

Seamless Integration of Parquet Files into Data Processing

Alice Rey,¹ Michael Freitag,¹ Thomas Neumann¹

Abstract: Relational database systems are still the most powerful tool for data analysis. However, the steps necessary to bring existing data into the database make them unattractive for data exploration, especially when the data is stored in data lakes where users often use Parquet files, a binary column-oriented file format.

This paper presents a fast Parquet framework that tackles these problems without costly ETL steps. We incrementally collect information during query execution. We create statistics that enhance future queries. In addition, we split the file into chunks for which we store the data ranges. We call these *synopses*. They allow us to skip entire sections in future queries.

We show that these techniques only add a minor overhead to the first query and are of benefit for future requests. Our evaluation demonstrates that our implementation can achieve comparable results to database relations and that we can outperform existing systems by up to an order of magnitude.

1 Introduction

Data is stored less and less in relational database management systems (RDBMS) [AAS13; A112; Id11; Ka14; O117]. Instead, users tend to store large amounts of data in data lakes with standardized file formats such as Parquet [ASF13]. Nevertheless, the users still expect the same performance as if the data resides in an RDBMS, which improvised tooling cannot achieve. Even though database systems are much better suited for data exploration, existing RDBMSs that support directly querying files still cannot reach the performance of database relations. They face the problem of not having any insights about the underlying data, which are crucial for efficient data access and query plan optimizations. Usually, an RDBMS knows the processed data well since it can collect all sorts of information while the user loads the data into the database.

Parquet files are one of the most used data structures for storing large amounts of data. Big companies like Twitter [De13], Netflix [WG17] or Skyscanner [SE19] use it to store large amounts of data for big data analysis. The column-wise storage format is close to how an RDBMS with a columnar storage engine would store the data, making the Parquet format a great candidate for integrating it into a data processing pipeline.

The Parquet file format tries to balance data compression and data access. The format supports different encoding and compression schemes. Therefore, additional decoding and

¹ Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany, {rey, freitagm, neumann}@in.tum.de

decompression steps are required to access the underlying values, which database relations typically do not have. Furthermore, the Parquet file offers a lot of optional fields such as the minimum and maximum values or the number of null values, which can be helpful for query execution. However, since these fields are optional, we cannot rely on them.

Naive file access would be very costly if the dataset resides on a remote server. The *lineitem* table with 10 million rows of the TPC-H dataset at scale factor 10 has a total size of 2 GB when we store it in a Parquet file generated by Spark [ASF14]. When accessed via HTTP, it would require 16 seconds to download the entire file in a typical local network with a bandwidth of 1 Gbit/s. If we only require one column, a smart access logic can minimize the download time to less than 2 seconds. Parquet files allow us to use byte-range requests to only download required columns. In addition, we also keep structural information about the data in a fixed number of synopses. We split the file into equally sized chunks and track the minimum and maximum value of the contained data, which allows us to skip entire chunks in future queries. Since this technique was already used in previous works under different names [E113; La16; Mo98; O117; Zi17], we chose the generic name *synopses*. To keep the computational overhead of these synopses small, we only compute the data ranges for columns that are currently required. If a user requests additional columns in the future, the corresponding ranges can be added incrementally.

Most users will switch to more complex analytical queries after an initial exploratory phase, where a fully-fledged database system with a query optimizer comes in handy. Database systems usually perform great on complex queries since they have prior knowledge about the data gathered while copying it into database relations. Since we skip this step, we save initialization costs and avoid accessing data we would never use. On the other hand, we lose crucial information that might have helped during future query optimization steps. Therefore, we take the Parquet view feature one step further by allowing a smooth transition from exploratory data analysis to more complex analytical queries. We will not add an initial scan over the Parquet file, but instead, we will collect samples and sketches of the used data while executing the queries. We do not expect complex operators during the exploratory phase so that we can accept some slowdown. We then benefit from the information we gathered during the exploratory phase for later queries. To the best of our knowledge, our implementation is the first to reach comparable execution times to database relations for Parquet files with these techniques.

The key contributions of this paper are:

1. A smart access logic for Parquet files that introduces minimal initial overhead to the query execution time and improves performance over time.
2. An incremental procedure for cheap statistics computation that enhances query optimization steps for future queries.
3. A remote access strategy that minimizes execution time overhead.

The remainder of this work is structured as follows: We start by briefly explaining the challenges the Parquet file format introduces to an RDBMS in Sect. 2. Second, in Sect. 3, we explain the techniques we used to conquer the imposed challenges and how scanning Parquet files can be integrated into database systems. Third, we perform a detailed evaluation of our implementation in Umbra [NF20] with different benchmarks (Sect. 4). Finally, we conclude with some related work in Sect. 5 and summarize the work in Sect. 6.

2 Background

Parquet files are an excellent match for databases that use a column-wise storage format due to their columnar file format. Nevertheless, Parquet files pose some challenges when integrating them into query processing. In this section, we will discuss the structure and versatility of the Parquet format in the first two subsections to understand the need for certain design decisions. Finally, we conclude the section with the challenges that the file format introduces.

