Vrije Universiteit Amsterdam

University Politehnica of Bucharest

Master Thesis

Adaptive on-the-fly compressed execution in Spark

Author: Ionut Boicu (2637444)

1st supervisor:	Peter Boncz	
daily supervisor:	Bogdan Raducanu	(Databricks)
2nd reader:	Animesh Trivedi	

August 19, 2019

Abstract

Modern query execution engines rely on hash tables when working with expensive operators like hash aggregate and join. Often, query performance is determined by the hash table representation and its peak memory size. This project investigates the use of compact, compressed data representations in hash tables to make data access faster. Specifically, in the case of Apache Spark, we propose an optimized design for the small hashmap found in the partial aggregation stage. Subsequently, we discuss two orthogonal on-the-fly data-compression approaches to reduce the size of the hash table records. We present an adaptive bit-packing technique that compresses the numeric data tightly, using outlier-free metadata, generated at runtime. Moreover, we explore Strings compression, by building dictionaries that map Strings to codes and, further, work with the codes inside the aggregation. For String compression, we restrict the scope of the project only to aggregations that feed off a Parquet scan, so that we can leverage the Parquet `dictionary_encoding` and build the dictionaries in sublinear time. Our experiments show that we can speed up query execution performance up to 5x by exploiting the `dictionary_encoding` from the Parquet files. We demonstrate the benefit of compressed execution on industry standard TPC-H *queries 1* and *16*, for which we observe a maximum speedup of 1.9x.

Contents

Chapter 1

Introduction

1.1 Context

Modern query execution engines use in-memory hash tables for operators like hash joins and hash aggregates. These hash tables can get large enough to become dominant factors of memory consumption, in query processing. Often, hash tables exceed the CPU cache size, in which case the memory latency can slow down query execution. If we reduce the hash table memory consumption, we can have better CPU cache locality, therefore the query execution performance could improve. While we do not focus on this here, compressing hash tables reduces memory consumption, such that tasks that would fail or spill, can work, in a single pass. In this thesis, we intend to apply compressed execution to speed up the Apache Spark (1) query execution engine, through hash table records size reduction.

1.2 Objective

Our objective is to employ compressed execution in Spark and reduce the hash table memory consumption. We propose to use an adaptive, on-the-fly bit-packing technique to compact hash table records. Compressing each row can accelerate aggregates and joins due to less computational effort and better cache locality. For this approach, we require data statistics, such as min-max information, to determine the smallest amount of bits needed for the representation of each value, in a record, and then pack multiple of these values into a single word. A first problem is that such statistics are unavailable, especially in big data environments. A second problem is that although specific and popular storage formats, such as Apache Parquet (2), may contain data statistics, these are susceptible to outliers. Very large datatypes are forced to be used when such outliers are present, even if the rest of the values can do with a smaller one. To overcome these problems, we infer our own adaptive data statistics, at runtime, where we have an overview of the values and their distribution. Having statistics that deal with outliers implies that not every row can be compressed using them. Therefore, our approach should be adaptive - bit-pack most data tightly; but be able to fall back to the initial design, in the case of outliers.

Strings are one of the most common types found in real applications (3, 4), even though, compared to Integers, they are larger and slower to process. Since bit-packing techniques cannot be applied on String data, we propose on-the-fly partial dictionaries that map numeric data to Strings. Apache Parquet supports dictionary compression, but uses a separate dictionary for every chunk of data (row group). This chunk is justified in order to allow parallel bulk loading and dictionary encoding. Parquet files neither have *unified* dictionaries nor they encode all data in a chunk if the dictionary grows too large. Compressed execution, needs a *unified* dictionary for all data from a column, such that the same *number* is used to encode each particular value. Therefore, partial dictionaries in Parquet row groups need to be unified on-the-fly, during query processing, in order to enable compressed execution. We investigate this compression and build the *unified* String dictionaries in sublinear time by leveraging the Parquet Dictionary encoding scheme. Further, we use the partial String dictionaries to work with encoded Strings in the aggregate operator and achieve improved query execution performance. In this thesis, we compress numeric primitive data types using bit-packing and Strings using the Parquet dictionaries, when available. Additionally, we instrument Spark to create the compression and work with this compact data inside the hash table.

Apache Spark uses just-in-time (JIT)(5) compilation, allowing code compilation at runtime. Moreover, Spark uses the whole-stage code generation (WSCG) to improve the execution performance of a query by fusing its operators into a single optimized function (a single Java file), eliminating virtual calls and leveraging CPU registers for intermediate data. Our goal of compressed execution is made more challenging because Spark uses a just-in-time compiled engine while the data statistics only become available at runtime. Thus, the code for our compressed execution must be generated ***before*** the statistics are known. Generally, it is an open research challenge how to combine JIT with adaptive query execution, because adaptivity is predisposed to explode the size of the generated code. Therefore we focus on finding data statistics to have some knowledge about the data and, consequently generate a very specific code to compress the data at runtime and work with it in a compact format, thus reducing the hash table memory consumption. However, WSCG optimizes the query before the metadata is generated. Therefore, we can not produce very specific and optimized code for compression, unlike the Parquet files situation in which we can obtain the metadata before WSCG. In this project, we solve this challenge by speculatively generating a limited variety of code paths that are fast but allow limited parametrization, such that we can still adapt the compressed hash table to the run-time statistics when they become known, after WSCG. The innovation in this work is robust compressed execution, using on-the-fly gathered statistics, and integration into the whole-stage code generation (WSCG) of the hash aggregate operator and implicitly into the query execution engine.

1.3 Related-work

The orthogonal approach of minimizing the size of a hash table, by increasing the loading factor, has been tackled by Cuckoo Hashing (6) and Robin Hood Hashing (7). Both of the approaches, for the hash table size reduction, can be used together to achieve higher size reduction ratio in hash tables. However, in this project, we explore only the hash tables records compression.

In the current state of the art, statistics and meta-information are used to tackle the problem of using compressed data during execution. Graefe et al. (8) compress attributes individually, employing the same compression scheme for all attributes of a domain, and then operate on this format until decompression is necessary. Abadi et al. (9) are among the first that mention the term ”compressed execution” and show how to integrate this method in the query engine. Their approach is to classify compression schemes to a basic set of properties and introduce an API, that abstracts from the different schemes, to extend the query operations that work directly on compressed data. Other systems, such as SAP HANA (10), Teradata (11), and Vectorwise (12) manipulate the min/max metadata for optimizations during query evaluation and in data storage. However, in Spark, the only data statistics available are the ones kept in Parquet files (2) or the ones automatically gathered by Delta Lake (13), in its transaction logs. Since this stored metadata might contain outliers, leveraging it can force our approach to mismatch the optimal compression strategy. Therefore, in this project, we compute our own statistics that deal with outliers and investigate on-the-fly compression inside hash tables.

Regarding the data compression, we examined the research related to compressing specific data types. Numeric values are amenable to different techniques, such as bit-packing (14), byte-slicing (15), bit-weaving (16). Funke et al. (17, 18) propose utilizing the technique of separating commonly used strings from infrequent strings, into hot and cold regions, to improve the cache locality. Moreover, String data can be de-duplicated using dictionaries (19, 20, 21). The dictionaries work as a translation between unique strings of a column and integer codes. It happens that the strings have a large distribution of values that cannot be mapped to integer codes altogether. Apache CarbonData (22) implemented one approach to global dictionaries, but it is expensive and requires data pre-computation. Carsten Binnig (23) and Franz Farber (24) explore strings encoded inside dictionaries, but they assume the existence of global dictionaries. In this project, we leverage the Parquet `dictionary_encoding`, when available, to create a global String dictionaries and work with compressed Strings in the aggregation.

1.4 Research Question

In our goal of investigating whether query execution performance can be increased by employing compressed execution, we try to answer the following research questions:

Q1: How do we efficiently build on-the-fly String dictionaries?

- How do we determine, at runtime, which String columns are dictionary encodable?

Q2: What kind of data statistics do we need to obtain before compressing the data on-the-fly?

- How do we deal with outliers?
- What is the overhead of detecting them?

Q3: How do we adaptively compress multiple columns together into fewer machine words?

- Which columns should be selected for compression?
- How do we make our approach effective and still robust?

Q4: How do we integrate our design into Spark whole-stage code generation?

- What compressed aggregation code should we generate before the runtime data statistics are known?
- How do we limit the number of efficient code paths to keep code generation size and compilation time small, while still providing enough flexibility to adapt to runtime data distribution information?

Q5: How do we evaluate our system's performance?

1.5 Thesis Outline

In this chapter, we set up the context for our project, presented the current related work, briefly mentioned our contribution and laid out the research questions we investigate in this project. In the following chapter, we provide background information related to the specifics of Spark, Parquet, Tungsten, and Databricks Delta. In the third chapter, we show our early data science findings, from studying Databricks workloads. Chapter 4 covers the design of the hash table that will operate on compressed data and presents an optimized design for this hashmap. Further, in Chapter 5, we introduce the String compression approach, while in Chapter 6, we highlight the application of the bit-packing technique. At the end of each of these three chapters, we provide the performance evaluation of each contribution and experimentally motivate our choices. The last chapter summarizes our findings, answers the proposed research questions and presents future work.

Chapter 2

Background

In this section, we introduce an overview of the Apache Spark architecture, followed by some of the Tungsten Engine components, and explain how the hash aggregate operator works. Then we provide background information required to understand our compressed execution design by explaining the specifics of Apache Parquet, the UTF8String format, and the Databricks Delta Lake system.

2.1 Apache Spark

Spark(1) is an open-source cluster computing framework for distributed data processing. A Spark application runs as a set of executor processes and a driver process. The driver is located on a single node in the cluster, called the driver node, while multiple executors run on worker nodes. The program execution is realized in tasks, the smallest form of computation that the driver delivers to the executors. Spark's core relies on the concept of resilient distributed datasets (RDDs (25)), which are immutable collections of records, horizontally partitioned, that support parallel processing on each partition.

Spark SQL and the DataFrame API are higher-levels of abstraction, which provide automatic query optimization over RDDs. The DataFrames allow expressing operations on data through manipulation of tables, rows and columns, while Spark SQL handles operations expressed through SQL queries. Regardless of the choice between Spark SQL or Dataframe API, the queries benefit from the same optimizations compared to the manually written RDDs that do not get optimized. The query optimization is done by Catalyst, a rule-based framework that works with query plans; trees of relational operators from a structured query. Catalyst optimizes queries in four phases: (i) the analysis of a logical plan to resolve references, (ii) logical plan optimization, (iii) the choice of an optimized plan using cost-based heuristics (26), and lastly, (iv) the code generation stage that compiles parts of the query to Java code. Both Spark SQL and DataFrames are eventually executed as operations on RDDs.

2.2 The Tungsten Project

The Tungsten project was a major change brought to the Apache Spark 1.5 execution engine, with the goal of optimizing Spark jobs, for memory and CPU efficiency. Recent Spark analysis (27) showed that the CPU is the new bottleneck, as opposed to network and disk I/O operations which are considered fast enough. Three different techniques are used in the scope of optimizing CPU throughput: (i) off-heap memory management, (ii) cache-aware optimizations, (iii) code generation. Further, in this subsection, we describe the off-heap memory management and the code generation as these features are overseen in this project.

2.2.1 Memory Management

In the JVM, applications typically rely on the Garbage Collector (GC) to analyze unused objects and safely manage memory. However, GC overhead becomes non-negligible when the number of JVM objects grows rapidly, a circumstance that happened often in Spark because it can store large amounts of data, represented as objects, in memory. To tackle the Java objects dependency and overhead, the Tungsten Engine uses the JVM internal API from `sun.misc.Unsafe` to allow explicit memory management without safety checks. Besides using this API for direct memory access to bypass the JVM in order to avoid garbage collection, it also features off-heap memory management using binary in-memory data representation. In Spark, the off-heap memory management is turned off by default (`spark.memory.offHeap.enabled`) and requires manual input for the allocated size, through the `spark.memory.offHeap.size` parameter, because off-heap memory can lead to OOM problems if normal JVM allocations run close to the machine resource limit already.

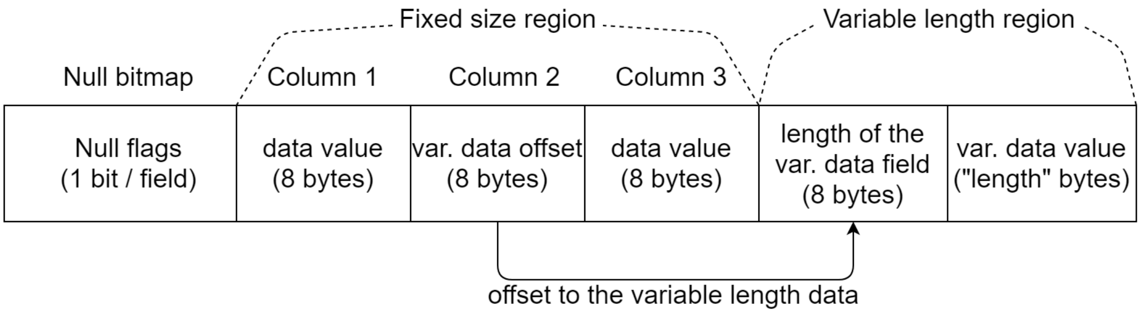

Fixed size region
Variable length region
Null bitmap
Column 1
Column 2
Column 3
Null flags (1 bit / field)
data value (8 bytes)
var. data offset (8 bytes)
data value (8 bytes)
length of the var. data field (8 bytes)
var. data value ("length" bytes)
offset to the variable length data

Figure 2.1: The UnsafeRow format

The Tungsten Project introduces an in-memory format that leverages the sun.misc.Unsafe API, called UnsafeRow and has an array-based internal structure, 8-byte word-aligned, where all data is stored as bytes. This representation is substantially smaller than objects serialized using Java or even Kryo serializers. The UnsafeRow structure, shown in Figure 2.1, has three parts: the null bitmap, the fixed-size region, and the variable-length region. First, the null bitmap is used to tell whether a field in the row has a valid or a null value, utilizing 1 bit for each column. Second, the fixed-size region allocates 8 bytes for each

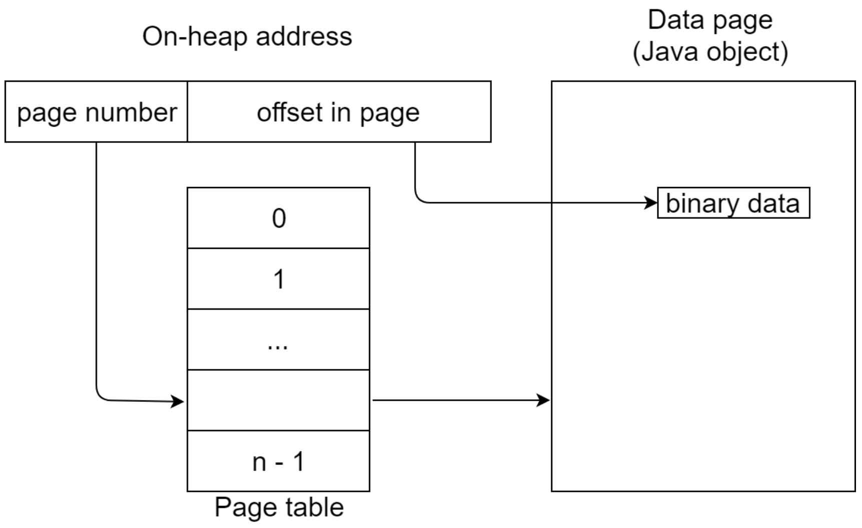


Figure 2.2: Address encoding in on-heap memory

column and stores either the inlined values for small data types or relative offsets into the variable-length section otherwise. UnsafeRows are always using 8 bytes for a column in the fixed-size region, the null bitmap and the variable region are multiples of 8 bytes, therefore the rows are 8-byte word-aligned. Even if the null bit is set for a column, its corresponding field in the fixed-size region still exists. Knowing that the embedded data is 8-byte aligned, the UnsafeRow manages it using byte-wise operations and byte-to-value conversions. Moreover, equality comparison and hashing can be performed on the raw bytes without requiring additional type-specific interpretation.

