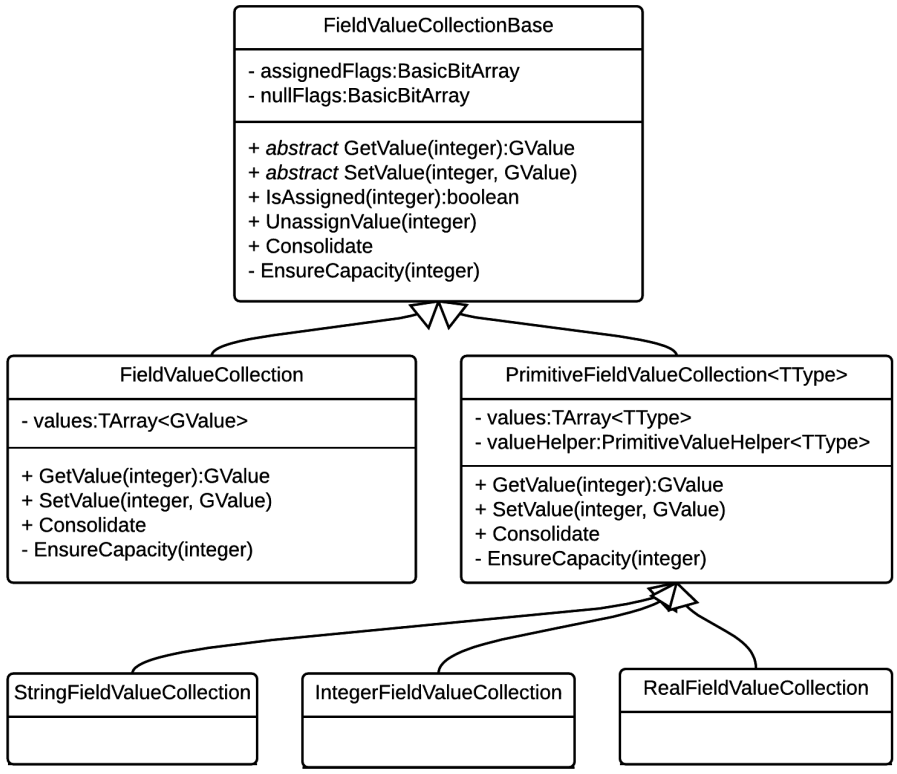


Figure 6.1: Column store class hierarchy after the introduction of primitive data type support. `FieldValueCollection` and `PrimitiveFieldValueCollection` extends a common base class, `FieldValueCollectionBase`. `PrimitiveFieldValueCollection` has one subclass per supported primitive data type. Although string is not a simple data type in *Delphi*, we still create a string primitive column subclass. The primitive value column structure still has an `GValue` interface in getters and setters.

Listing 6.2: `SetXMLValue` in `PrimitiveFieldValueCollection`.

```
procedure PrimitiveFieldValueCollection<TType>.SetXMLValue
( index : integer; xmlValue : string);
begin
  EnsureCapacity(index);
  values[index] := valueHelper.GetFromXMLValue(xmlValue);
  nilFlags[index] := FALSE;
  assignedFlags[index] := TRUE;
end;
```

we remove an extra layer of indirection to access a value and avoid the overhead related to `GValue`.

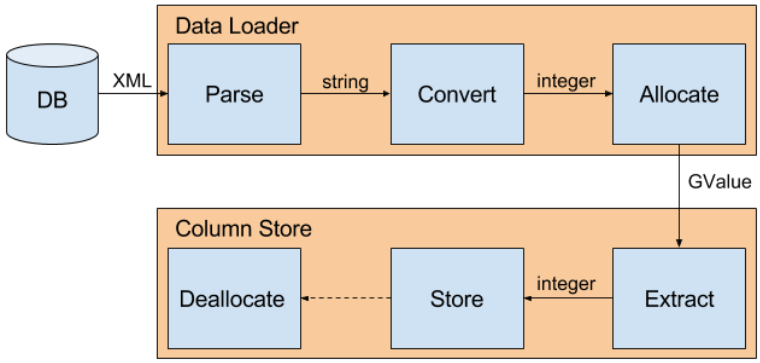
6.2.2 Column Selection

To make the new `PrimitiveFieldValueCollection` class work, we must pick the correct primitive type subclass for the different data descriptors in a data source. This is the responsibility of the column store. We extend `CompositionValueCollection` with a `GetFieldValueCollection` method which takes a data descriptor as an input and returns the correct column. In this method, the data descriptor is queried for its data type, and it picks the correct `PrimitiveFieldValueCollection` subclass. If no matching primitive data type column is found, the method falls back on the original `FieldValueCollection` from Chapter 5.

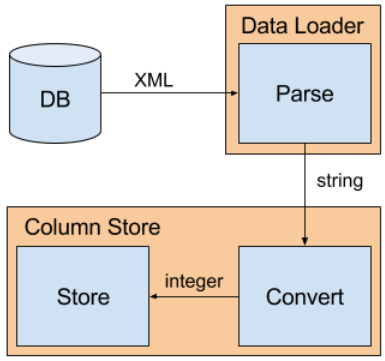
6.2.3 Loading Raw XML Values Directly

In *Genus App Platform*, composition objects in a data source are populated with data using a loading mechanism found in the platform's event handler. We refer to this mechanism as the *data loader*. The data loader queries the database infrastructure for data which is returned as XML. The loader parses the XML into strings, converts the values to the correct value type, and allocates `GValue`s, as seen in Figure 6.2a. Our observation is, with the primitive data type column implementation, that `GValue`s are surplus and causes additional memory management overhead.

Hence, we extend the `FieldValueCollectionBase` interface with a new method, `SetXMLValue`, which accepts a data source index as well as a string XML value as inputs. The value helper class is also extended accordingly. For the different primitive data types, this method is overridden and implemented with high performance standard library functions, like `StrToInt` and `StrToFloat`. For more advanced data types, like date, the correct *Genus App Platform* parser functions are used. We see the `SetXMLValue` implementation in Listing 6.2.



(a) Original implementation where all data is transferred as `GValues`.



(b) Enhanced implementation where data is passed to the column store as strings.

Figure 6.2: By loading raw XML strings directly into the columns, the load process is simplified, and we remove uneccessary memory allocations for `GValue`.

As a result of our changes, the loading process has been simplified and the number of memory allocations reduced. See Figure 6.2.

6.3 Results

Like the previous iteration, we assess our modifications by using Benchmark A.1, the *Data Mart Load Benchmark*. We use this benchmark to test our hypothesis that primitive value storage and direct value load contribute to reduced memory footprint and improved load time.

`GetValue` and `SetValue` in `PrimitiveFieldValueCollection` still use a `GValue` interface, which means memory for the `GValue` is allocated and deallocated on each method call respectively. We are, therefore, interested in testing the effects of this extra memory handling operations with Benchmark A.2, the *Write Benchmark*. We also believe the read-intense operations, join and source measure lookup, in Benchmark A.1 provides insight on this matter.

We test two configurations in this iteration which is the new primitive value column implementation with and without raw XML value load. We denote the configurations as *primitive w/ raw load* and *primitive* respectively. We compare the benchmark results with the previous iteration, and we denote the configuration from this iteration as *column store*. Due to time constraints, we run the benchmarks only three times and report the average result. However, all measurements had low variance with no outliers.

Full benchmark details are found in Appendix A.

6.3.1 Data Mart Load Benchmark

Benchmark A.1 was run with the new primitive value column implementation, with and without the new loading scheme. We run both scaling factors SF1 and SF10, but if we observe the same effects on both scaling factors, we only report for SF10.

As seen in Figure 6.3, primitive value columns reduce memory consumption per `LINEITEM` from 419 to 333 bytes and from 501 to 374 bytes for scaling factors SF1 and SF10 respectively. This corresponds to 21 % and 25 % reduction in memory footprint. However, the *primitive w/ raw load* configuration does not reduce the bytes per `LINEITEM` as much as the *primitive* configuration, although it is still lower than the original column store.

Load times are increased by the primitive column store, but it is still lower than the original *Genus App Platform* row storage implementation. However, with our raw XML loading mechanisms, the load time has increased even more, and is comparable original implementation. The results are presented in Figure 6.4.

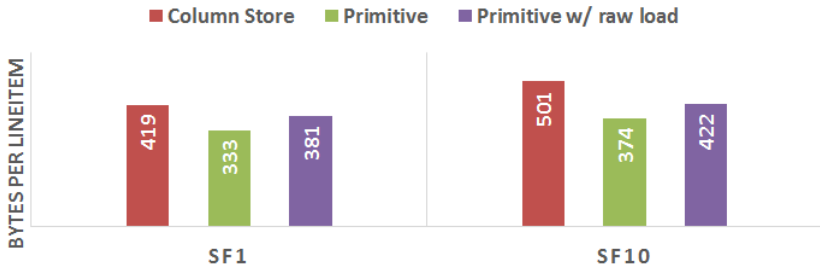


Figure 6.3: Bytes per `LINEITEM` used by the `FieldValueCollection` implementation from Chapter 5, primitive value columns, and primitive value columns with raw XML data load, Benchmark A.1 with scaling factors SF1 and SF10.

Figure 6.4: Data source load time for the `FieldValueCollection` from Chapter 5, primitive value columns, and primitive value columns with raw XML data load, Benchmark A.1, scaling factor SF10.

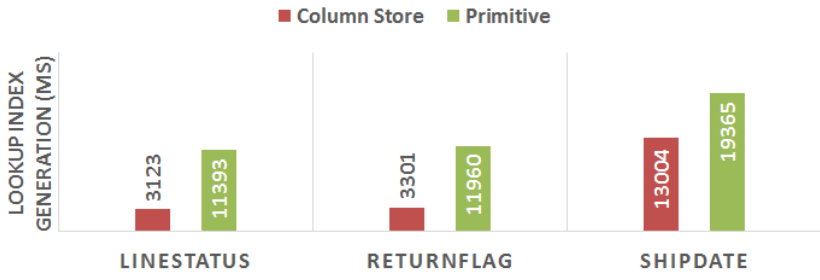


Figure 6.5: Lookup index generation performance for the `FieldValueCollection` from Chapter 5 and primitive value columns, Benchmark A.1, scaling factor SF10.

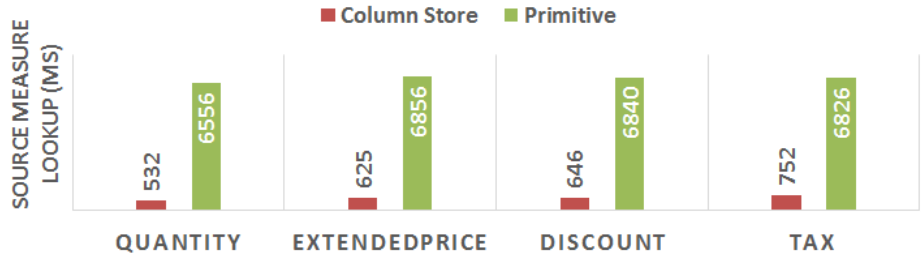


Figure 6.6: Source measure lookup performance for the `FieldValueCollection` from Chapter 5 and primitive value columns, Benchmark A.1, scaling factor SF10.

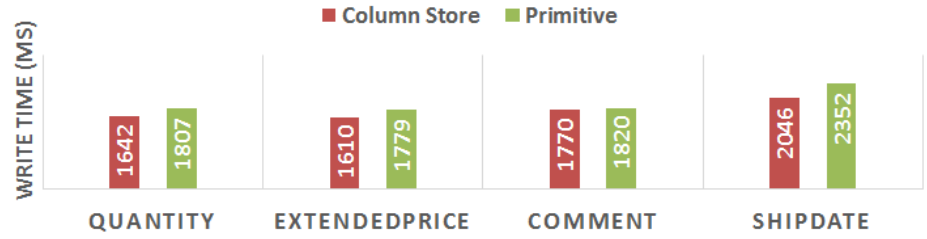


Figure 6.7: Write performance for the `FieldValueCollection` from Chapter 5 and primitive value columns, Benchmark A.2, 1000 elements.

Both read-intense operations in *Data Mart Load Benchmark* are drastically slowed down by the new primitive value storage format. As seen in Figure 6.5, lookup index generation performs 3-4 times worse than the `FieldValueCollection` implementation. Sorce measure lookup sees an even higher performance impact, and is, according to Figure 6.6, ten times slower as the implementation of the previous chapter, and almost 20 times slower as the original *Genus App Platform*.

6.3.2 Write Benchmark

Write performance has, as Figure 6.7 shows, degraded slightly with the new primitive data type column store. However, no operations are more than 15 % slower than the original.

6.4 Discussion

We see that the memory used per `LINEITEM` for scaling factor SF10 has been reduced by an additional 25 % compared to the `FieldValueCollection` implementation.

This observation confirms our hypothesis that the overhead associated with `GValue`s is significant.

We observe an increased load time for Benchmark A.1, where the time it takes to populate the data source is increased from 8.5 seconds to 13.6 seconds. We believe the extra steps needed to extract raw values from `GValue` instances causes this. Still, the load time is lower than the original *Genus App Platform* implementation.

Circumventing `GValue` creation and loading raw XML string values directly into the column store had no positive effect on the *Data Mart Load Benchmark*. Here, both the number of bytes per `LINEITEM` and load time increased. We did circumvent the creation of `GValue`s, but by doing so, we also disabled the existing caching mechanism in *Genus App Platform*. Thus, no strings are reused, and no conversion results are cached. We find it likely that this is the cause of increased memory usage and load time.

We see the consequences of having a `GValue` interface on the columns, which results in frequent allocations and deallocations when accessing values. The *write benchmark* shows slightly reduced write performance. However, we measure the largest performance impact on the read-intensive operations in the *Data Mart Load Benchmark*. For source measure lookup, which creates a real value array for a column using a tight loop, the performance has dropped and is now ten times slower than the `FieldValueCollection` column store due to the frequent memory allocations.

6.5 Iteration Conclusion

We conclude that primitive value column storage successfully reduces memory consumption in *Genus App Platform*. Compared to the original implementation, the bytes used per `LINEITEM` are reduced from 715 bytes to 374 bytes. Our changes have affected write performance negatively, but not by more than 15 %. We argue that the significant reduction in memory usage outweighs the slightly negative impact on write performance, and concludes that storing values as primitive data types and using a `GValue` interface is feasible.