2.1 Parquet File Format

Parquet stores data in columnar format and is not human-readable in contrast to CSV or JSON files. Instead, the goal of the Parquet format is to store data as densely as possible. Fig. 1 (b) shows the basic structure of a Parquet file. The data is first separated into row groups that split the dataset horizontally. Each row group is then split into column chunks vertically, storing each column in a separate column chunk. The actual elements of the column chunk are stored on one or more data pages. Parquet does not enforce the number of rows which should be stored per row group or page. In addition, the number of rows stored per page does not need to be synchronized between the column chunks of the same row group or the same column. In our example, we store the values of x on two pages in the first row group and on three pages in the second row group. Column y stores the values on four pages and column z on one page for both row groups.

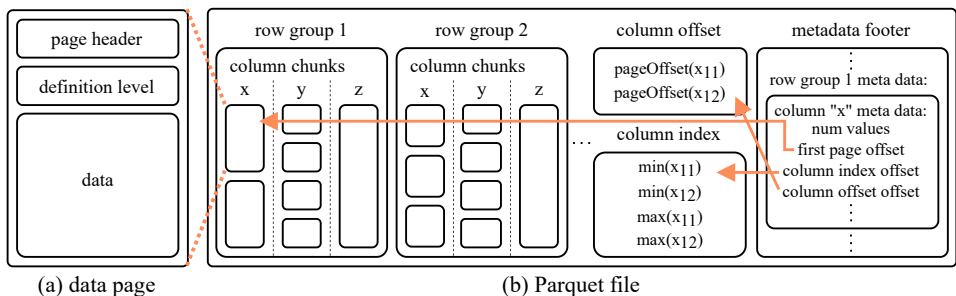


Fig. 1: Parquet File Structure

The schema and data arrangement are stored in the Parquet file's footer. In our example, we display the first row group's metadata of column x . In addition to much other information, the column chunk has to store the offset of the first page as the starting point. The *column index* and *column offset* data structures, which are usually located close to the metadata footer, are optional and can be referenced via their offset. The *column offset* stores the offsets to all pages of the column chunk, and the *column index* stores the minimum and maximum value of each page of the column chunk.

In Fig. 1 (a), we display the general structure of a data page. Each page starts with a page header that contains the page type, the number of contained values, the used encoding, and the compression. The page writers we examined used SNAPPY [GG11] as the default compression format for pages. The most commonly used page encoding we found is dictionary encoding, where the actual values are stored on a dictionary page, and the data pages store indices into the dictionary. The number of bits required to store the dictionary indices depends on the number of elements in the dictionary. The indices are stored in multiple run-length encoded or bit-packed encode runs. Nevertheless, the encoding can switch to plain pages if the number of distinct elements exceeds the dictionary size.

Parquet also supports nested data types with the record shredding and assembly algorithm presented in the Dremel paper [Me10]. Datasets containing nested types violate the first normal form, which poses a fundamental challenge for any relational database management system regardless of the specific data storage format [DLN21; Sh99]. Our framework is able to scan any nesting level. Connecting multiple levels is a problem we consider to be orthogonal to the objective of this paper. We plan to investigate possible solutions in future work using our current framework as the base layer. The main idea of the Dremel algorithm is to use definition and repetition levels to store how many elements belong to the same parent element and at which level a value is undefined. The *definition level* is located below the header and is also relevant for nullable columns. If the column chunk is nullable, each page keeps a *definition level buffer* at the beginning of the page, which stores for each row if it is NULL with run-length and bit-packed-encoded. The rest of the page contains the actual values. Since the definition level already encodes NULL values, only non-NULL values are stored in this section. Parquet supports a fixed set of physical types such as *BOOLEAN*, *INT64*, or *BYTE_ARRAY*. All other types combine a physical type with a converted type. To store decimal values in a Parquet file, the physical type *INT64* is combined with the converted type *DECIMAL*. The optional fields *precision* and *scale* store additional information about the type.

2.2 Versatility of Parquet Writers

In the previous section, we described the general structure of a Parquet file. However, the data in a Parquet file can be spread over the row groups and the pages using any encoding and compression the writer or user wants. This freedom leads to inhomogeneous files even though they have the same file format. Our goal is to have a fast Parquet framework that

can achieve good performance independently of the used Parquet writer. Therefore, in this section, we look at three different Parquet writers to show how much Parquet files differ even though they store the same data.

Generator	Rows per Row Group	Pages per Row Group	File Sizes		
			SF1	SF10	SF100
CSV	-	-	719 MB	7.2 GB	74 GB
Spark	3,000,000	150	192 MB	2.1 GB	20 GB
uncompressed	3,000,000	150	333 MB	3.3 GB	33 GB
DuckDB	100,352	1	281 MB	2.8 GB	28 GB
Arrow	67,108,864	15 - 1800	189 MB	2.0 GB	20 GB

Tab. 1: Parquet Writer Comparison

In Tab. 1, we listed the properties of three different writers: Spark [ASF14], Arrow [ASF16], and DuckDB [RM19]. To measure their differences, we let each generator write the *lineitem* relation of the TPC-H benchmark to a Parquet file for the scale factors 1, 10, and 100. Apart from the size of the generated Parquet files, we listed the size of the underlying CSV file created by the TPC-H generator to show the compression ratios that the different writers achieve. The *lineitem* relation contains 6 million rows times the respective scale factor. We also included a version where we force Spark not to use any compression to measure the benefit of page-level compression with SNAPPY.