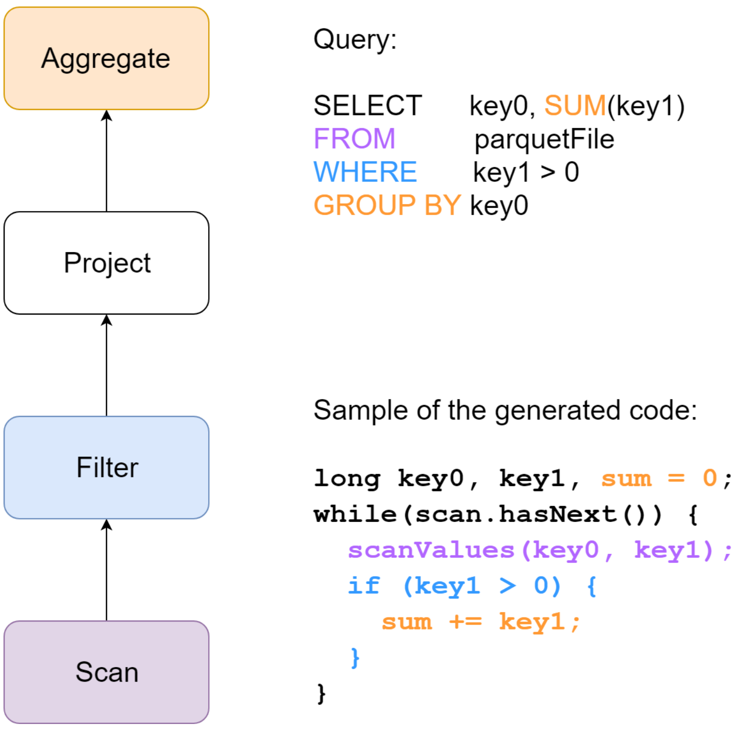
2.2.2 BytesToBytesMap

On the one hand, in on-heap memory, locations are addressed using a pair of a base reference object and an offset within this region of memory. On the other hand, when using memory that is not allocated by the JVM, off-heap, addresses can be referenced using 64bit raw memory pointers. Therefore, Tungsten uses an in-house "page table" abstraction that enables a more compact encoding of on-heap addresses by combining the two available methods. The addresses are encoded in the on-heap mode as a pair of a page number, using the upper bits of the address, and an offset in that page using the lower bits. The "page table", is a binary array that contains Java objects from the off-heap memory and is used to encode an on-heap address. This is done using the page number as an index in the page table, that gets the base object (memory page), and the offset to index into the memory contents and read or write at that location. In the Figure 2.2, we present a high-level overview of the on-heap address encoding.

These building blocks are used to create an off-heap hashmap which is used as a primitive inside the aggregation operator. The java.util.HashMap has object overhead, therefore the need for a hashmap with low space overhead and better memory locality, for a faster key lookup. In Project Tungsten, the off-heap memory management technique is leveraged to build a cache-efficient binary hashmap, called BytesToBytesMap. At a very high level, this data structure consists of an array which stores both the hash code of the object in the hashmap and a pointer to the key. This pointer is an offset, encoded with the addressing approach previously mentioned, into a memory page. This memory page contains a contiguous stream of keys and values, both represented in the UnsafeRow format. Therefore, the benefits of this in-house hashmap are lower space overhead because of minimizing the number of Java objects and their overhead, an excellent memory locality for a scan due to the juxtapositional hashmap records. closer to hand-written code (28).

2.2.3 Code generation

In the JVM, generic expression logic evaluation is expensive, because of virtual function calls, branches based on expression types and auto-boxing of the primitive data types. Neumann proposed, in (28), efficient query compilation into native machine code for *data-centric* processing; keeping the data in the CPU registers as long as possible. In his approach, the data-centric model is used to push the data to the operators, rather than letting them pull it, resulting in a better code and data locality. The data-centric model was used as a foundation for Tungsten's *whole-stage code generation*, a technique that compiles the performance-critical parts of a query to Java code. Therefore, Spark removes the generic expression overheads using the whole-stage code generation model to dynamically generate code that achieves performance



Query:

```
SELECT     key0, SUM(key1)
FROM       parquetFile
WHERE      key1 > 0
GROUP BY key0
```

Sample of the generated code:

```
long key0, key1, sum = 0;
while(scan.hasNext()) {
  scanValues(key0, key1);
  if (key1 > 0) {
    sum += key1;
  }
}
```

Figure 2.3: Whole-stage code generation fusing multiple operators in one file of code

Project Tungsten introduced, in Spark 1.5, the code generation for expression evaluation in SQL and Dataframes. In this process, a SQL expression is transformed into an abstract syntax tree (AST) for Scala code, that can be fed to the Spark query optimizer, called Catalyst (29). The optimizer is based on functional programming construct in Scala and evaluates the AST, improves it using cost-based heuristics and generates Java code at runtime. Compared with interpretation, the generated code reduces the boxing of primitive data types and avoids polymorphic function calls. In addition to this, Project Tungsten uses the Janino compiler to reduce the code generation time. In Figure 2.3, we show an example of how the whole-stage code generation fuses multiple operators in one file of code.

2.2.4 UTF8String format

UTF-8 (30) is a variable width character encoding defined by the Unicode Standard , that is able to encode all of the valid code points in Unicode using at most four 8-bit bytes. It was designed such that characters that occur more frequently to be encoded using fewer bytes and have lower numerical values. This way the first 128 characters of Unicode correspond one-to-one with the ASCII codes and are encoded using a single byte that has the same values in both ASCII and UTF-8 encoding. The latter has been the dominant encoding, and in the present, it accounts for more than 90% of all the web pages.

Spark uses internally the `UTF8String` format for Strings representation. Thus a string is encoded in UTF-8 and expressed as an array of bytes. Besides the underlying array that stores the bytes, the format has two other properties: an offset and the total number of bytes. They can be used together to create a virtual pointer inside the array and reference a any substring. These properties are leveraged, in our project, as part of the partial Strings compression.

2.3 Hash aggregate operator

Hash aggregate is a unary physical operator for aggregation, created when the logical query plan is transformed into a physical plan and the planning strategy selects the hash-based operator for the aggregation. With the introduction of the Tungsten Project, this operator got optimized by utilizing the code generation, the BytesToBytesMap, and the UnsafeRow format. In Figure 2.4, we can see how the flow of an optimized aggregation. First, the aggregation operator is going to consume a stream of input rows, that are projected according to the grouping keys. Second, this projection is converted to the optimized UnsafeRow format that is used for probing inside the hashmap. As mentioned in subsection 2.2.2, the BytesToBytesMap stores both the key and the value as a stream of bytes, the UnsafeRows, in the assigned memory page. Finally, once the key lookup provides the location in the hashmap, an update of the serialized data is triggered, using the value stored (partial aggregate result) at that location and the values present in the input row. Therefore, additional objects are not needed to compute the values. Once the aggregation finishes, the final values, resident in BytesToBytesMap, are ready to be scanned in a cache-friendly manner, by just reading the hashmap memory pages and leading their content to the next downstream operator.

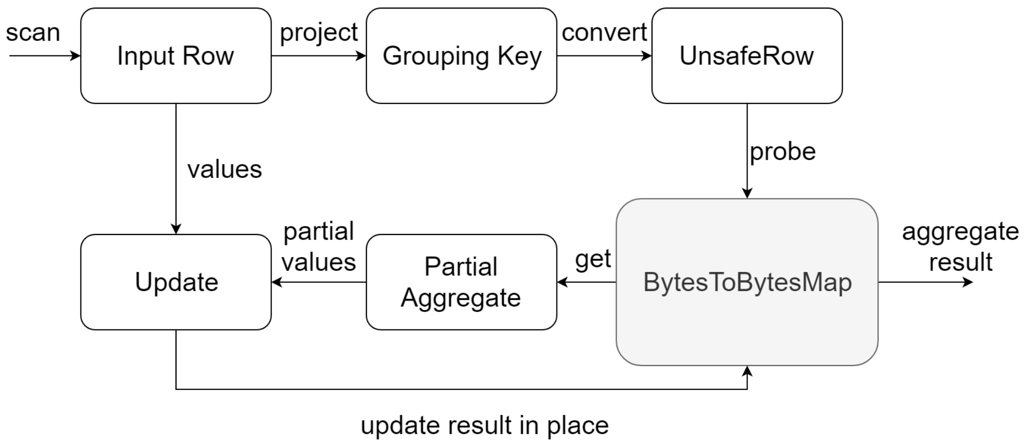


Figure 2.4: Flow of the Hash Aggregation

2.4 Apache Parquet

Apache Parquet (2) is an open-source compressed columnar storage format, that was designed to be used with online analytical processing systems. This columnar format provides better compression and throughput, by aligning and storing each column in its own chunk. The format supports both flat and nested data, and stores encoded metadata at different organizational levels of granularity, such as row groups, column chunks, and pages. The columnar storage format saves both IO and CPU cycles that would have been wasted from reading the rest of the unnecessary fields. Usually, Parquet datasets can be stored in multiple directories and files, following specific partitioning schemes, but are referred to using the root directory. The root sub-directories define a partitioning scheme where each directory represents a partition. For example, one partition can be a subset of data with a specific value or even the whole column. This way some of the data can be skipped using partition pruning.

The internal structure of one individual Parquet file consists of one or more row groups. Each row group is a logical horizontal partitioning of the data into rows and subsequently contains a vertical partition (a piece of data), called column chunk, for each of the columns within the rows. While row groups have no guaranteed physical structure and have a default size of 128MB, the column chunks live in a particular row group and are confirmed to be contiguous in a file. The actual data is stored in data pages inside a column chunk, where each page has default storage of 1MB. In Figure 2.5, we can see the logical organization of a Parquet file. In addition to this, we can identify in this diagram the data statistics at each level of granularity. The purpose of this metadata is the same as the one that partitioning has, to facilitate data filtering and reduce the amount of data that needs to be processed by a query. The data statistics kept at the data page level are the value count, minimum and maximum values for each column within that page. The upper levels of granularity, contain the aggregated values of the data statistics meaning the sum of all the count values and min/max values for each data page within a column chunk, respectively for each vertical partition within a row group.

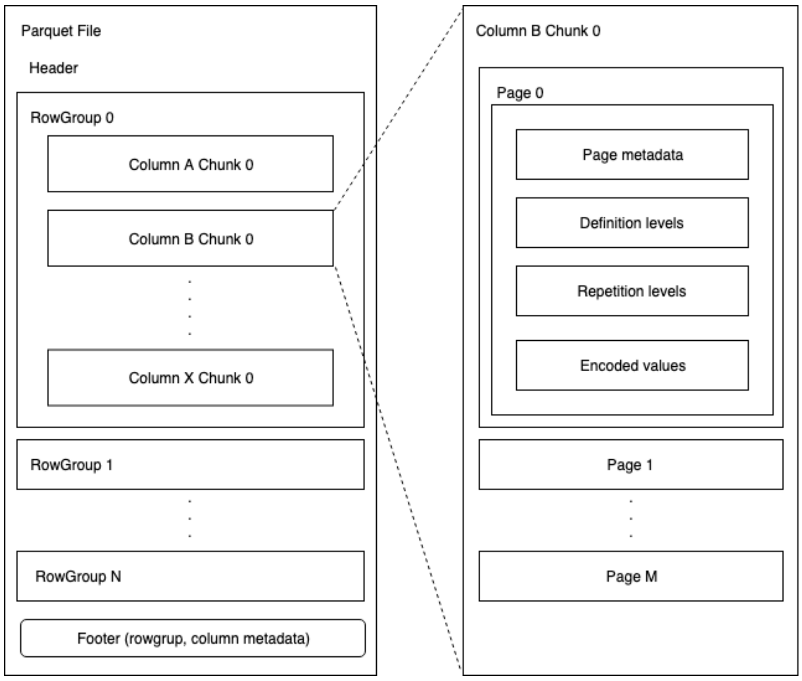


Figure 2.5: Apache Parquet organizational overview

In addition to the stored metadata, the Parquet format features several encoding schemes to store the data values. In the current design, there are six featured encoding schemes: `PLAIN`,`DICTIONARY`, `RLE`, `BIT_PACKED`, `DELTA_BINARY_PACKED`, `DELTA_BYTE_ARRAY`. Further, we explain the `PLAIN` and the `DICTIONARY` encoding, since these are prevalent in today's Parquet files and are the only ones used in our compressed execution design. First, the plain encoding is intended to be the simplest encoding, where values are written back to back. This encoding is used whenever no other more efficient can be used. Second, the dictionary encoding creates a dictionary, for each column, with all of the unique values encountered in that column. Mainly, this encoding relies on the hypothesis that values are repeatedly occurring in a dataset. A dictionary is created containing all the distinct values and then the values in the dataset are substitute with an entry in the dictionary. The references are stored as 32-bit Integers that serve as indexes inside the dictionary. The dictionary is stored in a special dictionary page per column chunk. Hence, in a row group, there can be at most the number of column chunks dictionary pages, one for each distinct column in a row. If the dictionary grows too big, either in the number of distinct values or in size, the encoding will fall back to the `PLAIN` strategy. The default threshold is set for 100.000 distinct values and a maximum size of 1MB per dictionary page. In Figure 2.6, we can see a high-level overview of how the dictionary encoding works in Parquet. The first dictionary encoding technique, called `PLAIN_DICTIONARY`, is the naive approach in which the values from the dataset are replaced with indexes in the dictionary. The second encoding approach is called `RLE_DICTIONARY` and it is based on `PLAIN_DICITONARY`, on which the Run Length Encoding (RLE) is applied. Therefore, instead of having an index for each value, multiple consecutive duplicate values are encoded using only one index and a number that indicates the number of repetitions for that index. In the Parquet 2.0 spec-

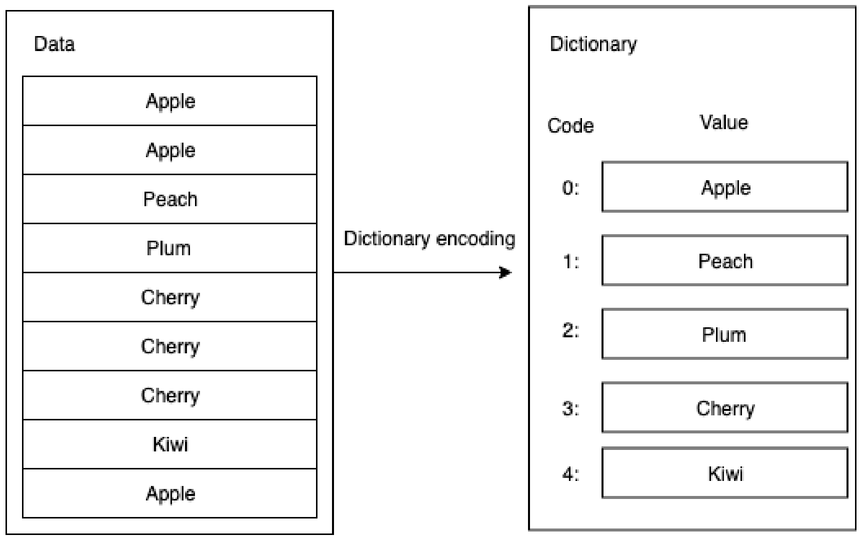


Figure 2.6: The Dictionary encoding in Parquet

ification, both of these techniques are used for the `DICTIONARY_ENCODING`, while for the Parquet 2.0+ files it has been decided that RLE is preferred for data pages and PLAIN for dictionary pages. In Spark, the 2.0 specification dominates and data is either `PLAIN` or `DICTIONARY` encoded.

