OPTIMIZED DATA PLACEMENT FOR REAL-TIME ANALYTICS ON SEMI-STRUCTURED DATA

by

Sahel Sharifymoghaddam

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2015 by Sahel Sharifymoghaddam
Abstract

Optimized Data Placement for Real-time Analytics on Semi-structured Data

Sahel Sharifymoghaddam
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2015

In this dissertation, we address the emerging demand for extending traditional relational support to semi-structured data, and for real-time data analytics. Semi-structured data (e.g., JSON) poses new challenges with large numbers of attributes, sparse attributes and dynamic changes in both workload and data set. In this context, we design, implement and evaluate a novel technique for vertical partitioning of data for optimizing performance of main memory databases. Our partitioning algorithm enables JSON data storage in main memory relational databases by intelligently decomposing JSON objects into different tables. It also adapts to changes in workload and dataset by dynamically refining the current layout.

Using the Nobench dataset for JSON data, we outperform Argo, a state-of-the-art data model that maps JSON into relational databases, by 15-30x. We also outperform Hyrise, a state-of-the-art vertical partitioning algorithm, by 1.6x for hybrid workloads with sparse data and perform as well as Hyrise for other workloads.
Acknowledgements

I would like to thank my supervisor, Professor Cristiana Amza, for her knowledge, patient guide, and support. It was my privilege and honor to work under Professor Amza’s supervision.

I would also like to thank my examination committee members: Professor Nick Koudas, Professor Ashvin Goel, and Professor Elvino S. Sousa for their valuable comments and feedback.

I am truly grateful to my colleagues and lab mates: Dr. Mihai Burcea, Dr. Ali Hashemi, Dr. Jin Chen, and Alan Lu for their knowledge, help, and support. I would especially thank Alan Lu for his close collaboration along this research project.

Last but not least, I would like to thank my family, especially my siblings Lina and Sayeh Sharifymoghaddam, for their understanding, love, and support.
Contents

1 Introduction 1

2 Background 6

2.1 Traditional Database Workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  2.1.1 OLTP Workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  2.1.2 OLAP Workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2 Main Memory Databases: Advantages and Challenges . . . . . . . . . . . . . . . 6

2.3 Partitioning and Data Layouts . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.3.1 Horizontal Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.3.2 Vertical Partitioning: Column-based, Row-based, and Hybrid Layouts . . 8

2.4 Optimal Data Layouts for Different Workload Types . . . . . . . . . . . . . . . . 9
  2.4.1 OLTP Workload . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.4.2 OLAP Workload . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  2.4.3 Hybrid Workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.5 Workload and Data Characteristics . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.5.1 Semi-structured Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.5.2 NoBench: A JSON Micro-Benchmark . . . . . . . . . . . . . . . . . . . . . 14

2.6 ARGO: A Data Structure for JSON in Relational Format Used for Comparison . 15

2.7 Hyrise: A Main Memory Database with Vertically Partitioned Layout Used for
  Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3 Design and Implementation 18

3.1 High Level Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 In-Memory Data Structure ........................................... 20

3.3 Query Execution ......................................................... 22
   3.3.1 Table creation ..................................................... 22
   3.3.2 Data Insertion ..................................................... 23
   3.3.3 Selection and Projection Queries ................................ 26
      3.3.3.1 Table-first Traversing Order .............................. 27
      3.3.3.2 Record-first Traversing Order ......................... 28
   3.3.4 Inner Join ........................................................ 29
   3.3.5 Repartitioning ...................................................... 29

4 Vertical Partitioning Algorithm ........................................ 33
   4.1 Vertical Partitioning Examples .................................... 33
   4.2 Impact of Sparse Attributes on Partitioning ...................... 35
   4.3 Partitioning Overview .............................................. 38
   4.4 Search Algorithm .................................................... 39
      4.4.1 Algorithm Termination ...................................... 41
      4.4.2 Search Space Pruning ....................................... 41
   4.5 Cost Model .......................................................... 41
   4.6 Initial Partitioning .................................................. 45

5 Evaluation Results ....................................................... 47
   5.1 Methodology and Configuration Set up ............................ 47
   5.2 Comparison with Argo, Row-based and Column-based Layouts .... 48
      5.2.1 Low Conflict ................................................... 48
      5.2.2 Medium Conflict .............................................. 50
      5.2.3 High Conflict ............................................... 52
   5.3 Comparison with Hyrise ............................................. 55
      5.3.1 Methodology ................................................... 55
      5.3.2 Record-First Traversing .................................... 56
      5.3.3 Table-first Traversing ...................................... 60
   5.4 Sensitivity to Alpha Parameter .................................... 62
## 6 Related Work

6.1 Vertical Partitioning in Relational Databases ........................................ 64
6.2 Other Data Layouts .................................................................................. 65
6.3 Main Memory Relational Databases ......................................................... 65
6.4 NoSQL Databases .................................................................................. 67
6.5 Mapping JSON to Relational Databases ................................................ 69

## 7 Conclusion and Future Work

Bibliography ......................................................................................... 74

Appendices ............................................................................................ 80

A CPU Configuration ............................................................................ 81
Chapter 1

Introduction

The ever-increasing integration of the Internet in more and more aspects of daily life has led to a corresponding increase in the demand for real-time analytics that provide up-to-the-minute reporting on various types of dynamic and rapidly changing data such as application log files or social network data. As an example, Facebook’s answer to their need of a framework that allows them to perform real-time analysis on various events occurring on Facebook pages, is a system named Scuba, developed in 2013 [1]. Scuba is a main memory database system used for real-time performance monitoring, trend analysis, and pattern mining purposes. In general, it is not feasible to try to process and analyze amounts of data experienced by companies such as Facebook in real-time, but applications like Facebook’s Scuba system can be used with sampled data that fits in the total memory of a cluster of servers. This type of performance requirements, originally encountered by Internet companies such as Amazon, Google, Facebook and Twitter, is becoming a challenge for many other companies who now also want to provide meaningful real-time services [2].

The need for real-time analytics has made main memory databases a lot more attractive lately. The increasing popularity of main memory database systems as a solution to real-time analytics is mainly due to 1) the much-faster response time of main memory compared to disk access [3], and 2) the continuing trend for faster and cheaper main memory, with significant size increases over the past decade [2]. These features help main memory databases enable real-time analysis on increasing amount of data [2].

Real-time analytics is not the only major need that modern database systems have; another
one is the ability to effectively support mixed OLTP (Online Transactional Processing) and OLAP (Online Analytical Processing) workloads in one single system. For example, some enterprise applications run OLTP-like queries to update stock levels, while at the same time processing OLAP-like queries to determine if there are enough stocks or resources to fulfill an order \[4,5\]. The traditional approach has been to divide database systems into OLTP or OLAP workloads, and companies that run applications in both areas would have two separate systems for these two kinds of workloads. In such arrangements, transactional processing is performed on a dedicated OLTP database system. A separate data warehouse is installed separately for analytical query processing. Periodically, data from the OLTP database system is extracted, transformed into a format suitable for the data warehouse system, and loaded into the data warehouse for future usage \[6\]. The problem with this setup, however, is data staleness, which occurs when data analytics applications do not run on the most up-to-date information available \[7\]; this issue represents a bottleneck in real-time data analysis that is difficult to overcome \[8\].

Previous performance studies of databases \[8–10\] have shown that cache behaviour has a very strong correlation with query execution time. This correlation is even stronger for main memory databases, due to eliminating the latency of disk accesses for these systems. The cited studies have also emphasized the importance of table layout for good cache and TLB behaviour. Even though row-based and column-based layouts are suitable for OLTP and OLAP workloads respectively, neither of these layouts performs well for mixed OLAP/OLTP workloads \[4,8\]. Vertically partitioning the data into different tables in an intelligent manner can lead to a layout that performs well for both query types of the mixed workload.

In addition to the lack of support for mixed OLTP/OLAP workloads, the increase in popularity of newer types of semi-structured data, such as JSON data also poses some challenges. JSON (Java Script Object Notation) is a simple but powerful data model for semi-structured data. The simplicity and compactness of the JSON model has made it easier to parse and process in comparison to alternative data models such as XML. JSON’s ease of use and its support in many languages such as Javascript, Python, Perl, PHP, and Ruby have made JSON the most common data model used in many Web 2.0 applications. Furthermore, JSON is widely used for data exchange among web services, and many web services support JSON as a data inter-
change format in their APIs. Google, Facebook and Twitter are examples of major commercial companies that have JSON API for their services [11].

Currently, the systems used to process JSON data are NoSQL (Not Only SQL) document stores [12,13]. The flexibility that they provide for programmers to load, store, and process data without defining any schema upfront is the main advantage that makes them very popular [14]. Despite their flexibility however, NoSQL systems suffer from several limitations compared to relational databases. Such limitations include: the absence of a standardized declarative query language, weaker concurrency and atomicity properties than ACID, and the lack of rich native query processing constructs (e.g., joins) [11]. One promising alternative to NoSQL systems is enabling JSON data management within RDBMSs (Relational Databases Management Systems) along with relational data [11,14,15].

To summarize, our motivations for this dissertation fall under three main categories: 1) the need for real-time analytics and main memory systems as solutions. 2) Hybrid workloads in modern applications and the need to support both OLTP and OLAP and hybrid layouts that vertically partition the data into tables as a solution. 3) Popularity of JSON data and problems with current NoSQL documents stores and mapping JSON into relational databases as the solution.

In this context, we identify several interesting research topics correlated with real-world needs that modern database systems have, and in this dissertation we propose an approach that effectively addresses these issues.

- We use a main memory database system to address the need for real-time analytics.
- We provide hybrid layouts that vertically partition the data into tables to satisfy the need for support of hybrid OLTP/OLAP workloads in modern applications.
- We present a solution for mapping JSON data into relational databases to provide support for JSON data without suffering from the limitations of current NoSQL document stores (the current typical framework for using JSON data).

The main contribution of our work, is to design, implement and evaluate a novel technique for vertical partitioning of the database schema for optimizing performance of main memory
Chapter 1. Introduction

databases. To do this, our partitioner considers workload characteristics (e.g. query frequency, selectivity, attributes accessed by each query, etc) to minimize TLB and cache misses while running the query workload. In addition to performance optimization, our dynamic vertical partitioning algorithm (DVP) enables JSON data storage in main memory relational databases by intelligently decomposing JSON objects into different tables.

Furthermore, our partitioner can adapt the current layout to workload and data changes, meaning that if the workload changes, or if new attributes appear in the data, the partitioner generates a new layout by incrementally refining the current layout. Through this repartitioning, our prototype engine constantly maintains good performance even in the case of workload changes, and it also supports schema changes, which are very common for JSON data with attributes dynamically appearing and disappearing in different objects.

To efficiently handle JSON data, DVP takes into account semi-structured data characteristics such as attribute sparseness while generating the data layout. To store JSON objects in relational tables, we first flatten nested objects and make each unique attribute a column in our schema. Then, we partition the schema into different tables. Since flattening JSON objects will result in hundreds or thousandth of columns, exhaustively searching through all possible layouts is not possible. We use simple heuristics to group attributes that are used together in queries into the same partition and to segregate sparse and non-sparse attributes into different partitions. The algorithm complexity of our partitioning algorithm is of polynomial complexity of number of columns.

While the use of these heuristics would intuitively improve spatial locality of access and avoid thrashing the cache, our main contribution is finding a simple encoding of these ideas into an efficient polynomial-time cost model.

In our experiments we use the Nobench benchmark and two state of the art baselines for comparison. In addition we use row-based and column-based layouts which are suitable for OLTP and OLAP workloads. We use Nobench data which is first introduced in the Argo and recently used in other related works. We evaluate DVP-generated layouts against Argo, a state-of-the-art data model that enables JSON mapping onto relational databases and Hyrise, a state-of-the-art vertical partitioning algorithm that is targeted to minimize the number of cache misses in main memory systems. To make a fair comparison with Hyrise, we use
a version of NoBench where we reduce the number sparseness of attributes since Hyrise is not optimized for handling attribute sparseness and doesn’t scale to thousandth of attributes. Our experiments include three different workloads with different conflict levels between attributes accessed by queries. All three workloads are generated by modifying Nobench query set and varying the selectivity of queries, and the query frequencies in the workload based on random distributions. The first two workloads consist of mainly OLAP queries, while the third one is a hybrid of both OLTP and OLAP queries.

Our evaluation results show that we outperform Argo by 15x-30x in all three conflict scenarios. For the first two workloads that have OLAP queries (which don’t access any sparse data), we generate the same layouts as Hyrise, and we outperform Hyrise by a factor of 1.6 for the hybrid workload that contains accesses to sparse data. We also compare the performance of DVP-generated layouts against row-based and column-based layouts, which are standard layouts for relational databases. DVP-generated layouts outperform the row-based layout by at least 2.1x and up to 7x for all three workloads. For the column-based layout, our approach outperforms it by 5% in the first two conflict scenarios, which are OLAP workloads where the column-based approach is known to perform very well. Finally, for the third conflict scenario, DVP layouts outperform the column-based layout by a factor of 6.