For each generator, we measure the number of rows and the number of pages that are stored per row group. The Spark and DuckDB Parquet writers store a fixed number of elements per page and a fixed number of pages per row group. Since Parquet does not force synchronization between the column chunks, there are writers such as Arrow that do not store the same number of elements per page. Arrow uses a fixed data page size between roughly 0.5MB and 1 MB. For DuckDB and Spark, the page sizes vary from 0.5 MB to 6 MB. Out of all three writers, DuckDB has the worst compression ratio of 2.6. The major difference between DuckDB and the other two formats is that DuckDB does not use any encoding for the columns of the *lineitem* table. Spark and Arrow have a very similar compression ratio of 3.6. If we force Spark to write the pages uncompressed into the Parquet file, the resulting file size is less than 20 % bigger than the DuckDB file. Even though we only cover three different Parquet writers, we have already observed two extremes. DuckDB and Arrow do not take advantage of the hierarchical data layout: DuckDB will only use one page per row group, and Arrow stores the entire dataset in one row group for scale factor 1 and 10 since each row group stores 67 million rows.

2.3 Parquet Format Challenges

In the two preceding sections, we pointed out the structure of the Parquet files and their versatility. We will conclude our preliminary discussion with the format's challenges on

the Parquet integration. We highlighted the freedom Parquet writers have in Tab. 1. The consequence of this freedom is that we cannot assume anything about how the data is split across the Parquet file hierarchy. Unfortunately, especially for parallel processing, this freedom makes scanning more difficult. If Parquet, for example, forces all writers to store the same amount of elements per page for each column, this would allow more optimized accesses. The elements of one row would all be stored at the same page index and the same page offset in the different column chunks, at least for non-nullable columns.

Parquet offers a lot of data structures such as the *column index* that greatly facilitate data access since we can directly jump to the page that contains the required value. Nevertheless, a lot of these structures are optional. If the data structure does not exist, we need to access all preceding page headers to compute the offset of the required page. Parquet’s encoding and compression techniques can minimize the size of Parquet files. On the other hand, it adds additional decompression and decoding steps when reading the data. Dictionary encoded pages use multiple run-length or bitwise encoded runs. Each run stores the number of values it encodes in its header. If we search for an element at a specific page offset, we have to access each run to compute the start of the following run until we reach the required offset.

The definition level encoding is another limiting factor. Storing only non-NULL values on the page helps decrease the overall size of the file. However, if we want to access an element at a specific page offset, we have to scan the definition level first to count the number of preceding non-NULL values to get to the actual offset. Some data types stored in Parquet also introduce additional transformation steps. For example, *BYTE_ARRAY* elements are represented as 32 bit values for the length followed by the actual value. We transform these into fixed-length 16 byte fields in our internal buffer. If the strings are longer than 12 bytes, we store the size of the string and the first 4 bytes of the value in the upper 8 bytes. The lower 8 bytes contain a pointer that points to the location where the entire string is stored [NF20].

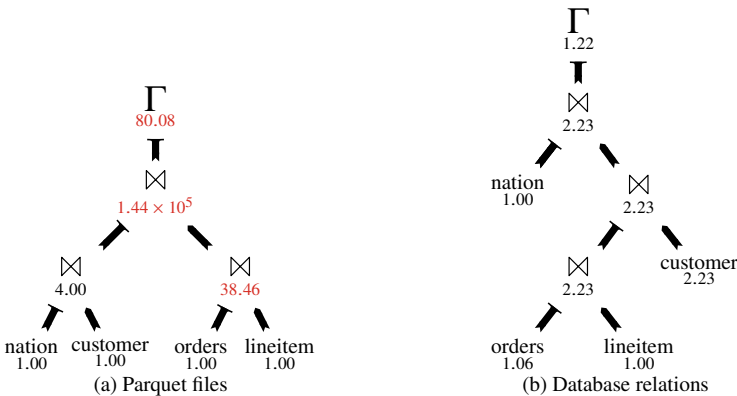


Fig. 2: Q-errors of the cardinality estimates of TPC-H query 10 with scale factor 10.

The only statistics available in Parquet files are the cardinality of the contained dataset and each page’s minimum and maximum values. Unfortunately, the minimum and maximum

values are optional fields, so Parquet writers are not forced to use them. Out of three Parquet writers we analyzed, only Spark stores minimum and maximum values. These minimum and maximum values, as well as the cardinality of the datasets, are the only sources available for performing cardinality estimates. Therefore, we get imprecise results since we do not know how the data is distributed within the given boundaries. As a consequence, we get erroneous cardinality estimates and suboptimal query plans.