In this iteration, we have laid an important foundation for optimizing the read-intensive operations in the *Data Mart Load Benchmark* by increasing data locality. More specifically, we have replaced the `GValue` pointers and now store values directly in the columns. Until we exploit this storage structure, source measure lookup and lookup index generation suffer severely due to memory allocation operations on each access. We investigate column operations in Chapter 8.

Our hypothesis that loading data as raw XML values into the column would reduce load time was wrong, at least for now. Circumventing `GValue`s also circumvents the built-in caching mechanism in *Genus App Platform* and results in higher memory usage and slower load times.

We have cut the application memory requirements in half without applying any form of compression. Now that we know data can be stored as primitive data types, the next step is to compress the data with techniques used by read-optimized databases.

6.5.1 Future Work

We are still keen to investigate the potential of loading XML values directly, but due to time constraints, we are unable to pursue this topic further. Future work should aim to replicate the existing caching mechanisms in *Genus App Platform*, but within the column store and without the creation of `GValue`s. Caching could, for instance, only be enabled if the data source is in a feeding state. Building on the idea of horizontal partitioning from Section 5.5.1, caching could also be used in newly created partitions, and once a partition is full, all caching structures are discarded, and the partition is optimized for read-only workloads.

Chapter 7

Iteration III: Compression

So far we have reduced the number of bytes per `LINEITEM` from 715 bytes to 374 bytes in the *Data Mart Load Benchmark*. This corresponds to a reduction of 48 %. However, we have not yet explored any compression techniques which we studied in Chapter 3. In this chapter, we investigate the effects of light-weight compression methods by implementing dictionary encoding, bitpacking, and property packing. Also, we inquire into a new technique for compressing null pointers, which we denote as *null pointer compression*.

This chapter forms the third and final iteration where the goal is to reduce memory. There is one more iteration, but this does not seek to reduce memory any further, only utilize the storage structures to increase performance for read-intense operations.

7.1 Introduction

In Section 3.3, we saw that compression not only reduces memory requirements but also comes with the benefit of increased performance. Compression is beneficial because it increases cache locality, turns processes CPU-bound from memory-bound and thus reduces the number of CPU cycles needed to process data. SIMD instructions might also increase processing throughput on compressed data. According to Abadi *et al.*, compression increases performance by a factor of two on average [16]. For compression to be beneficial the compression must be *light-weight* since the real benefit of compression can only be leveraged if the decompression effort is minimized.

One such light-weight compression technique is *dictionary encoding*, which we saw in Section 3.5. In a dictionary encoded column, each distinct value is stored once in a structure known as a dictionary, and instead of storing the actual values, keys to this dictionary are stored in the column. Not only does this technique save space, but it may increase query performance since comparisons are reduced to cheap integer operations. In this iteration, we extend our column store to support dictionary encoding and hope to leverage the benefits this technique gives us.

The second technique we implement in this iteration is bitpacking, which we studied in Section 3.4. In this compression scheme, no more bits than needed are used to store a value. This improves cache hit-rate and enables SIMD query processing. Bitpacking and dictionary encoding are commonly used in conjunction, and this what we will do in this iteration.

Last, we explore two techniques to save even more memory in *Genus App Platform*. The first is a technique we denote as *null pointer compression*. It is based on the observation that a `CompositionObject` has several properties, properties that are commonly `nil`. Thus, by moving the properties to the column store, we may save space by not allocating any values unless they are set. The second technique we apply is to compress the residual member variables in the `CompositionObject` class with a `packed record`.

7.2 Implementation

In this section, we explain how compression is implemented by creating new `FieldValueCollectionBase` subclasses for dictionary encoding, both with and without bitpacking. Then we proceed with null pointer compression and property packing.

7.2.1 Dictionary Encoding

In Section 3.5, we saw that dictionary encoding is commonly applied as a compression technique in OLAP-databases. There are many benefits with dictionary encoding

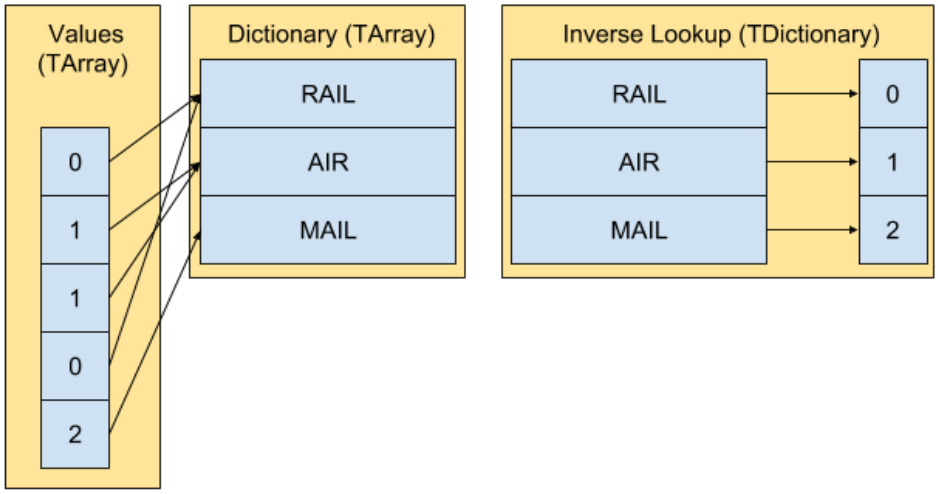


Figure 7.1: Dictionary encoded column implementation. Both value buffer and dictionary are stored using `TArray`. To ensure constant time write operations, we keep an inverse lookup from values to dictionary keys, which is implemented as a `TDictionary`. Read-only data sources may drop the inverse lookup to save memory.

in addition to a reduction in memory usage, like improved data locality and query speedup.

In a dictionary encoded column, the dictionary can be structured either for read or write performance, or both. For read performance, there must exist a lookup from a dictionary index to the actual value. For such lookup, a simple index-based list suffices, adding new keys to the end of the list as they appear. Hence, value lookups happen in constant time. However, with this implementation, a linear search to find the correct key in the dictionary is performed every time there is a write operation on the column. Here, an inverse lookup from dictionary values to indexes would be beneficial.

Since *Genus App Platform* must handle both transactional and analytical workloads, we implement a dictionary encoded column with both lookup structures, such that both read and write operations happen in constant time. We have depicted this in Figure 7.1. However, keeping both structures comes at the cost of increased memory. Thus, for read-only workloads, like *Genus Discovery*, we would like to discard the inverse lookup. We implement this in `Consolidate`; when this function is called, we deallocate the inverse lookup.

We implement a new class, `PrimitiveDictionaryFieldValueCollection`, to be our dictionary encoded column class. The class uses `TArray` for both values, or dictionary keys, and the dictionary itself. We choose this array type since the performance assesment from Appendix B indicates that `TArray` is the fastest array

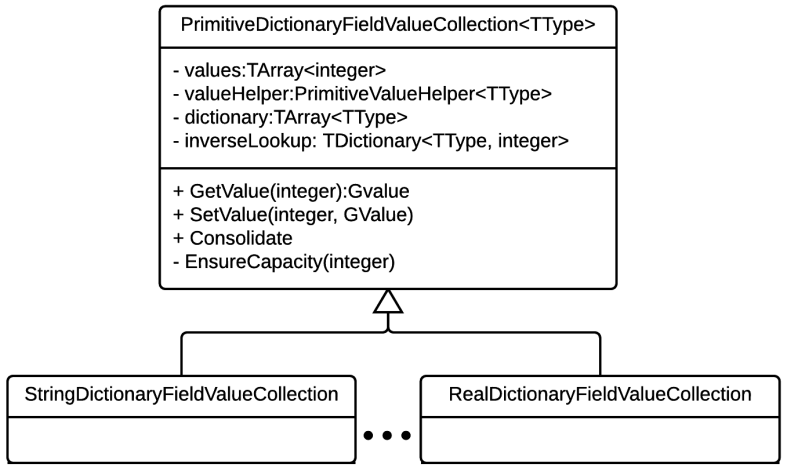


Figure 7.2: `PrimitiveDictionaryFieldValueCollection` class diagram. The class inherits from `FieldValueCollectionBase`, and has a subclass for each supported primitive type.

structure. For the inverse lookup, we use a `TDictionary`. For different primitive types, `PrimitiveDictionaryFieldValueCollection` has a subclass for each supported type which uses the same value helper as we introduced in the previous section. `PrimitiveDictionaryFieldValueCollection` extends `FieldValueCollectionBase`. A class diagram is shown in Figure 7.2.

The array with dictionary keys, or the value buffer, uses the same growth strategy as we discussed in Section 5.2.3. Like `FieldValueCollection` and `PrimitiveFieldValueCollection`, the value buffer is doubled on index overflow, and the `Consolidate` method resize the column to the exact data source size.

7.2.2 Bitpacking

Bitpacking is a compression technique where values are stored with no more bits than needed. As we saw in Section 3.5.2, dictionary encoding and bitpacking goes hand in hand. We, therefore, implement bitpacking in our dictionary encoded columns to save memory.

We implement the value buffer as an array of 8-byte chunks, where we denote each chunk as a *cell*. Unlike previous column implementations, a cell is added one at a time, instead of using a doubling strategy. We did this for simplicity reasons, and, since each cell contains multiple values, array reallocation happens less frequent than for an uncompressed column.

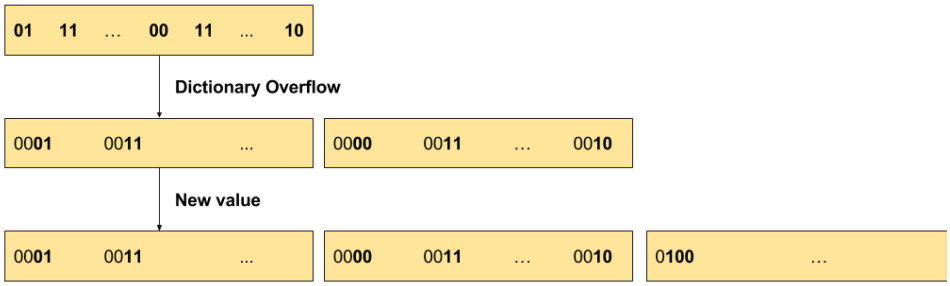


Figure 7.3: Bitpacked value buffer. First, we see a dictionary overflow, where the value buffer is rebuilt with extra padding. Second, if needed, a new cell is added to hold the new value.

One of the main challenges with bitpacking is word alignment. If keys are allowed to cross machine word boundaries, not only performance suffers, but the implementation complexity increase drastically. Therefore, we take a simpler approach, where the number of bits used to store each value must be a power of 2. As seen in Figure 7.3, a dictionary overflow causes the bits per value to go from two to four to avoid values stored across machine word boundaries. When a dictionary overflows, the columns rebuilds the entire value buffer using the new number of bits per value.

We name our bitpacked dictionary encoded column class `PrimitivePackedDictionaryFieldValueCollection`, and, like always, the class inherits from `FieldValueCollectionBase`. The value buffer is implemented using a `TArray<UInt64>`. Like the non-packed version, a `TArray` is used for the dictionary and `TDictionary` for the inverse lookup. In addition to this, the class holds some state regarding the bitpacking, and a private function `IncreaseDictionaryCapacity` which rebuilds the value buffer on dictionary overflow. The class diagram is seen in Figure 7.4.

7.2.3 Null Pointer Compression

So far, we have compressed data from the Information System and Information System Development Layers. However, several other structures belong to `CompositionObject`s which are used in the Method Engineering layer. These structures include lists of data validation and integrity errors, formatting rules, and more. Our observation is that these attributes are not fundamentally different from attributes from the other model-driven engineering layers, which means we can put these pointers into our column store too. Although one might initially think that we have only moved the problem and that the pointers now are stored in the column store instead in the composition objects, we may save memory based on the observation that these pointers are usually `nil`. In other words, these pointer columns do not need to be allocated before any values are set. We illustrate this scenario in Figure 7.5.

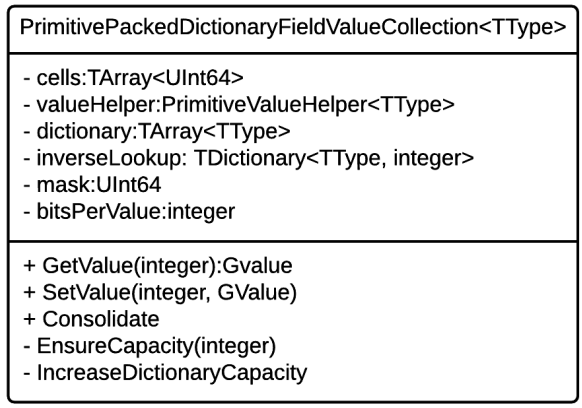


Figure 7.4: `PrimitivePackedDictionaryFieldValueCollection` class diagram. The class is very similar to `PrimitiveDictionaryFieldValueCollection`, but it has a different storage structure for its values and holds some state variables needed for bitpacking.

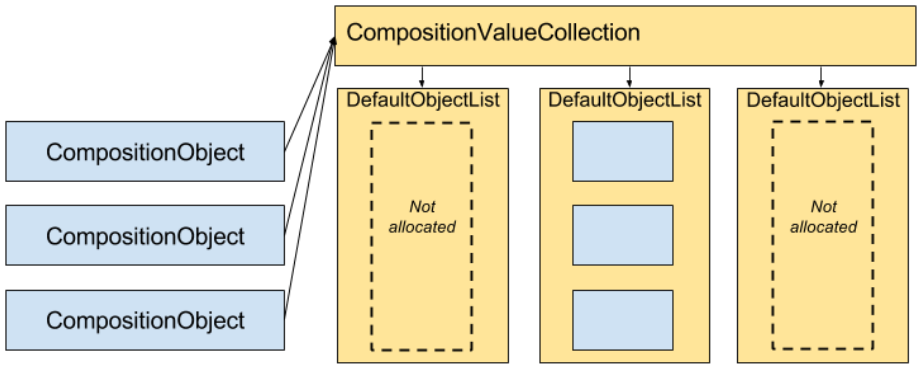


Figure 7.5: Null pointer compression. Based on the observation that most `CompositionObject` properties are normally `nil`, data can be compressed by not allocating value buffers until the properties are set.

Listing 7.1: Structure with packed data for `CompositionObject`.

```
PackedComObjData = packed record
    modifyCount,
    datasourceIndex     : integer;
    state               : EObjectState;
    filterStatus        : EFilterStatus;
end;
```

To enable null pointer compression, we implement a new generic column structure named `DefaultObjectList<TType>`. This class is an array structure that can be instantiated with any class type that will return `nil` if no values are set. If values are set, the array, or column, is allocated to accommodate these values. We extend the base column store class, `CompositionValueCollection`, with one such `DefaultObjectList` for each pointer variable in `CompositionObject`. Hence, objects must now query the column store and provide a data source index to access structures used in the method engineering layer, like validation errors and formatting rules. A total of six member variables from `CompositionObject` is moved to the column store.