2.5 Databricks Delta Lake

Databricks (31) was founded by the original creators of Apache Spark. This company offers a platform for deploying Spark clusters in cloud environments, featuring a web interface for writing, running Spark applications and deploying them on provisioned clusters. In addition to this, the platform uses an in-house optimized version of the open-source Apache Spark, for its applications.

In late 2017, the company introduced Delta Lake (13), a data management system that simplifies and combines large-scale data management with the reliability of data warehousing and the low latency streaming. It runs over Amazon S3 and stores the data in open-source formats like Apache Parquet and provides several extensions, such as ACID transactions and automatic data indexing over the transactions. Delta Lake stores its data in Parquet files and, to provide ACID transactions, it uses a transactional log, called Delta Log, to keep track of all the changes made to a table. Compared to the open-source version of Delta Lake (32), the one in Databricks automatically collects data statistics such as min-max information and stores them besides each commit, in the log. This metadata is later used for data skipping (33), at query time, to provide faster queries. Even though the Delta Lake data is stored in Parquet files, the data statistics stored in the Delta transactional log are course-grained and more accessible, being stored per file, compared to the finer-grained metadata stored in Parquet, at different levels of granularity.

Chapter 3

Data science

Our project started with a micro-benchmark experiment, in Spark, in which we tested how queries perform when applied on compressed data. We measured the total execution time needed by Spark SQL for aggregation with three keys (2 integer columns and one string column with a small number of distinct values) in comparison with another aggregation that uses the same keys but compressed into a single numeric column (strings were mapped as integer codes). In this experiment, we observed speedups between 1.4 and 1.8x, when varying the aggregation function. This provided early indication that compressed execution can speedup aggregation. But, how often does this opportunity arise in practice?

Furthermore, we present the results of our data science, performed on data collected during a short period, on selected anonymized workloads of some Databricks customers. We instrumented the workloads to collect statistics on the behavior of aggregations, never collecting values, but only detecting trends. Specifically, we gathered information on what data types where used as group-by keys. Here, we observed that String and integer data types are the most commonly found. In Figure 3.1 we show the chart with the data types distribution from the perspective of the number of rows processed, while in Figure 3.2 we processed it by giving to each data type the same weight.

Moreover, We instrumented Databricks Spark to collect relevant insights into the restricted characteristics of the value distributions used as group-by keys and aggregate values. Regarding columns of the integer data type (stored in Spark as an 8-byte integer), we observed, in figure 3.3, that in more than half of the cases, the numbers would also have fit into a single byte (i.e. their value is between 0 and 255). This indicates that users tend to use data types that are much bigger than what is actually required by the data. Based on the previous findings, we approximate the number of queries that would benefit from a possible bit-packing technique. Accordingly, in Figure 3.4, we notice that 68% of the sampled queries have at least 2 columns that can be squeezed together into one-word keys. Note that four is the average number of columns found in the group-by operator.

Overall, our data science on real customer workloads tells us that a significant percentage of aggregate queries (#queries) and aggregated data volume and time (#rows) involve string and integer columns. These string and integer columns very often have limited domains that could be represented much more compactly than Spark does by default. More than

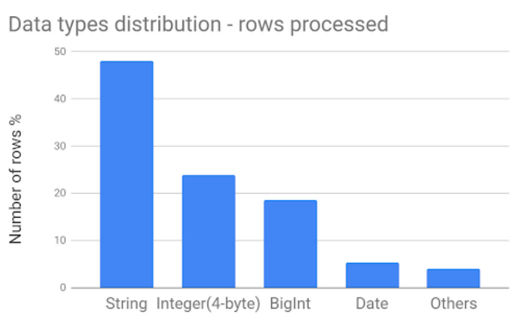


Figure 3.1: Data types distribution in the aggregate group-by operator, based on the number of rows processed by the query

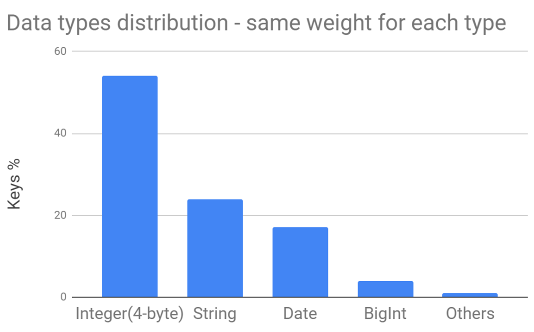


Figure 3.2: Data types distribution in the aggregate group-by operator; based on the total number of keys (each key has the same weight)

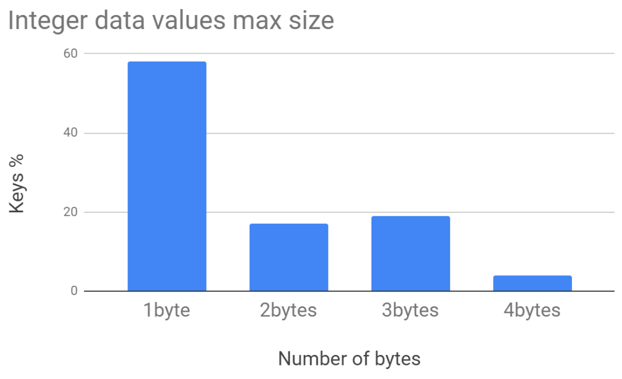


Figure 3.3: The number of bytes required by the Integer data to store their values

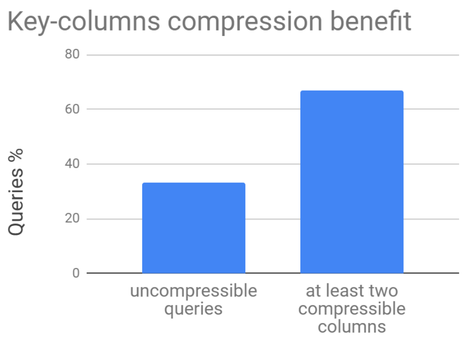


Figure 3.4: The number of aggregate queries that may benefit from Integer data compression

half of the queries in the Spark workload provide such optimization opportunities.

In order to preserve the privacy of the data of the Databricks customers, there are strong limitations to what can be profiled. Still, without requiring insight into the individual queries or users datasets, data science allows us to conclude that the techniques we set out to develop have strong applicability in real use cases.

Chapter 4

Small hashmap

In this chapter, we provide the knowledge required to understand the small hashmap that Spark uses during the partial stage of the hash aggregation, and cover the design of an optimized hashmap that is to become the base for compressed execution. First, we present the current implementation of the small hashmap, followed by an analytical discussion about the current design. Then, we describe a new design for the small hashmap and motivate our decisions using specific benchmarks.

4.1 Small hashmap design

The small hashmap was introduced as an optimization layer for the partial aggregation. It supports only non-null primitive data types and is much faster than the normal map, being designed as an append-only row-based hashmap that can act as a cache for extremely fast key-value lookups, while evaluating certain aggregation functions. The small hashmap introduction into the aggregate operator is depicted in Figure 4.1.

In Spark, the small hashmap stores the key-value entries as a batch of rows, in a contiguous memory region (memory page). Both keys and values are stored as `UnsafeRows`, therefore each record contains two `UnsafeRows`: one for the key and one for the value. The batch of

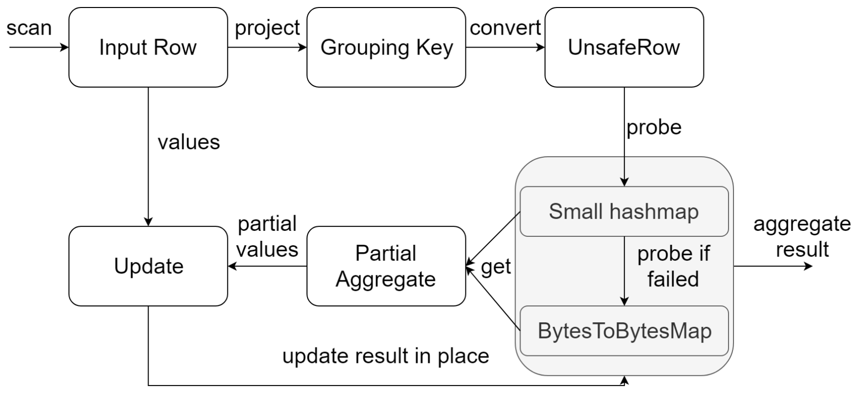


Figure 4.1: Small hashmap inside the Hash Aggregate operator

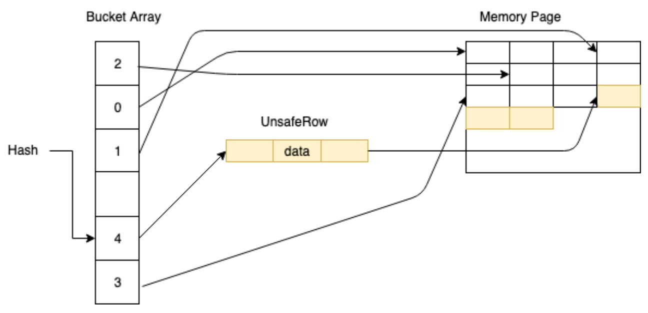


Figure 4.2: Current small hashmap design

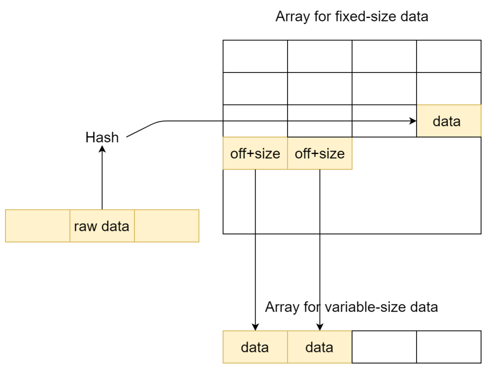


Figure 4.3: New fast hashmap design

rows is backed by a single memory page, that ranges from 1 to 64MB, depending on the system configuration. Whenever the page is full, the aggregate logic falls back to the second level hashmap, the BytesToBytesMap. The maximum capacity for the small hashmap is configurable through the `spark.sql.codegen.aggregate.fastHashMap.capacityBit` parameter. The default value is 16 bits, thus a total of 2^{16} records can fit into the small hashmap. Even though the maximum capacity of the small hashmap can be modified, the memory allocated is limited to the 64MB memory page. This one-page design is used to simplify the memory address encoding and decoding for each key-value pair.

The *load factor*, of a hash table, is the number of unique keys stored in the hashmap divided by its capacity. Two keys collide in a hash table if the hash function maps them to the same index. The higher the hashmap capacity, the less probable it is to have a collision between two different keys. However, in the small hashmap design, the capacity of the memory page is equal to the number of unique records that can be stored. To overcome the problem of requiring another memory page for a lower load factor, the current design has a bucket chained indexing for the records. This bucket array is a level of indirection for the memory page, used to virtually increase the capacity of the hash table. In the current version of Spark, it has twice the number of records in the memory page to support a load factor of 0.5 (fixed in implementation). In figure 4.2, we illustrate the small hashmap components and design.

This small hashmap is faster than the BytesToBytesMap not only because it is code generated, but also because it benefits from a few improvements, such as (i) an optimized underlying data structure for the batch of rows, (ii) row caching and (iii) a faster hash function. There are two implementations for the underlying batch of rows, based on the existence of variable-size datatypes. On the one hand, when dealing only with fixed-size data types, the batch of rows is used in an optimized form (i), where the logic for the rows offsets, inside the batch, is built on an arithmetic computation. In this implementation, the length of a record is equal to the sum between the lengths of the key-value `UnsafeRows` and 8 bytes used for the pointer to the next record. On the other hand, when dealing with at least one data type that has variable length, the batch of rows uses a generic implemen-

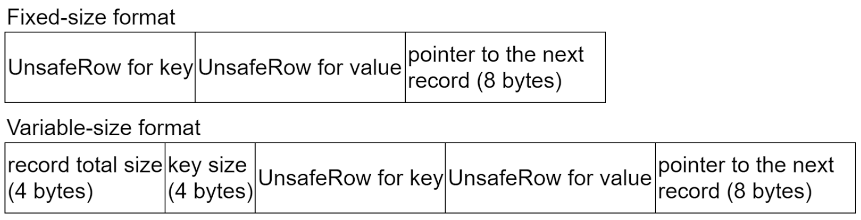


Figure 4.4: Records format in the small hashmap underlying batch of rows

tation, where additional metadata is stored for each row. For example, one record, in the generic implementation, keeps 4 bytes for the row total size, 4 bytes for the key length, the bytes for the `UnsafeRow` (key-value pair), the 8 bytes that point to the next record and another 8 bytes for the offset inside the memory page. In Figure 4.4, we can see the format for the records inside the fast map and the need for the extra 8 to 24 bytes for metadata.

The lookup, in the small hashmap, returns the row indicated from either calculating or retrieving the offset, based on the batch implementation. This operation is optimized (ii) by saving a reference to the last row accessed and returning it if the same key is used for multiple consecutive lookups. When either inserting or looking up a key, the data is converted to `UnsafeRow` and, later, this is copied to the memory page or used for rows equality checking. Unlike the BytesToBytesMap, that uses Murmur hash function for each datatype, the small hashmap uses it only when dealing with strings, while otherwise (iii), it uses a function that contains a few arithmetic operations such as multiplication with a magic number, sums, bit shiftings, and xors.

Even though this small hashmap is faster than the normal hashmap, it only works with primitive datatypes and does not support nullable keys. Besides this, it is limited to a maximum of two consecutive collisions for the same key. If the key is neither found nor inserted after two collisions, the small hashmap automatically falls back to the BytesToBytesMap.

4.2 Discussion

The goal of the small hashmap is to act as a cache for extremely fast key-value lookups. However, we identified, in the current design, some inefficiencies with respect to cache friendliness and performance:

- Linear hash tables normally have the advantage over bucket-chained hash tables, because only a single cache miss is generated when accessing them. But the small hashmap is not designed as a truly linear hash table, therefore it will generate two cache misses because of the bucket array.
- The management of records is done using the `UnsafeRow` format. It is not a very efficient format as it supports nulls[1], even though the small hashmap does not, and wastes at least (1 bit for each column and it is 8-byte aligned) an 8-byte integer on that. Furthermore, each column, in the `UnsafeRow`, uses 8-byte integers for all

[1]if a column is null, it still requires storage in the `UnsafeRow`, which is also not memory efficient

datatypes smaller than 8 bytes. A record, in the underlying batch of rows, uses two `UnsafeRows`, one for the key and one for the value, introducing additional overhead and one more cache miss. According to the batch of rows design, records store additional internal information, ranging from 8 to 24 bytes, for fixed-size, and respectively, for variable-size `UnsafeRows`. This repeated information for every row is again wasting space. Last but not least, the data embedded in `UnsafeRows` is accessed using byte-wise operations, increasing the data management overhead.