For efficient query processing, it is necessary to produce optimal query plans independently of the amount of metadata the Parquet file provides. Since cost-based query optimizers heavily rely on statistics and samples gathered during table initialization, the optimizer suffers a lot when these do not exist. In Fig. 2, we visualize the optimized query plans of TPC-H query 10 based on Parquet files on the left and database relations on the right. For each operator, the optimizer tries to estimate the resulting cardinalities. The q-error next to each operator gives us the deviation of the estimates from the actual cardinality [MNS09]. In the case of the Parquet file version, the estimated result cardinality of the join between *orders* and *lineitem* is off by a factor of 38. The q-error gets even worse with the following join with a factor of 144 thousand, which is the reason for the suboptimal query plan for the Parquet files. This shows how crucial a good cardinality estimate is for a Parquet scan to be an acceptable alternative to database relations. The Parquet scan cannot get close to the execution times of database relations as long as the query optimizer cannot choose the same query plans for the Parquet files.

3 Integrating Parquet Files into Query Processing

We now explain the different techniques for integrating Parquet files seamlessly into a query processing pipeline. We tackle their complex structure, versatility, and lack of metadata described in the previous section. We propose a parallelization technique with stable performance, independently of how the data is distributed over the Parquet file hierarchy. In addition, we minimize the amount of data we have to access, which is especially beneficial for remote files. Thirdly, we present a technique for collecting information about the dataset that benefits future queries without notably sacrificing the execution time of the currently executed query.

Figure 3 describes the concept of how future queries can benefit from information gathered in preceding scans. We visualize a simplified execution of three different queries, executed sequentially from left to right. All three queries access the same Parquet file during the *Parquet Scan* phase. The Parquet file is split into four row groups ($r1$, $r2$, $r3$, $r4$) and contains three columns, namely x , y , and z . Throughout the three queries, we incrementally collect information about the Parquet file. In the *statistics* we store HyperLogLog [F107] sketches to estimate the distinct values. We also keep a small set of rows as a sample for multi-column estimates and predicate selectivity estimates [FN19]. The *synopses* store the minimum and maximum value of each column chunk for each row group. To limit the required space of synopses, the user can choose an upper limit for synopses. If we have more row groups

than synopses, multiple row groups are grouped into one synopsis. With the help of the statistics, further queries can choose better query plans during the query optimization phase. The synopses can be used in future queries to skip the row groups that do not meet the restrictions of the queries. This is only applicable if the data in the Parquet file is implicitly clustered, which is often the case in real-world datasets [Mo98; Vo18].

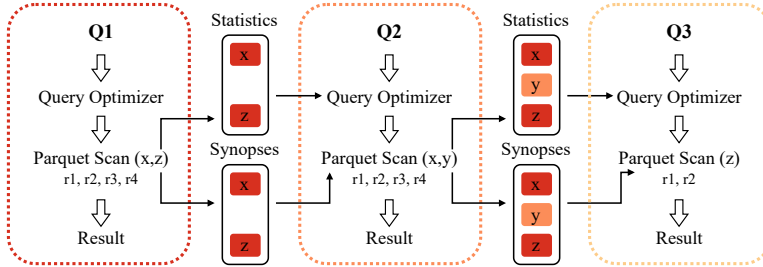


Fig. 3: Sequential execution of three queries (Q1, Q2, Q3) with the incremental computation of synopses and statistics for the accessed Parquet file.

The first query (Q1) in Figure 3 requires the columns x and z . We do not have any prior knowledge about the Parquet file during the query optimization phase, which we could consider. Therefore, we most likely end up with a suboptimal query plan. During the Parquet scan, we will access the column chunks of x and z of all row groups. We store statistics about those two columns and leave space between them for the second column y . We also track the data ranges of all four row groups for both requested columns and store them in the synopses. The next query (Q2) requests the columns x and y . The query optimizer can consider the statistics for column x that we gathered during the first query, which will help select a better query plan. While scanning the Parquet file for the second query, we will gather information about the column y , which we can use to fill in the gap in the statistics and the synopses.

Since Q1 and Q2 cover all columns, any following query will not have to compute any statistics or synopses. For query 3 that requests column z , the query optimizer can work with the statistics and develop a cost-optimized query plan as it could for any database relation. During the *Parquet Scan* phase, we can take advantage of the synopses. Query 3 restricts column z . Since we know the minimum and maximum values, we can check if the requested range and the row group range overlap for each row group. In the case of the third and fourth row groups, the ranges do not overlap. For this reason, we will only process the first and second row groups, which we reference with $r1$ and $r2$ in Figure 3. Query 2 cannot take advantage of the synopses yet, since we still need to gather information about column chunk y while executing that query. To compute correct statistics for column y , we need to consider all row groups of that column.

To achieve stable parallelization over the Parquet file, we looked at different possible ways of splitting the Parquet file, which we visualized in Fig. 4. We split the entire workload into

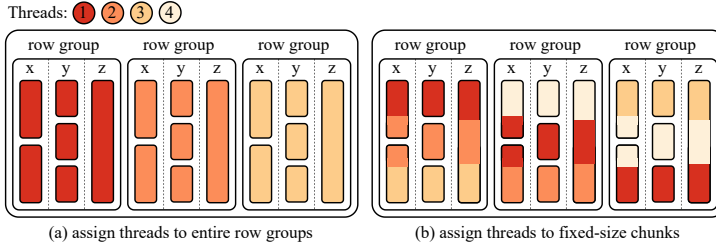


Fig. 4: Comparison of different parallelization approaches for Parquet files.

smaller work units to be able to process them independently of each other by all threads in a work-stealing framework such as the one provided by Umbra [NF20]. In our example, the Parquet file consists of three row groups that contain 60,000 rows each. We assume that we have four threads available for this Parquet scan. The sections that each thread is scanning are colored in the corresponding color.