7.2.4 Property Packing

In the last section, we saw how null pointers could be compressed by moving member variables in `CompositionObject` to the column store. However, the `CompositionObject` class has some value type properties that are always set, which includes a modify count, some state variables, and the much-needed data source index. Since these variables are always set, they cannot be compressed with null pointer compression.

To reduce the memory footprint used by these attributes, we compress them in a *Delphi* record type with the `packed` keyword. This keyword tells the compiler to pack data and disregard word boundaries [13]. Thus, we take all the residual properties from the `CompositionObject` class and move it to a `PackedComObjData` record type. Listing 7.1 shows the structure.

As a result of both null pointer compression and property packing, a `CompositionObject` has no member variables except for the `PackedComObjData` record and a pointer to the column store.

7.3 Results

We test the changes from this iteration with the *Data Mart Load Benchmark*, Benchmark A.1 and the *Write Benchmark*, Benchmark A.2. Like the previous

	Distinct Values	Dictionary Encoding?
LINEITEMKEY	600,000	No
LINESTATUS	3	Yes
RETURNFLAG	3	Yes
SHIPDATE	2526	Yes
QUANTITY	50	Yes
EXTENDEDPRICE	598,966	No
DISCOUNT	11	Yes
TAX	9	Yes

Table 7.1: Column selection for Benchmark A.1. Low-cardinality columns are dictionary encoded, while the others are not. The numbers are based on scaling factor SF10, where there is a total of 600,000 LINEITEM rows.

two iterations, we are curious to see how compression techniques affect memory consumption, load time, read performance, and write performance.

We read in Section 3.5 that dictionary encoding is only effective when there is a small number of distinct values compared to the number of total values. Hence, not all columns in the *Data Mart Load Benchmark* should be encoded with dictionary encoding. Therefore, we pick certain LINEITEM properties by hand to be dictionary encoded, based on the column cardinality in the LINEITEM table. Table 7.1 shows the results, and indicates that most columns benefit from dictionary encoding.

In this iteration, we test four new configurations in addition to the primitive column configuration from the previous chapter:

- *Primitive* is the configuration from the last iteration, which is column store with non-compressed, primitive values.
- *Dictionary* uses dictionary compression from Section 7.2.1. No bitpacking.
- *Dictionary /w Raw Load* is the same dictionary compression configuration as above, but with raw XML string value load as outlined in Section 6.2.3.
- *Packed Dictionary* is configuration where we use the bitpacked dictionary compression from Section 7.2.2.
- *Full Compression* configuration uses all compression techniques discussed in this chapter: Bitpacked dictionary encoding with null pointer compression and proprety packing.

Like the previous iterations, we run each benchmark three times and report the mean measurements. All results had low variance, within 15 % of the average measurement.

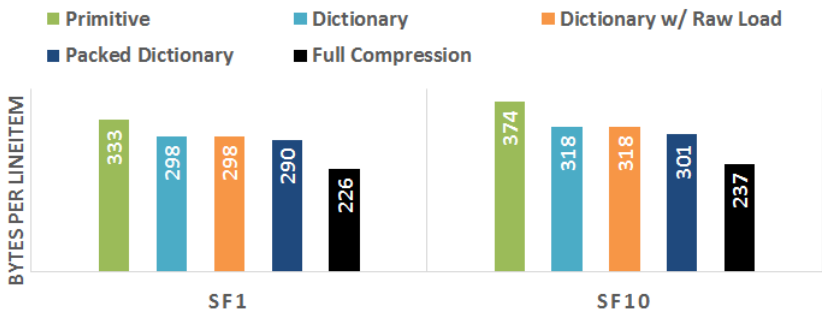


Figure 7.6: Bytes per `LINEITEM` used by all configurations tested in this iteration of data compression, Benchmark A.1 with scaling factors SF1 and SF10.

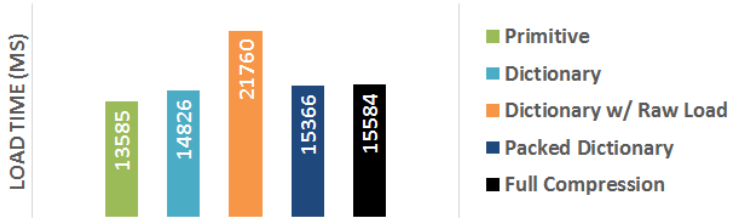


Figure 7.7: Data source load time for all compression configurations tested in this iteration, Benchmark A.1, scaling factor SF10.

7.3.1 Data Mart Load Benchmark

Benchmark A.1, the *Data Mart Load Benchmark*, is run with all configurations enumerated above. We use both scaling factors, SF1 and SF10, but if we observe the same effects with both, we only report for SF10.

We see in Figure 7.6 that the *dictionary* configuration reduce memory by 11 % and 15 % for scaling factors SF1 and SF10 respectively. Also, there is no longer a difference in memory footprint for original and raw XML value loading mechanism, a difference we observed in Section 6.3.1. Null pointer compression and property packing significantly reduce memory consumption, while bitpacking only reduces the bytes per `LINEITEM` by 5 %. For SF10, the total reduction from *primitive* to *full compression* is 137 bytes, or 37 %.

As we see in Figure 7.7, load times have increased as a result of compression. For SF10, dictionary encoding adds 1.2 seconds load time, bitpacking adds 0.5 seconds, and null pointer compression and property packing adds 0.2 seconds more. Even though load time has been increased, it is still less than the original implementation. Loading raw values into the columns increases load time to 21.8 seconds for SF10.

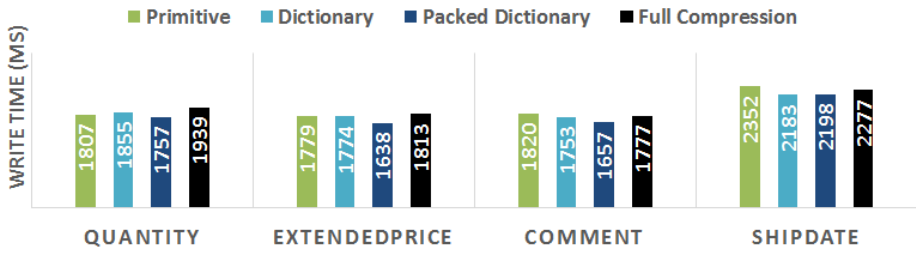


Figure 7.8: Write performance for configurations tested in this iteration, Benchmark A.2, 1000 elements.

We benchmark the read-intense operations, lookup index generation and source measure lookup, for all configurations in this iteration. However, since the measurements are not significantly different from the previous iteration, we do not present them in this chapter.

7.3.2 Write Benchmark

Write performance has not been significantly changed compared to primitive column storage, as seen it Figure 7.8. Bitpacking is the fastest of all implementations, but only by very little.

7.4 Discussion

We see that dictionary encoding is successful in compressing data, and for SF10 in the *Data Mart Load Benchmark*, memory is reduced by 15 % compared to the primitive column structure from the previous chapter. Bitpacking also contributes to reduced memory footprint, but not more than five percent. The increased compression rates come at the cost of higher load times, and it now takes 15,4 seconds to load the full data mart into memory.

Null pointer compression and property packing contribute to large memory savings and reduces the storage per `LINEITEM` from 301 bytes to 237 bytes, or 64 bytes. Since we only removed six pointers, or 48 bytes, it means property packing contributes to saving 16 bytes per element. Neither of these two techniques increases load time significantly.

None of the applied techniques has changed the write performance, which strengthens our hypothesis that write operations have much overhead associated integrity, data validation, and formula calculation. However, read-intense operations in the *Data Mart Load Benchmark* still suffer severely from the modification applied to *Genus App Platform* so far, and we believe this is caused by the `GValue` interface.

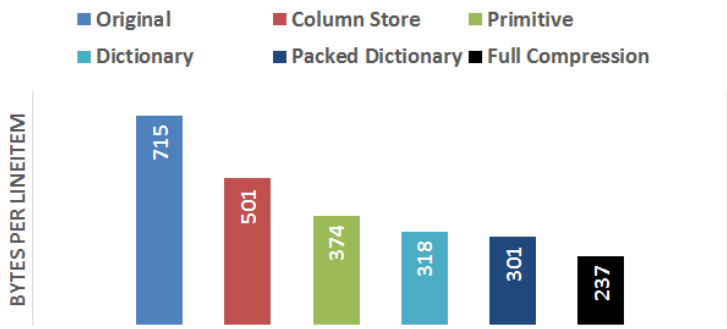


Figure 7.9: The bytes used per `LINEITEM` for different configurations tested during the course of this research, Benchmark A.1, scaling factor SF10.

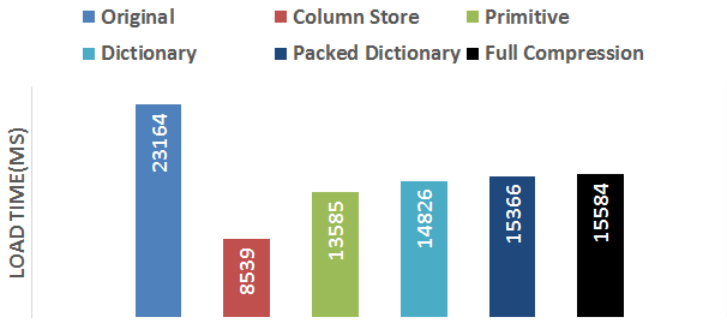


Figure 7.10: The number of milliseconds used to load the *Data Mart Load Benchmark* for different configurations tested during the course of this research, Benchmark A.1, scaling factor SF10.

If we load raw XML string values directly into a dictionary encoded column, we observe that we save the same amount of memory as the built-in caching mechanism in *Genus App Platform*. In other words, the number of bytes for dictionary encoding and dictionary encoding with raw value load is the same. However, the load time is still significantly longer if we load raw XML values. We believe this is because the caching mechanism not only re-uses data but also stores string-to-primitive conversion results. We discuss how a similar caching mechanism can be implemented in our column store in Section 7.5.1.

7.5 Iteration Conclusion

In this iteration, we have reduced the memory required by Benchmark A.1 by 37 %. This reduction is mainly caused by dictionary encoding and null pointer

compression. Compared to the original *Genus App Platform* implementation, the bytes per `LINEITEM` is reduced by 67 %. The reduction in memory usage throughout the iterations of this research illustrated in Figure 7.9. Although the introduction of primitive data types and compression has increased load time from the first `FieldValueCollection` implementation, the time it takes to load the *Data Mart Load Benchmark* is still reduced by 33 % compared to the original implementation. This is illustrated in Figure 7.10.

Concluding this iteration, we are left with two major challenges. The first challenge was discovered in the previous iteration: Read-intensive operations must be adapted to utilize the new storage format. For instance, one of source measure lookup operations in Benchmark A.1 now takes 6.9 seconds, an operation which previously took 0.3 seconds. We address this challenge in Chapter 8.

The second challenge, which is a result of dictionary compression, is to select the correct column format for different object class properties. In this iteration, we picked columns that would benefit from dictionary encoding by hand. With this approach, the *Genus App Platform* expert users need knowledge about data cardinality, which is, in our opinion, a knowledge they should not need to have. In Chapter 9, we investigate how *Genus App Platform* can pick the correct storage format without modeler intervention by using database-provided statistics.

7.5.1 Future Work

Like the last iteration, we encourage studying the built-in caching mechanism in *Genus App Platform*. We have proved that dictionary encoded columns re-use values as efficiently as the existing mechanism, but if caching is disabled, the load time is significantly higher. We believe this is caused by the caching mechanism re-using string-to-primitive value conversions. One straight-forward solution to the problem would be to add a `rawInverseLookup` to the `PrimitiveDictionaryFieldValueCollection` class which translates raw XML strings to dictionary indexes.

In our bitpacking implementation, cells are added one at a time to a column instead of doubling capacity. Although one cell normally contains more than one value, the effects of the doubling growth strategy should be investigated.

Chapter 8

Iteration IV: Column Operations

In this chapter, we investigate the potential in the data structures used by our column store. We do so by defining operations that work on entire columns at a time and show how the changes improve read-intense operations by several orders of magnitude.

This chapter contains the fourth and final iteration of our research.

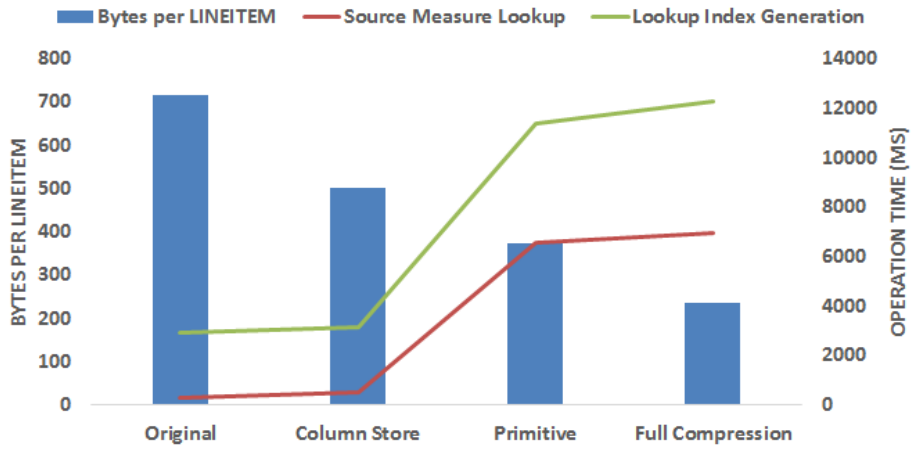


Figure 8.1: As memory footprint has gone down through the course of this research, read-intense operation performance has decreased, especially after the introduction of primitive value storage.

8.1 Introduction

So far, the techniques applied in this research give a reduction in memory usage and load time. However, as the previous iterations have pointed out, we have not yet exploited the potential the data structures in our column structure. In Figure 8.1, we see that both lookup index generation and source measure lookup performance from Benchmark A.1 has been severely slowed down, especially after the introduction of primitive value storage. In this chapter, we aim to increase the performance of these operations.

When we say *exploiting the potential in the data structures*, we refer to making operations in *Genus App Platform* more friendly for modern CPUs. As we saw in Section 3.9, independent instructions, cache hit-rates, and branch avoidance is key to achieve full CPU throughput.