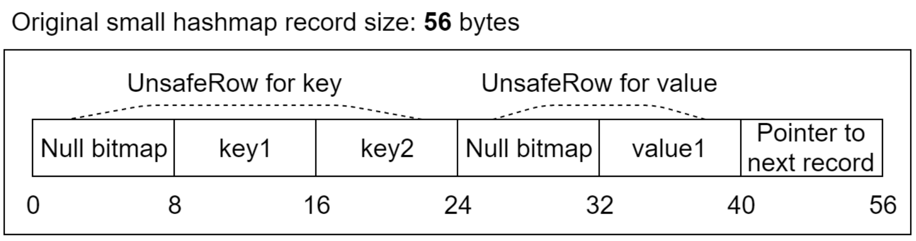
- The murmur hash function is used when dealing with String data in the small hashmap. While Murmur is good general-purpose hash functions, it is quite slow for this hashmap that has a limited capacity. Because it is small, the hashmap will eventually get filled even when using a fast but worse hash function. If the hashmap is full and new keys that contain Strings are coming in, the Murmur hash function is computed twice for both of the hashmaps, slowing down the query execution.

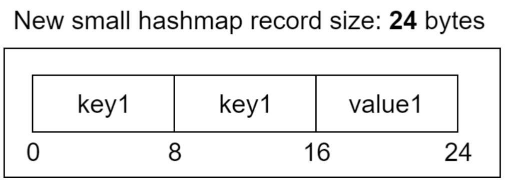
4.3 Optimized small hashmap design

Since the current small hashmap design is memory wasteful, we introduce a simpler and optimized design. It is implemented as a true linear hash table, without any levels of indirection, using a row-wise layout. Unlike the column-wise layout, that generates a cache miss for every column, in the row-wise layout, a hash table record often takes much less space than a cache line, so there can be only one cache miss generated for a lookup. The new hashmap design achieves compaction and evades the JVM memory management, being built as only two large 8-byte integer arrays. The structure of these arrays is similar to the `UnsafeRow` format, where datatypes that use at most 8-bytes store their values directly in the first array (for fixed-size data), while the others store only a reference (offset and length) that points to their value location in the second array (for variable-size data). Strings are the only variable-size data that the small hashmap supports and the only ones stored in the second array. We use an 8-byte representation for Strings because we think that manipulating longs is faster than manipulating bytes. The longs are mapped on machine words, hence, they can be manipulated naturally and fast by CPUs. Moreover, Strings hashing is slightly faster, for more than 8-byte long Strings, if done one long at a time, instead of the original approach that works on 4 bytes. Besides the new data structure, we use a fast hash function which we use on all the supported datatypes (Strings included). The simplified layout of the new small hashmap is available in Figure 4.3.

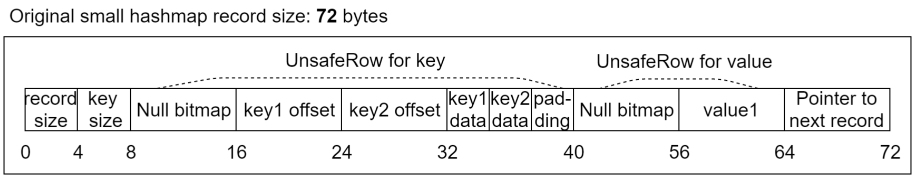
Even though the underlying data structure of the new design looks similar to the memory-wasteful `UnsafeRow` format, we conduct an in-detail study for both hashmaps, and show how the old one wasted space. For this experiment, we do a bytecount per record for both small hashmap designs. The actual values of the columns are unnecessary for the count and we require only the length of the variable datatypes.

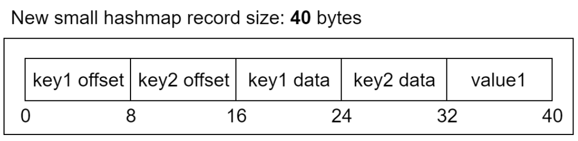
- First case study: two 4-byte integer keys and one 8-byte integer value





- Second case study: two String keys of 3 and, respectively, 4 characters long; and one 8-byte integer value





The above bytecount experiment is to show only the space wasted by the old design, while storing the records. Because our hashmap represents Strings as 8-byte integers, it wastes, on average, 4 bytes compared to a 1-byte representation. In the above cases, for a small hashmap that is full of records(64k), the new design uses 1.5-2.6MB, fitting in the CPU L2 cache level (4.5MB), while the old design requires 4-5.3MB (additional 8 bytes for the bucket array), typically residing in the L3 CPU cache level. In addition to the higher memory space requirement, the old hashmap also has overhead when managing the data because of using the two `UnsafeRow` objects. This overhead is analyzed in the next section, along with the system performance.

4.4 Benchmarks

In this subsection, we present the evaluation of our new small hashmap design. To assess our system's performance, we use synthetic benchmarks that are either manually created or come from the industry standards. Our experiments are composed of micro-benchmarks that motivate our system's design choices and specific benchmarking queries to measure the overall performance. When evaluating a query's performance, we take into account only the time spent in aggregation because other operators can get optimized on multiple subsequent runs, and the end-to-end time spent in a query can vary. For this, Spark UI tool provides a visualization of the query plan and for each operator, it shows statistics, as in Figure 4.5. From this visualization, we extract the time spent in aggregation, specifically the `"aggregate time total"` metric from the HashAggregate operator. For all of the experiments, we ran the queries multiple times and selected the shortest aggregation time obtained. Further, in this section, we detail the environment characteristics, then we present the obtained results.

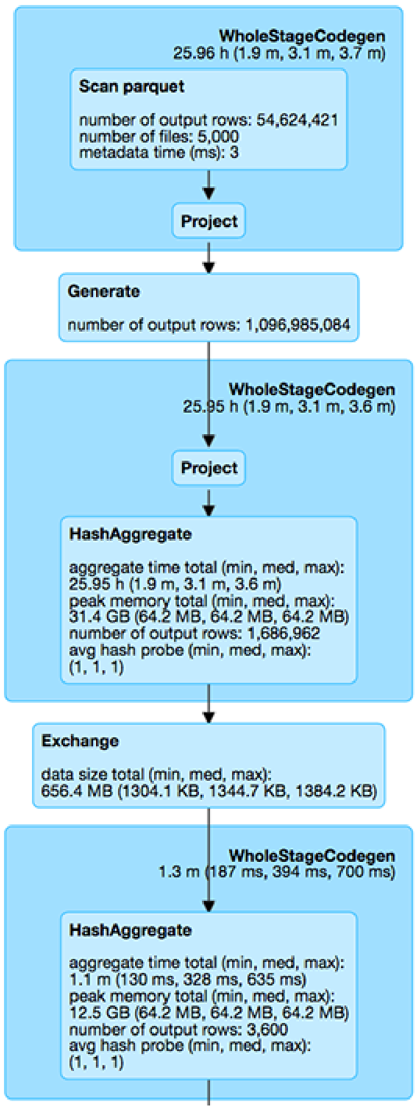
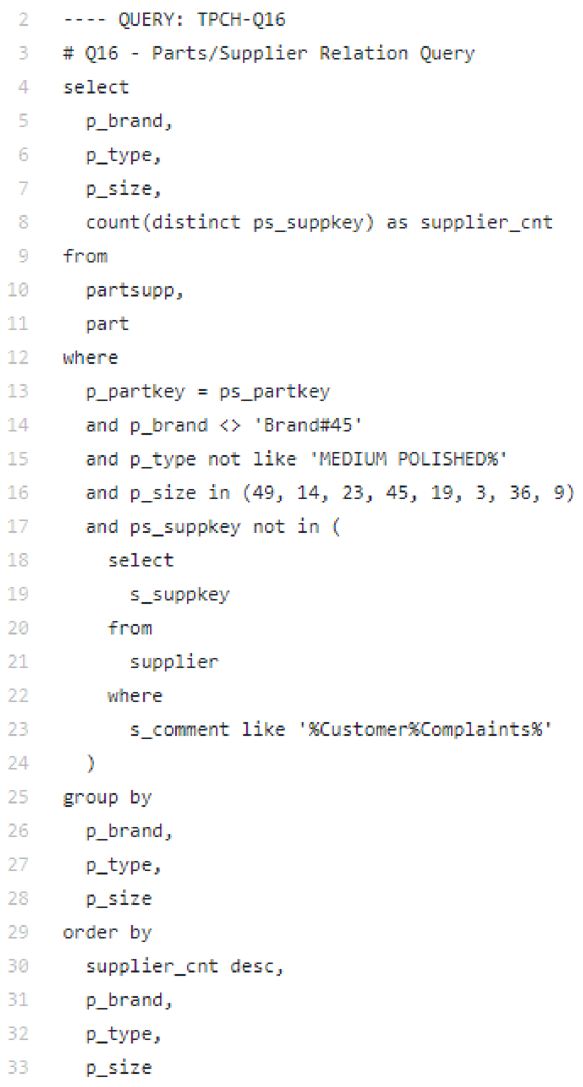

WholeStageCodegen
25.96 h (1.9 m, 3.1 m, 3.7 m)
Scan parquet
number of output rows: 54,624,421
number of files: 5,000
metadata time (ms): 3
Project
Generate
number of output rows: 1,096,985,084
WholeStageCodegen
25.95 h (1.9 m, 3.1 m, 3.6 m)
Project
HashAggregate
aggregate time total (min, med, max):
25.95 h (1.9 m, 3.1 m, 3.6 m)
peak memory total (min, med, max):
31.4 GB (64.2 MB, 64.2 MB, 64.2 MB)
number of output rows: 1,686,962
avg hash probe (min, med, max):
(1, 1, 1)
Exchange
data size total (min, med, max):
656.4 MB (1304.1 KB, 1344.7 KB, 1384.2 KB)
WholeStageCodegen
1.3 m (187 ms, 394 ms, 700 ms)
HashAggregate
aggregate time total (min, med, max):
1.1 m (130 ms, 328 ms, 635 ms)
peak memory total (min, med, max):
12.5 GB (64.2 MB, 64.2 MB, 64.2 MB)
number of output rows: 3,600
avg hash probe (min, med, max):
(1, 1, 1)

Figure 4.5: Visualization plan of an executed aggregation query



```
---- QUERY: TPCH-Q16
# Q16 - Parts/Supplier Relation Query
select
  p_brand,
  p_type,
  p_size,
  count(distinct ps_suppkey) as supplier_cnt
from
  partsupp,
  part
where
  p_partkey = ps_partkey
  and p_brand <> 'Brand#45'
  and p_type not like 'MEDIUM POLISHED%'
  and p_size in (49, 14, 23, 45, 19, 3, 36, 9)
  and ps_suppkey not in (
    select
      s_suppkey
    from
      supplier
    where
      s_comment like '%Customer%Complaints%'
  )
group by
  p_brand,
  p_type,
  p_size
order by
  supplier_cnt desc,
  p_brand,
  p_type,
  p_size
```

Figure 4.6: TPC-H Query 16

4.4.1 Benchmark characteristics

The experiments were executed in Spark 3.0 with the default configuration and no optimization flags. The environmental characteristics are related to the usage of a two-node cluster consisting of one driver and one worker running on the same AWS i3.2xlarge instance, with the following specification:

- 61 GB RAM memory
- 1.9TB NVMe SSD
- Intel Xeon E5-2686 v4 physical CPU
  - Operating at 2.3 GHz
  - Running 8vCPUs
- CPU cache level info:
  - L1I & L1D - 576KB (18x32 KB 8-way set associative, per core)
  - L2 - 4.5MB (18x256 KB 8-way set associative, per core)
  - L3 - 45MB (18x2.5 MB 20-way set associative, shared)

The queries chosen for micro-benchmarks are targeting average case scenarios (Data science, Section 4) because they have four columns set in the group-by operator. For different experiments, we vary the datasets and the keys datatypes, but we use the following base aggregation query in all of the micro-benchmarks:

Micro-benchmarks Query:
```
SELECT key0, key1, key2, key3, count(*)
FROM parquet_file
GROUP BY key0, key1, key2, key3
```

As far as the data location is concerned, the datasets reside in the AWS storage and are eventually fetched, when needed, from the cloud. For the micro-benchmarks, we manually created datasets for each query and saved them as Parquet files. In the next subsection, we present the benchmarking results, highlight the dataset characteristics and their relevance to each experiment.

4.4.2 Small hashmap evaluation

Linear small hashmap
This experiment shows the benefit of having a true linear hash table, by removing the bucket array from the old design while still working with the `UnsafeRows`. In the table below we present the dataset characteristics.

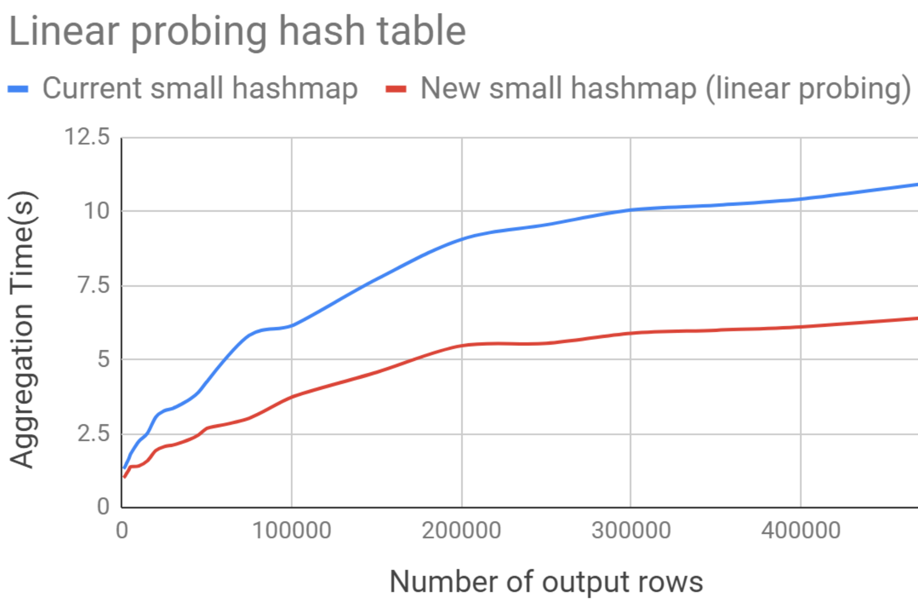


Figure 4.7: Comparison of the original Spark 3.0 with a version that removes the bucket array from the small hashmap. This shows the benefit of having a true linear small hashmap.

Dimension	Value
Size	100M rows
Keys Datatypes	4 x 8-byte integer
Data values	pseudo-randomly distributed
Number of output rows	ranging from 1000 to 500.000
Small hashmap load factor	0.5
Small hashmap record size	64 bytes (40 bytes UR-key, 16 bytes UR-value, 8 bytes next record pointer)

Table 4.1: Dataset characteristics

As we can see in Figure 4.7, the true linear probing version of the small hashmap is faster than the original design. The overhead is notable when the hashmap size exceeds the CPU cache size (L1 - 10k rows, L2 - 75k rows) and the bucket array, from the old design, starts producing additional cache misses.

Small hashmap underlying data structure

This experiment is done to motivate the row-wise layout choice. It compares three different data structures used for the true linear small hashmap. The managed data structures are as follows: (1) `UnsafeRows` from the old batch of rows structure, (2) row-wise layout using a single contiguous memory region (8-byte integer array), (3) column-wise layout using specific arrays for each column's datatype.