In Fig. 4 (a), each thread is assigned to an entire row group. One of the threads is idle in this case because we do not have enough row groups in the Parquet file. As we know from Tab. 1, there are Parquet file generators such as Arrow that only use one row group to store 60 million rows. In that case, all threads except for one would be idle. Therefore, we opted for another more adaptive approach, as visualized in (b). We work with a fixed-sized chunk size. For this example, we assume a chunk size of 20,000 rows. Our goal is to keep all threads busy while ensuring that the helper structures required for the scan do not take up too much space. Therefore, we limit the number of row groups we process in parallel. We tested different spaces between 265 MB and 4 GB and achieved the best results by limiting the required space to 1 GB.

3.1 Parallelizing the Parquet File Scan

Achieving stable parallelization over Parquet files requires parallelizing the scanning below row group level. We split the Parquet files into independently processable chunks to reach similar performance to queries that are based on database relations and executed in a work-stealing framework. The maximum granularity level is the row group size. We further split each row group into fixed-sized chunks, which we call morsels [Le14], depending on the row group size. The threads that are processing morsels of the same row group will have to access and prepare the same file ranges. We therefore reduced the amount of duplicate work as much as possible while avoiding contention. In Fig. 5 we describe how we parallelized our framework and how the different aspects we already mentioned work together.

During the setup of the scan, we use the synopses to filter out row groups that do not meet the restrictions, which are the filter predicates of the query (Line 3). If the synopses are

```

1 def setupScan:
2   if !computeStatistics:
3     requiredRowGroups = synopses.filter(rowGroups)
4   else:
5     if noStatsAvail: computeSamplePos()
6     else: loadSamplePos()
7   computeConcurrentRowGroupCount()
8   rowSize = sum(requiredColumnSizes)
9   blockSize = bufferSize / rowSize
10  bufferPointers = computeColumnWriterOffsets()
11
12 def prepareDictionary(column c, dictionary d):
13   if d.isCompressed:
14     singleAccessBlocking() => {
15       rowGroupBuffer[c][d] = decompress(d)}
16   if c.type == 'BYTE_ARRAY':
17     singleAccessBlocking(convertDictionaryEntries)
18
19 def prepareDataPage(column c, page p):
20   if p.isCompressed:
21     singleAccessBlocking() => {
22       rowGroupBuffer[c][p] = decompress(p)}
23   if p.dictEncoded:
24     prepareDictionary(c, rowGroupBuffer.dictionary)
25   if p.dictEncoded or c.type == 'BYTE_ARRAY':
26     singleAccessNonBlocking(computeSparseOffsets)
27
28 def moveToOffset(row, column):
29   page = moveToPage(row)
30   prepareDataPage(column, page)
31
32   if sparseOffsetsComputed:
33     moveToSparseOffset(row)
34     moveToRowOffset(row)
35
36 def readColumn():
37   moveToOffset(morsel.currentRow, column)
38   insertElements(column)
39
40 def readNextBlock(morsel):
41   if computeStatistics:
42     for column in reqColumns: readColumn()
43     computeStatistics()
44   else:
45     matches = list(range(0, blockSize))
46     moveToPageWithMatches(restrictedColumns)
47     for column in resColumns:
48       readColumn()
49       checkRestrictions(matches)
50     for column in unresColumns: readColumn()
51
52 setupScan()
53 threads.doInParallel((morsel) => {
54   # wait until rowGroup in rowGroupBuffer
55   singleAccessBlocking(setupRowGroupProps)
56   while not morsel.allRowsProcessed():
57     readNextBlock(morsel)
58     processBlock()
59 })

```

Fig. 5: Pseudocode for scanning Parquet files in parallel

not available yet, or if we need to compute statistics for a column that is accessed for the first time, all row groups will be processed. As part of the statistics, we also retrieve a data sample. If the file is accessed for the first time, we compute 1024 random sample positions (Line 5). We allocate enough space to store the sample values of all columns. Then, when the user queries the file again and requires other columns this time, we incrementally add samples for these columns as well. We also choose the number of row groups we will process in parallel such that all threads are busy while the required space is limited. The next step is to prepare for each thread a fixed-sized buffer in which we will write the values of all required columns (Lines 8 to 10). At first, we compute the space we would require to store one row, which is the sum of the sizes of all required columns. We group the data inside the buffer by column for the vectorized evaluation of restrictions and to be able to copy entire blocks if possible. We use a fixed-sized buffer between the Parquet file layer and the next operator since it allows consistent processing of all columns independent of how they were stored inside the file. For strings, the database engine has to use an out-of-line format to allow fixed-sized elements. Many columns are dictionary encoded by the writers we examined in Tab. 1 and many data types require additional transformation steps to data types available in the RDBMS. Therefore, we have to touch most of the tuples either way and storing them into a buffer only adds minor overhead. The buffer makes our approach applicable for both push-based execution models [Ne11] as well as vectorized execution models [BZN05].