Currently, read-intense operations, like source measure lookup and lookup index generation in Benchmark A.1, access values in tight loops using the `GetValue` function. Hence, every loop iteration requires a series of function calls and layers of indirection to access the value, including a dictionary lookup. More importantly, a `GValue` is allocated per iteration, and likely discarded shortly after. As we saw in Section 3.9, this is a recipe for poor performance.

We solve the above challenge by introducing *column operations*, or operations that work on all values for an object class property in a data source. In other words, we move certain functionality from outside the `GValue` interface to the column store itself. This modification enables the operations fully to utilize the storage format:

Listing 8.1: Returning the value buffer pointer in a `RealFieldValueCollection`.

```
function RealFieldValueCollection.GetDoubleArray
: TArray<double> ;
begin
  Result := values;
end;
```

Listing 8.2: Creating a double array for a dictionary encoded integer column.

```
function IntegerDictionaryFieldValueCollection.GetDoubleArray
: TArray<double>;
begin
  SetLength(Result, maxDatasourceIndex + 1);
  for i := 0 to maxDatasourceIndex do
    Result[i] := dictionary[values[i]];
end;
```

Improved cache locality, vectorized execution, and low degree of freedom. Moreover, the column operations are free of branches and function calls, which fills the CPU pipeline, enable loop unrolling, and superscalar processing.

So far, four operations that would benefit from the column store are identified in *Genus App Platform*. These include source measure lookup and join operations from Benchmark A.1, as well as the generation of identifier indexes and predicate bitmaps. In this iteration, we implement these operations and measure the performance impact using Benchmarks A.1, A.3, and A.4.

8.1.1 Source Measure Lookup

As written in Appendix A, the source measure lookup operation creates an array of double precision floating point values for a column in a data source. This array is the base unit in *Genus Discovery* and is used for calculations, grouping and aggregation. For simplicity reasons, all numeric types which work as measures in a data mart are converted to 64-bit floating point numbers, even 32-bit integers. In the primitive and uncompressed column class `RealFieldValueCollection`, we already use such array to store the data, so ideally there should be no need to generate a new array by accessing values one-by-one using `GetValue`.

We implement a new function, `GetDoubleArray` in `FieldValueCollection`. This function will, for all supported column types, return a *Delphi* array of `double`. If the column does not support the operation, `nil` is returned. For a `RealFieldValueCollection`, the method needs only to return a pointer to its internal value buffer, as seen in Listing 8.1.

Listing 8.3: `GetLookupIndex` implementation.

```
function ObjectHandleFieldValueCollection.GetLookupIndex
( identifierIndex : TDictionary<string, CompositionObject>)
: TArray<integer>;
begin
  SetLength(Result, maxDatasourceIndex + 1);
  for i := 0 to maxDatasourceIndex do
  begin
    if identifierIndex.TryGetValue(values[i], comObj) then
      Result[i] := comObj.DatasourceIndex
    else
      Result[i] := -1;
  end;
end;
```

However, since the array of double precision floating point numbers is also required on integer measures, we see the potential of creating the array inside the columns. By iterating all values and converting them to floating point numbers *within* the column, there is no branches or function invocations, nor any layers of indirection. This implementation enables loop unrolling and instruction pipelining, which in turn maximizes CPU throughput. We also implement `GetDoubleArray` in dictionary encoded columns, as seen in Listing 8.2.

8.1.2 Lookup Index

A common operation in *Business Discovery* is to create a lookup index from one table to another, which maps data source indexes from one data source to another. Another name for this operation is *join*. In databases, a join normally takes two columns as input and outputs a mapping between the columns. There are several algorithms for joining, however, as we saw in Section 3.11, the *nested loop* is most popular for in-memory databases. This algorithm creates a hashmap on the inner, or smaller, relation first and then probes this hashmap with values from the outer relation.

Genus Discovery uses a lookup index to join tables within a data mart, but as we have seen, creating such index is inefficient with the current implementation. We observe that this operation can benefit from the column store structure and avoid the unnecessary `GValue` and `GetValue` interface. Therefore, we define a new column operation, `GetLookupIndex`. Luckily, the hashmap of the inner relation is already available for us in *Genus App Platform*, in the form of an *identifier index*. We may, therefore, use this structure as an input argument in `GetLookupIndex`. The operation returns a `TArray<integer>` which contains a mapping from the data source indexes of the column to the data source indexes in the identifier index. Listing 8.3 shows the implementation.

Listing 8.4: `GetIdentifierIndex` implementation.

```
function PrimitiveFieldValueCollection<TType>.GetIdentifierIndex
( compositionObjects : CompositionObjectCollection )
: TDictionary<string, CompositionObject>;
begin
  Result := TDictionary<string, CCompositionObject>.Create;
  for comObj in compositionObjects do
  begin
    id := valueHelper.ToString(values[comObj.DatasourceIndex]);
    Result.Add(id, comObj);
  end;
end;
```

8.1.3 Identifier Index

We read about the *identifier index* in the previous section, and how it was used in the `GetLookupIndex` operation. This structure maps an identifier, typically a database primary key, to composition objects, and is used several places in *Genus App Platform*. The structure is created by the data source on-demand. We implement an operation named `GetIdentifierIndex` in `PrimitiveFieldValueCollection`. This method iterates a list of composition objects, looks up the column value based on the datasource index, and creates a mapping between the identifier and object. Since all identifiers in *Genus App Platform* are strings, the value helper class must convert the column contents before using it as a key in the dictionary. Most objects are ordered by their datasource index, which means cache locality is maximized. We see the `GetIdentifierIndex` implementation in Listing 8.4.

The `GetIdentifierIndex` is not implemented in the dictionary encoded column classes because it is not needed: If a column contains object identifiers, or primary keys, there will be nothing but unique values in the column, and such column never benefits from dictionary encoding.

8.1.4 Predicate Evaluation

Until now, we have focused on *Genus Discovery* and the *Data Mart Load Benchmark*. However, one common operation in *Genus App Platform* is to move certain objects from one data source to another based on a filter, an operation normally performed in tasks. Filters are composed of several predicates, including equal, not equal, greater than, less than, null, and other comparisons.

We believe our column store can aid this filter operation by implementing predicate evaluation in the `FieldValueCollection` interface, such that unnecessary memory allocations and layers of indirection are avoided. Hence, we introduce a new column operator, `GetBitmap`, that takes an operator and a value as input, and returns a

Listing 8.5: Creating a bitmap for the equal operator.

```
function PrimitiveFieldValueCollection<TType>.GetEqualBitmap
( value : CGValue )
: BasicBitArray;
begin
  Result := BasicBitArray.Create(maxDatasourceIndex + 1);
  nativeValue := valueHelper.ExtractValue(value);
  for i := 0 to maxDatasourceIndex do
    Result[i] := valueHelper.ValueEqual(values[i], nativeValue);
end;
```

bitmap of matching rows. An implementation of the equal predicate evaluation is shown in Listing 8.5. If a filter contains more than one predicate, they can be efficiently combined with bitwise `AND` and `OR` operations, much like bitmap indexes depicted in Figure 3.10.

In Listing 8.5, we see the implementation in the generic `PrimitiveFieldValueCollection` class, which means the value helper class must be used to evaluate the predicate. Using this class causes extra overhead in calling the `ValueEqual` function, particularly as it is a virtual function which cannot be inlined. However, all `PrimitiveFieldValueCollection` subclasses are free to override the predicate evaluation methods to use the *Delphi* comparison operators directly.

We also implement predicate evaluation in the dictionary encoded columns. Here, we first scan the dictionary and add keys satisfying the predicate to a set. Then we create the bitmap by iterating the value buffer and probing the set of keys.

8.2 Results

In this section, we test the performance impact of the column operations we introduced in this iteration. For this, we use a combination of Benchmarks A.1, A.3, and A.4, depending on the operation. Most results are presented on a logarithmic scale.

8.2.1 Source Measure Lookup

We use Benchmark A.1 with scaling factor SF10 to measure the performance impact of using `GetDoubleArray` in the source measure lookup operation. As seen in Figure 8.2, the time it takes to perform the operation is reduced by more than one order of magnitude from the original implementation, and more than two orders of magnitude from the compressed implementation. The non-compressed `EXTENDEDPRICE` column

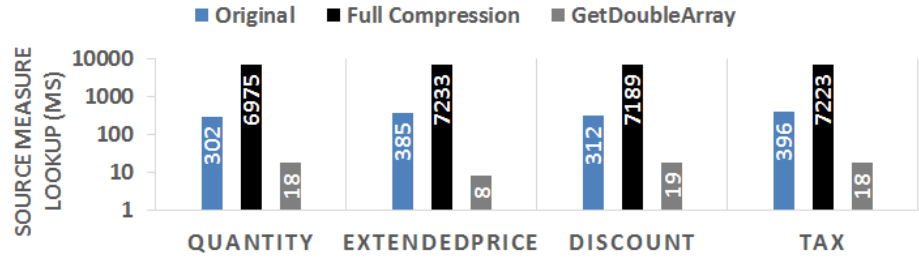


Figure 8.2: Source measure lookup time for original, compressed column store, and compressed column store with `GetDoubleArray` column operation, Benchmark A.1, scaling factor SF10. Logarithmic scale.

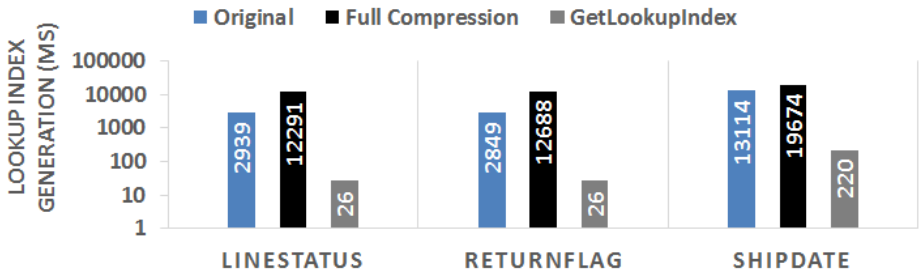


Figure 8.3: Lookup index generation time for original, compressed column store, and compressed column store with `GetLookupIndex` column operation, Benchmark A.1, scaling factor SF10. Logarithmic scale.

yields the best performance result, where the source measure lookup operation now takes 8 ms for 600,000 elements.

8.2.2 Lookup Index

Like as for source measure lookup, we used the *Data Mart Load Benchmark* SF10 to investigate the performance benefits of the `GetLookupIndex` operation. Figure 8.3 shows the result of this benchmark. The time it takes to perform the join operation is reduced by two orders of magnitude compared to the original implementation, and almost three orders of magnitude compared to the compressed implementation.

8.2.3 Identifier Index

To test the performance of the identifier index operation, we use a benchmark which we have not yet employed in this research; Benchmark A.4. This benchmark is

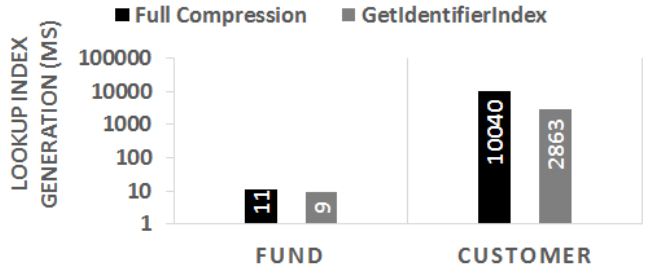


Figure 8.4: Lookup index generation time for compressed column store and compressed column store with `GetIdentifierIndex` column operation, Benchmark A.4. Logarithmic scale.

similar to *Data Mart Load Benchmark*, but this benchmark contains a join between two tables with 200,000 rows and 400,000 rows respectively. Thus, by using this test, we expect to make the effects of `GetIdentifierIndex` more apparent. Here, we only compare operation time with the *full compression* configuration.

Figure 8.4 shows the results of Benchmark A.4. The time it takes to join `Customer Balance Daily` with `Customer` has been reduced with almost four times. The smaller join, between `Fund` and `Customer Balance Daily`, goes from 11 ms to 9 ms, but this is likely to be caused by inaccuracy in the measurements.

8.2.4 Predicate Evaluation

To test predicate evaluation operators, we use the *Filter Benchmark*, Benchmark A.3. This benchmark moves data from one data source to another based on a filter. The results are presented in Figure 8.5.

For all high-selectivity predicate evaluations, the new predicate operator function increases performance, but is, in general, slower than the original implementation. Low selectivity predicates, like `EXTENDEDPRICE < 10000` (selects 44) and `EXTENDEDPRICE = 42995.94` (selects one) are faster than the initial implementation, where the latter is three orders of magnitude faster.

8.3 Discussion

The benefits of utilizing the data structure available in the column store and making column operations that are friendly to modern CPUs are significant. Column operations have resulted in both read-intensive operations in Benchmark A.1 have had their performance increased by one or two orders of magnitude compared to the original *Genus App Platform* implementation. Performance on large joins is

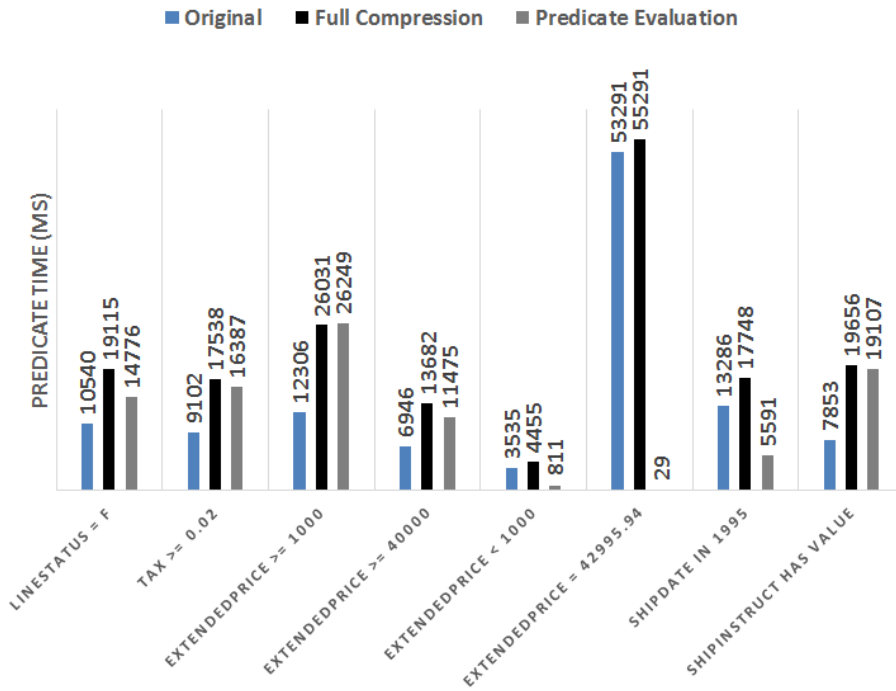


Figure 8.5: Predicate evaluation results for original, compressed column store, and compressed column store with predicate evaluation operators, Benchmark A.4.

even further improved by the `GetIdentifierIndex` operation, where the join time Benchmark A.4 is reduced by 75 %.