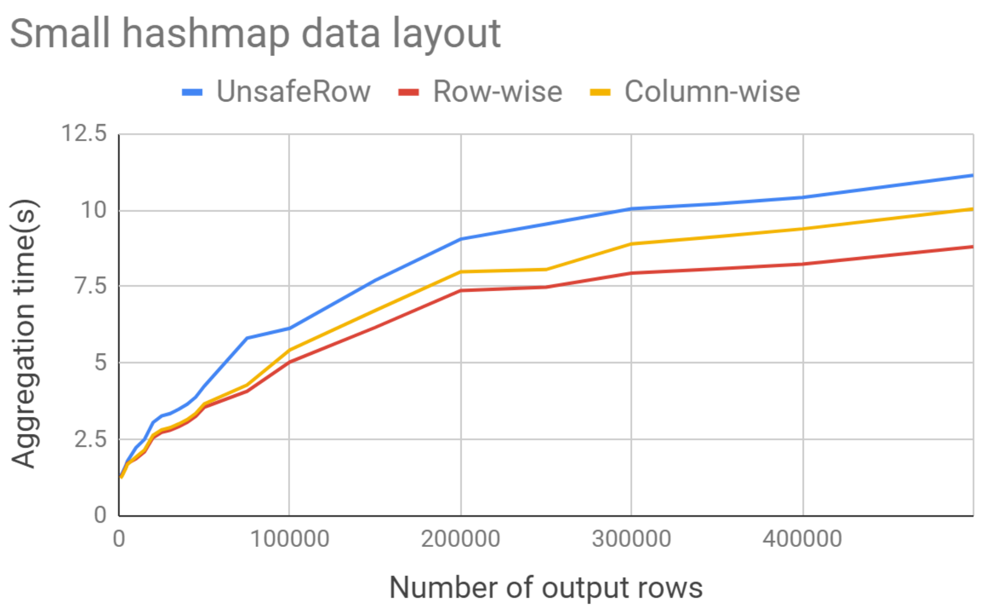


Figure 4.8: New small hashmap with three layouts: UnsafeRow, column-wise layout, row-wise. The results show that the row-wise layout is the best, especially when the hashmap does not fit into the CPU cache.

Dimension	Value
Size	100M rows
Keys Datatypes	2-byte integer, 4-byte integer, 2 x 8-byte integer
Data values	pseudo-randomly distributed
Number of output rows	ranging from 1000 to 500.000
Small hashmap load factor	0.5
Small hashmap record size, when using UnsafeRows	64 bytes (the same as in the previous experiment)
Small hashmap record size, when using the row-wise layout	40 bytes (32 keys, 8 values)
Small hashmap record size, when using the column-wise layout	30 bytes (22 keys, 8 values)

Table 4.2: Dataset characteristics

We observe, in Figure 4.8, that removing the `UnsafeRow` usage from the small hashmap has a minor benefit in query execution performance. This is mostly due to the `UnsafeRow` embedded data manipulation through byte-wise operations and its memory-wasteful format. As far as the hashmap layout is concerned, it was expected that the column-wise layout to be fast as long as it fits (L1 - under 12k rows, L2 - under 150k rows) into the CPU cache. At some point, when the number of records grows, we can see the balance tilting in favor of row-wise layout. This is because the column-wise layout that can have a cache miss for each column, while, in the row-wise layout, one record requires less space than a cache line and generates at most one cache miss.

Variable-size data representation

In this experiment, we evaluate the benefit of having the variable-size region store and manage data 8-bytes at a time. As previously mentioned, there is a chance for fast data manipulation, because of the natural mapping to machine words. Additionally, this representation would help in achieving faster String key equality checking and hash code generation, when dealing with strings bigger than 4 bytes (old design checks 4 bytes at a time).

Dimension	Small Strings dataset	Large Strings dataset
Size	50M rows	50M
Keys Datatypes	3 x String Columns	3 x String Columns
Data values	4 characters long	Between 20 and 40 characters
Number of output rows	100k	100k
Small hashmap load factor	0.5	0.5
Small hashmap max record size	56 bytes	152 bytes

Table 4.3: Datasets characteristics

Looking at the results in Figure 4.9, we can tell that there is not much of a real benefit when dealing with Strings stored in a long array. Moreover, when dealing with very small Strings, the byte array implementation is favorable because there is no memory space wasted.

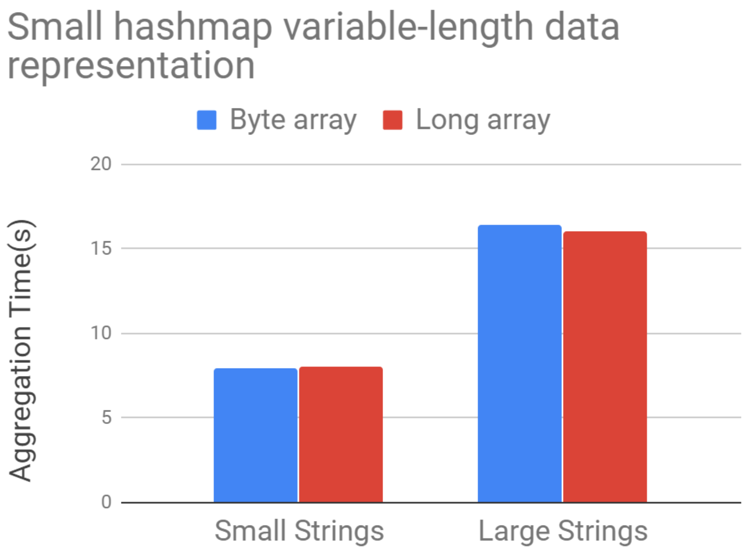


Figure 4.9: Comparison of the 1-byte and the 8-byte representation of the variable-length data region in the linear probing small hashmap. The minor differences show the costs and possible gains of having the hash code and the equality computed 8 bytes at a time.

System evaluation

In this experiment, we show the overall benefit of our hashmap design in improving the query aggregation time. For this test, we use two synthetic standard industry queries (TPC-H (34)), available in figure 4.6 and figure 4.12, and the custom query used in micro-benchmarks. We selected the TPC-H queries *1* and *16* because they both contain an aggregation and have different number of keys in group-by. Moreover, they have some important features that we want to analyze in the following chapters. One characteristic useful for this evaluation is that the number of output rows is less than 64k, thus we make sure that most of the records fit in the small hashmap. For the TPC-H queries, the dataset was generated for scale factor 100. Note that the small hashmap supports only primitive data types, therefore for decimal type present in TPC-H queries, we converted them to a double type.

TPC-H Q1 evaluation

The results present in Figure 4.10, show the speedup that can be obtained in aggregation using our small hashmap design. If we look at the TPC-H Query1 we notice that it has two string keys in group-by and seven computing-intensive aggregation functions. Even though the new small hashmap design is cache friendlier and has optimizes memory management, the TPC-H Q1 is focused on the CPU computing power.

The complex aggregation functions and the arithmetic operations make this query CPU bound. Moreover, the resulting number of output rows is 4, where each record has 104 bytes, meaning that the whole small hashmap can fit in the CPU cache level 1. Therefore the 10% improvement is coming from managing primitive data (8-byte integers), instead of the inefficient `UnsafeRows`.

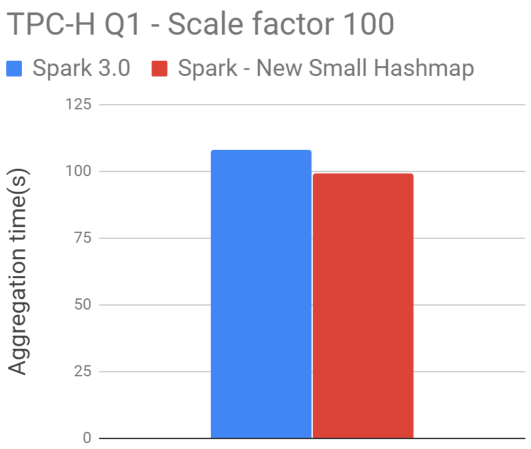


Figure 4.10: TPC-H Query1: comparison of the time spent in aggregation, between the old and new small hashmap designs

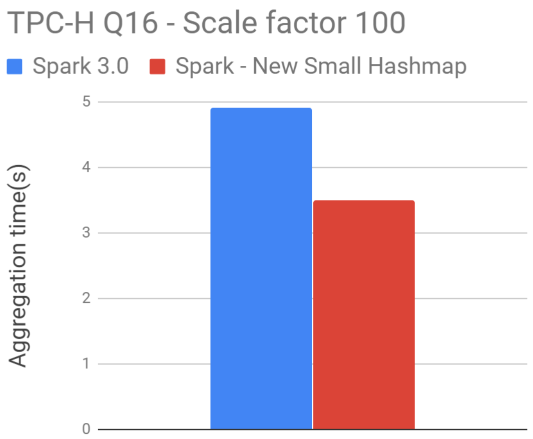
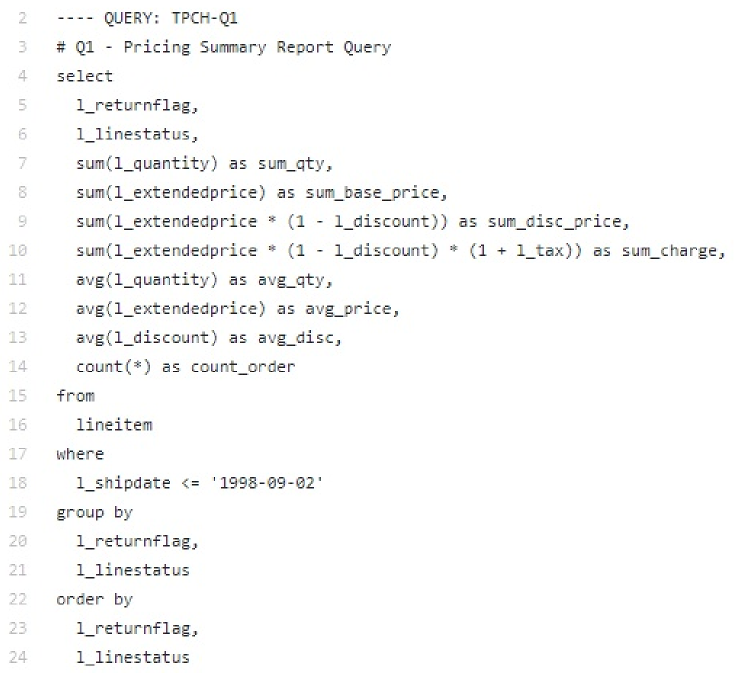


Figure 4.11: TPC-H Query16: comparison of the time spent in aggregation, between the old and new small hashmap design



```
---- QUERY: TPCH-Q1
# Q1 - Pricing Summary Report Query
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= '1998-09-02'
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus
```

Figure 4.12: TPC-H Query 1

TPC-H Q16 evaluation

Similar to the previous evaluation, looking at figure 4.11, we analyze the obtained speedup of 1.4x. TPC-H Query16 has two String keys ("p_brand", "p_type") that contain values around 25 characters. This means that a part of the CPU computing effort is focused on working with large strings. Having around 18k output rows and 90 bytes in record size, we obtain a 1.6 MB hashmap that fits into CPU cache level 2. Since this query does not have compute-intensive aggregation functions, the obtained speedup shows the benefit of the new optimized small hashmap design, while fitting in the CPU cache.

Custom Query evaluation

```
SELECT key0, key1, key2, key3, count(*)
FROM parquet_file
GROUP BY key0, key1, key2, key3
```

Dimension	Value
Size	100M rows
Keys Datatypes	4 x 8-byte integer
Data values	pseudo-randomly distributed
Number of output rows	ranging from 1000 to 500.000
Small hashmap load factor	0.5
Record size in the original small hashmap	64 bytes
Record size in the new small hashmap	40 bytes

Table 4.4: Dataset characteristics

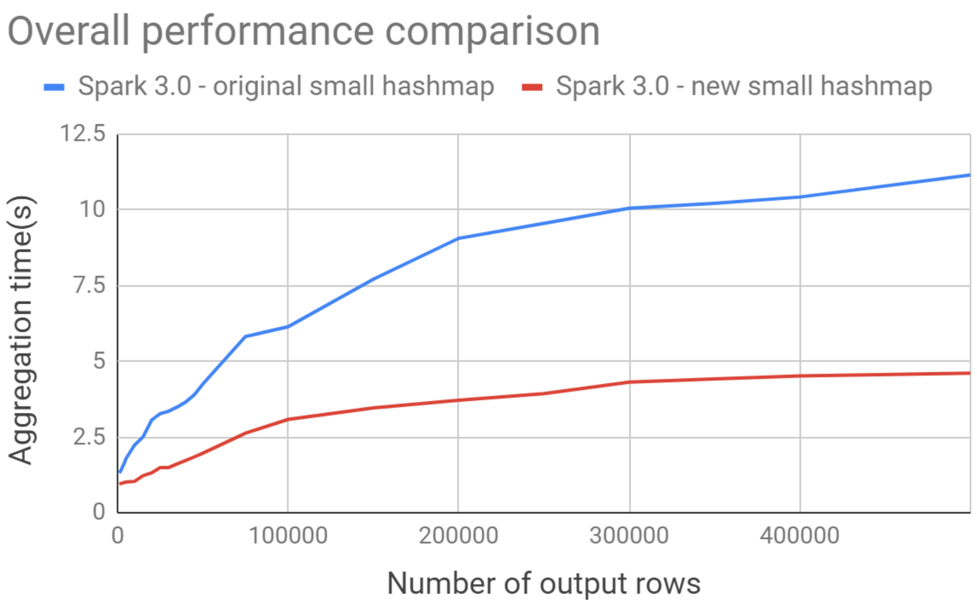


Figure 4.13: Custom Query: comparison of the time spent in aggregation, between Spark 3.0 and Spark 3.0 with our new design for the small hashmap, showing the performance improvement obtained.

From the results present in Figure 4.13, we notice a maximum speed of 2.7x. Compared to the other queries run, this custom query neither works with Strings nor does it have any compute-intensive aggregation functions. Moreover, all four keys are represented as 8-byte integers, to avoid any type conversions. Therefore this dataset is a proper best-case scenario, making the query to be oriented on memory management. In the TPC-H queries, we validated the speedup when the hashmap was in cache, while in this experiment we noticed a larger benefit in query execution when comparing the two small hashmap versions while outside of the CPU cache (L1 - 10k/15k rows, L2 - 70k/110k rows).

With respect to data locality and memory management, we observed that the original small hashmap was inefficient. To assess the new design's cache friendliness, we compared the number of CPU cache misses resulted from both of the small hashmap designs, while outside of the CPU L2 cache (last experiment, where the number of output rows is 200k for both versions). For this, we used the async-profiler tool (35) that provides out-of-the-box flamegraph (36) support for cache misses. The results were obtained by running the profiler, with a sampling interval of 0.2ms, on the executing query. Even though the number of the cache misses generated by the old small hashmap represents 10% of the total, when compared to the results of the new design, we observed a 70% reduction in the number of cache misses generated by this hashmap.

We created a PR of the new small hashmap and we propose adopting it, in Spark, on a future occasion.

Chapter 5

String Compression

Our approach to String compression is to build a unfied (per task) dictionary that maps Strings to unique codes and further work with a compact format in the aggregation. Since we have no prior knowledge about the number of distinct values in the dataset, such a unified dictionary can become unmaintainable and grow large enough to exceed the memory allocated for a task. However, the Parquet files store the data encoded and the `dictionary_encoding`, if available, already provides a String dictionary, typically one per row group. Therefore, in this chapter, we are going to restrict the scope of our problem only to those aggregations that immediately feed off a Parquet scan, so that we can leverage the Parquet dictionary pages. Further, in this section, we highlight the challenges faced while exploiting the Parquet `dictionary_encoding` and present the integration of String compression into the new small hashmap design. Finally, we conclude with a series of benchmarks that show the possible gains of using this approach.

5.1 String dictionary

As stated in Section 2.4, the Parquet `dictionary_encoding` is used to compress the column chunk data into dictionary pages and, by default, is available only if the number of distinct values, in a chunk, is less than 100k and the total size of the dictionary page is less than 1MB. In the Parquet file organization, each row group contains one chunk for each column in the dataset, therefore, if the `dictionary_encoding` is available, a row group has only one dictionary page for each column.