After the initial setup, all threads can work in parallel (Line 52). The row groups are split into morsels that are assigned to free threads until no morsels are left. Whenever a thread starts processing a specific morsel, the thread has to check at first if the currently required row group is loaded into the row group buffer. Then, it sets up the data structures for the row group which is only done by one morsel per row group. Afterwards, the thread processes the morsel in `blockSize` chunks that fit into the fixed-sized buffer by repeatedly calling the `readNextBlock` function until no data is left.

For each column, we will at first move to the correct offset and then insert the elements at the correct position in the buffer, given by the `bufferPointers`. Each morsel can independently compute the start position of the current block since we know how many rows are stored per row group and how many values are stored per page. Given the currently required row number, we will move to the page that contains the row we need, prepare the page and then move to the correct offset inside the page. For the page preparation (Line 19), we let only one thread decompress the page and store it in a buffer where all other morsels can access it afterwards. If the page is dictionary encoded (Line 12), one morsel per row group decompresses the dictionary and converts the dictionary into a fixed-sized out-of-line representation in case of a *BYTE-ARRAY* column. For all other data types, the index can be multiplied by the data type size to get to the correct offset in the dictionary page. The same holds for data pages. If Parquet stores the values plainly, we can get to any offset by multiplying the required offset by the data type size. If the data page is dictionary encoded or stores plain *BYTE-ARRAY* values, we can not easily jump to a certain offset. Given we have multiple morsels that work on the same page, they all have to start reading from the start of the page to reach their required offset. Letting one morsel compute sparse offsets distributed over the data page (Line 26) helps future morsels to reach the required offset earlier. To avoid any contention at this step, we do not let the other threads wait until the offsets are available. If they are available, the other threads can use them (Line 31). In the last step we move to the actual row offset, either starting at the beginning of the page or starting from the closest sparse offset if available.

For the loading step (Line 37), we favor copying whole data blocks at once for each column, which is only possible for fixed-size, plainly stored, non-nullable columns that need no transformation step. For nullable columns, we first have to scan the run-length-encoded and bit-packed-encoded runs of the *definition level buffer*. Since the pages store non-null values densely, we will have to check how many non-null values we have to read. In our buffer, we do not store the non-null values densely. Instead, we store the null information for each tuple in one byte and leave an empty gap if the element is null. Leaving gaps for null values makes the access cheaper since we can precompute all offsets independently of other rows.

If we access columns for the first time, we will compute statistics and synopses for each block independently after all the values of all columns are loaded into the buffer (Line 42). If the Parquet file stores min/max information, we use these to compute the synopses only once per row group.

If we do not have to compute statistics (Lines 43 to 49), we will at first skip as many pages as possible based on our restrictions and the min/max information from the Parquet file. Afterwards, we start loading the restricted columns one after the other into our buffer and evaluate the restrictions with vectorized functions. In the matches list, we store which indices of our current block are still valid. At the beginning, all indices are stored in the list since all are valid. Each restricted column evaluates its restriction on its values and updates the matches list accordingly. Further restricted and unrestricted columns use the matches list to only load the values of rows that are still valid.

3.2 Parquet File Access

The data retrieval becomes our main bottleneck if the Parquet file is located on a remote server. We will not download the entire file at once since this would include a lot of unused data, especially if the user only requires a small fraction of the entire dataset. We can guarantee that we only request the required data with byte-range requests, given our remote source is accessible via HTTP. As with query 3 in Fig. 3, we can use the synopses computed in preceding requests to exclude row groups that do not match the restrictions of the query, and we only retrieve the column chunks that the query requires.

During data exploration, users will likely access the data of a queried Parquet file again in the following queries. Therefore, if the user works on a remote Parquet file, it is crucial to cache the used data locally to reduce the access costs for future queries. Since we want to integrate the Parquet files into an existing database system, we can utilize its buffer manager. The buffer manager in Umbra is based on LeanStore [Le18] with the addition of variable-size pages [NF20]. At first, we will request the size of the Parquet file with a *HEAD* request. If the file is smaller than 64 KB, we will download the entire file and load it into one 64 KB page. Otherwise, we will first access the metadata block at the end of the file to get some general information about the contained data, such as the data types, the number of row groups, and the cardinality. Then, when our morsels start processing, we will request the data as needed. Each column chunk is requested separately and stored on a buffer page. The only difference between the Parquet file pages and the standard database pages is that we drop the page instead of writing it to disk when it is evicted. Therefore, we can not reload an evicted page, but instead we have to download it from the remote source again.

For local Parquet files, we could, in theory, work with memory-mapped files, but its usage was discouraged by Crotty et al. [CLP22]. Therefore we treat local and remote Parquet files the same and load both as needed into our buffer manager.

3.3 Query Plan Optimizations

Databases heavily rely on cardinality estimates when it comes to query plan optimizations. This information is usually provided by the user or computed by the database engine. The

user can give hints to the database with primary and foreign keys. The database computes its metadata during the *INSERT* statement. We compute HyperLogLog sketches for single columns and retrieve samples from our data [FN19; NF20].