In Figure 8.2, we observe that source measure lookup on `EXTENDEDPRICE` is twice as fast as for the other three columns. This is because `QUANTITY`, `DISCOUNT`, and `TAX` columns are dictionary encoded. `EXTENDEDPRICE` only needs to pass its array pointer to the lookup operation, while the other columns must allocate and populate this double array using the dictionary. However, this operation is still fast; it takes approximately 20 ms to populate 600,000 elements. For `QUANTITY` column, values must be converted from integers to floating point numbers, but this does not affect performance.

In the column store, the predicate evaluation operators from Section 8.2.4 have speeded up Benchmark A.3 if using a compressed column store. However, most of these operations are still slower than the original *Genus App Platform* implementation. We believe this is because *Genus App Platform* is not designed for the new storage format; data is moved from one data source to another using rows, or `CompositionFieldValueCollection`s, filled with `GValue`s. However, the potential in column predicate evaluation is apparent on low-selectivity filters: The time it takes to evaluate `EXTENDEDPRICE = 42995.94` is reduced from 53.3 seconds to 29 ms, almost 2000 times faster.

8.4 Iteration Conclusion

There is much to gain by exploiting the potential in the data structures in our column store. By introducing *column operations*, we see that several read-intensive operations are speeded up by several orders of magnitude. We see the largest performance impact in this iteration on an equality predicate evaluation in the *Load Benchmark*, where the time it takes to perform the operation has been reduced by approximately 2000 times.

The four column operations described in this chapter serve as a proof-of-concept on the potential in column storage; they are just the tip of the iceberg. Other parts of *Genus App Platform* should be rewritten to use these operations and new column store functionality should be defined. We elaborate in Section 8.4.1.

This chapter concludes our design and experiment part of the research. We have reduced memory by 67 %, increased load time by 36 %, and the performance for most operations benchmarked in this research has been increased. Some operations are still slower than the original *Genus App Platform* implementation, like those in Benchmark A.3. However, we have laid some important groundwork for restoring and improving the performance of these operations.

The next chapter discusses other interesting topics investigated in this research, which is column selection and UTF-8 strings.

8.4.1 Future Work

Future work should aim to identify more column operators that can increase performance in *Genus App Platform*. As we have seen throughout the research iterations, read-intensive operations suffer from the `GValue` interface, and if no attention is paid to such operations, performance decrease by as much as one order of magnitude. These read-intensive operations, however, are also the operations that have the potential to benefit the most from columns. Hence, future work should identify such operations, implement them in the `FieldValueCollection` class, and apply them in *Genus App Platform*.

For instance, we have seen the potential in the predicate evaluation operation, where Benchmark A.3 yielded 2000 times performance improvement for a low-selectivity predicate. However, since *Genus App Platform* have not been adapted to the column store, moving data from one data source to another generally takes longer time than the original implementation. We believe this operation can be significantly faster by creating an `ExtractFromBitmap` function in `CompositionValueCollection`; a function that takes a bitmap as input and returns a new `CompositionValueCollection` with a subset of the data based on the bitmap. This way, composite queries can evaluate predicates using the `GetBitmap`, combine them with `AND` and `OR` operations, and extract data using the new `ExtractFromBitmap` method. We link such approach to the *late materialization principle* from Section 3.10.3, where no rows are materialized until needed, or in this case; never.

Chapter 9

Other Topics

In this chapter, we cover some other topics investigated in this research. First, we see how memory usage is reduced by applying UTF-8 instead of UTF-16 encoding to strings in the column store. Second, we discuss how database statistics can be used to select the correct column implementation automatically.

The topics in this chapter yield some interesting results which are worthwhile to discuss. However, the techniques are not tested thoroughly enough to draw any general conclusions.

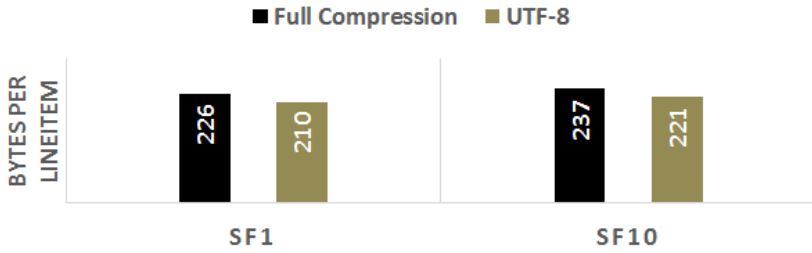


Figure 9.1: Bytes per `LINEITEM` used by with and without UTF-8 string encoding, Benchmark A.1 with scaling factors SF1 and SF10.

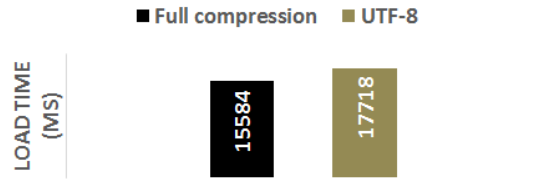


Figure 9.2: Data source load time for full compression with and without UTF-8 string encoding, Benchmark A.1, scaling factor SF10.

9.1 UTF-8

According to the *Delphi* documentation, the built-in `string` type uses UTF-16, because the Win32 platform uses this format [12]. UTF-16 is a Unicode encoding which uses two or four bytes per character. However, another encoding, UTF-8, exists for the Unicode charset, which encodes the 128 ASCII characters with one byte only.

Our observation is that most strings used in *Genus App Platform* only use these 128 ASCII characters, which means a transition from UTF-16 to UTF-8 could potentially reduce the memory requirements for strings in half. Therefore, we extend our primitive column subclasses with one more type: *Delphi*'s `UTF8String`. Then, we see how load time and memory usage are affected.

9.1.1 Test Results

Figure 9.1 shows that the number of bytes used per `LINEITEM` is reduced by 7 % for both scaling factors. However, as seen in Figure 9.2, load times are increased by roughly 14 %.

9.1.2 Discussion, Conclusion, and Future Work

We observe that UTF-8 encoding reduces memory consumption slightly. However, this comes at the cost of increased load time. We believe that the increased load time is caused by the fact that *Genus App Platform* consequently uses UTF-16 encoded strings throughout the application source code. Hence, when data is loaded into a data source, there are many conversions between UTF-8 and UTF-16 that causes performance to slow down. The results from Benchmark A.1 might also be the tip of the iceberg; the entire platform might be slowed down due to encoding conversions between `string` and `UTF8String`. Also, UTF-16 is the default encoding in *Delphi* because it is supported by the underlying system. In other words, all communication with the OS must convert all strings to UTF-16.

If this topic is to be pursued further, all strings in *Genus App Platform* must be changed to the `UTF8String` type. Only then can a true performance assessment be made, and to see if the reduced memory consumption outweighs the benefits of having strings stored as the same format as the underlying operating system.

9.2 Column Selection

During the research, we have implemented different column formats with the `FieldValueCollection` interface. Until now, we have selected by hand which properties in an object class uses which storage format. However, this disregards an important aspect of *Model-Driven Development*: The modelers should be shielded from the underlying complexities of data storage and should not need to know anything about how the data is best represented internally. In other words, *Genus App Platform* should ideally decide which `FieldValueCollection` implementation which is better suited for different use cases and object class properties.

First, according to our benchmarks, we concluded in Chapter 5 that the column store did not hurt performance. Hence, we may safely use `CompositionValueCollection` for representing data. Second, we concluded in Chapter 6 that discarding `GValues` and instead storing data as primitive data types significantly reduces memory consumption and does not affect write performance by much. We may, therefore, use the `PrimitiveFieldValueCollection` class on all supported types. Third, we observed no negative performance impact on the bitpacked dictionary compared to the non-packed dictionary.

We saw a significant space saving by choosing dictionary columns on low and medium cardinality rows. However, for columns with high cardinality, a dictionary is not beneficial due to increased memory usage and load time.

Thus, it all sums down three implementations:

- `PrimitiveFieldValueCollection` for high-cardinality columns; object properties with many distinct values.

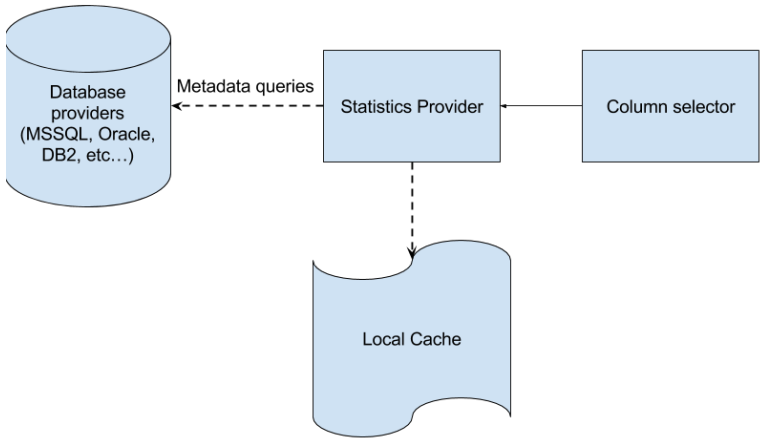


Figure 9.3: Using statistics to select the proper storage format. A statistics provider service will, for a given data descriptor, return relevant statistics, preferably the number of rows, number of unique values, and average data element size. The module might cache, restructure and calculate some of the statistics internally, as the different database providers have different interfaces.

- `PrimitivePackedDictionaryFieldValueCollection` for low- and medium-cardinality columns; object properties with few distinct values.
- `FieldValueCollection` for object properties with types not supported in the primitive value columns.

To check whether a primitive type is supported is trivial, however deciding column cardinality is challenging. To overcome this challenge, we implement a database statistics module in *Genus App Platform*. This module will, for a given data descriptor, return relevant statistics for a column, including the number of rows, number of unique values, and average data element size. As seen in Figure 9.3, the module queries the database infrastructure for metadata, and caches them locally. For instance, *Oracle Database* provides an SQL interface to enable clients to access statistics used by the query optimizer. The acquired data is then utilized by the column selector to pick between dictionary encoded or regular columns.

9.2.1 Data Source Filters

One of the major limitations in our statistics provider is that it provides statistics for all values for a given object class property. However, it is quite common to add a filter to the data source on tasks and analyses, for instance on an analysis of data for a given date range. What this means is that even though our statistics provider picks the correct storage format for all data in a column, the storage format might not be optimal for filtered subsets of the data.

One solution to this problem would be to increase the granularity of the statistics. Instead of only using data descriptors, the statistics module should also accept a data source with corresponding filters as input. This extra input would increase the precision of the statistics, but it has certain challenges. For example, the database infrastructure does not provide statistics for data subsets. Thus, the statistics would need to be calculated on-demand by slow queries like `COUNT` and `COUNT DISTINCT`, which in turn increase load time.

The problem from the above paragraph can be overcome if *Genus App Platform* first loads the entire column into a non-compressed column and generate statistics for this column in a low-priority thread. Then, on consecutive loads, these statistics are used to make a qualified decision whether to use dictionary encoding or not. Although this might not be as accurate as using statistics from the database infrastructure directly, it is likely to be more precise than making decisions based on unfiltered data.

We can build on this idea even further. Instead of having a low-priority thread to calculate database statistics, we can have a low-priority thread to restructure the data into the most suitable format. In other words, we load data directly into regular columns. Then, we start a low-priority thread which will analyze the data and rebuild columns to dictionary encoding, but only if they have low- or medium-cardinality. Once rebuilt, the low-priority deallocates the regular columns and replace them with the dictionary encoded equivalents. This technique has the benefit of increased load performance, as loading values into a dictionary encoded column is more time consuming than a regular column, and it does not rely on statistic support in the underlying database infrastructure.

Another, but less precise way to overcome the challenges with filtered data sources, is to utilize predictability in columns and extrapolate the statistics for a given filter. For instance:

- Low-cardinality columns tend to have the same amount of unique values no matter how many more values are added (typically gender, age, etc.).
- High-cardinality columns tend to have a linear relationship between the number of values and the number of unique values.
- Some column types are predictable. For instance, the number of distinct values in a code domain is constant, and some data, like timestamps, is ordered.

The challenge here is to exploit the predictability to make qualified decisions on column selection.

9.2.2 Special Cases

There are cases where storage format should completely disregard database statistics. For instance, *Genus Discovery* uses an array of floating point numbers as a base unit for calculation. Hence, to avoid data duplication, a column used as a measure

Property	Column implementation
LINEITEMKEY	PrimitiveString
LINESTATUS	PrimitiveDictionaryObjectHandle
RETURNFLAG	PrimitiveDictionaryObjectHandle
SHIPDATE	PrimitiveDictionaryCalendarTime
QUANTITY	PrimitiveDictionaryInteger
EXTENDEDPRICE	PrimitiveReal
DISCOUNT	PrimitiveReal
TAX	PrimitiveReal

Table 9.1: Column selection for Benchmark A.1 using the statistics module.

in an analysis should never be dictionary encoded, since *Genus Discovery* does not yet work on dictionary encoded values.

9.2.3 Precision of Statistics

So far, we have assumed that statistics provided by the database infrastructure or by *Genus App Platform* are only estimates. Let us imagine that they are not, but rather exact numbers. First and foremost, the growth strategy from Section 5.2.3 would be surplus as we could allocate the correct size for the value buffers immediately. Thus, we avoid unnecessary reallocations and data movement. Second, bitpacked columns would benefit from these exact values: The number of bits needed to represent each value is known before populating the column, which means we avoid value buffer rebuilding.

9.2.4 Results

We test the statistics module with Benchmark A.1 and present the results in Table 9.1. We observe that the column implementation is the same as the hand-picked implementations from Chapter 7, except for `DISCOUNT` and `TAX`. These two properties are used as measures in the benchmark and should not, according to the reasoning in Section 9.2.2, be put in a dictionary encoded column.