The remainder of this document is organized as follows: Chapter 2 introduces several key concepts and provides a context for our work. In Chapter 3 we present a high-level view of our prototype system and describe several important components of our framework. Chapter 4 provides a detailed description of our Vertical Partitioning algorithm. In Chapter 5 we show a thorough experimental evaluation of our system. We discuss related work in Chapter 6 and present our conclusions and plans for future work in Chapter 7.
Chapter 2

Background

2.1 Traditional Database Workloads

In this Section an overview of OLTP and OLAP workloads is provided.

2.1.1 OLTP Workloads

An Online Transaction Processing (OLTP) workload consists of queries that access all or most of the attributes of a few records within a table. OLTP workloads are normally insert or update-intensive and need high throughput. For instance, a simple query that returns all information about a specific employee by using his ID is an OLTP query.

2.1.2 OLAP Workloads

An OLAP (Online Analytical Processing) workload contains read-heavy queries that access a small number of columns for a large number of records. OLAP queries can perform different statistical calculations over the whole data. For example, counting the number of employees that are between 20 and 50 years old is an OLAP query.

2.2 Main Memory Databases: Advantages and Challenges

Unlike a traditional RDBMS, main memory databases store data in the main memory of the system. The motivation behind keeping databases in the main memory is the fact that retrieving
data from main memory is orders of magnitude faster than accesses to disk [3]. In addition to higher performance, the increase in the size of main memories and their lower price in recent years (starting from early 2000) are other factors that have led to a greater popularity of main memory databases.

Another reason why main memory systems have become more attractive in recent years, is their ability to provide high performance real-time analysis for both OLAP and OLTP workloads. While traditional OLTP systems have been optimized to minimize the insertion, update, and deletion of a small subset of records in the database, these approaches have traditionally had poor performance for OLAP workloads. The reason for this poor performance is the access pattern of OLAP-like queries. OLAP queries are typically read-heavy queries that scan a few columns of all records (e.g., count, average and aggregation queries). Reading a few columns of each record can severely affect the performance of traditional OLTP systems due to random accesses to disk. The traditional solution for this problem is maintaining a separate data warehouse system that is optimized for heavy reads [16]. In this separated configuration, insert and update queries apply the changes only to OLTP tables. Therefore, to maintain data in the warehouse, periodic loading of the new data from the OLTP system to the OLAP system is required. Batch-mode data maintenance in analytical systems introduces staleness in the accessed data, which makes it impossible to provide real-time analytics on the most recent data. On the other hand, main memory systems are capable of running OLAP and OLTP queries on the same system, since random accesses to RAM don’t introduce any additional latency.

While main memory databases have the potential to offer faster analytics and remove the need for a separate OLAP system, fully exploiting their potential presents some challenges. Data persistence and failure recovery is one of the challenges of storing data in volatile media rather than disk. One potential solution is using NVDIMM (Non-volatile Dual In-line Memory) [17]. The current approach to address this problem is appending updates to log files and periodically storing snapshots of memory to the hard disk to make failure recoveries faster [18].

Another challenge of using main memory databases is the limited size of data that can be stored in the main memory of a single machine. Distributed main memory systems [1, 19–21] solve this problem by scaling out across several cluster nodes. One other common solution is to use data aging to keep only recent data (hot data) into the main memory of the system, and
spill cold data to disk \[1,3\].

Even though the above-mentioned features help with scalability and durability of main memory systems, adding them to our main memory system is beyond the scope of this thesis. The focus of this work is on optimizing data layouts of tables in main memory to achieve high performance for mixed OLTP/OLAP workloads and also suitable for supporting JSON data.

### 2.3 Partitioning and Data Layouts

Data layout is the information that determines how data must be placed in different tables. Different partitioning techniques generate different layouts that divide data into multiple tables in order to improve performance of a database system. This Section provides an overview of horizontal and vertical partitioning \[22,23\] techniques along with the motivation behind each of them.

#### 2.3.1 Horizontal Partitioning

This technique partitions data of a single table into multiple tables based on the value of a given attribute, such that each table corresponds to a specific range of values. One common usage of this technique is data aging in main memory databases. Since the size of main memory is limited, only recent records are kept in main memory tables and older data is shifted to tables on the hard disk. In this usage, the partitioning criterion is the time of the insertion or last update. Another common usage of horizontal partitioning is in distributed databases \[19,20\]. For example, the information of users from different countries can be stored on different servers distributed over a wide area where each server stores only local users. In this case, the partitioning criterion is the location of the user.

#### 2.3.2 Vertical Partitioning: Column-based, Row-based, and Hybrid Layouts

This technique partitions attributes of a single table into multiple narrower tables, named partitions. The goal of vertical partitioning is to group together attributes that are all being accessed by the same queries. By reducing the number of attributes in each table, a specific
query may potentially skip some of the tables. This skipping leads to lower I/O costs in disk-based databases, as well as better cache locality in main memory databases.

A partition can contain 1 to N attributes, where N is the total number of attributes in the data set. Thus, a vertically partitioned layout can vary from one partition with N attributes to N partitions each with one attribute. A layout with one partition is equivalent to the traditional row-based layout, while a layout with N partitions that stores each attribute in a separate table is equivalent to a column-based layout. We will refer to a layout with any number of partitions in between as Hybrid layout.

Most of the current main-memory database systems are using one of the two common layouts: column-based or row-based. For example, Scuba [1] from Facebook uses a row-based approach and systems such as Dremel [24] and PowerDrill [25] from Google all use a column-based approach. To exploit the advantages of both layouts, some systems let the system administrator specify one of the column-based and row-based layouts [21], while others maintain copies of one table with both formats. For instance, the Oracle main memory system database [26] has a dual-format architecture that enables tables to be represented in main memory using both layouts simultaneously, and automatically sends each query to one of the copies to improve performance.

2.4 Optimal Data Layouts for Different Workload Types

This Section provides an overview of performance of OLAP, OLTP, and mix workloads on column-based and row-based layouts.

2.4.1 OLTP Workload

As a generic example of OLTP-like queries, consider the simple query: "SELECT * (all attributes) FROM table_name WHERE attribute_name = some_value". In a row-based layout, once the condition holds for the condition attribute of a record, other attributes of the record that are stored next to the condition attribute are already in the cache, and accessing them doesn’t introduce any extra cache misses. On the other hand, in a column-based layout, different attributes of a record are stored in different tables (different memory addresses), and
retrieving them will introduce extra cache misses. One may argue that once data is brought into the cache for one record, it will still be there for the following records that get selected. This can be true if the selectivity of the query is high enough to select records that are close to each other. However, this is usually not the case for OLTP queries since they have low selectivities, and selected records are distributed among different rows of a table.

Figure 2.1 shows the number of cache misses for an OLTP-like query that selects one single record with attributes a1-a8. In a row-based layout (represented on the left), the query retrieves all 8 attributes with one cache miss since all the attributes of the record are adjacent to each other and can be fitted into a single cache line. In the column-based layout represented on the right, multiple cache misses will occur, since the 8 attributes from the selected record are spread out far apart in memory, and accessing each attribute incurs one cache miss.

2.4.2 OLAP Workload

A typical access pattern of an OLAP query is accessing a few columns of all records, while leaving the other columns untouched. The column-based layout has the best performance for OLAP queries, since it isolates accessed columns for running a query from other columns of the table, which leads to better cache locality. In other words, by keeping each attribute in a separate column, unwanted columns are not being brought into the cache, thus avoiding unnecessary cache pollution.

Figure 2.2 shows the number of cache misses for an OLAP-like query that selects one single attribute (a1) from all objects when some condition (a4) is true. In the row-based layout, represented on the left, the query has to bring 8 different blocks to the cache in order to retrieve a1 from all records. On the other hand, in the column-based layout (represented on the right), bringing all the necessary information to the cache only incurs 2 cache misses, since all a1 attributes are stored in a separate table.

2.4.3 Hybrid Workloads

In Sections 2.4.2 and 2.4.1 we have explained the performances of OLTP and OLAP workloads on row-based and column-based layouts. However, in the cases where both types of queries happen in the same workload, neither of the two extreme layouts works well. Mixed OLTP/OLAP
workloads are common \[8\] in recent data analytics. The solution for these hybrid workloads is chunking the table schema into smaller tables that lie somewhere between the row-based and column-based layouts in terms of their number of attributes. Vertical partitioning of data leads to best performance when some information about queries of the workload is known and partitions can be customized based on workload information \[3,7\].

2.5 Workload and Data Characteristics

In this Section, we will explain the data and query types that will be our focus. The first Section is about semi-structured data and JSON in particular, the advantages of using JSON data, and the second Section provides an overview of Nobench \[11\], which is a synthetic benchmark for
2.5.1 Semi-structured Data

Semi-structured data is schema-less, meaning that the users do not have to define a rigid and predetermined schema upfront. This lack of schema provides more flexibility for users, because they don’t have to know the whole schema in advance. In other words, different objects can have different structures and properties, and attributes can dynamically appear/disappear in different objects. Semi-structured data is self-describing because it contains the structure of the data along with the actual values. Storing the structure of the data makes it easier for applications to parse the data and generate web pages automatically. All these advantages have made the semi-structured data format the dominant standard for data exchange through JSON data.
JSON (Java Script Object Notation) and XML (Extension Mark-up Language) data models are the two main models that are designed for semi-structured data. Since the XML model is the older one, there are more studies about different ways of storing XML data, adapting it to relational databases, indexing it, compressing it, and there are more standard query languages and benchmarks for it. On the other hand, JSON is a newer model and it does not have any standardized query languages or benchmarks.

Despite a relatively low number of studies on it, JSON is becoming more popular these days. There are many reasons for the recent increase in popularity of the JSON model. First, the JSON model is simpler than XML while having the same data presentation power. Indeed, the JSON data model is a simple yet powerful format that supports four primitive types: String, Numbers, Boolean, and Null. JSON also supports arrays and nested objects for expressing complex data. Second, since JSON is a more lightweight format in comparison to XML, it is much easier to parse and use it in different programming languages such as Javascript, Java, C++, Python, Perl, PHP, etc. The third reason is JSON’s adaptability. It’s trivial to convert data from XML to the JSON format and it is appealing for many systems that are using XML to switch to the JSON format. All these advantages have made JSON the dominant format for data exchange among web services. Examples include the APIs for many of the services provided by Twitter, Facebook, and Google.

The increasing use of the JSON data model comes with the demand for effective JSON-based data management. While NoSQL document stores are popular for JSON data management, they lack several of the advantages of relational database management systems (RDBMS), including complex queries such as joins, ACID properties, a standardized query language, etc. Therefore, there have been recent studies on adapting JSON data to relational databases to exploit the benefits of both RDBMSs and the JSON data model.

When designing a database system for JSON data, several important factors need to be taken into account: first, the lack of an upfront schema will lead to attributes that only exist in some records rather than in all of them; this is known as data sparseness. Second, JSON objects can be hierarchical, meaning there can be any number of levels of nested arrays and objects. Due to these two reasons, flattening JSON data into a relational table can be extremely
inefficient because of the huge number of null values that must be stored. Third, the values of an attribute can have different data types in different records. For instance, an attribute called "age" can have an integer value of 10 in one object and a string value of "middle-aged" in another one. This means that in order to be able to support storing JSON data in relational databases, a system needs to be able to handle dynamic data types. All of these characteristics make mapping from JSON to relational databases challenging. However, an efficient mapping can preserve the flexibility of the JSON model while still providing the advantages of relational databases.

2.5.2 NoBench: A JSON Micro-Benchmark

In this Section we present the Nobench benchmark that we use in our experiments. This benchmark was introduced in the Argo work [11] on providing support for JSON data in relational databases. Nobench generates a configurable number of JSON objects. Each object is a set of structured "key":"value" pairs. To represent real life data and to cover all characteristics of the JSON model, Nobench data includes dynamic typing, sparse data, nested objects and nested arrays with an arbitrary number of elements.