In Spark, it is often the case that a task works with only one row group. However, there are two configuration flags - `openCostInBytes`, `maxPartitionBytes` - that decide the number of Parquet row groups per task. The first configuration flag is used when dealing with multiple small files, so that these are read together to eliminate additional overhead. The second one indicates the maximum number of bytes that can be read from a single partition. If these flags are modified or the Parquet file has been written with a non-default configuration, then a task can have multiple row groups and implicitly multiple dictionary pages for each column. Since the number of row groups per task can vary and the Parquet

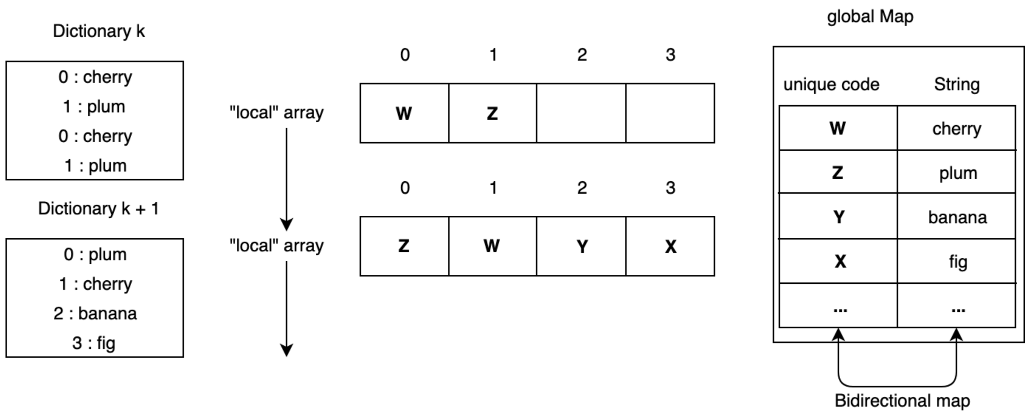


Figure 5.1: Unified String dictionary creation, while deduplicating the Parquet codes

files do not have a unified dictionary per column, we create our own unfieid (per task) String dictionary, at runtime, using the Parquet dictionary pages.

Aggregating multiple Parquet dictionaries, means that we can deal with two unfavorable situations where: (i) different integer codes map to the same Strings and (ii) the same codes map to different Strings. The solution to this problem is to deduplicate the integer codes and remap the Strings to unique values. While the scan operator is reading the data, we apply the deduplication algorithm and create the unified String dictionary. At runtime, we read the integer code for a String column, from the Parquet dictionary page. Then, we use an array (*local array*), where the indexes are integer codes coming from the dictionary page and the values are unique deduplicated codes, to probe the integer code read. If the value is not found, it means that we check for the deduplicated code inside our *unifiedlMap* (per column global dictionary), otherwise we know that its corresponding String is present in this dictionary and therefore the provided value can be used later, in the aggregation. The *unifiedlMap* is the dictionary that we create, which bidirectionally maps distinct Strings to generated, unique codes. The bidirectional map is composed of: (1) a hashmap from Strings to 8-byte Integer codes and (2) an array, called *stringRegion*, that contains appended Strings as a stream of bytes, where each code from the hashmap is a tuple (offset and size) that references a String in the array. The next step, in the deduplication algorithm, is String decoding from the dictionary page and probing it in the *unifiedlMap*. Now, we either retrieve the stored unique code or, if not found, insert a new entry (the probed String and a generated unique code) in the hashmap. Finally, we update the *local array* with the unique code so that further integer codes, from the same dictionary, can get the value immediately without looking it up in the *unifiedlMap*. When the scan operator starts reading from the next dictionary page, the contents of the *local array* will be cleared. For each dictionary page, the algorithm decodes the String associated with each distinct integer code, updates the *unifiedlMap* and then uses the local array to act as a cache for further codes. In figure 5.1, we illustrate an example for the deduplication algorithm where the local array changes its values for each dictionary, according to the *unifiedlMap*(global Map) codes.

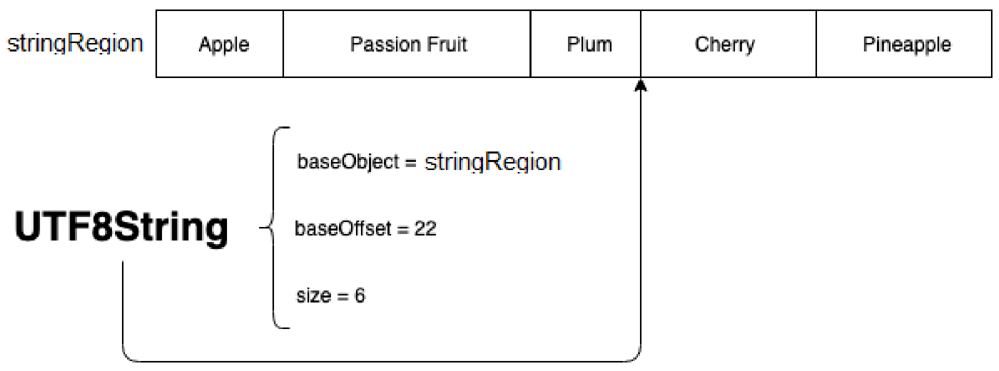


Figure 5.2: a compact String that uses the `UTF8String` API

When we have multiple row groups per task, we also have to deal with the problem that not all of them can have a dictionary page available for the same column. In this situation, we can compute only a partial unified dictionary and, once again, it is unmaintainable to build a complete dictionary since we have no knowledge about the number of distinct Strings available in the column chunks that could not provide a dictionary page. Even though, a lower limit is known (100k distinct values or 1MB in size), we can not say anything about the upper limit. Because of this high degree of uncertainty, we make the aggregation work with compact Strings only where the key-columns are part of the partial dictionary. In Spark, the Strings are represented and used in the `UTF8String` format, thus we exploit this format to make the aggregation transparently work with both normal and compact Strings.

5.2 Custom compact Strings

The `UTF8String` API works using three parameters: *baseObject*, *baseOffset*, size. To distinguish the normal from compact Strings, we use one contiguous memory region (the textitstringRegion that is part of the *unifiedlMap*), as *baseObject*, for the Strings that come from the Parquet dictionaries. As far as the *baseOffset* and the *size* are concerned, they will be the unique codes from the *unifiedlMap* that point to different locations in the *stringRegion*, according to the referenced String. As a whole, such a Strings look like a normal `UTF8String`, while under the hood it is compressed utilizing only an offset and a size to reference a String from the *unifiedlMap*. In Figure 5.2, we can see an example of how a custom `UTF8String` looks like - the only difference is the *baseObject* that is referencing the array used to store the unique dictionary Strings. This way, we deal with both normal and compact Strings. In either case, the scan operator produces an `UTF8String` to feed to the aggregate operator.

For each String key-column, that is part of both the Parquet scan and the aggregate operators, we build such a partial String dictionary and use the compact form, where possible. To identify these key-columns, we analyze the query plan starting from the aggregation and searching recursively in all of its children plans. The keys found are susceptible to having available the Parquet dictionary encoding and, for them, we modify the scan operator to build the partial unified dictionaries and to feed the compact `UTF8Strings` to the aggregation.

Knowing that all of the compressed Strings from the same key-column are part of the same contiguous memory region (the *stringRegion*) and each String is uniquely identified by a tuple (offset and size), then, two `UTF8Strings`, that have all three parameters equal are referencing the same dictionary String. Therefore, we optimize the small hashmap equality checking when dealing with compact Strings. To further make use of the String dictionary, we save, in the *unifiedMap*, the hash code for each dictionary encodable String, so that we reduce the overhead of computing it at every lookup in the small hashmap.

5.3 Benchmarks

In this section, we present the evaluation of our small hashmap that leverages the custom-built String dictionary. Similar to the small hashmap evaluation, we assess the system performance, using synthetic benchmarks that are both manually created and coming from industry standards. The environmental characteristics are identical to those described in Section 4.4. We use the same TPC-H queries for evaluation because they contain key-columns that have the dictionary encoding available. Additionally, we evaluate a custom query and a modified version of TPC-H Query 1, to observe the maximal benefit of this compression strategy. Note that the String compression strategy is implemented and tested in the new small hashmap design.

TPC-H Q1 evaluation
The results presented in figure 5.3, show a 1.3x speedup for applying the String compression, compared to the original Spark 3.0 and 1.2x in comparison to our new small hashmap, without compression. Similar to the explanation found in Section 4.4.2, the TPC-H Query1 is CPU bound because of the complex aggregation functions. However, 20% speedup (total of 30% minus 10% from the new small hashmap) is obtained only from leveraging the Parquet dictionary encoding. In TPC-H Query 1, both String columns are dictionary encodable because they have only 4 values of 1 character each. Even though these are very small Strings, decoding millions of them adds overhead. In our approach, this overhead is reduced by decoding each unique String only once per row group.

Modified TPC-H Q1 evaluation
In this experiment, we removed all of the compute-intensive aggregation functions from TPC-H Query1 and ran it on the same dataset of scale factor 100. The purpose of this experiment is to assess the gains of the compressed execution, while the query is no longer CPU bound. Therefore, in Figure 5.4, we notice a speedup of 2.65x between the original version of Spark and the one that uses compressed Strings in the new small hashmap design.

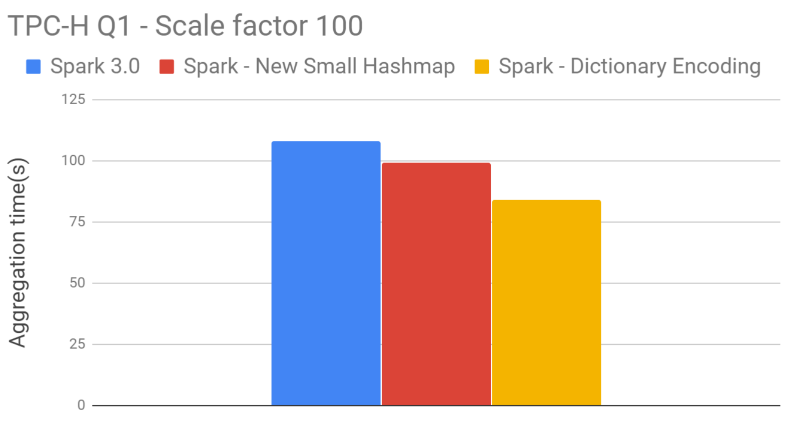


Figure 5.3: TPC-H Query1: comparison of the time spent in aggregation between three versions of Spark: (blue) original, (red) only with the optimized small hashmap, (yellow) using both the new small hashmap and the String compression

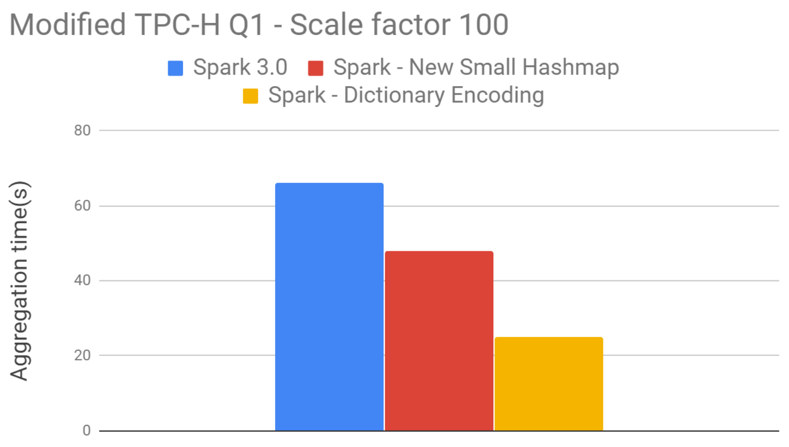


Figure 5.4: Modified TPC-H Q1: comparison of the time spent in aggregation between three versions of Spark: (blue) original, (red) with the modified small hashmap, (yellow) using both the new small hashmap and the String compression

TPC-H Q16 evaluation

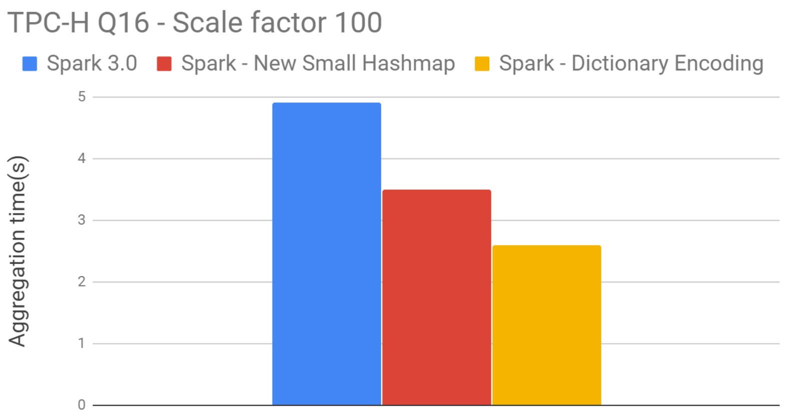


Figure 5.5: TPC-H Query16: comparison of the time spent in aggregation between three versions of Spark: (blue) original, (red) with the modified small hashmap, (yellow) using both the new small hashmap and the String compression

TPC-H Query16 has two String key-columns, out of which only one has the Parquet `dictionary_encoding` available. This justifies the small increase in speedup, available in Figure 5.5. We notice a speedup of 1.9x comparing the third version with the first one, but only 40% of the performance increase is because of applying String compression, while the rest is from using the new small hashmap design. This speedup is obtained from decoding the Strings once per row-group and from using the fast String equality checking, by comparing the reference pointers of the compact Strings.

Custom Query evaluation

```sql
SELECT key0, key1, key2, key3, count(*)
FROM parquet_file
GROUP BY key0, key1, key2, key3
```

Dimension	Value
Size	50M rows
Keys Datatypes	3 x String columns
Data values	Strings of 24 characters
Values distribution	Pseudo-randomly chosen from a list
Encoding scheme	`Dictionary_encoding` available for all of the keys
Number of output rows	50.000
Small hashmap load factor	0.5

Table 5.1: Dataset characteristics

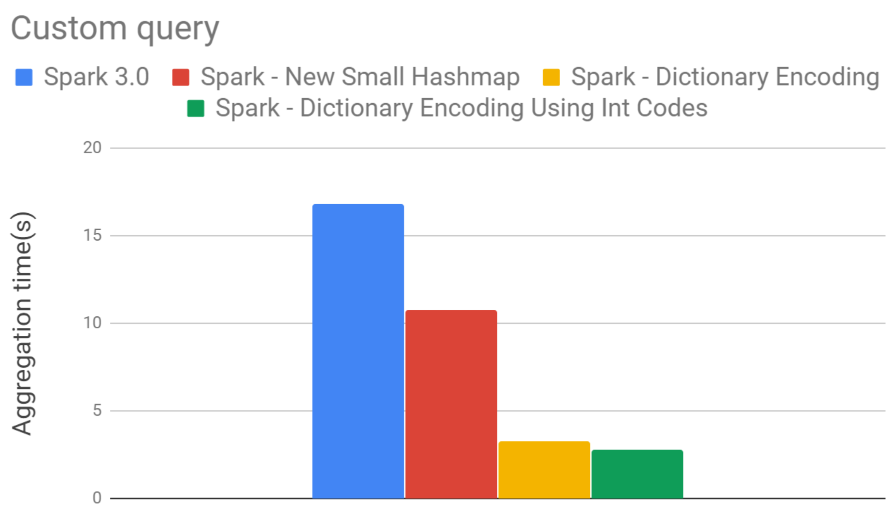


Figure 5.6: Custom query: comparison of the time spent in aggregation between four versions of Spark: (blue) original, (red) with the modified small hashmap, (yellow) using both the new small hashmap and the String compression, (green) using the new small hashmap and feeding the integer codes, that come from the Parquet dictionaries, directly in the aggregation

This experiment is created not only with the purpose of providing a best-case scenario, but also to compare the unified String dictionary approach with another compression that makes the aggregation work on integer codes rather than on String data. The integer codes are obtained directly from the Parquet dictionary pages and used in the new small hashmap. At the end of the aggregation, the integer codes are transformed back to Strings, using the Parquet dictionaries. We manually crafted the dataset and, therefore, guarantee `dictionary_encoding` availability for all of the key-columns and for each row group. For this experiment, we observe in Figure 5.6, the 5x speedup when using the unified dictionary approach, compared with the original Spark, and a slowdown of 17% when compared with the new compression approach. This slowdown, is mainly due to no longer building the unified dictionary and handling 8-byte integers instead of the `UTF8String` objects. Since this is the best-case scenario, the 17% slowdown is not significant while considering the fact that `dictionary_encoding` can not be guaranteed for all of the row groups, in real workloads. Another issue with this approach is that the String data is not used in the aggregation anymore and, therefore, aggregation functions, that want to manipulate the String data, will not work, when applied on integers.