9.2.5 Discussion, Conclusion, and Future Work

We see a definitive potential in using database statistics and other clever tricks to select the correct column. However, several challenges must be overcome to utilize this potential fully. For instance, data sources with filters impose a challenge. Second, to support multiple database backends, vendor-specific changes must be made to the database adapter in *Genus App Platform* to enable database metadata queries. The column selection should not only rely on database statistics; one

should also consider what the data is used for. For example, measures in *Genus Discovery* should not be dictionary encoded.

We believe this topic should be pursued further. The primary motivation behind *Model-Driven Engineering* is to close the gap between the problem and the implementation. Having the platform select the best-suited storage format without modeler intervention is a step towards closing the gap. Future work should not only focus on database statistics for correct data format selection but also see how the data is used in *Genus App Platform*. For write-intensive operations, perhaps the original `FieldValueCollection` from Chapter 5 is better suited than a dictionary encoded, bitpacked column with primitive data types.

Part III

Discussion and Conclusion

Chapter 10

Discussion

In this chapter, we discuss our findings in a broader perspective. We study the implications of our research for *Genus App Platform*, *Model-Driven Engineering* in general, and for traditional programming languages. We also provide a section on limitations and critics.

10.1 Discussion

In this section, we discuss the implications of the findings in Part II in a broader perspective.

10.1.1 Implications for Genus App Platform

So far in this research, we have reduced memory consumption, improved load time, and increased operation performance for our benchmarks. Even though these benchmarks only test a small subset of *Genus App Platform* functionality, we believe the advantages will appear throughout the platform. Even without dictionary compression and the column selector functionality discussed in Section 9.2, storing data in columns, either as `GValue`s or primitive data types, is a good idea. However, we also expect to see functionality in *Genus App Platform* slowed down by our column store implementation. Reduced performance is likely to be observed on operations that are not adapted to the new storage format; operations that heavily rely on `GValue`s and row storage, like the filter operation in Benchmark A.3.

In the thesis introduction, we said *Genus AS* have focused on keeping the source code readable and maintainable. However, as we have seen, this emphasis have also resulted in performance issues in *Genus App Platform*. With the introduction of `CompositionValueCollection`, the code is more slightly complicated and less readable. Still, the `CompositionObject` class is intact and works exactly as before, and it has a `GValue` interface. In other words, the application works like before, the `GetValue` and `SetValue` on `CompositionObject` have only got a new implementation. In addition, the interface to the column store, or `CompositionValueCollection`, is clean, and requires `CompositionFieldDescriptor` or data source index, or both, to access data.

We believe further development in *Genus App Platform* may proceeds as before. However, two things should be kept in mind. First, developers must be aware the cost of `GValue` allocation when getting values from a `CompositionObject`. Thus, read-intensive operations like source measure lookup in the *Data Mart Load Benchmark* perform poorly if no special attention is given. Second, developers should identify new operations or optimizations that benefit from the column data structure. These two things are intertwined; read-intense operations that allocate `GValue`s frequently are also likely to benefit from column store exploitation.

10.1.2 Implications for Model-Driven Engineering

This research does not provide an extensive overview of *Model-Driven Development* tools and their particular challenges, but we believe that their ability to handle and analyze large datasets is commonly outperformed by hand-made, traditional

Figure 10.1: The *Oracle Database* dual format. Data is stored as both rows for transactional performance, and columns for analytical performance. (Adapted from [45])

programs. Our research shows that by applying database technology used in read-optimized databases, *Model-Driven Engineering* tools can gain this ability. We have shown that internal data representation is key for low memory usage and high performance, also in *Model-Driven Development*. Clever data structures increase *Model-Driven Engineering* versatility and enable new functionality, for instance *Business Intelligence* and *Business Discovery*.

We have also discovered that changing internal data representation only affects the method engineering layer in *Model-Driven Engineering*if implemented correctly. By using database statistics or other tricks, modelers in the information system development layer do not need to be affected by the underlying storage technology. In other words, object classes and other data models are still expressed such that they are close to the business domain, while the underlying *Model-Driven Engineering* infrastructure optimizes data representation. This decoupling is different from a traditional programming language, where class definitions serve as both business problem abstractions and data containers.

There exist many different *Model-Driven Development* tools with various approaches to the *Model-Driven Development* methodology. Some are simple code-generation tools, while other provides an entire infrastructure. For layered systems that loads and process data in-memory, like *Genus App Platform*, there is much potential in data structure optimization: Different tasks in the platform should use whichever format is better-suited for that particular task. In Section 3.14, we read abut mixed workloads and that consistency, correctness, and data freshness can be sacrificed for performance. We also know that row-wise storage normally outperforms columns on transactional workloads. *Model-Driven Engineering* tools may have it all. For instance, *Business Discovery* funcionality should load data as immutable, read-optimized, and compressed columns while write-intense should use structures that store data in rows. This is similar to the *Oracle Database* dual format, depicted in Figure 10.1.

Listing 10.1: Decorators in Python that specifies storage engine.

```
class Person:
    @storage(type='column')
    first_name = None

    @storage(type='column')
    last_name = None

    @storage(type='dictionary')
    gender = None
```

10.1.3 Implications for Traditional Programming Languages

In *Model-Driven Engineering*, we have changed the internal data representation without affecting the modelers in the information system development layer. However, as far as we are aware, traditional programming languages do not offer the same functionality, as classes serve the role of *both data structures and business abstractions*. Although developers are free to move class attributes into database-inspired structures, like columns, this degrades code readability: Data gets decoupled from the objects they belong to, and class definitions no longer contain available properties. The implementation of the internal data structures is also tedious, which increase application development time.

We believe even traditional programming languages can benefit from the techniques we have studied in this research. For instance, the Python programming language implements the *decorator pattern* which dynamically changes the source code of a function or attribute [10]. We believe decorators can be used to configure the underlying storage engine for class properties. Listing 10.1 exemplifies a `Person` class where first and last name are stored in regular columns, while a dictionary encoded column stores the person's gender.

In Section 7.3.1, we saw how null pointer compression significantly reduced the number of bytes needed to store a `CompositionObject`. We believe traditional programming languages could leverage this technique by not allocating buffers for pointers before they are set.

10.2 Limitations and Critics

Our research has several limitations, one of them is benchmark coverage. We only tested a limited subset of *Genus App Platform*. There might exist other parts of the platform with unexpected effects, both regarding performance and correctness. We tested write performance with Benchmark A.2 but this benchmark is simple and does not test all edge cases. Neither did we use any benchmarks to test for

correctness. However, the column store implementation is already in use internally in *Genus AS*, and so far no correctness or performance issues have been reported. We are, therefore, confident that most parts of *Genus App Platform* work as before, and only a few components are affected negatively by our changes.

Another limitation is how we executed the benchmarks. Due to time constraints, we run all benchmarks only three times. Ideally, the tests should have been run more times to obtain a statistical foundation in our results. Also, all benchmarks were run with the compiler set to debug mode. We are unaware of the differences between debug and production mode in the *Delphi* compiler, but we believe it is likely that production mode applies more optimizations, like loop unrolling and function inlining. However, we would like to point out three things:

1. All changes have been tested in debug mode, so we expect the relative changes to be the same in production.
2. All measurements had low variance.
3. The relative changes we have measured have been substantial; up to three orders of magnitude at the most.

In other words, despite poor statistical foundation and benchmarks run in debug mode, we believe conclusions can be drawn from the benchmark results.

One major limitation is that we have based our entire research on *Genus App Platform*, but many *Model-Driven Engineering* supporting infrastructures are fundamentally different from this platform. Thus, our changes might not apply to all *Model-Driven Development* tools. There might exist systems which solve the problem of handling large datasets using other techniques, and systems that do not require this functionality at all. Nevertheless, we believe our modifications to *Genus App Platform* serves as a proof-of-concept for *Model-Driven Development* tools challenged with these issues by showing the importance of internal data representation.

This research has focused on read-optimized databases because it was for read-intense operations and analytical workloads *Genus App Platform* had the largest performance issues. Thus, we disregarded technologies used in write-optimized OLTP databases, like row-storage and indexes. We are confident that *Genus App Platform* could benefit from technology used in these databases as well. Still, *Genus AS* experiences no performance issues on transactional workloads. Also, data update operations are dominated by integrity and validity checks, formula calculation, and more, which makes such operations less sensitive to the underlying data representation.

Chapter 11

Conclusion

This chapter concludes this research and points at interesting directions for future work.

11.1 Conclusion

In this research, we set out to answer the following research question:

> **RQ1: How can technology used by in-memory, read-optimized databases improve *Genus App Platform*'s ability to handle and analyze large datasets, and what can *Model-Driven Engineering*, in general, learn from database technology?**

We conclude that techniques used in read-optimized, in-memory databases have improved *Genus App Platform*'s ability to handle and analyze large amounts of data. First, column storage with primitive data types, dictionary encoding, bitpacking, and null pointer compression significantly reduce memory consumption. In the *Data Mart Load Benchmark*, the number of bytes needed per `LINEITEM` has been lowered by 67 %. Second, load time has improved by 36 %. Regarding performance, we see that *Genus App Platform* benefits from the same principles as OLAP databases: By avoiding branches and improving cache locality, CPU throughput is maximized, and join, lookup, and filter operations are one, two, and even three orders of magnitude faster than the original implementation. Write performance is not slowed down as a result of the new internal data representation, although certain read-intense operations which are not adapted to the new structures are.

Our research has, by using the modifications in *Genus App Platform* as a proof-of-concept, introduced evidence that *Model-Driven Engineering* benefits from database technology. Due to the wide variety of approaches used by *Model-Driven Development* tools, we cannot draw any general conclusions whether the exact techniques we applied to *Genus App Platform* can be employed by all *Model-Driven Engineering* supporting infrastructures, but we have shown the importance of internal data representation. Thus, *Model-Driven Engineering* tools which need support for analysis of large datasets, like *Business Discovery* functionality, should look at the database technology for inspiration.

All of our changes lie in the method engineering layer. The tool optimizes internal data structure without affecting modelers in the information system layer. Hence, expert users can continue to focus on the business problem, and not worry about the underlying implementation. We pointed out that similar techniques can be applied to traditional object-oriented languages, where the problem is that classes serve the role as both business abstractions and data containers.

11.2 Future work

In this research, we have only implemented and discovered a few methods used by read-optimized, in-memory databases. Future work should aim to identify more techniques that can be used in the context of *Model-Driven Engineering*, also methods used in OLTP. Such techniques include parallelism and scaling, delta stores, indexes, result caching, code generation, query optimization, and more.

Database technology has been a major research field for decades, so there should be plenty of techniques that can be applied to *Model-Driven Development* supporting infrastructures.

In Section 9.2, we saw how *Genus App Platform* decides whether to apply dictionary encoding to a column or not by using database statistics and in Section 10.1.2 we claimed that data should be represented in the best-suited format for different application tasks. Future work should pursue this topic. *Model-Driven Engineering* has a major advantage over general-purpose databases: *Model-Driven Development* tools has extensive knowledge of data interpretation, and how and when the data is used. Hence, this knowledge, combined with database statistics, should be exploited to make intelligent decisions on which storage format that should be used for different parts of the application.

Last, but not least, this research has focused on what *Model-Driven Engineering* can learn from database technology. What if we turn the problem around and ask ourselves: What can database technology learn from *Model-Driven Engineering*? Future work should investigate if techniques, terminology, and technology well known in *Model-Driven Engineering* applies to databases as well. For instance, could databases benefit from having an extra layer of data interpretation on top of data types? Should database schemas contain not only data type definitions and relations, but also information on how and when the data is used?

11.2.1 Genus App Platform

Above we discussed general topics for future work. However, the research has resulted in an extensive list of ideas specific for *Genus App Platform* which we enumerate in this section. Some of the ideas are fetched from Chapters 5-9, while others are new.

Genus Discovery Specific Optimizations

The original motivation for this research was to improve analytic capabilities in *Genus App Platform* by studying how the *Business Discovery* functionality in the platform could be improved. However, we have only implicitly improved *Genus Discovery* by reducing memory usage and load time. *Genus Discovery* still works on arrays of floating points exclusively and does not exploit the data structures implemented in this research. Future work should improve and modify *Genus Discovery* to work such that it can operate on dictionary encoded columns directly, and also apply state-of-the-art techniques for joining, grouping, and aggregation.

Horizontal Partitioning and Delta Store

In Chapter 5 and Chapter 6, we briefly discussed horizontal partitioning. Future work should investigate the effects of horizontal partitioning in *Genus App Platform*. `CompositionObject` and `CompositionValueCollection` could be extended with a partition index. This way *Genus App Platform* could leverage all benefits of this technique, which is metadata pruning, parallelization, and better support for data skew. At the same time, the effects of a delta store could be investigated: Newly created partitions could be optimized for data inserts and updates, while full partitions could be stored in read-optimized, immutable structures.

Dynamic Storage Formats

In the current implementation, *Genus App Platform* tries to make a qualified decision whether to apply dictionary encoding to a column or not. However, as we saw in Chapter 7, dictionary encoded columns takes a longer time to load. Hence, it might be beneficial to load all data into non-compressed columns first, and only then make a decision whether data should be rebuilt as a dictionary encoded column. Then the restructure operation could happen in a separate, low-priority thread. Future work should investigate the effects of such dynamic storage formats. *Genus App Platform* should also try to deallocate data in data sources that are no longer in use. For instance, the entire column can be dropped after a floating point array is extracted in the source measure lookup operation.

Server State

An important principle in *Genus App Platform*, is the *stateless* application backed, *Genus App Services*. Although this has several benefits regarding simplicity and scalability, it has certain disadvantages. One disadvantage, is that clients talk with the database infrastructure directly, mostly using XML, which *Genus App Platform* compress the data after it has traversed the network. Ideally, the compression should happen in the database infrastructure, or in *Genus App Services*. Hence, future work should investigate the effects of applying state to the server. Not only would server state reduce network traffic, but it would also improve client load time if the data is already structured as columns. This could also relax memory requirement for clients as more data processing and aggregations can be performed on the server.

Bibliography

[1] Calling procedures and functions (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Calling_Procedures_and_Functions_(Delphi). Accessed: 2016-4-21.

[2] Classes and objects (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Classes_and_Objects_(Delphi). Accessed: 2016-5-23.

[3] Columnstore indexes described. https://msdn.microsoft.com/en-us/library/gg492088.aspx. Accessed: 2015-10-28.

[4] Expressions (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Expressions_(Delphi). Accessed: 2016-4-21.

[5] Genus. https://www.genus.no/?PageKey=91ff59b3-216e-4c89-8bd8-d9d366cc8479. Accessed: 2016-4-7.