Each object in Nobench consists of 19-25 attributes including some string, boolean, and long attributes, a nested object with two attributes, a nested array with length 0-7, two dynamic type attributes, and 10 sparse string attributes. The sparseness ratio is 1%, which means that each sparse attribute appears in 1% of the objects. There are 100 groups of sparse attributes and each group contains 10 attributes, for a total of 1000 sparse attributes in the schema. To assign different sparse attributes to different JSON objects, one sparse group is randomly assigned to each object; once all the groups are assigned, all 10 attributes of each group appear in the objects. Even though Nobench is the only benchmark for JSON data to the best of our knowledge, it models the characteristics of semi-structured data collections very well [15]. Table 2.1 contains a list of the queries present in the Nobench benchmark.
Table 2.1: Nobench Sample Queries

Q1  SELECT str1, num FROM nobench_main
Q2  SELECT nested_obj.str1, nested_obj.num FROM nobench_main;
Q3  SELECT sparse_xx0, sparse_xx9 FROM nobench_main;
Q4  SELECT * FROM nobench_main WHERE str1 = XXXXX;
Q5  SELECT * FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY;
Q6  SELECT * FROM nobench_main WHERE XXXXX = ANY nested_arr;
Q7  SELECT * FROM nobench_main WHERE sparse Xxx = YYYYY;
Q8  SELECT COUNT(*) FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY GROUP BY thousandth;
Q9  SELECT * FROM nobench_main AS left INNER JOIN nobench_main AS right ON (left.nested_obj.str = right.str1) WHERE left.num BETWEEN XXXXX AND YYYYY;
Q10 LOAD DATA LOCAL INFILE file REPLACE INTO TABLE table;

2.6 ARGO: A Data Structure for JSON in Relational Format

Used for Comparison

Argo was introduced as a mapping layer that transforms JSON objects to a relational format to enable storage of JSON objects in traditional relational databases [11]. Argo provides two mapping methods: Argo1 and Argo3.

Argo1 uses a simple schema to store all JSON objects without customizing the layout based on the attributes that are in the object. It uses a single 5-column table to store every object. The first column of the table stores object IDs, the second column stores attribute keys, and for each value type there is a separate column (string, number, boolean). To store an object, each attribute of the object and its corresponding value are stored in a separated row in the table. The first cell of the row stores the object ID of the object, the second cell is used to store the key of the attribute, and depending on the data type of the attribute’s value, it will be stored in one of the third, fourth, or fifth cells, with nulls in the remaining two other cells that are not used. The left-hand side table in Figure 2.3 shows an example of a 5-column table used in...
Argo1. As the Figure shows, the value of the "age" attribute is 58, which is a long number. It is stored in the fourth column, and the third and fifth columns contain null values.

In the Argo1 design there are two null values for each attribute-value pair. To optimize the storage space, Argo3 stores data in three separate tables (one for each of data type). Thus, there are three tables in total and each table has three columns. Similar to Argo1, the first two columns store the object IDs and attribute keys while the last column stores the attribute values. Depending on the value's data type, a key and its value are stored in one of the three tables (right-hand side tables in Figure 2.3).

Figure 2.3: Transformation of JSON objects into relational format using Argo1 and Argo3: a single 5-column table with one column per data type is used in Argo1, while Argo3 stores different data types separately in three different tables.
The data structures for both Argo1 and Argo3 have the advantage of being flexible for JSON data and easy to use, as users do not need to provide a schema upfront. The table representation takes care of issues such as sparse attributes, hierarchical data, and dynamic typing. However, the performance of both designs suffers due to flattening objects to key-value pairs. Inserting each attribute of every object into a separate row results in tables with roughly 20x more rows. The large table size becomes problematic when a query needs to scan all rows of a table. Scanning the whole table occurs when a query accesses an attribute (e.g., the condition attribute of a selection query) for each object. Since offsets of the same attribute may vary significantly from object to object (due to both arbitrary order of attributes in JSON objects and sparse attributes), skipping rows based on row numbers is not possible.

We implemented both Argo1 and Argo 3 in the same language that we used for our system, C++. To handle keys and string values, a dictionary technique that will be described in Section 3.2 is used and hashed values rather than strings are inserted into actual main data table(s). We believe that this hashing technique improves Argo performance, as comparing integer values is faster than comparing strings.

2.7 Hyrise: A Main Memory Database with Vertically Partitioned Layout Used for Comparison

Hyrise (2010) is a main memory database that supports hybrid layouts. It uses an accurate cache miss model to calculate the estimated runtime of all queries in the workload for a given layout. Hyrise uses an exhaustive search to find the layout with the minimum estimated execution time among all possible layouts. Even though the Hyrise layout generator performs and exhaustive search which is of exponential complexity of number of attributes, it scales up to a few hundred number of attributes by implementing different optimizations. While Hyrise manages to scale up to a few hundreds of attributes by pruning parts of the search space, it is still unable to generate a layout for a dataset with 1000 attributes. We ran the Hyrise layout generator on the Nobench dataset and the program did not terminate even after several hours of execution. We eventually had to halt the program.
Chapter 3

Design and Implementation

In this Chapter we discuss the design of our proposed system and provide detailed implementation details. We begin by giving a high-level overview of the architecture of our system and a brief description of its main components in Section 3.1. We present the In-Memory Data Container in detail in Section 3.2. Finally, in Section 3.3 we explain significant aspects of the execution of various types of queries in our system and address the topic of repartitioning.

3.1 High Level Architecture

A high level view of the implemented system’s components is shown in figure 3.1. The system consists of the following four main components:

1. An in-memory data container which contains contiguous table-like structures to store data.

2. A partitioner which takes workload and data characteristics as input and intelligently generates the optimal layout. The partitioner uses the proposed algorithm to vertically partition the data into different tables.

3. A data manager which receives data layout from the partitioner and enters data into the container by using layout information.

4. A query engine that parses and executes queries one by one by accessing the data container and retrieving desired data.
As shown on the left side of figure 3.1, our system receives JSON data streams and queries from the outside world as inputs and retrieves query results as the output. The system can profile the received queries and data over a period of time to extract their important characteristics, such as: the number of different types of queries, selectivity and frequency of each type, number of attributes in the dataset, sparse attributes and their sparseness ratio, access patterns of attributes by different queries, etc.

Once useful information about data and workload characteristic are gathered by profiling inputs and outputs for a while, the partitioner can use this information to come up with a layout that optimizes performance. It’s important to mention that, since the optimization is based on the history of the system, any performance benefit is expected only when the workload and data characteristics are stable and will not dramatically change for a while.

Subsequently, the data manager uses the generated layout to set meta data (Schema) of the in-memory data container. The data container in turn builds tables in main memory for each partition and populates them with data. An example of dividing the dataset into different smaller tables (3 tables in this example) is illustrated in the right side of Figure 3.1.
3.2 In-Memory Data Structure

Figure 3.2: An example of data storage within the in-memory data container: schema tables store the name and type of each attribute, while main data tables store actual data. Data types are represented as follows: long and boolean values are stored as they are, null values are stored as -111111, string values are hashed in a separate hash map and their corresponding hashed values are inserted into data tables.

Our in-memory data container component takes the partitioning layout from the partitioner and stores the following meta data for each attribute in the dataset.

- **Name:** in our implementation, we decided to flatten nested objects and use their path from the root object as their names. For example attribute "foo" of a "nested_object" is flattened as a separate attribute and referred as "nested_object.foo" (see [11][24] for more details). Similarly we treat each index of the array object as a separate attribute and to access different indexes of an array object we use object_name + index format. For instance, the name of first index of array "arrayFoo" object will be "arrayFoo[0]". Note that flattening nested data to a 2D table doesn’t introduce any noticeable overhead in
our experiments because the maximum nesting level in Nobench is two 11.

- **Type:** type of the attribute, four supported primitive types are long, string, boolean, and null value.

- **Table ID:** ID of the table to which the attribute belongs to (host table).

- **Table Offset:** offset of the attribute in its host table.

After storing meta data for each attribute, the system create a table object for each partition in the layout. As illustrated in figure 3.2 in our data structure, each Table object has the following parts:

- **Schema table:** this is a string array that stores names of the attributes in the table.

- **Main data table:** an array of long type that stores the actual data. In this long type array, long values are stored as they are. Boolean values are stored as either 0 or 1 to represent false and true, respectively. Null values are stored as -11111111. Since it is necessary to represent a null as a long value, -11111111 specific value is chosen assuming that it never appears as a long value in any actual data.

To be able to access each attribute of a record by its offset rather than scanning the whole record from its beginning, it is necessary to have fixed size attributes. So, it’s not possible to store string values as they are. Two alternative solutions are: 1) to use a hash table and enter the hashed values in the array. 2) to store strings in a separate table and keep a reference to them in the array table. The former method allows to compare strings based on their hashed values (hashed values are unique) instead of comparing string values directly. For example, when a query such as `Select A, B, C Where D = "stringFoo"` is executed, it’s not necessary to access actual values of `D` in every record to compare them with `"stringFoo"` and it’s enough to compare their hashed value with hashed value of `"stringFoo"` and access actual values only when the condition is true. This technique is called late materialization 35. To retrieve both corresponding hashed value of a string and equivalent string value of a hashed number in $O(1)$ we maintain two hash tables, one translating from String to Long and the other one vice versa (Figure 3.2).
As mentioned in Section 2.5, some attributes are sparse and don’t exist in all records. If all attributes of a table are missing in a record, we skip the record and don’t store a record with all null values in the table. So, some records are missing in some tables and it’s not possible to access attributes of a record by using its row number in a table. In Figure 3.2’s example, the second object is skipped in Table 2 because all of its attributes in this table (B, D) are missing.

To keep track of all attributes of a record in separate tables, object IDs are added to the beginning of each record in every table, as shown in Figure 3.2. By using the object IDs, we are able to extract attributes of a record from different tables and reconstruct the whole record.

### 3.3 Query Execution

This Section provides some details about execution of different queries in the system.

#### 3.3.1 Table creation

Before any data can be populated into and retrieved from the engine, tables need to be created. The engine takes a partitioning layout as an input and creates tables. Next, it sets the tables' meta data by populating the schema table in each of them. Before creating the main data table some optimizations are done to minimize number of TLB and cache misses while accessing the data.

**Cache Collision Prevention:** the starting addresses of data tables are aligned to page size addresses in order to exploit TLB entries as much as possible. Since the number of sets in the L1 cache is a divisor of page size, this alignment causes similar offsets of different tables to be placed into the same cache line [8]. For example, the first 64 bytes of all tables will be placed into set 0 of the L1 cache. This means that only a limited number of tables can be accessed at the same time (up to a number equal to the associativity of the cache). Thus, if a query needs to access more tables, it is not efficient to process records one by one by going through all tables. To overcome this problem, the Data Manager allocates a data table in page size-aligned addresses that are shifted by one cache line size compared to the previous data
table. By doing so, we can concurrently access a number of tables equal to the associativity of the cache \times the number of sets in the cache, without introducing any cache collisions between different attributes of the same record that are in different tables.

**Narrow Padding:** In some cases, it is beneficial to add padding to the end of the records of a table in order to make their size cache-friendly; in other cases, padding will introduce extra cache misses in addition to its memory overhead. As a simple example for a table with padded records, when a query needs to scan the whole table, it will experience a higher number of cache misses, since a table with padded records is larger than the original one. On the other hand, when a query accesses the first 40 bytes of 120-byte records on a machine with a cache line size of 64 bytes, the original table will experience an average of 1.5 misses per record, while a table with 8-byte padding (and a total size of 128 bytes) will only experience an average of 1 cache miss per record.

In order to figure out whether padding would be beneficial or not for a specific table, we predict the total number of cache misses for all possible simple projection queries over the table. If the average cache miss per record is lower for the padded version, then we add the padding; otherwise, we leave the table unpadded. We use the projection miss formulas from Hyrise \[8\] to predict the number of cache misses for the projection. The padding size is calculated by using Equation \[3.1\]

\[
\text{Padding Size} = \text{cache line size} - (\text{record size} \% \text{cache line size})
\]

**Null Rows Insertion:** recall that by default if all attributes of a table are missing in a record, we skip the record and don’t store it in the table. But we also have the implementation option that one can enable inserting null records to guarantee that all records are present in all tables and access records by row number. This option is useful for queries with low selectivities since they can access few records directly instead of scanning the whole table.

### 3.3.2 Data Insertion

After creating all tables based on the input layout, a bulk insert need to be executed to populate the tables before starting execution of other queries. However, just like other queries individual
and bulk inserts can be executed during the actual query running process as well.

During the bulk insert, the data populator reads objects from the object file one by one (each object is written in a separate line). Once the line is stored in a string variable, a reader from JSONCpp [36] (an open source JSON parser project) parses the line and creates the in-memory JSON object. Next, the engine calls a recursive parser method to parse the JSON object and transform it to its flattened version. The output of the parser is a map of key-value pairs of the object in which keys are the flattened name of attributes (see Section 3.2). The pseudo code of the parser method is provided in algorithm [1].

The parser function has two arguments: the first argument is the JSON object that is going to be flattened (JSON_object) and the second one is the flattened path of object’s parent (parent_path). The output of the function is a hash map with flattened attribute names and their values. The function is initially called with a JSON object and an empty string (the parent is null), then for each attribute of the object if the attribute is an object itself (i.e. a nested object) the function will be recursively called for its attributes with the "parent_path" updated to "parent_path.object’s name". If the attribute is an array object, the function will be recursively called for all attributes of the array and for each index the parent_path will be updated to "parent_path[index]". Finally, if an attribute is neither a nested object nor an array, its value as well as "parent_path.attribute_name" will be added to the map.