When we access our Parquet file for the first time, we do not have any prior knowledge about the contained data. Nevertheless, we do not want to waste time with an additional initialization step before starting to work on the actual query. Therefore, we will collect the information we need while executing the first query. We can use the statistics for query plan optimizations starting from the second query. Typically, users will start with exploratory queries when they access Parquet files for the first time. These queries are simple and not very complex, so we do not need metadata to get a relatively good query plan estimate. However, suppose the users still want to execute complex queries on a Parquet file immediately. In that case, they could trigger the computation of the statistics beforehand by accessing all required columns with a simple scan request. For the first execution, we use vague estimates for the query plan optimization: For each column chunk that uses dictionary encoding, we use the number of values in the dictionary as a distinct value estimate. This estimate can be wrong since the encoding of a column chunk can change back to plain encoding.

During execution of the first query, we will compute HyperLogLog sketches and store a data sample from the file for each required column, similar to the synopsis computation. For the sample, we compute 1024 random row numbers (Line 5 in Fig. 5). We allocate enough space to store the sample values of all columns. Then, when the user queries the file again and requires other columns this time, we will incrementally add sketches and samples for these columns as well. Since strings are stored using a fixed-sized, out-of-line format, and all other types required to store the Parquet types have a fixed length, we can precompute the required space we need. All morsels can fill the sample in parallel since the sample positions are stored in a sorted list. Each morsel knows its row range in the Parquet file and can independently check if it contains one or more of the sample positions. The index in the sample positions list tells the thread the offset of the row in the sample. We can compute the required position in the data sample with the sample positions' index, the index of the required column in the schema, and the type size. Since strings are stored using an out-of-line string format, we allocate additional space at the end of the data sample for string columns, as needed. The string format has to work with offsets to allow relocations of the data sample when the data sample outgrows its current memory location.

In Fig. 5, we depict how the morsels are scanned: We fill our fixed-sized buffers with as many rows as possible and then push them to the parent operator. Before we call the parent operator, we reuse the buffer representation to cheaply retrieve all the information we need for our sketches and statistics computation (Line 42). This step saves much time since we do not have to decode and decompress the values from the Parquet file separately. We mentioned that we first load the restricted columns into the buffer and check the restrictions before loading the values of the qualifying rows for the remaining columns. To compute correct statistics, we need all rows, and we will therefore load all values when statistics have to be computed, even though they do not meet the restrictions.

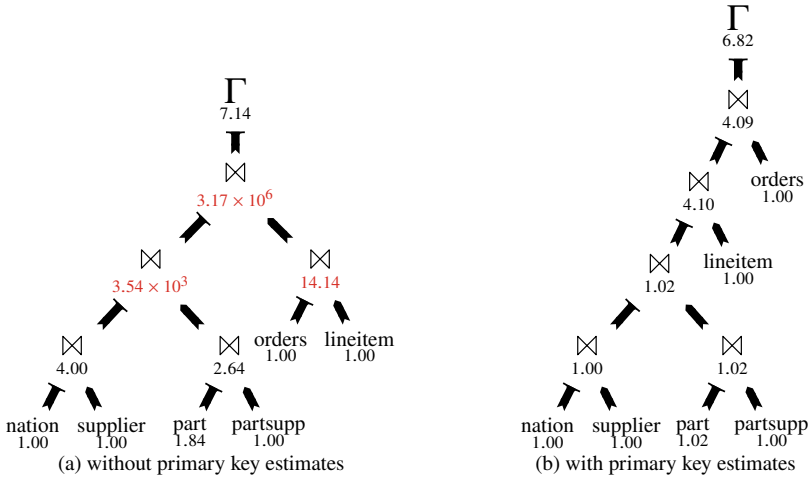


Fig. 6: Q-errors of TPC-H query 9 with scale factor 10 on Parquet files.

The data sample and the HyperLogLog sketches already help to generate optimal query plans. Nevertheless, we still have a significant performance loss for queries that heavily rely on primary key information, which we do not have in Parquet files. For example, query 9 of the TPC-H benchmark contains multiple primary/foreign key joins. The impact of the primary key information can be seen in Fig. 6. The query plan without primary key information is displayed in Fig. 6(a). Fig. 6(b) displays the usage of primary key estimates. Without primary key estimates, the q-errors are significantly higher, which leads to a suboptimal query plan.

To enhance query plans such as the one depicted in Fig. 6, we try to retrieve the primary key from the given Parquet file. A lot of work has already been done on exact methods for the detection of unique column combinations (UCCs) and functional dependencies (FDs) [AGN15]. However, discovering all minimal UCCs is an NP-hard problem. Checking if a column combination is unique, is very costly, therefore we only work with estimates here. In the literature, some approaches already exist that define different heuristics either to select a primary key from a given set of primary key candidates [JN20; PN17] or to estimate them from scratch [MK17].

Since we know the query already, we can also benefit from the query plan properties, as already discussed by Andersson [An94]. We need the primary keys in our example to know whether a join is a primary key/foreign key join. Therefore, we only consider those columns used in an equality condition of a join operator. We might miss the correct primary key. However, if the primary key is not part of the join condition, knowing the primary key would be useless since it does not influence the join order. For our approach, we only look at possible primary keys that consist of one or two columns to keep the complexity of our

estimates low. In addition, according to Papenbrock, Naumann, and Jiang [JN20; PN17], it is more likely to have short keys because it is easier to work with these.