[6] Indexes. http://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.htm. Accessed: 2015-11-26.

[7] Language overview - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Language_Overview. Accessed: 2016-4-21.

[8] Memory management - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Memory_Management. Accessed: 2016-4-21.

[9] Procedures and functions (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Procedures_and_Functions_(Delphi). Accessed: 2016-4-21.

[10] PythonDecorators - python wiki. https://wiki.python.org/moin/PythonDecorators. Accessed: 2016-4-12.

[11] Simple types (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Simple_Types_(Delphi). Accessed: 2016-4-21.

[12] String types (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/String_Types_(Delphi). Accessed: 2016-4-21.

[13] Structured types (delphi) - RAD studio. http://docwiki.embarcadero.com/RADStudio/Berlin/en/Structured_Types_(Delphi). Accessed: 2016-4-21.

[14] The underlying technology of QlikView — DBMS 2 : DataBase management system services. http://www.dbms2.com/2010/06/12/the-underlying-technology-of-qlikview/. Accessed: 2015-10-28.

[15] Understanding tableau data extracts. http://www.tableau.com/about/blog/2014/7/understanding-tableau-data-extracts-part1. Accessed: 2015-11-25.

[16] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 967–980. ACM, 9 June 2008.

[17] Russell Lincoln Ackoff. Ackoff's best: his classic writings on management. John Wiley & Sons, 1999.

[18] Siah Hwee Ang. Research Design for Business & Management. Sage, 2014.

[19] C Atkinson and T Kuhne. Model-driven development: a metamodeling foundation. IEEE Softw., 20(5):36–41, 2003.

[20] R Barber, G Lohman, I Pandis, V Raman, R Sidle, G Attaluri, N Chainani, S Lightstone, and D Sharpe. Memory-efficient hash joins. Proceedings VLDB Endowment, 8(4):353–364, December 2014.

[21] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, and Frederick Ho. Business analytics in (a) blink. IEEE, 2012.

[22] Truls A Bjørklund. Column Stores versus Search Engines and Applications to Search in Social Networks. PhD thesis, Norwegian University of Science and Technology, January 2011.

[23] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining query execution. In CIDR, volume 5, pages 225–237, 2005.

[24] Peter Alexander Boncz. Monet; a next-Generation DBMS Kernel For Query-Intensive Applications. 2002.

[25] Kjell Bratbergsengen. Storing and Management of Large Data Volumes. 27 March 2015.

[26] Kalen Delaney. SQL server In-Memory OLTP internals overview. Technical report, March 2014.

[27] EXASOL. EXASolution technical whitepaper: A peek under the hood. Technical report, 2014.

[28] Franz Farber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Muller Hannes Rauhe, and Jonathan Dees Sap Ag. The SAP HANA database – an architecture overview. IEEE, 2012.

[29] Martin Faust, Pedro Flemming, David Schwalb, and Hasso Plattner. Partitioned Bit-Packed vectors for In-Memory-Column-Stores. In Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics, page 2. ACM, 31 August 2015.

[30] Alberto Ferrari. VertiPaq vs ColumnStore comparison. August 2012.

[31] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In 2007 Future of Software Engineering, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Genus AS. Genus app platform - evolve, scale, adapt. Technical report, 31 March 2016.

[33] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. In-memory performance for big data. Proceedings VLDB Endowment, 8(1):37–48, September 2014.

[34] Martin Henkel and Janis Stirna. Pondering on the key functionality of model driven development tools: The case of mendix. In Perspectives in Business Informatics Research, Lecture Notes in Business Information Processing, pages 146–160. Springer Berlin Heidelberg, 29 September 2010.

[35] Allison L Holloway and David J DeWitt. Read-optimized databases, in depth. Proceedings VLDB Endowment, 1(1):502–513, 1 August 2008.

[36] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. Proceedings VLDB Endowment, 1(1):622–634, August 2008.

[37] Neelesh Kamkolkar, Ellie Fields, and Marc Rueter. Tableau for the enterprise: An overview for IT. Technical report, 2015.

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

[39] T Lahiri, S Chavan, M Colgan, D Das, A Ganesh, M Gleeson, S Hase, A Holloway, J Kamp, Teck-Hua Lee, J Loaiza, N Macnaughton, V Marwah, N Mukherjee, A Mullick, S Muthulingam, V Raja, M Roth, E Soylemez, and M Zait. Oracle database In-Memory: A dual format in-memory database. In Data Engineering (ICDE), 2015 IEEE 31st International Conference on, pages 1253–1258, April 2015.

[40] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-Store 7 years later. 21 August 2012.

[41] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L Price, Srikumar

Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL server column stores. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pages 1159–1168, New York, NY, USA, 2013. ACM.

[42] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding up queries in column stores. In Data Warehousing and Knowledge Discovery, Lecture Notes in Computer Science, pages 117–129. Springer Berlin Heidelberg, 30 August 2010.

[43] Mauri Leppänen. An integrated framework for meta modeling. In Advances in Databases and Information Systems, Lecture Notes in Computer Science, pages 141–154. Springer Berlin Heidelberg, 3 September 2006.

[44] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. Proceedings VLDB Endowment, 4(9):539–550, 1 June 2011.

[45] Oracle. Oracle database In-Memory. Oracle White Paper, 07 2015.

[46] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. Proceedings VLDB Endowment, 8(12):1816–1827, August 2015.

[47] Hasso Plattner. The impact of columnar In-Memory databases on enterprise systems. Proceedings VLDB Endowment, September 2014.

[48] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. Scaling up mixed workloads: A battle of data freshness, flexibility, and scheduling. In Performance Characterization and Benchmarking. Traditional to Big Data, Lecture Notes in Computer Science, pages 97–112. Springer International Publishing, 1 September 2014.

[49] Qlik. The associative experience: QlikView's overwhelming advantage. Technical report, October 2010.

[50] Qlik. QlikView architecture and system resource usage. Technical report, April 2011.

[51] Qlik. What makes QlikView unique. Technical report, January 2014.

[52] QlikTech International AB. Reference Manual, April 2011.

[53] V Raman, G Swart, Lin Qiao, F Reiss, V Dialani, D Kossmann, I Narang, and R Sidle. Constant-Time query processing. In Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, pages 60–69, April 2008.

[54] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2

with BLU acceleration: So much more than just a column store. Proceedings VLDB Endowment, 6(11):1080–1091, August 2013.

[55] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. Efficient transaction processing for hyrise in mixed workload environments. INTERNATIONAL WORKSHOP ON IN-MEMORY DATA MANAGEMENT AND ANALYTICS, 2014.

[56] Bran Selic. The pragmatics of model-driven development. IEEE Softw., 20(5):19, 2003.

[57] Stoimen. Computer algorithms: Data compression with run-length encoding. `http://www.stoimen.com/blog/2012/01/09/computer-algorithms-data-compression-with-run-length-encoding/`. Accessed: 2015-11-25.

[58] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented DBMS. In Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, pages 553–564, Trondheim, Norway, 2005. VLDB Endowment.

[59] Transaction Processing Performance Council (TPC). Transaction Processing Performance Council, 13 November 2014.

[60] Victor Lavrenko. Indexing 6: delta encoding (compression), 20 January 2014.

[61] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. ACM SIGMOD Record, 29(3):55–67, 1 September 2000.

[62] Wikipedia contributors. Business intelligence. `https://en.wikipedia.org/w/index.php?title=Business_intelligence&oldid=691203830`, 18 November 2015. Accessed: 2015-11-22.

[63] Wikipedia contributors. Call stack. `https://en.wikipedia.org/w/index.php?title=Call_stack&oldid=683600635`, 1 October 2015. Accessed: 2015-12-4.

[64] Wikipedia contributors. Database. `https://en.wikipedia.org/w/index.php?title=Database&oldid=693373678`, 2 December 2015. Accessed: 2015-12-3.

[65] Wikipedia contributors. Delta encoding. `https://en.wikipedia.org/w/index.php?title=Delta_encoding&oldid=684583969`, 7 October 2015. Accessed: 2015-11-25.

[66] Wikipedia contributors. Loop unrolling. `https://en.wikipedia.org/w/index.php?title=Loop_unrolling&oldid=678495551`, 29 August 2015. Accessed: 2015-12-4.

[67] Wikipedia contributors. Online analytical processing. https://en.wikipedia.org/w/index.php?title=Online_analytical_processing&oldid=693548894, 3 December 2015. Accessed: 2015-12-3.

[68] Wikipedia contributors. Online transaction processing. https://en.wikipedia.org/w/index.php?title=Online_transaction_processing&oldid=691859924, 22 November 2015. Accessed: 2015-12-3.

[69] Wikipedia contributors. Short-circuit evaluation. https://en.wikipedia.org/w/index.php?title=Short-circuit_evaluation&oldid=691686570, 21 November 2015. Accessed: 2015-12-4.

[70] Wikipedia contributors. SIMD. https://en.wikipedia.org/w/index.php?title=SIMD&oldid=690570287, 14 November 2015. Accessed: 2015-12-2.

[71] Wikipedia contributors. Superscalar processor. https://en.wikipedia.org/w/index.php?title=Superscalar_processor&oldid=687770458, 27 October 2015. Accessed: 2015-12-3.

[72] Wikipedia contributors. Delphi (programming language). https://en.wikipedia.org/w/index.php?title=Delphi_(programming_language)&oldid=714796109, 11 April 2016. Accessed: 2016-4-21.

[73] Thomas Willhalm, Ismail Oukidyz, Ingo Mullery, and Franz Faerber. Vectorizing database column scans with complex predicates. ADMS, 2013.

[74] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proceedings VLDB Endowment, 2(1):385–394, 1 August 2009.

[75] Roberto V Zicari. In-memory database systems. interview with steve graves, McObject. http://www.odbms.org/blog/2012/03/in-memory-database-systems-interview-with-steve-graves-mcobject/, 16 March 2012. Accessed: 2015-9-21.

[76] M Zukowski, S Heman, N Nes, and P Boncz. Super-Scalar RAM-CPU cache compression. In Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on, pages 59–59, April 2006.

Appendices

Appendix A

Benchmarks

In this appendix, we elaborate on the benchmarks run to measure performance in this research. There are five benchmarks in total, all aiming to test different parts of *Genus App Platform* where memory usage or performance might have been affected. The last benchmark, Benchmark A.4, is based on real customer data from one of *Genus AS*' customers.

We measure memory usage for various states in the program by querying the built-in *FastMM* memory manager. To measure the entire *Genus App Platform* memory footprint, including the .NET layer and the graphical user interface, we use the built-in memory manager in the *Windows* operating system.

We run all benchmarks in *Genus Desktop* in *Delphi* debugger mode. Test output and debugging statements are extracted through *Trace Log*, a logging tool developed by *Genus AS* for their applications. Before running any benchmark, *Genus Desktop* and the debugger was restarted such that the effects of caching are minimized.

All tests are run on a Windows 10 Dell workstation with a 64-bit, 2.40 GHz Intel®Xeon®E5620 processor, and 8.00 GB RAM. The *Delphi* version used is 23.0.21418.4207.