Figure 3.3 shows an example of flattening hierarchical and nested attributes of JSON objects in our system. "Kid.Name" is used to represent the attribute called "Name" within the nested object called "Kid" and "Friends[0]" to "Friends[2]" are used to represent index 0 to 2 of "Friends" attribute respectively. You can also see null values due to data sparseness in the picture. For example, nested object "Kid" is missing in left object and all "Kid.*" attributes of this record have null values in the table.

Before inserting each parsed object into its tables, a new row, along with a system-wide unique object ID, is added to every table that is involved in storing at least one of the attributes in the current parsed object. If a table has reached to its maximum capacity, it will be copied to a new array with twice capacity before adding the new row and object ID.
Algorithm 1: JSON recursive flattening algorithm

**Input:** JSON\textsubscript{object}, parent\_path

**Output:** A map of flattened attribute names and their corresponding values.

**Method:** parseObject

if JSON\textsubscript{object} is a boolean, string, or long value then
  Insert (JSON\textsubscript{object}’s value, parent\_path + . + JSON\textsubscript{object}’s name) to the map.
else if JSON\textsubscript{object} is an array then
  foreach index of the array do
    Call parseObject with JSON\textsubscript{object}[index] and parent\_path+"["+index+"]"
    Add the result to the map.
if JSON\textsubscript{object} is an object then
  foreach attribute in JSON\textsubscript{object} do
    Call parseObject with JSON\textsubscript{object}.attribute and parent\_path+.+attribute name
    Add the result to the map

return map

Figure 3.3: Example of how the system flattens JSON objects for storage in relational format.
3.3.3 Selection and Projection Queries

Select, support of many conditions, how to handle, if now missing rows access table by index, two ways of table traversing pros and cons

Generic form of selection Queries is "SELECT column-names FROM table-name WHERE some-conditions" and the generic form for projection queries is "SELECT column-names FROM table-name". In other words, a projection query is equivalent to a selection query with an always true condition part. The condition part of selection queries can have arbitrary number of conditions from the following list:

- Equality (attribute-name = some-value): this condition type works for all attribute types including strings and booleans.
- Inequality (attribute-name != some-value): this condition type works for all attribute types including strings and booleans.
- Other comparison operators (¡, ≤, ≥, and ≥): all these condition types are supported for number variables only. Although they can be used for strings as well, we decided not to support them. Because they need comparison of actual strings rather than their hashed values. It means that late materialization is not possible for these operations and the running query needs to access dictionary table for extracting actual values of the string condition variable for each record.
- Range (attribute-name BETWEEN value1 AND value2): Since range queries are pretty common for number variables, we decided to support them directly rather than combining two comparison operators.

Selection queries scan condition attribute(s) of records to check whether the condition(s) is/are true or not. If all conditions for one record are met, attributes in the selection part of the query will be retrieved from the record. Since records are vertically partitioned into different tables, it is possible for a selection query to access different tables for condition evaluation and attribute retrieving. Therefore there are two different orders in which a query can access the different tables. The first option is to retrieve the attributes from each table sequentially which is called Table-first traversing and the second option is to access all necessary tables in parallel,
which is called Record-first traversing. In the following Sections details of both implementations are provided.

3.3.3.1 Table-first Traversing Order

In this method of traversing, to run a selection query the engine scans the tables of condition attributes one by one. While scanning the first condition table, it creates a list of object IDs for which the first set of conditions (conditions from first table) are true. Once the scanning of the first condition table is complete, the engine continues with scanning the remaining condition tables. For these tables, it skips the objects that do not exist in the object ID list and evaluates the condition sets only for the objects that are still in the list. If evaluation of a condition set fails for an object ID, the engine removes the ID from the list. Therefore, as the engine scans more tables, object ID lists becomes smaller and smaller. The IDs that remain in the list after scanning all condition tables, are those that satisfy all conditions in the condition part of the query.

Once the object ID list is finalized, the engine scans the selection tables one by one and retrieves the desired attributes from objects that are in the object ID list. As an optimization, if the last condition table that the engine scans is also a selection table, its desired attributes are retrieved right away. This way the engine scans this table only once rather than once for condition evaluation and once for attribute selection.

Figure 3.4 shows a simple example of table-first traversing. In this example, the query "SELECT a1, a2, a3, a4 WHERE a1 = some-value" is executed on a layout with the first two attributes in one partition and the other two in another partition. Assuming that the condition is only true for a1 in objects 1 and 3, the engine first scans the condition table and inserts object IDs of 1 and 3 while skipping the second object. It retrieves a1 and a2 from the same table right away and finally scans the second table to retrieve attributes a3 and a4 that are inserted into the object ID list.

**Post-Select Data Reordering**

While accessing tables one by one seems easier, it has the overhead of post selection data reordering. This is because, when attributes of the same object are vertically partitioned into different tables, accessing the tables one by one will retrieve all attributes of one table next to
each other rather than keeping attributes of one record next to each other. However, when a subset of attributes of an object are requested, it is normally desired to append attributes from different tables to each other and reconstruct the original object.

To retrieve the data in correct format, an additional step is necessary to reorder the intermediate results and stitch them back into object-based tuples before returning the results to the user. To reconstruct the tuples from selected data, a list with the size of the selected tuples is created. Each element of the list maintains one partially reconstructed tuple. By traversing the intermediate results, each attribute is appended to the tuple element with its corresponding object ID. Once the traversing is completed, the tuple list is returned as the final result to the user.

### 3.3.3.2 Record-first Traversing Order

In record-first traversing mode, the engine scans all necessary tables in parallel. The query execution starts with assigning a pointer to the starting address of each desired table. Then the engine iterates through rows of condition tables one by one, and evaluates all condition sets from different tables for each object ID. If all conditions are met for an object ID, the engine retrieves selection attributes of same object from all selection tables and then increments all pointers to point to next row otherwise, it only increments pointers of condition tables.

If a specific object ID is missing in one of the condition tables, it means that the corresponding object doesn't have the condition attributes of the table. In our implementation, not having an attribute is treated as not meeting the condition for the attribute. Therefore, condition evaluation fails for the missing object ID and pointers to condition tables get incremented without accessing selection tables. On the other hand, if an object ID is missing in a selection table, the engine skips the table for this specific object ID and retrieves its attributes from other selection tables. Note that since desired attributes are retrieved record by record rather than table by table, reconstructing retrieved parts of objects as tuples is as easy as appending attributes next to each other while they are being retrieved.

Figure 3.5 shows record-first traversing of the same query in 3.4. Similarly assuming that the condition is only true for a1 in objects 1 and 3, the engine scans both tables in parallel. It evaluates the condition for first record and retrieves a1 and a2 from the first and a3 and a4
from the second table if a1’s value satisfies the condition and skips the record otherwise. The engine repeats the same procedure for the next two records.

3.3.4 Inner Join

Generic form of an inner join query is "SELECT column-names-from-either-of-the-tables FROM table1 INNER JOIN table2 ON join-condition [WHERE some-conditions-over-table1]".

Join Queries retrieve attributes from two tables when the join condition and the optional conditions over first table are true. We use hash tables to implement join queries. Our join implementation has two phases. In first phase, the join-attribute of first table is scanned and hashed into a hash table with attribute values as keys and row numbers as values. Optional conditions over the first table are also evaluated in this phase and records that violate any of the conditions are not inserted into the hash table. Once the first table is completely scanned, the second phase starts. During the second phase, the engine scans the second table row by row. For each row, it looks for the join-attribute of the row in the hash-table. If the engine finds the join-attribute key, it then extracts the corresponding value of the key which is a row number from the first table. Finally, it retrieves the selection attributes from both current row of the second table and the row with the extracted row number of the first table.

3.3.5 Repartitioning

Our engine is adaptive to workload and data changes, meaning that if the workload changes, or if new attributes appear in the data, the partitioner generates a new layout by incrementally refining the current layout (see Chapter 4 for more details). Once the new layout is generated, the query engine invokes repartitioning. The repartitioner first creates meta-data (schema) of the new tables by parsing the new layout file. After creating the new tables, it fills the tables with data from the current tables. As an optimization, if the schema of a table is not modified in the new layout, the whole table is copied at once. Copying data from old tables to new tables can be time consuming if there are many rows in tables.

In order to maintain high performance of the engine during repartitioning, a separate thread runs repartitioning in the background. While new tables are being populated by data from old tables, the query engine continues to run the queries on old tables. Once the building of new
tables is completed, the repartitioning thread signals the main thread to atomically switch to new tables. After this atomic switch, the memory of old tables is deallocated. In order to minimize the interference of repartitioning in query performance, the repartitioner thread is bound to a different CPU core.
Query: SELECT a1, a2, a3, a4 WHERE a1 = some-value (The condition is true only for row 1 and row 3.)

Layout:
P1: a1, a2
P2: a3, a4

Object ID list:
1 -> 3 -> ...

1- Evaluates the condition for a1, condition met, adds the ID to object ID list, adds a1 to the answer array.
2- Adds a2 to the answer array.
3- Evaluates the condition for a1, condition not met, skips the record.
4- Evaluates the condition for a1, condition met, adds the ID to object ID list, adds a1 to the answer array.
5- Adds a2 to the answer array.
6- ID 1 is in Object ID list, adds a3 and a4 to the answer array.
7- ID 3 in Object ID list, adds a3 and a4 to the answer array.

Answer Array:

Figure 3.4: Table-first traversing example of how the query "SELECT a1, a2, a3, a4 WHERE a1 = some-value" is executed.
Query: SELECT a1, a2, a3, a4 WHERE a1 = some-value
(The condition is true only for row 1 and row 3.)

Layout:
P1: a1, a2
P2: a3, a4

Object ID list:
1 -> 3 -> ...

1- Evaluates the condition for a1, condition met, adds the ID to object ID list, adds a1 to the answer array.
2- Adds a2, a3, a4 to the answer array.
3- Evaluates the condition for a1, condition not met, skips the record.
4- Evaluates the condition for a1, condition met, adds the ID to object ID list, adds a1 to the answer array.
5- Adds a2, a3, a4 to the answer array.

Answer Array:

Figure 3.5: Record-first traversing example of how the query "SELECT a1, a2, a3, a4 WHERE a1 = some-value" is executed.
Chapter 4

Vertical Partitioning Algorithm

This Chapter describes in detail our proposed vertical partitioning algorithm. We begin by a simple example of vertical partitioning in Section 4.1. We analyse the impact of sparse attributes on partitioning in Section 4.2. We give an overview of our algorithm and its two main components, a search algorithm and a cost model, in Section 4.3. We then present these two components in detail in the following two sections: Section 4.4 describes our search algorithm, while Section 4.5 covers the cost model for our partitioning algorithm. Finally, in Section 4.6 we explain the initial partitioning approach that our algorithm uses in specific scenarios.

4.1 Vertical Partitioning Examples

As we have already mentioned, cache utilization has a strong correlation with query execution time in main-memory database systems. In this section, we show the impact of vertical partitioning on cache utilization by giving simple examples. Figure 4.1 shows the cache miss patterns for running the query "SELECT a1, a5, a9, a16 (100% selectivity)" on the records with a1-a16 attributes. The average number of cache misses per record is two for the non-partitioned layout vs. 0.5 for the partitioned layout. In this example, partitioning avoids bringing unwanted data into the cache by isolating required data from the rest. Figure 4.2 shows the cache miss patterns for running the query "SELECT * WHERE (only true for r1 and r4)" on the same records as previous example. The average number of cache misses per record is two for the non-partitioned layout vs. three for the partitioned layout. In this example partitioning brings
extra unwanted data into the cache. This is because attributes from consecutive records are unlikely to be retrieved by queries with low selectivities. The two examples show that there are two sources of low cache utilization caused by bringing unwanted data into the cache: 1- Running queries with low selectivity on multiple partitions causes the next record to be fetched but not needed. 2- Retrieving few attributes from each record fetches some other attributes of the records into the cache without being needed. In this chapter, we present an algorithm that estimates the cost of unwanted data from each source for different layouts and finds the layout with minimum average cost.

Query 1: Select a1, a5, a9, a16 (100% Selectivity)

Figure 4.1: Cache miss patterns of a simple query that selects a1, a5, a9, and a16 from records with attributes a1-a16: non-partitioned layout on the top vs. partitioned layout on the bottom.
Chapter 4. Vertical Partitioning Algorithm

4.2 Impact of Sparse Attributes on Partitioning

One important characteristic of JSON data is attribute sparseness, which arises when the attributes have non-null values in a subset of records rather than in every record. Data sparseness is an important consideration in partitioning the data, because different layouts would contain different amount of null values for storing the same dataset. In general, wider tables contain more null values, which in turn will degrade cache locality for the actual non-null values. Moreover, during query execution, null values still need to be read and compared before the engine can distinguish them from non-null values, which takes additional time to perform. In this Section, we investigate the effect of sparse attributes on partitioning by a simple experiment.