Chapter 6

Numerics Compression

As far as the numeric data is concerned, we squeeze multiple columns together, at runtime, and work with this compact format in the aggregation. In Chapter 4 (data science), we observed that most of the data used 4-byte integers, while requiring only 1-byte to store its values, and noticed a 1.6-1.8x benefit in a possible application of a bit-packing technique. Bit-packing is a compression technique that saves space by reducing the number of bits necessary to represent a value. Bit-packing makes the hash tables smaller, and thus somewhat faster. Another benefit is that when multiple keys are packed into a single word, only one hash code and comparison computation is needed, which reduces the CPU cost of a lookup. Fixed-size data types use all of the available bits to represent their values, regardless of that value. The principle behind bit-packing is to use only the bits that are actually needed for values representation. This can extrapolate to datasets, where the number of bits needed is based on the maximum value present. For example, if we have a dataset with only one column whose values range from 0 to 10 (the binary representation of a 4-byte integer with the value 10 is: 00000000 00000000 00000000 00001010), then we only require 4 bits to represent all of the possible values within the dataset. In this chapter, we focus on compressing numeric datatypes through bit-packing. Although we did not investigate this, we considered applying bit-packing on compact Strings, but we used a whole 8-byte word to store the String reference(4-byte offset and 4-byte size of). At most, we could have moved the size at the beginning of each String and apply bit-packing on the offsets and compress them two by two.

Even though the optimal compression is obtained by analyzing each row's data and tightly bit-packing it, this requires storing information about how each row was compressed, so that, at the end of the aggregation, the records can be uncompressed. Storing additional information is opposed to our goal of minimizing the hash table records size, therefore we use the same bit-packing compression scheme on a batch of rows, significantly reducing the amount of stored information. To avoid reading the data twice (first, to compute the compression schemes and, second, to apply them and compact the data), we require data statistics, such as min-max information, for each batch of rows. This metadata can be used to create the compression schemes without having to look at the data. However, in Spark as well as in Cloud, such information is unavailable when dealing with big data. The only metadata available is the min-max statistics that can be found either in the Parquet files, at different levels of granularities (Section 2.4), or in the Delta Lake transactional log

(Section 2.5). The min-max information can be extracted from the Parquet files by reading each column chunk or page header, while in Delta the metadata is present in each commit, therefore its extraction requires aggregating all of the commits for a file version and then obtain the statistics per file. This stored metadata is available if we restrict the scope of our problem to aggregates that feed off Parquet files in original format or delta format. In addition to this, the min-max statistics are susceptible to outliers and can not guarantee an optimal compression. Therefore, in this chapter, we generate our own statistics from data samples, remove the outliers and optimally compress multiple numeric columns together, without restricting the scope of our project.

6.1 Data statistics

Because no data samples are available, we generate the metadata, at runtime, while looking at a sample of processed rows. We can see the actual values available in this sample and, hence, we designed an algorithm that generates min-max statistics while avoiding the possible outliers. In the aggregation, the rows are read and processed one by one; therefore, we select a fixed amount of rows read and save them in memory to use for metadata generation. After we obtain the outlier-free data statistics, we process, first, the saved rows and then continue reading and processing the remaining row found in the current batch. This way, we can apply the compression scheme, obtained from the generated statistics, on the sampled rows as well.

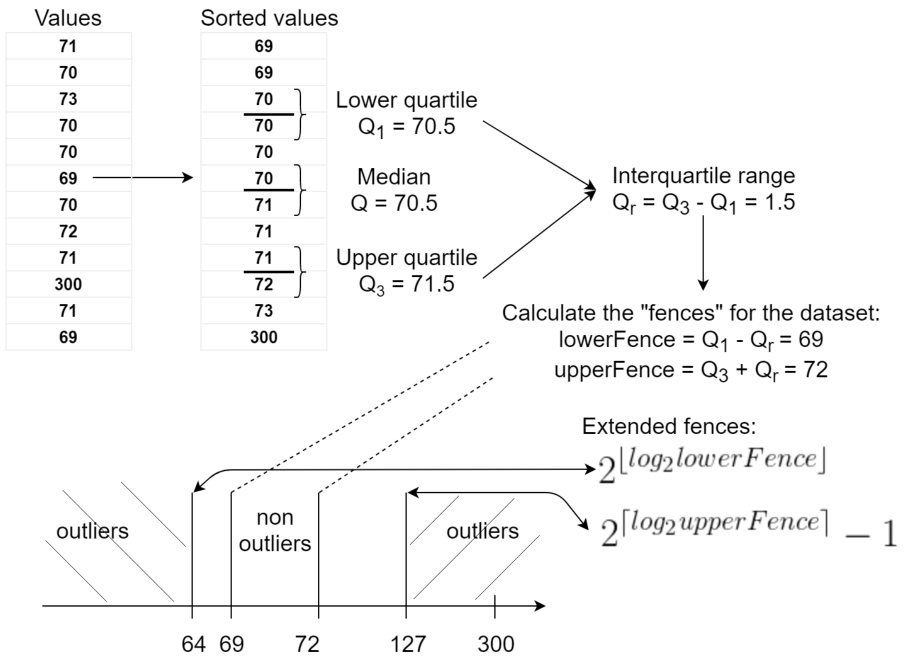


Figure 6.1: Example of how the outlier detection algorithm works

Algorithm 1 Min-Max generation and outlier detection algorithm

```
1: global variables
2:     $numCols$, number of columns
3:     $maxSampledRows$, number of sampled rows
4:     $data[numCols][maxSampledRows]$, sampled data
5:     $minMax[numCols][2]$, min-max information for each column
6: end global variables
7: function PROCESS_MINMAX($data$)
8:     for $i := 0$ to $numCols$ step 1 do
9:         $data[i]$ =sorted($data[i]$)
10:        $Q_1 = data[i][\lfloor maxSampledRows/4 \rfloor]$
11:        $Q_3 = data[i][\lfloor maxSampledRows/4 * 3 \rfloor]$
12:        $Q_r = Q_3 - Q_1$
13:        $lowerFence = Q_1 - Q_r$
14:        $upperFence = Q_3 + Q_r$
15:        $minBound = 2^{log_2 \lfloor lowerFence \rfloor}$
16:        $maxBound = 2^{log_2 \lceil upperFence \rceil} - 1$
17:        for $j := 0$ to $maxSampledRows$ step 1 do
18:            if $data[i][j] >= minBound$ && $data[i][j] < maxBound$ then
19:                $minMax[i][0]$ =MIN($minMax[i][0], data[i][j]$)
20:                $minMax[i][1]$ =MAX($minMax[i][1], data[i][j]$)
21:            else
22:                outlier identified
23:            end if
24:        end for
25:    end for
26:    Return $minMax$
27: end function
```

To avoid outliers, we use a simple algorithm, available in the above pseudocode (Algorithm 1), that sorts the sampled dataset, calculates the median (Q) and the lower and upper quartiles (Q1 and Q3). Using the quartiles, it calculates the interquartile range (Qr) and then computes the "fences" (lower and upper values that border most of the data) of the dataset by subtracting the interquartile range from the lower quartile and, respectively, adding it to the upper quartile. We expand these "fences" to include all of the values that require the same amount of bits for their representation. These values are found between (left-closed interval) the highest power of 2, smaller than the "lower fence" and the smallest power of 2, bigger than the "upper fence". All of the values, that are within the "fences", are valid for data statistics while the rest are considered outliers. In figure 6.1, we illustrate an example of outliers detection using this algorithm.

From the generated data statistics we compute a rule, called *packing configuration*, that specifies how the rows from a batch are going to be compressed. From the sorted per column min-max information, we compute the *packing configuration* by adding as many keys as can fit in an 8-byte integer, until all of the columns are in a "compressed" format (e.g. the number of bits of 5 min-max columns are: 60, 42, 20, 15, 14, 8 and they require three 64-bit columns as follows: 60, 42 + 20 and 15 + 14 + 8). A *packing configuration* is characterized by three features for each column: the number of bits required for its value, the column index inside the compact row and the offset inside the value at that column index. Every time a new row is processed, it is checked against the packing configuration and either compressed accordingly or labeled as "unpackable". Since sending all of the unpackable rows to the BytesToBytesMap can slow down query execution performance, we use two small hashmaps, one for the packable rows and the other one for the rest, while the regular map is used only for fallback cases. The small hashmap, used for packable rows, is also created in the WSCG. The code generation happens before query runtime, therefore, we use generic code to handle numeric datatypes compression in this small hashmap. We generate the code such that the small hashmap works with a general packing configuration, implying overhead from additional for-loops, conditional branches and variables. Code genericity is the cost we pay for having data statistics without outliers. In the next section, we highlight how a better code can be generated.

6.2 Code generation

Specific code generation is desired for faster query execution, and it can be achieved when leveraging the Parquet min-max statistics or extracting the Delta Log statistics, before runtime. In this case, we suffer from the metadata extraction overhead and from a non-optimal compression due to possible outliers present in the metadata. We analyze this tradeoff in the benchmarking section.

When we have our own generated statistics, we deal with generic code latency. However, the small hashmap is part of WSCG, which gives us the freedom to create multiple code paths for it and decide, at runtime, which code path should be used. This decision is based on the resulted packing configuration. The amount of generated code is limited, in Spark, to 64KB, meaning that we can not generate specific code for each possible packing configuration. Therefore, we restrict ourselves to generating only a few code paths based on heuristics and speculations. For example, we can speculate of each key-column found in an aggregation group-by require at most 15 bits for its value and, thus, we can compress 4 columns together in one-word key (or 3 columns compressed using maximum 21 bits each). Guessing the number of bits used for each key is very luck-based (64 bits to guess for each column), but the most reliable speculation is related to the number of columns that result after compression, being a number lower or equal to the number of numeric columns found in the group-by operator. Only with this information, we can optimize the generated code with specific values and conditions. The resulted columns means that we know the size of the *packing configuration* and we use techniques like loop unrolling to escape some of the generic code latency. We limit the number of efficient code paths to a maximum of half the number of columns (when at least two columns can be packed together), to keep the time for code generation low, while still providing enough flexibility to optimize parts of the code, according to the obtained data statistics. From the data science we know that the average number of keys found in the group-by operator is 4, therefore we expect that the maximum number of columns to be a low number.

Specific code:
```
if (key0 < 32000 && key1 < 128000) {
  packedValue[0] = key0 + key << 15;
  HashMap.lookup(packedValue);
}
```

Generic code:
```
if (key0 < configValue[0] && key1 < configValue[1]) {
  for ((k,i) in keys.zipWithIndex()) {
    packedValue[configIndex[i]] += (k << configBits[i]);
  }
  HashMap.lookup(packedValue);
}
```

Speculative code:
```
if (key0 < configValue[0] && key1 < configValue[1]) {
  packedValue[0] = key0 << configBits[key0.Index] +
                   key1 << configBits[key1.Index];
  HashMap.lookup(packedValue);
}
```

Figure 6.2: Sample of the small hashmap lookup method, code-generated with 3 different strategies

In Figure 6.2, we present three snippets of the same code, generated using each strategy: (1) specific code using the min-max statistics, (2) generic and (3) speculative code, based on the packing configuration. The illustrated code is part of the small hashmap lookup method and is generated for a query with 2 keys that can be compressed as a one-word column. The snippets are used to highlight the specific code simplicity and show that the speculative code can get optimized leveraging the information presumed. In the next section, we analyze and compare the performance of these three strategies on custom-made benchmarks.

6.3 Benchmarks

In this section, we present the evaluation of employing the bit-packing technique in our small hashmap. Unlike in the previous benchmarks, we use only custom benchmarks to evaluate the performance because the TPC-H queries 1 and 16 have only one numeric column and two String columns, on which we do not apply our technique yet because we use the whole 8-byte for the compact String representation. The environmental characteristics are identical to those described in Section 4.4. Further, we briefly present the datasets and the query ran, and then we analyze the results.

The first and the fourth datasets are used to illustrate the best and worst cases, when all of the rows are compressible, and respectively, none are compressible. The other two datasets are mainly used to provide different scenarios for the speculative code generation, in which the 4 keys will be compressed as either one-word or two-word columns. For evaluation, the following query was executed multiple times on all of the four datasets and we selected the shortest aggregation time obtained, for each test.

Custom Query:

```
SELECT key0, key1, key2, key3, count(*)
FROM parquet_file
GROUP BY key0, key1, key2, key3
```

Dimension	Dataset1	Dataset2	Dataset3	Dataset4
Size	200M rows	200M rows	200M rows	200M rows
Keys Datatypes	4 x 8-byte integer	4 x 8-byte integer	4 x 8-byte integer	4 x 8-byte integer
Data values	use less than 15 bits	use between 15 and 40 bits	use between 21 and 40 bits	use more than 40 bits
Number of output rows	200k	200k	200k	200k
Number of compressible rows	200k	150k	100k	0
Small hashmap load factor	0.5	0.5	0.5	0.5
Small hashmap max record size	40 bytes	40 bytes	40 bytes	40 bytes

Table 6.1: Datasets characteristics

Query execution performance

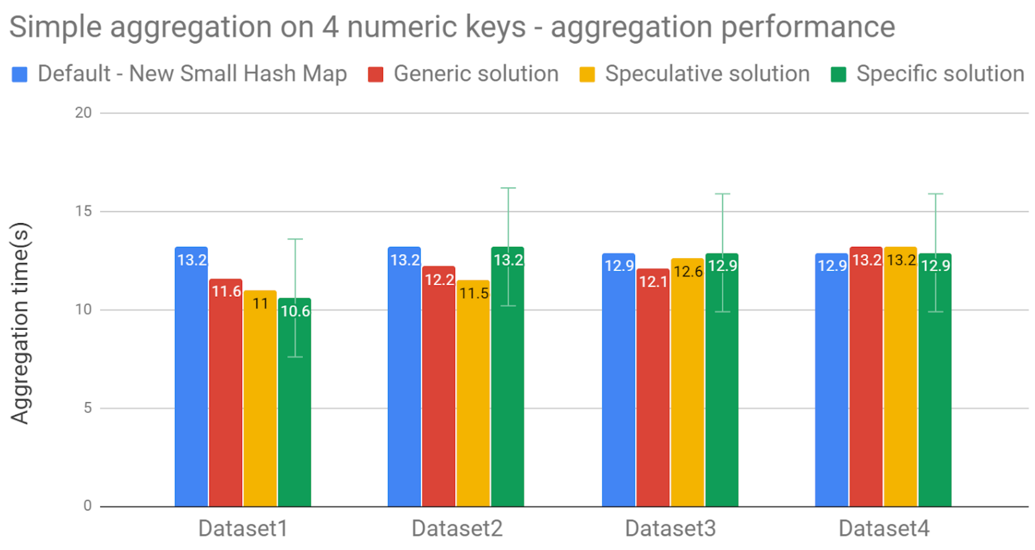


Figure 6.3: Comparison of total time spent in aggregation between the default version of Spark, using our optimized small hashmap, and three versions (for each code generation strategy) of Spark that employ numerics compression in the new small hashmap

In figure 6.3, we compare the performance of all of the code-generation strategies, when employing bit-packing inside the hashmap, against a Spark version that uses our small hashmap design. For the specific code solution(green bar) we read the min-max information from Delta Log and its error rate represents the actual time ($\tilde{3}$ s) needed for metadata extraction. This is represented as an error bar due to the fact that the metadata processing happens before code generation and it is not counted in the hash aggregate operator metrics. For the last three scenarios, the specific solution (green bar) has outliers in the min-max information and, therefore, is unable to compress the data. From the results, we can easily say that the speculative code generation is the best solution as long as the assumptions made are correct. The third scenario shows a slowdown compared to the generic solution because the chosen code path was not optimal, compared to the generic solution(e.g. in the following test, the selected code path for speculative version compressed the columns in two-word key). However, we can optimize the speculative version to check if the selected code-path provides a better compression than the generic solution and, otherwise, fall back to the generic code-path. In the fourth scenario, we see that numerics compression does not work, therefore every row is stored uncompressed and the solutions have 0.3 seconds overhead, from generating the data statistics for all of the rows samples. In all of the scenarios, the first 5000 rows of each batch were used to create the metadata and the packing configuration. Nevertheless, the results do confirm the benefit of employing bit-packing, at query runtime, and show a speedup of 20%. An important part of this speedup is mostly due to the reduced CPU cost used for hashing and comparison.