A.1 Data Mart Load Benchmark

In this benchmark, we measure analytical performance in *Genus App Platform* using using a benchmark inspired by and based on the Q1 in the TPC-H Benchmark. As we saw in Section 3.15, the TPC-H benchmark is made for decision support workloads and consists of a suite of business oriented ad-hoc queries. The schema is composed of eight separate tables, as seen in Figure 3.23. The table columns have a variety of different data types, including integers, floating point numbers, variable and fixed width string, identifiers, and booleans.

```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
        sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

Figure A.1: TPC-H Q1 query. (Adapted from [59])

Q1, or the *Pricing Summary Report Query*, provides a summary of all items shipped within a date range [59]. It aggregates price with and without discount, and with and without tax, as well as average quantity, price, and discount. The results are grouped by `RETURNFLAG` and `LINESTATUS`. The original query is shown in Figure A.1.

To accommodate our needs for this research, as well as adapt the query to fit *Genus App Platform* and *Genus Discovery*, we perform some modifications. First, `RETURNFLAG`, `LINESTATUS`, and `SHIPDATE` are extracted as code domains such that they can be used as dimensions in the data mart. Also, we remove unnecessary table columns irrelevant to the query from the mart. The resulting data mart is shown in Figure A.2. Second, since *Genus Discovery* does not work with an SQL interface, but rather with a user interface with graphs and boxes, we translated the query to a reporting dashboard with bar charts and selection boxes. The resulting user interface is shown in Figure A.3.

We populate the database with the `dbgen` tool provided by the *Transaction Processing Performance Council*. However, since the default scaling factor for the TPC-H benchmark is too large for *Genus App Platform*, we created two new dataset sizes, SF1 and SF10, with 1 % and 10 % of the original data respectively. The data was downscaled by keeping every 10th or 100th row in `LINEITEM` and removing all orders, suppliers, customers, and parts that no longer were connected to any

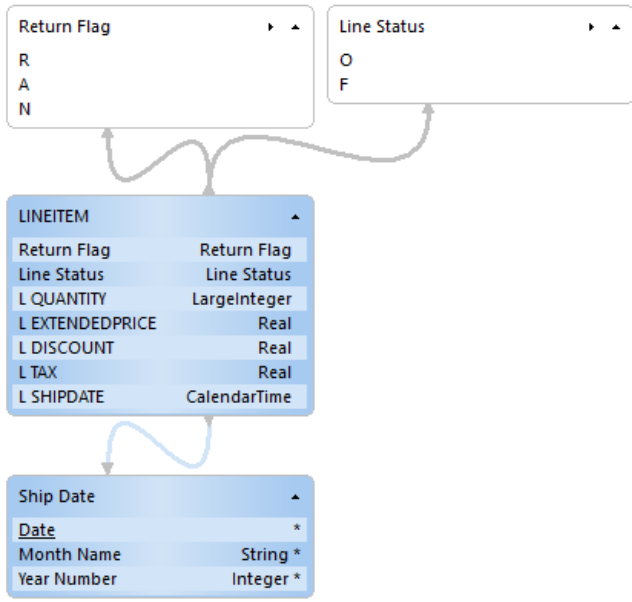


Figure A.2: Data mart for the *Data Mart Load Benchmark*.

elements in this table.

We focus our testing on time and memory footprint using to load this data mart and corresponding GUI into *Genus Discovery* rather than measuring the time it takes to answer each query. The reason for this is that we have not looked at *Genus Discovery* specific optimizations in this research, only modifications that this component can use reduce memory footprint and decrease load time. Loading a data mart takes the majority of the time anyway, and once the data mart is loaded, the application answers queries efficiently.

A.1.1 Test Input

Two different scaling factors: SF1 with 60,000 `LINEITEM` rows, and SF10 with 600,000 `LINEITEM` rows.

A.1.2 Test Output

We list the benchmark's output below. The output is recorded in the time between the shortcut button for the *Genus Discovery* instance is pressed until the user interface (Figure A.3) is displayed on the screen.

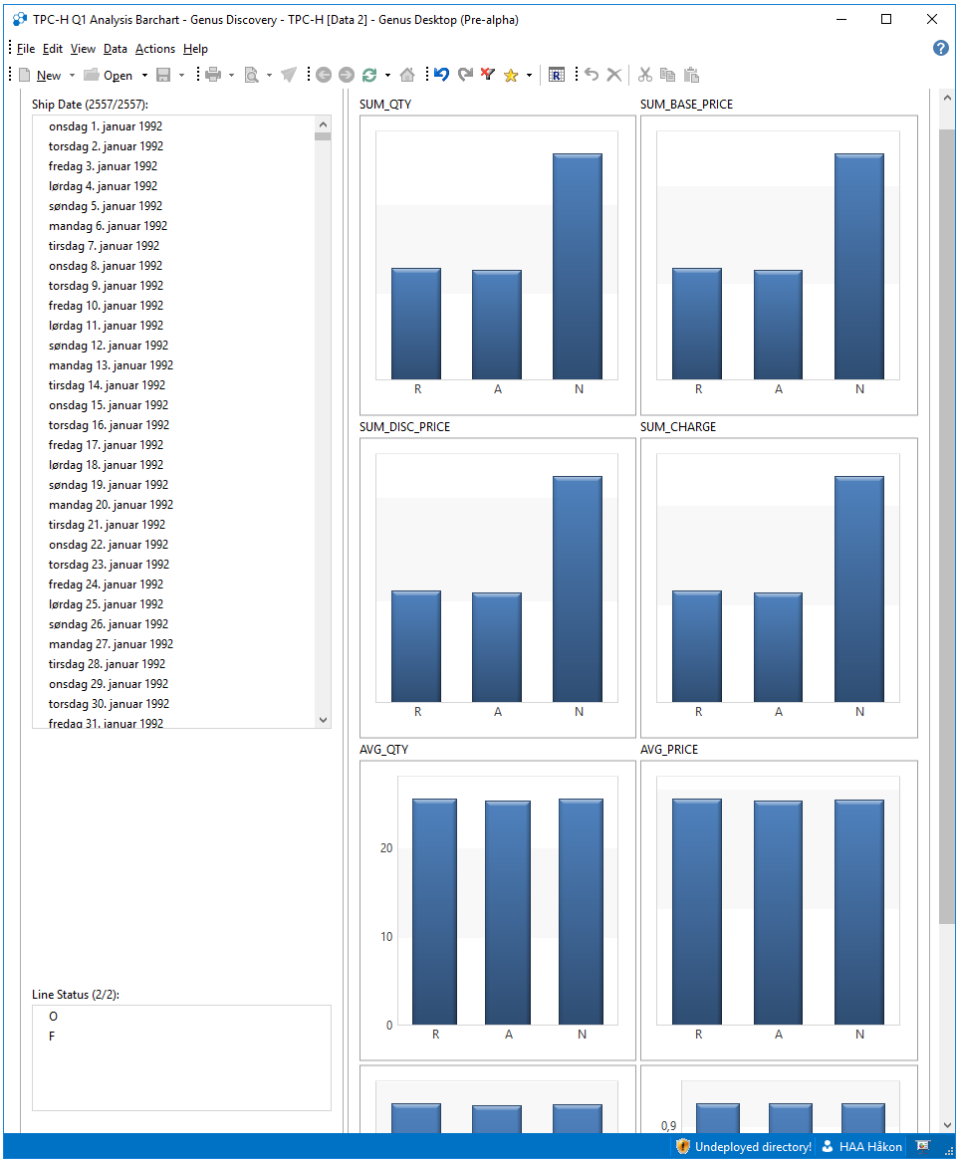


Figure A.3: *Data Mart Load Benchmark* user interface in *Genus Discovery*.

Bytes Per LINEITEM The number of bytes allocated per `LINEITEM` in the memory manager is reported. We measure memory usage before and after the `LINEITEM` data source load and then divide by the number of loaded items in the data source.

Load Time The benchmark reports the time it takes for the `LINEITEM` data source to be loaded. The output disregards the overhead in creating composition objects and parsing XML from the underlying database infrastructure. Measurements are reported in milliseconds.

Lookup Index Generation (Join) This output shows how many milliseconds it takes to create lookup indexes between tables in the data mart. This operation is considered as a join operation. A join is performed on all relations in the mart, which is the `LINESTATUS`, `RETURNFLAG`, and `SHIPDATE` in the `LINEITEM` table. A measurement is made for every relation. Creating a lookup for `SHIPDATE` takes longer than the two others because it is a larger join and dates are harder to parse.

Source Measure Lookup In *Genus Discovery*, all calculations and aggregations are performed on arrays of double precision floating point values. Hence, after all data for a given data source has been loaded, such arrays are extracted for all measures in the data mart. In the *Data Mart Load Benchmark*, such structures are created for integer column `QUANTITY` and floating point columns `EXTENDEDPRICE`, `DISCOUNT`, and `TAX`. For each column, the number of milliseconds used to create this lookup is measured.

Total Memory Footprint We measure the total memory footprint of the *Data Mart Load Benchmark Genus Discovery* application by using the Windows 10 memory manager such that all overhead in the .NET and user interface layers is included. The *In Use* memory metric is recorded before the *Genus Discovery* instance is loaded and when the UI is shown to the user. Since the measurement includes more than our application and is affected by background tasks and other running program, the total memory footprint output is not very accurate. However, it serves as a confirmation on other memory measurements, as well as an indication that total application memory consumption is reduced.

A.2 Write Benchmark

We test write performance in *Genus App Platform* to see that it is not negatively affected by the changes we make in this research. We base our benchmark on the `LINEITEM` table in the TPC-H benchmark. The main reason for using this table is the availability of different data types, which we see in Figure A.4.

LINEITEM Table Layout

Column Name	Datatype Requirements	Comment
L_ORDERKEY	identifier	Foreign key reference to O_ORDERKEY
L_PARTKEY	identifier	Foreign key reference to P_PARTKEY, Compound Foreign Key Reference to (PS_PARTKEY, PS_SUPPKEY) with L_SUPPKEY
L_SUPPKEY	identifier	Foreign key reference to S_SUPPKEY, Compound Foreign key reference to (PS_PARTKEY, PS_SUPPKEY) with L_PARTKEY
L_LINENUMBER	integer	
L_QUANTITY	decimal	
L_EXTENDEDPRICE	decimal	
L_DISCOUNT	decimal	
L_TAX	decimal	
L_RETURNFLAG	fixed text, size 1	
L_LINESTATUS	fixed text, size 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPINSTRUCT	fixed text, size 25	
L_SHIPMODE	fixed text, size 10	
L_COMMENT	variable text size 44	

Compound Primary Key: L_ORDERKEY, L_LINENUMBER

Figure A.4: `LINEITEM` table layout. (Adapted from [59])

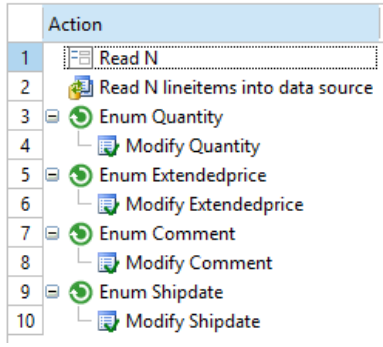


Figure A.5: The task definition for the Write Benchmark. There is one loop per attribute.

The benchmark reads N items into a data source, for all the elements, the following fields are modified:

- `QUANTITY = QUANTITY + 1` to measure integer performance.
- `EXTENDEDPRICE = EXTENDEDPRICE + 1.0` to measure floating point performance.
- `COMMENT = COMMENT + "1"` to measure string performance.
- `SHIPDATE = SHIPDATE PLUS 1 DAYS` to measure date performance.

We create one loop per property, such that there is a total of four loops in this task. The full task definition is seen in Figure A.5.

A.2.1 Test Input

The number of `LINEITEM`s loaded into the data source and modified, which is specified in a dialog box before the task is run. For all tests performed in this research, we use 1000 elements as input.

A.2.2 Test Output

For each property modified, the task reports the time it takes to enumerate the entire data source and modify the property. In addition to total time, the average time per modification is listed as well.

Predicate	Selects (of 100,000)
LINESTATUS = F	49,864
TAX >= 0.02	66,851
EXTENDEDPRICE >= 1000	99,956
EXTENDEDPRICE >= 40000	45,408
EXTENDEDPRICE < 1000	44
EXTENDEDPRICE = 42995.94	1
SHIPDATE IN 1995	15,435
SHIPINSTRUCT_NULL HAS VALUE	74,991

Table A.1: Predicates in the *Filter Benchmark*. The second column indicated selectivity by showing the number of selected rows (of 100,000 total).

A.3 Filter Benchmark

To test *Genus App Platform*'s ability to move data from one data source to another based on a filter, we establish a benchmark based on the `LINEITEM` table from the TPC-H benchmark. As for the *Write Benchmark*, this test is used for the various data types available.

A list of predicates which compose the benchmark, and objects that satisfy each predicate are moved from the original data source to a filtered data source. We select the predicates to span a variety of different data types, selectivities, and operators. Since there are no `null` columns in `LINEITEM`, we create a new column `SHIPINSTRUCT_NULL` and assign `null` to all cells with the value *NONE*. We list the different predicates and respective selectivities in Table A.1.

A.3.1 Test Input

The number of `LINEITEM`s loaded into the original data source. We only use 100,000 as input in this research.

A.3.2 Test Output

The timing result for the filter operation for each predicate; the time it takes to move the objects from the original data source to the filtered data source. Reported in milliseconds.

A.4 Data Mart Load Benchmark II

We run the same benchmark as Benchmark A.1, but with a subset of data from one of *Genus AS*' customers. The data mart for the *Genus Discovery* analysis is

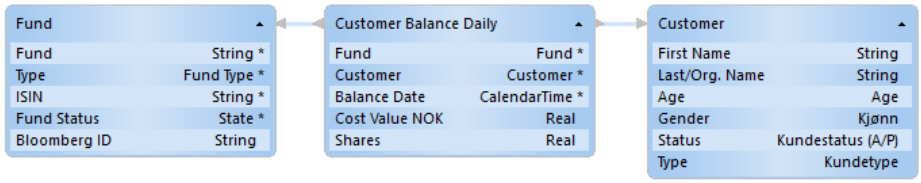


Figure A.6: Data mart for the *Data Mart Load Benchmark II*.

composed of three tables; `FUND`, `CUSTOMER`, and `CUSTOMER BALANCE DAILY`, with 40, 442,715, and 203,519 elements respectively. `CUSTOMER BALANCE DAILY` links the two other tables by indicating how many shares each customer has in each fund. Figure A.6 shows the data mart definition for the benchmark.

The reason why we choose to introduce another data mart load benchmark is that this benchmark contains a larger join than the *Data Mart Load Benchmark*. In *Data Mart Load Benchmark II*, the lookup index generation operation must create a lookup index between `CUSTOMER` and `CUSTOMER BALANCE DAILY`, which have 442,715 and 203,519 elements respectively.

A.4.1 Test Input

None.

A.4.2 Test Output

For each table, the bytes per element and load time is reported. In addition to this, the benchmark prints the time it takes to perform each of the two joins in the data mart.

Appendix B

Array and Data Type Performance in Delphi

B.1 Introduction

In this microbenchmark, we test the performance and memory usage of different numeric data types in *Delphi*. We also study the performance of various array types found in the programming language and the standard library. We will assess `integer`, `Int64`, and `double` data types and `array of *`, `TArray`, and `TList` array types.

The motivation for this microbenchmark is to find the most efficient data types for column storage in *Genus App Platform*, both regarding memory usage and performance.

B.2 Test Setup

For this microbenchmark, we write a small *Delphi* program that fills an array structure of choice with numbers from 0 to 9 n times. Then, we iterate the numbers and accumulate the sum into an accumulator variable. Listing B.1 shows the program.

We measure memory usage before and after allocation and array population. Then, we measure the average time it takes to sum all numbers in the array. We choose $n = 4000000$ for this benchmark.

The test is run on a Windows 10 Dell workstation with a 64-bit, 2.40 GHz Intel®Xeon®E5620 processor, and 8.00 GB RAM. The *Delphi* version used is

Listing B.1: Array Performance Benchmark

```
SetLength(l_arrNumbers, 4000000 * 10);
for l_i := 1 to 4000000 do
  for l_j := 0 to 9 do
    l_arrNumbers[l_i * 10 + l_j] := l_j;
l_oMeasurer.After('List', 4000000*10);

for l_i := 1 to 10 do
begin
  l_dSum := 0.0;
  l_oTimer.Start('Sum ' + IntToStr(l_i));
  for l_j := 0 to (4000000 * 10) - 1 do
    l_dSum := l_dSum + l_arrNumbers[l_j];
  l_oTimer.Stop('Sum ' + IntToStr(l_i));
end;
```

	integer	Int64	double
array of	4.00 bytes	8.00 bytes	8.00 bytes
TArray	4.00 bytes	8.00 bytes	8.00 bytes
TList	4.00 bytes	8.00 bytes	8.00 bytes

Table B.1: The memory used per element for different data types (columns) and array types (rows).

23.0.21418.4207. We query the built-in *FastMM* memory manager to measure memory.

B.3 Results

As seen in Table B.1, all array implementations use equally many bytes per element, which means any overhead associated with these array types is negligible. As expected, the `integer` type takes 4 bytes, while `Int64` and `double` takes 8 bytes.

The results of the performance test are shown in Table B.2. We see that `array of`

	integer	Int64	double
array of	131 ms	134 ms	157 ms
TArray	130 ms	134 ms	157 ms
TList	215 ms	221 ms	237 ms

Table B.2: The average time it takes to sum all numbers in the benchmark for different data types (columns) and array types (rows).

and `TArray` types are equal regarding performance. `TList` is 1.5 - 2 times slower than these types. Also, integer addition is faster than floating point addition. There is not a significant difference in performance between 32-bit and 64-bit integer types.

B.4 Conclusion

We conclude that either `array of` or `TArray` is better suited to store column data than `TList`. In addition, the `integer` or `Int64` data type should be used over `double` whenever possible.