The experiment is executed over a synthetic JSON dataset that consists of 20 non-sparse attributes (a1-a20) and 120 sparse attributes (s1-s120) with a 5% sparseness ratio. Sparse attributes come in 20 groups of 6 attributes each. To generate an object, a group is first randomly assigned to the object; next, all 6 attributes of the selected group get values in the object; the remaining sparse attributes (i.e., from the other 19 groups) will not appear in the

![Figure 4.2: Cache miss patterns of a simple query that selects all attributes from r1 and r4 records: non-partitioned layout on the top vs. partitioned layout on the bottom.](image)

**Figure 4.2:** Cache miss patterns of a simple query that selects all attributes from r1 and r4 records: non-partitioned layout on the top vs. partitioned layout on the bottom.
object. All values are randomly generated double-precision floating point numbers within the [0,1] interval.

Once a dataset with 200,000 objects is generated, we store the data in tables using layouts with different partition sizes and run a "Select * Where xxx" query on different layouts to see the impact of the frequency of null values on execution time. Results for the experiment with 25% selectivity are shown in Figure 4.3.

There are 16 layouts with different partition sizes in the experiment. In all layouts, the condition attribute (a1) is stored in a separate partition, all non-sparse attributes in selection part of the query (a2-a20) are stored in another partition together, and sparse attributes are chunked into partitions with specific sizes according to the layout. For example a layout with partition size of 1 results in 120 partitions for sparse attributes and a layout with partition size of 120 results in 1 partition for sparse attributes. The execution time for all divisors of 120 are presented in Figure 4.3. The number of sparse attributes is selected as 120 since it has a high number of divisors.

![Query Execution Time for "SELECT * WHERE ... (25% selectivity)"

Figure 4.3: Execution time of Select * Where (25% selectivity) query over layouts with different partition sizes

As presented in Figure 4.3, neither of the two extreme sizes is the sweet spot for partition sizes. In absence of sparse attributes the layout which groups all attributes of the selection
part into one partition has the best performance for SELECT * queries. However, when some attributes are sparse, chunking them in smaller partitions will result in better performance. This is because the objects that don’t have any attributes in common with a table are not inserted in it. Thus, the number of null values is lower in tables with fewer attributes, as shown in Figure 4.4. The minimum amount of null values appears in layouts with very small partition sizes (1-6). However, the performance of layouts with these small sizes suffers from the overhead of accessing too many partitions (see Section 4.5). Indeed, the sweet spot for this dataset is somewhere between 6-12 since sparse attributes appear in groups of 6. This also explains the fact that in some cases, going from one partition to a bigger one doesn’t lead to an increase in the number of null values, or in one case (from 10 to 12), the number of null values even decreases.

To summarize, when some attributes are sparse, it’s always beneficial to isolate sparse attributes from non-sparse attributes. Otherwise the amount of null values for the sparse attribute will increase by a value equal to 1 minus the sparseness ratio. Moreover, to avoid performance decrease due to null values, it’s necessary to chunk sparse attributes into smaller groups.
4.3 Partitioning Overview

We propose a partitioning algorithm that evaluates a range of possible table layouts, and identifies the layout with the minimum workload cost. Several criteria can be considered for workload costs; query execution time, data insertion and update time, and storage space are all common metrics. Since our benchmark contains no update queries, an inserted record is never changed subsequently, and thus data update time is not important for us. Our goal is to reduce workload execution time, while also taking into account storage space as a secondary factor.

Our proposed algorithm evaluates workload costs on different layouts based on both query and data characteristics such as query execution times, selectivities, and frequencies, attribute spareness ratios, and access patterns (which attributes are accessed together).

The algorithm has two main components: the first component is an iterative search algorithm that starts with a current layout or a workload-aware initial partitioning (The initial partitioning is smart in the sense that it groups attributes into different partitions based on query characteristics and attribute access patterns rather than randomly chunking them into different partitions), and repeatedly modifies different partitions to generate the final layout. The second component is the cost model used to evaluate candidate layouts in each iteration of the search. We describe our search algorithm in Section 4.4 and present our cost model in Section 4.5.

To find an optimal layout, we map the problem to a graph partitioning problem. Each attribute of the data set is a node in the graph and each partition in the layout is a partition of the graph. The attributes that are accessed with the same queries are connected to each other by a weighted edge. The weights of edges are assigned based on the workload and data characteristics such as selectivities and frequencies of queries as well as data sparseness ratio. Once the mapping is done, the goal of the problem is to find a graph partitioning that minimizes our cost model. In common partitioning problems, the number of partitions is fixed (k-way partitioners) and the cost model that is being minimized is the summation of weights of the edges that lie between two partitions. In contrast, in the case of our problem, we don’t
know the number of partitions in advance, nor is the number of partitions a fixed number. Thus, using the same cost model leads to putting all nodes in one partition with zero cost of cross partition edges. To let our cost model find the optimum number of partitions, we assign penalties to partitions based on the number of attributes that are in the partition but not accessed by the same queries.

### 4.4 Search Algorithm

Graph partitioning is an NP-hard problem even when the number of partitions is fixed. In fact, given \( N \) attributes, the number of all possible ways for partitioning them is of exponential complexity of \( N \) (Bell’s \( N^{th} \) number with \( O(n^n) \) complexity \[^{37}\]), which means that exhaustive searches will not scale to hundreds or thousands of attributes. Since we aim to support a large number of attributes (more than one thousand) and perform the partitioning in real-time, we use a heuristic algorithm to start from the current layout, map it to the graph partitioning problem, greedily refine the partitions, and come up with an updated layout.

Since our goal is to dynamically adapt the layout to data or workload changes, each time the partitioner is triggered, the algorithm starts with the current layout and tries to optimize it rather than generating a new set of partitioning from scratch with common top down or bottom up approaches \[^{37}\]. Generating a new layout by refining the current one has the advantages of both faster convergence of the algorithm and lower cost of recreating the tables.

Taking all explained factors into account, our search approach is an incremental cost-function based heuristic that starts from the current layout (or an initial partitioning) and, in each iteration, it goes through all existing attributes and partitions; for each attribute-partition pair, it calculates the gain of migrating the attribute to that specific partition. At the end of each iteration, the algorithm chooses the attribute that would provide the maximum gain, and actually migrates this attribute to its new partition. The gain of a migration is the difference between the cost function values before and after the migration. The algorithm completes when the maximum gain during an iteration is less than or equal to zero, as shown in Algorithm \[^{2}\]
Algorithm 2: Partitioning Algorithm

Input: attributes, queries, [initial_layout]

Output: new_layout: list of attributes in each partition

if initial_layout is not available then
    Call Initial_Partitioning to get initial_layout

foreach attribute pair \((a_1, a_2)\) in attributes do
    Calculate their connecting edge’s weight (equation 4.6)
    current_layout = initial_layout
    do
        max_gain = -1
        Calculate clc (current_layout_cost) by equation 4.8
        foreach attribute \(a\) in attributes do
            foreach partition \(p\) in layout do
                Temporarily change \(a\)’s layout to \(p\)
                Calculate nlc (new_layout_cost) by equation 4.8
                if clc - nlc > max_gain then
                    max_gain = clc - nlc
                    moving_attribute = a
                    target_partition = p
            if max_gain > 0 then
                Migrate the moving_attribute to its target_partition
                Update current_layout
        while max_gain > 0;
    new_layout = current_layout
4.4.1 Algorithm Termination

At each iteration of the algorithm, a new temporary layout is generated by migration of one attribute to a partition. We refer to the sequence of these temporary layouts from initial partitioning to the final layout as $L$ sequence ($L_i$ in this sequence is the temporary layout resulting from the $i^{th}$ iteration).

In order to prove that the algorithm terminates, it’s enough to show that no temporary layout appears more than once in the layout sequence from initial partitioning to final layout ($L$). Recall that the condition to change one temporary layout to another one is decreasing the layout cost (i.e., the cost of each layout ($L_i$) in the sequence is always lower than the cost of its previous layout ($L_{i-1}$)). Thus, the layout costs in the sequence are descending and repeating a layout from the past is not possible because it will have a higher cost compared to the current layout.

Since no temporary layout appears more than once and the total number of layouts is finite, the algorithm is guaranteed to terminate.

4.4.2 Search Space Pruning

We call all the layouts that can be generated from an identical layout by single attribute migration sibling layouts and prove that from every set of sibling layouts at most one layout can appear in the layout sequence.

In Section 4.4.1 we explain that layout cost sequence from initial partitioning to final layout is descending and all future layouts will have lower costs than the current layout in the layout sequence. Since at each iteration the layout with minimum cost among all siblings will be selected as the current layout and future layouts will have lower costs, the current layout’s siblings will never appear in the layout sequence afterward.

4.5 Cost Model

In this Section we will first explain the motivation behind the cost function’s definition and then provide a high-level description of what it considers and at the end we will explain the actual mathematical equations in detail.
During a query's execution, the query may access different attributes from different tables. But other attributes will be brought into the cache by the query while accessing necessary attributes. For the remainder of this document, we will refer to these unnecessary attributes as redundant attributes and to the cost of accessing them as redundant access cost. Redundant attributes simply waste cache space and reduce our ability to fully benefit from cache locality, and it's important for a partitioner to avoid these redundant attributes as much as possible.

By placing each attribute in a separate partition, one can avoid redundant access costs completely. However, the resulting layout (column-based) penalizes the queries with random accesses to many attributes (OLTP-style queries), since they have to access a large number of tables, which, in turn, increases the number of cache misses for each row retrieval. For the remainder of this document, we will refer to this overhead as cross partition cost, because it is the price that a query pays while accessing different partitions. So, an efficient partitioning algorithm should solve the trade off between redundant access cost of accessing some attributes of wide partitions and cross partition cost of going through many narrow partitions.

Our cost function evaluates a layout by estimating redundant access and cross partitioning costs of running a given workload over it. To estimate the mentioned costs, it considers attribute sparseness ratio and access pattern as well as query frequency, selectivity, and here are a few key terms that are used throughout this work to quantify the estimation.

- \( sel(q, a) \) is the selectivity of query \( q \) for attribute \( a \) (equation 4.1).
- \( sel(q, p) \) is defined as the maximum selectivity of query \( q \) for all attributes of partition \( p \) (equation 4.2).
- \( spa(a) \) is the sparseness ratio of attribute \( a \) (equation 4.3).
- \( spa(p) \) is the maximum sparseness ratio of all attributes in partition \( p \) (equation 4.3).

In equation 4.4 \( RAC \) stands for Redundant Access Cost, which represents the total cost of accessing redundant attributes, for all queries, over all partitions. \( Q \) and \( P \) are query and partition sets respectively, and \( hasAttr(p, q) \) is a boolean function that is true if and only if there exists an attribute in partition \( p \) that is accessed by query \( q \). Finally \( w(q) \) is the frequency of the query \( q \). \( w(q) \) is used as a coefficient because different queries have different execution
costs that are based on their relative frequencies in the overall workload. So, their impact on
RAC is dependent on their relative importance.

`hasAttr(p, q)` in equation [4.4] shows that the increment of RAC for a query over a partition
that is not accessed by the query is zero. Equation [4.4] further indicates that, for each query,
there are two types of attributes belonging to an accessed partition that increase the RAC:
attributes that are not accessed by the query but brought in as part of a cache line access, and
attributes that are accessed by the query, but at a lower frequency in comparison to the query’s
other accessed attributes of the partition. For example, attributes that appear in the selection
part of a select query are accessed by the query with a frequency equal to frequency of the
query multiplied by its selectivity, while attributes that are in condition part are accessed by
the query’s own frequency.

The last summation in equation [4.4] shows that the amount by which these types of attributes
increase the RAC is proportional to the difference between the access rate and sparseness ratio
of the attributes, and the access rate and sparseness ratio of the underlying partitions of these
attributes. Specifically, larger differences imply more non-accessed or null values in partitions,
leading to poorer cache locality.

\[
\text{sel}(q, a) = \begin{cases} 
1 & a \in \text{condition part of } q \\
\text{sel}(q) & a \in \text{selection part of } q \\
0 & a \notin q
\end{cases} \quad (4.1)
\]

\[
\text{sel}(q, p) = \max \{ \text{sel}(q, a) \mid a \in p \} \quad (4.2)
\]

\[
\text{spa}(p) = \max \{ \text{spa}(a) \mid a \in p \} \quad (4.3)
\]

\[
\text{RAC} = \sum_{q \in Q} \sum_{p \in P} \text{hasAttr}(p, q) \times w(q) \\
\times \sum_{a \in p} (\text{spa}(p) \times \text{sel}(q, p) - \text{spa}(a) \times \text{sel}(q, a)) \quad (4.4)
\]
As mentioned before, placing each attribute separately in a different partition eliminates the RAC. However, the overhead of accessing several tables for queries that need wide accesses to different attributes can be unacceptably high. Therefore a heuristic formula is necessary to estimate the cross partition cost of accessing different tables.