Small hashmap memory consumption

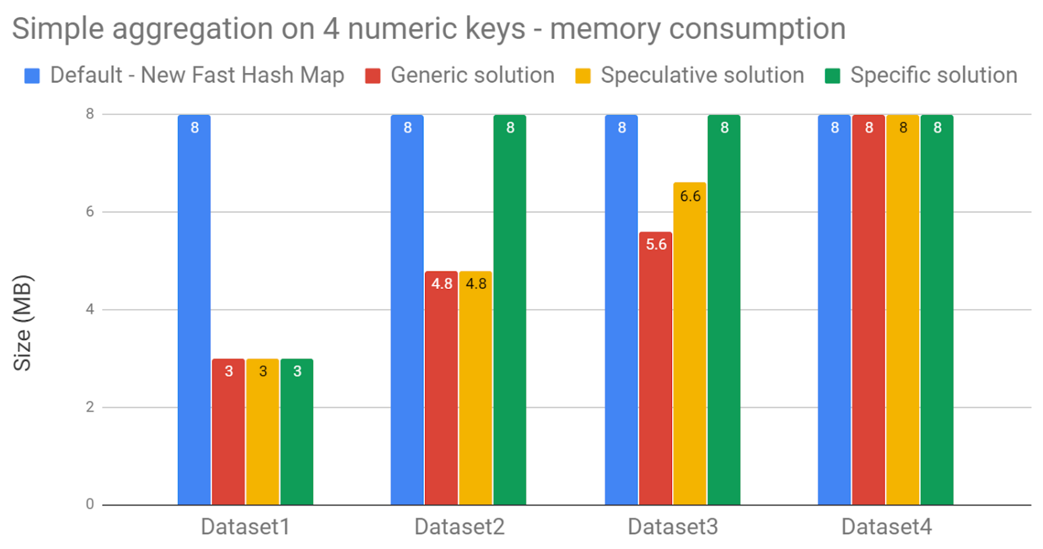


Figure 6.4: Comparison of the small hashmap memory consumption between the default version of Spark, using our optimized small hashmap, and three versions (for each code generation strategy) of Spark that employ numerics compression in the new small hashmap

We can see in figure 6.4, that the generic solution has the lowest memory consumption because it creates and uses the optimal packing configuration, while the speculative solution may greedily select the first code path(heuristic) that works. In the third scenario, the speculative code chooses a code path that compress the four keys in two, while the optimal solution is a one-word column. Each dataset allows a different number of rows to be compressed (100%, 75%, 50%, 0%), thus the difference in size from one test to another. All in all, even though the bit-packing technique brings a small improvement in query execution performance, it has a greater impact on the hashmap memory consumption (almost 3x reduction). In the current experiments, the L2 CPU cache size is 4.5 MB and the default size required by the stored records is 8MB. For the first dataset, after the compression, the records have only 3MB, easily fitting in the CPU L2 cache.

Chapter 7

Conclusion

In this thesis, we proposed an improved design for the small hashmap, present in the partial aggregation stage. Moreover, we presented two orthogonal approaches of compressed execution and integrated them in this optimized small hashmap. The first compression technique targets only String data and restricts the scope of the project only to the aggregations that feed off a Parquet scan, to leverages its `dictionary_encoding` scheme. The second method employs bit-packing on numeric datatypes to squeeze multiple columns together and reduce the size of the hash table records. While evaluating the performance of these contributions, we obtained up to 5x speedup for custom queries and up to 1.9x on TPC-H Query 16 . We also presented experiments in which our contribution had little to no benefit. Overall, we observed improved query execution performance when applying compressed execution. Nevertheless, with this project, we answer the proposed research questions:

- **RQ1:** How do we efficiently build on-the-fly unified String dictionaries? How do we determine, at runtime, which String columns are dictionary encodable?

  In our objective of compressing Strings, we restrict the scope of our problem only to aggregations that feed off a Parquet scan, and leverage its dictionary encoding, where available. The unified String dictionary is efficiently built, in sublinear time, by decoding only the distinct Strings found in the Parquet dictionary pages. As far as the columns identification is concerned, we analyze the query plan, at runtime, and discover which key-columns are part of both the group-by and the scan operator. The selected columns are marked for specific code generation and modified such that we can leverage the stored Parquet dictionary.

- **RQ2:** What kind of data statistics do we need to obtain before compressing the data on-the-fly? How do we deal with outliers? What is the overhead of detecting them? How do we quantify the precision of our generated statistics?

  We require min-max information, so that we have some knowledge about the data when applying bit-packing. The only available min-max statistics are available either in the Parquet files or in the Delta log, but these are susceptible to outliers and, in this case, the compression may not be optimal. Therefore, at query runtime, we look at a small sample of rows and compute our own data statistics that avoid outliers.

In chapter 6, we detail how the metadata generation works and how it avoids the possible outliers. Even though, in the benchmarks in Section 6.3, we show the 0.3s overhead from detecting the outliers, we did not evaluate the tradeoff between data statistics precision and outlier elimination overhead. This is an open question that we want to investigate in the future work.

- **RQ3:** How do we adaptively compress multiple columns together? How do we make our approach effective and still robust?

  We use the generated statistics to create a packing configuration that dictates(Chapter 6) how each numeric column should be compressed. Since the WSCG happens before query runtime, we speculatively generate multiple code paths for the small hashmap. At runtime, we decide, based on the computed *packing configuration*, which code path should be used for the small hashmap. The speculations aim at optimizing the generated code and making it closer to specific code. To make our approach robust, we use an additional small hashmap only for the unpackable rows, avoiding the latency for the immediate fallback to the `BytesToBytesMap`.

- **RQ4:** How do we integrate our project into the whole-stage code generation (WSCG)? What compressed aggregation code should we generated before runtime, for when the data statistics are known? How do we limit the number of efficient code paths to keep code generation time low, while still providing enough flexibility to adapt to runtime data distribution information?

  The first compressed execution technique, String compression, is part of both the scan and the aggregate operator. In the scan operator, the unified String dictionary is built from the Parquet dictionary pages and provides compact Strings (Section 5.1.2) for the aggregation. The second compression technique also works with both scan and aggregate operators. In the former, we generate the data statistics and compute the packing configuration, while in the latter, we compress the data and work with it in a compact form.

  For when the min-max information is known before runtime, we generate a very specific code that eliminates conditional branches, and replaces known variables with constants. Otherwise, when the min-max is obtained at runtime, we speculatively generate multiple code paths and let the *packing configuration* decide which code path to select for the current batch of rows. Unlike the generic code, the speculative generation has fast specific regions of code. In this project, we did not investigate when the generation time is overburdened and slowed down by the high number of code paths. However, from the data science (Chapter 3), we know that the average number of keys found in group-by is 4 and we limited the maximum number of generated code paths to half the total number of numeric keys (considering that at least two columns can be compressed as one), to keep the generation time as low as possible.

- **RQ5:** How do we evaluate our system's performance?

We evaluate our system using both manually created benchmarks and TPC-H Query 1 and 16 with a scale factor of 100. The custom benchmarks are used to motivate the decisions we made in our project and to provide some examples of best-case scenarios, while the TPC-H queries are used to validate our system against standard industrial queries. The selected TPC-H queries are favorable for the String compression approach, supporting dictionary encoding. In our evaluation, present in Sections 4.4, 5.3, 6.3, we obtained a speedup of 1.9x for TPC-H Q16 and 5x in custom benchmarks.

7.1 Future work

Looking at the overall benchmarking results, we can say that compressed execution has achieved improved query execution performance. We believe that there are more regions to explore in this project, therefore, we highlight the possible future work:

- For the moment, the small hashmap operates only on primitive numeric types and Strings. If we support complex types, such as decimal, we can leverage the Parquet dictionary encoding for them as well. Additionally, we want to evaluate the bit-packing technique when applied on Strings.
- Since the hash join operator also uses an in-memory hashmap, we would like to apply all of the contributions, presented in this project, to joins and evaluate the possible gains. The `UTF8String` format is the universal representation of Strings in Spark, therefore the String compression would work out-of-the-box. We still have to investigate how the numerics compression will fit into the hash join operator.
- The small hashmap is available only in the partial aggregation. Therefore, we want to apply some of the compression approaches to the `BytesToBytesMap`, so that the benefit of compression can work even if the partial aggregation stage does not exist.
- The outlier accuracy and detection overhead was not investigated in this project, as well as the precision of the obtained data statistics. Therefore, these research questions are open for analysis.
- Spark can generate data samples, of a specified number of rows, from the whole dataset. Having such samples available before query runtime means that we can move the data statistics extraction before WSCG. Hence, we can generate a very specific fast code while being able to avoid the outliers in our metadata.

References

[1] **Apache Parquet**. https://spark.apache.org/, 2013. [last accessed March-2019]. 5, 9

[2] **Apache Parquet**. https://parquet.apache.org/, 2013. [last accessed March-2019]. 5, 7, 14

[3] DBTEST. **dbtest.io**. http://dbtest.dima.tu-berlin.de/, 2018. [last accessed April-2019]. 5

[4] Alfons Kemper Viktor Leis Tobias Muehlbauer Thomas Neumann Manuel Then Adrian Vogelsgesang, Michael Haubenschild Jan Finis. **How Benchmarks Fail to Represent the Real World**. http://dbtest.dima.tu-berlin.de/media/DBTEST.io_Presentations/dbtest_vogelsang_18-06.pdf, 2018. [last accessed April-2019]. 5

[5] Andreas Krall. **Efficient JavaVM just-in-time compilation**. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pages 205–212. IEEE, 1998. 6

[6] Rasmus Pagh and Flemming Friche Rodler. **Cuckoo hashing**. *Journal of Algorithms*, **51**(2):122–144, 2004. 6

[7] Pedro Celis, Per-Ake Larson, and J Ian Munro. **Robin hood hashing**. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE, 1985. 6

[8] Goetz Graefe and Leonard D Shapiro. **Data compression and database performance**. In *[Proceedings] 1991 Symposium on Applied Computing*, pages 22–27. IEEE, 1991. 7

[9] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating compression and execution in column-oriented database systems**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006. 7

[10] **SAP HANA**. https://developers.sap.com/topics/sap-hana.html, 2010. [last accessed April-2019]. 7

[11] **Teradata**. https://www.teradata.com/, 1979. [last accessed April-2019]. 7

[12] **Vectorwise**. `https://www.actian.com/analytic-database/vector-analytic-database/`, 2014. [last accessed April-2019]. 7

[13] **Databricks Delta Lake**. `https://docs.databricks.com/delta/index.html`. [last accessed March-2019]. 7, 16

[14] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. **DB2 with BLU acceleration: So much more than just a column store**. *Proceedings of the VLDB Endowment*, **6**(11):1080–1091, 2013. 7

[15] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. **Byteslice: Pushing the envelop of main memory data processing with a new storage layout**. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 31–46. ACM, 2015. 7

[16] Yinan Li and Jignesh M Patel. **BitWeaving: fast scans for main memory data processing**. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2013. 7

[17] Florian Funke, Alfons Kemper, and Thomas Neumann. **Compacting transactional data in hybrid OLTP&OLAP databases**. *Proceedings of the VLDB Endowment*, **5**(11):1424–1435, 2012. 7

[18] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. **HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing.** *IEEE Data Eng. Bull.*, **35**(1):46–51, 2012. 7

[19] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. *Superscalar RAM-CPU cache compression.* IEEE, 2006. 7

[20] Ingo Müller, Cornelius Ratsch, Franz Faerber, et al. **Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems.** In *EDBT*, pages 283–294, 2014. 7

[21] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. **Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation**. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326. ACM, 2016. 7

[22] **Apache CarbonData**. `https://carbondata.apache.org/ddl-of-carbondata.html`, 2017. [last accessed April-2019]. 7

[23] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. **Dictionary-based order-preserving string compression for main memory column stores**. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 283–296. ACM, 2009. 7

[24] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. **SAP HANA database: data management for modern business applications**. *ACM Sigmod Record*, **40**(4):45–51, 2012. 7

[25] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. **Spark: Cluster computing with working sets.** *HotCloud*, **10**(10-10):95, 2010. 9

[26] Wenchen Fan Ron Hu, Zhenhua Wang and Sameer Agarwal. **Cost Based Optimizer in Apache Spark 2.2**. `https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html`, 2017. [last accessed April-2019]. 9

[27] **Spark Performance Analysis**. `https://kayousterhout.github.io/trace-analysis/`, 2015. [last accessed April-2019]. 10

[28] Thomas Neumann. **Efficiently compiling efficient query plans for modern hardware**. *Proceedings of the VLDB Endowment*, **4**(9):539–550, 2011. 12

[29] **Catalyst Optimizer**. `https://databricks.com/glossary/catalyst-optimizer`, 2015. [last accessed April-2019]. 13

[30] **UTF-8**. `https://unicode.org/standard/standard.html`, 1991. [last accessed April-2019]. 13

[31] **Databricks**. `https://databricks.com/`, 2013. [last accessed March-2019]. 16

[32] **Open-source Delta Lake**. `https://delta.io/`. [last accessed March-2019]. 16

[33] **Processing Petabytes of Data in Seconds with Databricks Delta**. `https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html`, 2018. [last accessed March-2019]. 16

[34] **TPC-H: industry standard**. `http://www.tpc.org/tpch/`. [last accessed March-2019]. 28

[35] **Java Profiling tool**. `https://github.com/jvm-profiling-tools/async-profiler`. [last accessed March-2019]. 31

[36] Brendan Gregg. **Flamegraph**. `http://www.brendangregg.com/flamegraphs.html`. [last accessed March-2019]. 31