To represent the mentioned cost, we map the problem to a graph partitioning problem, where attributes are graph vertices, and the weight of an edge between an attribute pair shows the pair’s affinity. In this mapping, the cost of queries accessing different partitions is equivalent to a summation of the weights of all edges that lie between different partitions. For an accurate mapping, we should assign weights to edges based on workload information (e.g., attribute sparseness ratios and their access patterns by different queries).

Equation 4.6 shows the formula for calculating the weight $w(a,b)$ of the edge between two arbitrary attributes $a$ and $b$, where $Q_{ab}$ is the set of queries that are accessing both $a$ and $b$ together. Note that $w(a,b)$ is the penalty cost that a layout would pay if it decides to map $a$ and $b$ to different partitions. Thus, the cost is higher when the two attributes have similar sparseness ratios and are accessed by many queries with similar access rates. Moreover, putting sparse and common attributes in the same partition introduces null values in the equivalent table which waste cache space while running queries. Thus, $w(a,b)$ between those attributes are small regardless of how frequently they are accessed together by different queries.

Equation 4.7 presents the total Cross Partition Cost ($CPC$) for the entire workload over all partitions as the summation of the $w(a,b)$ values of all attribute pairs that belong to different partitions.

$$Q_{ab} = \{ q \in Q | a \in q \land b \in q \}$$  \hspace{1cm} (4.5)
\[ w(a, b) = \frac{\min(\text{spa}(a), \text{spa}(b))}{\max(\text{spa}(a), \text{spa}(b))} \times \sum_{q \in Q_{ab}} w(q) \times \frac{\min(\text{sel}(q, a), \text{sel}(q, b))}{\max(\text{sel}(q, a), \text{sel}(q, b))} \]  

(4.6)

\[ CPC = \sum_{a, b \in A} \{w(a, b) \mid p_a \neq p_b\} \]  

(4.7)

So far, we have introduced \( RAC \) and \( CPC \) as the overheads of having redundant attributes in accessing partitions versus accessing several partitions during the queries’ executions. Equation 4.8 shows the formula for calculating the total Layout Cost (\( LC \)). We define \( LC \) as the weighted average of the normalized values of both \( CPC \) and \( RAC \). Normalization is necessary for mapping the absolute values of \( CPC \) and \( RAC \) to the \([0 \ldots 1]\) interval for a meaningful summation. The \( \alpha \) coefficient is a workload-dependent parameter that represents the relative importance of \( CPC \) and \( RAC \) for the specific workload. Note that \( CPC \) and \( RAC \) get their maximum possible values in column-based and row-based layouts, respectively.

### 4.6 Initial Partitioning

In some cases, starting from the current layout is not possible or beneficial e.g., when generating the layout for the first time, or after a significant change in the workload characteristics has occurred. In such cases, the algorithm needs to generate the initial layout, and two obvious candidates are the row-based and column-based layouts. However, since the number of attributes in our benchmark is in the order of thousands, having a single table or more than one thousand tables will not lead to an ideal layout. Therefore, starting from either of those extremes is not a good choice, as they are far from proper number of tables and would converge more slowly to an efficient solution, and they would also be more likely to converge to a local minimum near the inefficient extreme initial layout. For these reasons, we designed our own heuristic to generate the initial layout.

Our heuristic starts by sorting queries in descending order of their weights which is for now equal to their relative frequencies. For each query, all of its attributes that are not already
assigned to any partitions are placed together into one new partition. After iterating once through all the queries, the only unassigned attributes are those that are not accessed by any queries. Storing these attributes separately in any format has no impact on workload execution time. However since the workload is dynamic and it’s possible that some of those attributes will be accessed by queries in future, we choose to store them in column-based format because of its higher maintainability. This is because when recreating the tables for newly accessed attributes no change is necessary for tables of other yet unused attributes.

\[
LC = \alpha \times \frac{CPC}{CPC_{\text{max}}} + (1 - \alpha) \times \frac{RAC}{RAC_{\text{max}}} \quad (0 \leq \alpha \leq 1)
\]  

(4.8)
Chapter 5

Evaluation Results

In this Chapter we present our experimental testbed, performance results, and comparisons with baselines. In the first set of experiments, performance of DVP generated layouts is compared with four different baselines: row-based and column-based layouts as two common layouts that are being used in most commercial databases \[1,20,21,38\], as well as Argo1 and Argo3 layouts that are designed for JSON data. In the second set of experiments we compare the performance of DVP generated layouts against Hyrise \[8\] layouts. Hyrise is, to the best of our knowledge, a state of the art partitioner for main memory relational databases.

In the last set of experiments we show our system’s sensitivity to the alpha parameter used in our partitioning formula and examine its impact on the overall performance of our system.

5.1 Methodology and Configuration Set up

All experiments are executed on a Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz machine with 32KB L1D cache, 256KB L2 cache, a shared 20MB LLC cache, and a 32GB main memory. Caches are 8-way set associative and cache line size is 64B. To exploit maximum cache spatial locality of L1 and L2 caches, the engine is bound to a specific core, (Our detailed CPU configuration is illustrated in Appendix A.)

All experiments have the following configuration parameters. The experiments in Section 5.3 includes some additional configurations that will be explained in place.

- Data set: 200000 JSON objects (400MB) generated by Nobench benchmark which is
explained in Section 2.5.2.

- Query set: the queries are adopted from Nobench benchmark. However, in order to create/control conflicts between queries, some of the queries are modified/replaced. Our experiments have 3 sets of queries to represent different levels of conflicts between query access patterns, low, medium, and high conflict scenarios simulate workloads that have little, average, significant conflicts respectively (Tables 5.1-5.3). Different scenarios consist of both OLTP-like and OLAP-like queries to represent a hybrid workload. Uniform distribution is used to assign each query a random selectivity percentage on a 0-100 scale.

- Workload: to generate a workload based on each query set, first the frequency of each query is randomly generated, and then a query log of 1000 queries is populated based on the relative frequency of queries.

- Experiment Configuration: All reported results are an average of 5 independent runs after omitting any outliers. Error bars are not shown since the variance is less than 1% in all experiments. In all of the experiments the DVP results are reported for 3 different values for the alpha parameter ($\alpha$: 0.1, 0.5, 0.9). The time to retrieve actual string values from our dictionary table is not reported in any of the results since it's the same for all of the layouts.

### 5.2 Comparison with Argo, Row-based and Column-based Layouts

In this Section, we compare the performance of DVP layouts with that of Argo1, Argo3, Row-based, and Column-based layouts performances with three conflict scenarios.

#### 5.2.1 Low Conflict

Figure 5.1 shows execution times of the low-conflict query mix. All queries of this simple scenario are OLAP-like queries. Therefore, Column-based layout has the best performance among all baseline layouts. Since queries have conflicts over only one attribute, finding the
optimal layout for this workload is trivial: The optimal layout keeps the conflicting attribute in a separate partition and for each query, it groups its other accessed attributes in a single partition. DVP with all alpha values finds this optimal layout and that's why it outperforms column-based layouts in queries that are selecting multiple attributes (Q5, Q6). Row-based layout is not designed for OLAP-like workloads, especially in a schema with a large number of attributes. In queries with higher selectivities (Q3, Q6, Q7) Row-based performs worse, because selecting a few attributes that are broadly distributed in a wide record causes one cache miss per attribute; this affects queries with higher selectivities more since the select more attributes.

Argo1 and Argo3 have the worst performance among all baselines, because they store each attribute of every object as a separate row in their table and as a result their tables have a factor of 20 more rows. Since Argo layouts don’t have any schema and offsets of the same attributes in objects vary from one object to another object [11], in the absence of indexing, a query must scan the whole table to access its desired attributes. Scanning of large tables causes high execution times even for simple queries.

Figure 5.2 shows total execution time of different layouts for the low-conflict workload. All DVP layouts outperform Column-based (the best performing baseline) by more than 4%. Further improvement is not possible since DVP generates the optimal layout for this workload.

<table>
<thead>
<tr>
<th>Query#</th>
<th>Freq(%)</th>
<th>Sel(%)</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>23.2</td>
<td>12</td>
<td>SELECT str1 WHERE num BETWEEN 0 AND 24000</td>
</tr>
<tr>
<td>Q2</td>
<td>14.4</td>
<td>12</td>
<td>SELECT str2 WHERE num BETWEEN 0 AND 24000</td>
</tr>
<tr>
<td>Q3</td>
<td>9.7</td>
<td>71</td>
<td>SELECT bool WHERE num BETWEEN 0 AND 142000</td>
</tr>
<tr>
<td>Q4</td>
<td>22.4</td>
<td>15</td>
<td>SELECT thousandth WHERE num BETWEEN 0 AND 30000</td>
</tr>
<tr>
<td>Q5</td>
<td>3.5</td>
<td>8</td>
<td>SELECT nested_arr[0],nested_arr[1] WHERE num BETWEEN 0 AND 16000</td>
</tr>
<tr>
<td>Q6</td>
<td>8.5</td>
<td>41</td>
<td>SELECT nested_obj.num,nested_obj.str WHERE num BETWEEN 0 AND 82000</td>
</tr>
<tr>
<td>Q7</td>
<td>18.5</td>
<td>24</td>
<td>SELECT dyn1,dyn2 WHERE num BETWEEN 0 AND 48000</td>
</tr>
</tbody>
</table>

Table 5.1: Low conflict query set.


![Table 5.2: Medium conflict query set.](image)

### 5.2.2 Medium Conflict

Figure 5.3 shows the average execution times of Medium-conflict queries (Table 5.2). This query mix is still an OLAP-like workload since all queries of the set access a few columns of many records in the tables. While there are more conflicts among queries in comparison to the previous workload, results are mostly the same.

In this scenario $\alpha = 0.1$ generate same layouts as Column-based. Therefore, they have exact same performance for all queries. Q1-Q4 are common between low-conflict and medium-conflict scenarios (with different selectivities). Similar to low-conflict, Column-based, $\alpha = 0.1$, and $\alpha = 0.5$ layouts again have the best performance for first four queries since they are still ideal for these queries. However, $\alpha = 0.9$ generates a coarser-grain layout by merging str1, str2, bool, and thousandth, to num’s partition. This coarse-grain layout increases redundant attributes in the partition that is accessed by Q1-Q4 which in turn results in slightly higher execution time. Q5-Q7 have conflicts over str1 and num attributes $\alpha = 0.5$ performs the best for these queries by grouping their conflicting attributes in one partition. On the other hand $\alpha = 0.9$ has the same performance of Column-based and $\alpha = 0.1$ despite grouping the same attribute.
<table>
<thead>
<tr>
<th>Query#</th>
<th>Freq(%)</th>
<th>Sel(%)</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>19.8</td>
<td>19</td>
<td>SELECT * WHERE nested_obj.num BETWEEN 0 AND 38000</td>
</tr>
<tr>
<td>Q2</td>
<td>6.7</td>
<td>29</td>
<td>SELECT num, str1, str2, bool, thousandth, nested_obj.str,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nested_obj.num, nested_arr[0], nested_arr[1], nested_arr[2] WHERE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>nested_obj.num BETWEEN 0 AND 58000</td>
</tr>
<tr>
<td>Q3</td>
<td>20.3</td>
<td>46</td>
<td>SELECT sparse_000, sparse_010, sparse_020, sparse_030, sparse_040 WHERE num BETWEEN 0 AND 92000</td>
</tr>
<tr>
<td>Q4</td>
<td>23.2</td>
<td>11</td>
<td>SELECT sparse_110, sparse_120, sparse_130 WHERE num BETWEEN 0 AND 22000</td>
</tr>
<tr>
<td>Q5</td>
<td>24.4</td>
<td>23.5</td>
<td>SELECT str1 WHERE nested_obj.num BETWEEN 0 AND 47000</td>
</tr>
<tr>
<td>Q6</td>
<td>2.4</td>
<td>35</td>
<td>SELECT str1, bool, thousandth WHERE num BETWEEN 0 AND 70000</td>
</tr>
<tr>
<td>Q7</td>
<td>3.2</td>
<td>34.5</td>
<td>SELECT str1, str2, num WHERE num BETWEEN 0 AND 69000</td>
</tr>
</tbody>
</table>

Table 5.3: High conflict query set.
since the advantage of keeping conflicting attributes in one partition is offset by the inclusion of other attributes in the coarser-grain partition that force redundant attribute accesses for queries. Row-based and both the Argo layouts have poor performance over the medium-conflict scenario due to reasons similar to the low-conflict scenario.

Figure 5.4 shows total execution time for the medium-conflict workload (Table 5.2). DVP with $\alpha = 0.9$ has 4% higher execution time in comparison to Column-based layout. The reason for this increase is poorer performance of Q1-Q4 on $\alpha = 0.9$’s coarse-grain layout. On the other hand DVP with the $\alpha = 0.5$ layout improves Column-based layout’s performance by 5% since it merges conflicting attributes of Q5-Q7 into one partition. Again further improvement is not possible since Column-based layout performs really well for OLAP workloads.

5.2.3 High Conflict

Figure 5.5 shows average execution times of high-conflict queries (Table 5.3). This workload is a hybrid of both OLTP and OLAP queries. The queries have high degree of conflicts among their accessed attributes. Therefore, either Row-based or Column-based layouts are not suitable for this workload and DVP layouts that lie somewhere in between have the best performance. The
Chapter 5. Evaluation Results

53

Figure 5.2: Total low-conflict workload execution time for DVP (α : 0.1, 0.5, 0.9 ), Argo1, Argo3, Row-based, and Column-based Layouts

The following paragraph provides an in-depth analysis of each individual query.

Layout with α = 0.9 has slightly higher execution time for Q1 because it has coarse grain partitions and there are other attributes in Q1’s accessed attributes (str1 and nest_obj.num ) partition that are interfering cache locality. While Column-based and the two other fine grain layouts isolate these two attributes in one or two partitions without any interfering attributes. α = 0.9 has the best performance for queries Q2, Q3, Q5, and Q7 since it has a coarse-grain partition for all non-sparse accessed attributes which leads to the best performance. However it has slightly higher execution times for Q4 and Q6 in comparison to other DVP layouts since in α = 0.9 layout condition attribute of Q4 and Q6 (num) is stored in the coarse-grain partition for non-sparse attributes. Thus, these queries pay higher redundant attribute costs while scanning their condition attribute.

In Section 5.2.1 we explain that regardless of the number of attributes that are being selected by a query, both Argo layouts need to scan the whole table. This scanning overhead is independent from the number of attributes in the selection part and that’s why execution times of different queries are almost similar and there isn’t any significant increase from OLAP queries to the Select * query.

The Column-based layout, which performs well for all OLAP queries, has poorer perfor-
Figure 5.3: Average execution time for Table 5.2 queries over DVP ($\alpha$: 0.1, 0.5, 0.9), Argo1, Argo3, Row-based, and Column-based Layouts.

Performance in Q5 and Q7 since it needs to access a large number of tables. Indeed, it has the worst performance for select * query because of accessing more than 1000 tables in parallel. Even though the Row-based layout is designed for OLTP queries, it still has a poor performance for Select * query. Since storing semi-structured data in Row-based tables results in a huge number of null values which in turn increases the number of cache misses.

Figure 5.6 shows the total execution time of the high-conflict workload. DVP generated layouts (with all alpha values) are more than 5 times faster than both Column-based and Row-based layouts. As expected the DVP layout with $\alpha = 0.9$ has the best performance among all DVP layouts in a high-conflict scenario (5% lower workload execution time). This improvement in $\alpha = 0.9$ is because of the fact that by coarser-grain partitioning, it provides the best performance for 4 queries (including the select * query) while sacrificing performance of some OLAP queries that need finer-grain partitions.
5.3 Comparison with Hyrise

5.3.1 Methodology

Hyrise uses an exhaustive search through all possible layouts. It manages to scale up to a few hundreds of attributes by pruning parts of the search space [8]. However, it doesn’t scale to thousands of attributes. We ran the Hyrise layout generator for the Nobench data. The execution didn’t finish within a few hours and we halted the program at the end.

In order to compare the performance of DVP layouts with Hyrise layouts, we shrunk the number of attributes in the Nobench benchmark. Our modified version of Nobench contains 20 groups of sparse attributes with 5 attribute in each group. Therefore, there are 100 sparse attributes in total rather than 1000 and the sparseness ratio is 5% rather than 1%. Non-sparse attributes are not changed.

To have an accurate comparison, we are using Hyrise’s original source code to generate the layout. We are using our engine for running the experiment with both layouts. To have a fair comparison, all optimizations of the Hyrise engine are implemented in our engine as well. In other words, we support narrow padding as well as cache line wide offsets for starting addresses of different tables (More details of implemented optimizations is provided in Section 3.3.1). We also let the user to decide whether to insert missing rows (rows with all null values) or not and
Figure 5.5: Average execution time for Table 5.3 queries over DVP ($\alpha : 0.1, 0.5, 0.9$), Argo1, Argo3, Row-based, and Column-based Layouts

report results of both experiments for Hyrise.

Table 5.3 presents the query set that is used for running the experiment. We repeated the experiment with low conflict and medium conflict scenarios that don’t include any sparse attributes to compare the two partitioners in the absence of sparse data. DVP generates identical layouts to Hyrise for both scenarios. In the following two Sections, results of record-first and table-first traversing are presented.

5.3.2 Record-First Traversing

Figure 5.7 illustrates the average execution times of the query mix in Table 5.3 for different layouts. The main difference between Hyrise layout and DVP layouts are sparse attributes. Hyrise decides to put all sparse attributes that are only accessed in Q7 into one single partition, since they are accessed always together and don’t have any conflicts with other queries. While such a decision will lead to best performance in non-sparse attribute cases, for Nobench data it’s always better to store sparse attributes in smaller groups to avoid interferes of null values in cache spatial locality.

The layout with $\alpha = 0.9$ has slightly higher execution time for Q1 because it has coarse
Figure 5.6: Total medium-conflict workload execution time for DVP ($\alpha = 0.1, 0.5, 0.9$), Argo1, Argo3, Row-based, and Column-based Layouts

grain partitions and there are other attributes in str1 and nested_obj.num partition that are interfering with cache locality. In contrast, Hyrise and the two other fine-grained layouts store these two attributes either in one isolated partition or in two separate partitions without any interfering attributes. $\alpha = 0.9$ has the best performance for queries Q2, Q3, Q5, and Q7 since it groups non-sparse accessed attributes in one partition which leads to the best performance. However, it sacrifices Q4 and Q6 by storing their condition variable (num) within the coarse-grain non-sparse attributes layout.

The two fine-grain layouts have the best performance for Q4 and Q6 since they store their condition variable (num) in a separate partition. Even though Hyrise isolates the num attribute as well, it fails to get the same performance because it keeps sparse attributes of these two queries in one partition apart from the rest of sparse attributes and doesn’t chunk them further because of Q7 which is a select * query.

The Hyrise layout suffers from null values for Q3, Q5, and Q7 since the accessed attributes in these queries are grouped with sparse attributes and overhead of interfering null values is significant. To summarize, for all queries of Table 5.3, there is at least one $\alpha$ value for which DVP outperforms Hyrise. $\alpha = 0.9$ outperforms Hyrise in 5 queries while sacrificing the other 2. Note that, since the high conflict scenario is a hybrid of both OLTP and OLAP queries with
highly overlapping accessed attributes, there is no single layout that is ideal for all queries and sacrificing some queries for the best of workload is unavoidable.

Figure 5.7 shows the total execution time of the workload. All DVP generated layouts outperform Hyrise by more than 50% irrespective of alpha values. Since the workload is generated by randomly assigning selectivity and frequency in the query mix of Table 5.3, total execution time of the workload is a fair metric for performance measurement. However, we use geometrical mean as another performance metric to let different queries have equal impact on the total average execution time regardless of their absolute frequencies. (Figure 5.9) We are using geometrical mean rather than normal averaging since the normal average will be skewed by the query with highest execution time, while geometrical mean lets different queries to have equal impacts on the total average execution time regardless of their absolute execution times.

The results indicate that DVP generated layouts have lower average execution times in comparison to the Hyrise layout. Indeed, they improve geometrical average execution time of queries by between 15.4% and 19.3%. The difference between geometrical and total execution time improvement is because of the fact that DVP assigns different priorities to different queries based on their frequency in the workload and favors queries with higher frequencies. Note that
Figure 5.8: Total workload execution time for Hyrise and DVP layouts ($\alpha : 0.1, 0.5, 0.9$)

Figure 5.9: Geometrical Mean of execution times for Table 5.3 queries over Hyrise and DVP layouts
the layout with $\alpha = 0.9$ has the lowest total execution time while its geometrical mean execution time is higher. This is because of the fact that $\alpha = 0.9$ is the coarsest grain layout and it favors OLTP type queries which have high execution times as well as high frequencies (see Table 5.3).

### 5.3.3 Table-first Traversing

In Section 3.3.3 we explain that while running a query different orders of traversing tables is possible. In record-first traversing, a query retrieves different parts of a record from different tables in parallel, appends them together and reconstructs a record. On the other hand, in table-first traversing, a query accesses different tables sequentially one after another and then reconstructs the result records by reordering the retrieved data. The time that it takes to create final results by reordering the intermediate results is referred as reordering-time. In this Section, the graphs are results of running queries in table-first format.

Figure 5.10 presents average execution times of Table 5.3 queries over Hyrise and DVP layouts ($\alpha : 0.1, 0.5, 0.9$). The reordering time is excluded from execution times. By comparing the results of the record-first and table-first experiments (Figures 5.7 5.10) we can see an increase in execution times of all 7 queries for table-based format. The main reason for this increase is
that in table-first order a running query scans the object ID table multiple times (one for each table) while in record-first order it is scanned at most once (for Join queries) and not even kept for other queries. An outcome of scanning the index table as many times as accessing tables, is that queries that need to access a higher number of tables (Q5 and Q7) suffer from higher overhead in comparison to the rest of the queries. The second and more important difference between these two experiments is the fact that finer-grain layouts ($\alpha = 0.1, \alpha = 0.5$) have worse relative performance in table-first traversing order. This is because running queries over them demands accessing a larger number of tables.

To summarize, results of table-first traversing order show that, $\alpha = 0.9$ still has the best performance since in addition to all previous advantages, coarser-grain tables help with accessing a lower number of tables, which in turn results in lower overhead of scanning the index table.

Figures 5.12, 5.11 show the workload execution time for DVP and Hyrise layouts with and without reordering time. Both Hyrise and DVP $\alpha = 0.1$ have the highest execution time due to their relatively finer-grain layouts, while DVP $\alpha = 0.9$ is 1.8 times faster than them. More importantly, the reordering time of all DVP layouts is 2.3 times lower than Hyrise’s reordering
5.4 Sensitivity to Alpha Parameter

Figure 5.13 shows the normalized execution times of DVP layouts with different alpha values (0.1, 0.5, 0.9) for all three conflict scenarios. The normalization is based on the execution time for \( \alpha = 0.1 \) layout. As presented in Figure 5.13, \( \alpha = 0.1 \), \( \alpha = 0.5 \), and \( \alpha = 0.9 \) have the best performance for low, medium, and high conflict scenarios, respectively. This result is consistent with the fact that higher \( \alpha \) values generate coarser-grain layouts and coarser-grain layouts perform better for higher conflict workloads.

If the user provides the conflict level of the workload as a guide to the partitioner (e.g. a simple heuristic that calculates the fraction of conflicting attributes over all accessed attributes),
Figure 5.13: Normalized execution times of DVP layouts with different $\alpha$ values (0.1, 0.5, 0.9) over three conflict scenarios, normalization is based on execution times of $\alpha = 0.1$.

The resulting layout will be best tuned for the workload. However, even in the absence of such a simple guide, picking a random value for the $\alpha$ parameter will result in at most 5% performance penalty in all cases.
Chapter 6

Related Work

In this Chapter we present research related to our own work and discuss how our approach is different from previous works. We start with an overview of relevant works that address vertical partitioning in relational databases in Section 6.1 and we present works that use alternative data layouts in Section 6.2. We then go through related projects for main-memory relational databases in Section 6.3 and for NoSQL databases in Section 6.4. Finally, we discuss works that address the topic of mapping JSON data to relational databases in Section 6.5.

6.1 Vertical Partitioning in Relational Databases

Vertical partitioning has been researched extensively in the past. Traditionally, vertical partitioning has been targeted mainly to minimize I/O cost in disk-based relational databases [22, 39, 40]. Some works have addressed vertical and horizontal partitioning problems together [23, 41], while other works focused on layout optimizations in distributed databases [42, 43]. Recently, by eliminating the overhead of disk accesses in main memory databases, the new purpose of vertical partitioning is to optimize the data layout for better cache utilization [8, 14, 45].

Although some previous approaches (e.g., Hyrise) generate layouts that show good performance for relational databases, to the best of our knowledge, none of them consider data sparseness, which makes them unsuitable for partitioning of semi-structured data.
6.2 Other Data Layouts

In addition to vertical partitioning, there are other techniques that customize data layouts for better cache utilization: the PAX \[9\] data layout groups together all values of each attribute within each page. The PAX layout is similar to the row-based approach at the file level, since each page consists of values of all attributes for some records. On the other hand, within each page, the PAX layout is similar to the column-based layout since the values of each attribute for all records in the page are grouped together. PAX improves cache utilization without introducing extra random I/O overhead while accessing different attributes. However, it doesn’t perform well for queries that need to access many attributes; furthermore, it has a high cost for tuple reconstruction \[44\], \[45\]. Data Morphing \[45\] and Trojan \[44\] are two extensions of the PAX model that both group together values of all attributes that are accessed together, rather than grouping values of each attribute separately. This coarser-grain grouping reduces the number of groups in each page/data block, which in turn leads to a lower tuple reconstruction cost \[44\], \[45\]. Since accessing different tables in main memory databases doesn’t introduce any random I/O costs, column-based or vertically partitioned layouts improve cache utilization as well as PAX and its extensions, without the overhead of maintaining meta-data within each page and without an expensive tuple reconstruction.

6.3 Main Memory Relational Databases

Recent years have seen a constantly-increasing demand for real-time data analytics in many systems. Since access to main memory is orders of magnitude faster than access to disk, main memory databases are a solution widely-used to address this demand.

Scuba is a fast, distributed main-memory database system developed by Facebook (2013). It is designed to address Facebook’s real-time analysis demand \[1\]. Scuba is used for interactive, ad-hoc, analysis queries that run in under a second over live data. In most cases, only recent data needs to be processed by queries (e.g., queries related to performance monitoring). To only keep the most recent data in memory, data aging is supported in Scuba. Scuba uses a row-based layout for storing data in main memory tables, and it also supports selection and aggregation queries. As mentioned in \[1\], we believe that Scuba can benefit from other
layouts such as column-based or vertically partitioned layouts, since their workload (selections and aggregations) will have a higher performance on finer-grained layouts. Although Scuba manages to reach a high throughput for real-time analytics, its current (row-based) layout is not suitable for JSON data that has a high number of sparse attributes.

H-Store (2008) [20] and its streaming version S-Store (2014) [46] are other examples of distributed, main memory relational database systems. Both systems are targeted for high-performance OLTP processing. Therefore, the only layout that is supported in their implementations is the row-based layout, which is not efficient for other types of workloads (OLAP and mixed OLTP/OLAP workloads) or for semi-structured data.

Hyper (2011) [6] is a high performance main memory database system designed to address hybrid OLTP and OLAP workloads. Hyper uses a column-based layout to achieve the best performance for OLAP queries. To have a high performance for OLTP queries, it benefits from virtual memory snapshots and allows queries to be executed on arbitrary current snapshots. Although Hyper is targeted for hybrid workloads, we believe that its column-based layout will not lead to high performances for OLTP queries over extremely wide tables. For example, a SELECT * query that retrieves more than 1000 attributes for a record will have a poor performance due to accessing a lot of tables.

SAP HANA (2011) [21] is a distributed main memory database system designed for handling both OLTP and OLAP workloads. It supports both column-based and row-based layouts for relational data to achieve high performances for both workloads. Furthermore, it uses graph-based and text-based layouts to store and process semi-structured data. Even though SAP HANA supports both relational and semi-structured data, it uses a different query processor for semi-structured data instead of mapping the semi-structured data to the relational database. Therefore, all the problems of graph-based and document-based storage formats [6.4] for semi-structured data are still an issue in this design. In contrast, our approach maps the semi-structured data to the relational database, thus avoiding all the aforementioned problems.

Hekaton (2013) [38] is Microsoft SQL’s main memory database system, and it is integrated into the SQL Server. Hekaton is designed for OLTP workloads. It achieves a high performance for these type of queries by using latch-free data structures and an optimistic multi-version concurrency control technique. Similar to H-store and Scuba, it is not designed for hybrid
workloads, and therefore OLAP queries will have a poor performance while running on this engine.

Powerdrill (2012) is a data management system designed by Google, with a column-based database as its core component. Even though it has a very fast query execution time compared to other column-based systems developed by Google (e.g., Dremel), like many other systems it only supports a column-based layout and is not designed for OLTP/ mixed workloads.

Hyrise (2010) is a main memory database that supports hybrid layouts. It uses an accurate cache miss model to calculate the estimated runtime of all queries in the workload for a given layout. Hyrise uses an exhaustive search to find the layout with a minimum estimated execution time. While Hyrise manages to scale up to a few hundreds of attributes by pruning parts of the search space, it is still unable to generate a layout for a dataset with 1000 attributes. We ran the Hyrise layout generator on the Nobench dataset and the program did not terminate even after several hours of execution. We eventually had to halt the program.

In order to be able to compare our work with Hyrise, we shrunk the Nobench dataset to 100 attributes. Evaluation results for this modified benchmark show that our DVP-generated layout is 1.6x faster than the Hyrise layout. This is because Hyrise is designed for relational data, and it doesn’t take into account data sparseness when predicting cache misses.

6.4 NoSQL Databases

NoSQL databases are designed to address high-performance processing of big data. Among different kinds of NoSQL databases, key-value and document stores have been used to handle semi-structured data and JSON in particular.

Redis (2009) is a main memory key-value store which is implemented in C. Its advantage is support for complex data structures such as hashes, sets, etc. Fast recovery is another advantage of Redis, as it periodically stores snapshots of the main memory to the disk to accelerate recovery time. RAMcloud (2010) is another main memory key-value store. It is a distributed system designed for low latency, high availability and high performance. Both key-value store systems have high performance for simple OLTP queries, as they only need to read or update a few records. However, they don’t support complex queries like joins, and have
Document stores can support more complex data than key-value stores. Since they store the structure of data in addition to the values, they are the most appealing NoSQL stores for JSON data. Amazon’s SimpleDB, Apache’s CouchDB, and MongoDB are the most popular document stores.

As its name implies, SimpleDB (2007) uses a simple data model and only supports simple operations (Select, Delete, GetAttributes, and PutAttributes) on documents. SimpleDB’s data model doesn’t support nested documents, which is a serious limitation for storing JSON data, as hierarchy and nested objects are some of the key characteristics of this type of data.

Apache CouchDB (2008) is a flexible, distributed, and fault-tolerant document store that stores data in JSON format. Its query access is based on views. Different queries can be distributed with map-reduce jobs. However, CouchDB’s view mechanism puts more burden on users. Another downside of CouchDB is that update operations are performed on whole documents. Thus, even if a client wants to modify a single value in a document, he first has to load the document, make the modifications on it, and then send the whole document back to the database.

MongoDB (2009) is the richest document-oriented NoSQL database, written in C++. MongoDB’s data model is BSON (binary JSON), which is suitable for JSON data. MongoDB achieves a high performance by using memory-mapped files for storing data. Therefore, it’s fully a main memory database if the data can fit in RAM. MongoDB supports indexing and aggregation within a single collection. A collection is a set of documents that is equal to a table (a set of tuples) in relational databases. Because it supports secondary indexing and aggregations, MongoDB is the NoSQL document store that gets closest to a relational database. However, MongoDB does not support secondary indexing or aggregations across collections, and it also does not support join queries (which are also cross-collection).

To summarize, even though NoSQL databases have both better scalability and higher flexibility in comparison to relational databases, their performance comes with the cost of weaker concurrency and atomicity properties than ACID. They cannot support complex indexing, analytics, and join queries. They also suffer from lack of a high level standardized language like SQL and finally, they don’t provide good performance for OLAP queries as they are not
Chapter 6. Related Work

designed for read-heavy queries. We believe that by mapping JSON data to main memory relational databases, one can benefit from the high performance of main memory systems, the SQL language, and support of complex queries all at the same time.

6.5 Mapping JSON to Relational Databases

Mapping JSON data to relational databases with the purpose of benefiting from the advantages provided by highly-optimized relational databases, is the best possible approach to handle JSON data while at the same time avoiding the downsides of NoSQL databases.

To the best of our knowledge, the first work that proposes a mapping is Argo (2013) [11]. The main idea in this work is to use generic tables to store JSON data in a relational format. In these generic tables, every attribute of a JSON object is stored as a separate record. Using a generic format has the advantage of processing data without having to provide a schema upfront. However, because it shreds objects to different attributes and stores them as different records, Argo exhibits poor performance when reconstructing JSON objects [15]. Furthermore, Argo suffers from having tables with large numbers of rows while running queries that need to scan entire tables. Our experimental results show that our system is 15x-30x faster than Argo for different workloads.

Oracle has recently published a paper about JSON data management in RDBMSs (2014) [15]. To support JSON format within a RDBMS, each JSON object instance is stored as one aggregate object in a separate column without any relational decomposition. In this design, a JSON object collection is modelled as a table with one column storing JSON objects. Each row in this table holds a JSON object instance in the collection. As there is no object decomposition, the format for storing objects is adapted from document store databases. Therefore, OLAP queries will still have poor performance in this design since they need to extract few values from arbitrary complex objects at runtime.

Sinew (2014) [14] is an SQL system designed for support of multi-structured data. In Sinew’s data storage model, after flattening nested objects each unique key becomes a logical column. To avoid having a table with hundreds or thousands of attributes, only a subset of attributes are transformed to physical columns while the remaining columns make a serialized binary
column in the database. Users see a logical view of the database with all logical columns, and transforming queries from the logical view into queries over the physical layout is accomplished by using a mapping layer.

We believe that OLAP queries over the serialized column will have poor performance, since the query needs to access objects rather than their few columns. Even if the layout is initially designed to store OLAP accessing columns independently, after a change in workload it’s possible to a new OLAP query to appear and access the serialized column. Another possible challenge of such a design is the extra overhead caused by the transformation layer. To conclude, we believe that Sinew’s design is not capable of adapting to workload changes, since changing the columns that are grouped into one serialized column (adding or removing columns) can be very expensive.

To summarize, we believe that vertically partitioning JSON data into different tables is the best approach to achieve high performance for both OLTP an OLAP query types. However, such a system should support a fast layout change and a repartitioning tool to adapt its schema to any changes in the dataset or workload.
Chapter 7

Conclusion and Future Work

In this dissertation, we designed, implemented and evaluated a novel technique for vertical partitioning of the database schema for optimizing performance of main memory databases. To do this, our partitioner considers workload characteristics (e.g., query frequency, selectivity and attributes accessed by each query, etc) towards generating a data layout to minimize TLB and cache misses while running the queries. In addition to performance optimization, our dynamic vertical partitioning algorithm (DVP) enables JSON data storage in main memory relational databases by intelligently decomposing JSON objects into different tables. To efficiently handle JSON data, DVP takes into account semi-structured data characteristics such as attribute sparseness while generating the data layout. Furthermore, our engine can adapt to workload and data changes, meaning that if the workload changes, or if new attributes appear in the data, the partitioner generates a new layout by incrementally refining the current layout. Through this repartitioning, our engine constantly maintains good performance, even in the case of workload changes, and it also supports schema changes, which are very common for JSON data where attributes dynamically appear and/or disappear in different objects.

We use JSON data generated with the Nobench benchmark to evaluate DVP-generated layouts against Argo, a state-of-the-art data model that enables JSON mapping onto relational databases and Hyrise, a state-of-the-art vertical partitioning algorithm that targets minimizing the number of cache misses in main memory systems. Our experiments include three different workloads with different conflict levels between attributes accessed by queries. The first two workloads consist of mainly OLAP queries, while the third one is a hybrid of both OLTP and
OLAP queries. Our evaluation results show that we outperform Argo by 15x-30x in all three conflict scenarios. In order to compare the performance of DVP generated layouts against Hyrise layouts, we shrunk the number of attributes in the Nobench benchmark, because Hyrise didn’t scale to 1000 attributes. Even after reducing the number of sparse attributes, in some cases, Hyrise generated the layouts within minutes, while DVP took at most a few seconds to generate the layouts because of its polynomial complexity. Our evaluation results show that for the first two workloads that have OLAP queries (which don’t access any sparse data), we generate the same layouts as Hyrise. We outperform Hyrise by a factor of 1.6 for the hybrid workload that contains accesses to sparse data.

We also compare the performance of DVP-generated layouts against row-based and column-based layouts, which are standard layouts for relational databases. DVP-generated layouts outperform the row-based layout by at least 2.1x and up to 7x for all three workloads. For the column-based layout, our approach outperforms it by 5% in the first two conflict scenarios, which are OLAP workloads where the column-based approach is known to perform very well. Finally for the third conflict scenario DVP layouts outperform the column-based layout by a factor of 6.

In summary, we made the following contributions in this dissertation:

1. Designed, implemented and evaluated a novel technique for vertical partitioning of the database schema for optimizing performance of main memory databases.

2. Enabled support of JSON data in relational databases by intelligently decomposing JSON objects into different main memory tables and repartitioning the layout in the case of schema changes.

3. Performed a detailed evaluation of the performance of our system, including the partitioning algorithm, and showed its effectiveness in different workloads.

Going forward, our future work will focus on the following areas:

1. Adding features that are commercially available in enterprise systems such as automatic data aging (dumping old data to disk), update logging and memory snapshots for failure
recovery and improved system reliability to our current implementation of main memory system.

2. Support of indexing on different columns and adding compression techniques to our string dictionary table.

3. Implementation of a performance monitor to automatically trigger the repartitioner in the case of workload and/or schema changes.
Bibliography


[48] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazieres, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum,


Appendices
Appendix A

CPU Configuration

Figure A.1: CPU configuration of the machine which is used for running the experiments.