In their work [MK17], Motl and Kordík present different features that they consider for their primary key predictions. According to their evaluation, the ordinal position is the most relevant factor. Therefore, we check the columns in positional order. The first column used as an equality condition by a join operator that has a distinct values estimate close to the cardinality of the relation will be our estimated primary key. If we cannot find a single-column candidate, we repeat our search from the beginning with column pairs. Both columns have to be a part of an equality comparison in a join, and their combined selectivity should select almost only one tuple to be considered the primary key. We can compute very accurate distinct value estimates for single columns with the HyperLogLog sketches we computed during the first query execution. Our combined selectivity estimates are based on our retrieved samples.

4 Evaluation

In this section, we measure the performance of our Parquet scan framework, which we implemented in Umbra. We start with the statistics and synopses computation. We evaluate the computational overhead they introduce and their benefit by measuring the speedup the statistics introduce and the data transfer savings of the synopses. The second step is a look at the scalability of our system. Afterward, we compare our system with three other systems, namely DuckDB [RM19], Hyper [KN11], and Trino [Se19; TSF20]. We end our evaluation by comparing the performance of Parquet files and database relations. We ran all our experiments on an Intel Xeon Gold 6338 CPU with 32 physical cores and 64 logical cores running at 2.0 GHz. The server has 256 GiB of main memory, and all Parquet files were placed on a local Samsung 850 Pro SSD with 2 TB of storage space. For our benchmarks, we used the decision support benchmark TPC-H with different scale factors and the join order benchmark (JOB) [Le15]. All measurements were repeated ten times, and we always selected the fastest execution to measure performance with warm caches.

4.1 Impact of Statistics and Synopses

When a user accesses a Parquet file for the first time, we will compute statistics and synopses. This section evaluates the overhead and the benefit of the statistics and synopses for future queries using the TPC-H benchmark with scale factor 10. In Fig. 7, we display the execution time of the first and second execution for all 22 TPC-H queries. By *first execution*, we mean a pass where we compute statistics while running the query. For the *second execution*, we have statistics available that we can use to optimize our query plan. The *first execution* and the *second execution* are repeated 10 times. For the *first execution*, we distinguish between the statistics computation and the query execution time. On average, the statistics

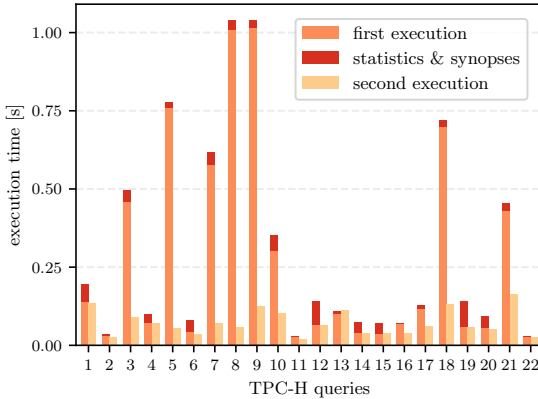


Fig. 7: Overhead and benefit of computing statistics on the performance of Parquet scans in Umbra.

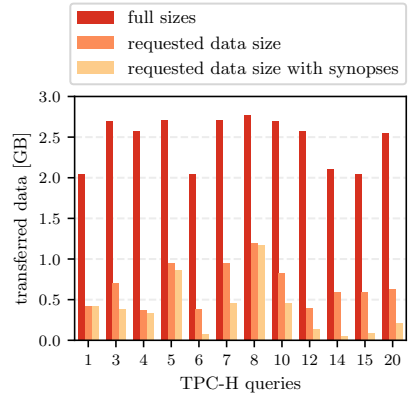


Fig. 8: Impact of synopses on the amount of transferred data for the TPC-H queries.

and synopses computation takes up 20% of the overall execution time of the first pass. In addition, the statistics and synopses for the *lineitem* Parquet file of the TPC-H dataset only require 260 KB of space, independently of the scale factor when we limit the number of synopses to 100. Even though there are queries that do not benefit from the statistics computation, since the naively chosen query plan is already optimal, they speedup the queries by 3.6 on average.

To demonstrate the benefit of the synopses, we assume the TPC-H datasets are sorted by the timestamp columns. In real-life datasets, we will most likely have some bias [Mo98; Vo18]. For example, the data might have been collected over time, which results in sorted timestamps. We used the compressed Parquet dataset with a scale factor of 10 for the measurements and stored the data on a remote server. In Fig. 8, we visualize the amount of data we have to transfer from the remote server to evaluate the TPC-H queries that are restricted by timestamps. We compare the total size of all required files with the amount of data we request from the server, once without synopses and once with synopses available for each row group. Even without the synopses, the amount of data we request is significantly smaller than the actual file sizes since we only request the required column chunks. On average, we request a fourth of the total file size. The benefit of the synopses is dependent on the restrictiveness of the queries. For queries 1 and 8, we cannot exclude any row groups, but for query 14, only one of the sixteen row groups meets the restrictions.

4.2 Scalability

Since we work with the buffer manager of Umbra to process local and remote Parquet files, we can also process files bigger than the main memory. We measured the execution times of all TPC-H queries for scale factors 1, 10, 100, and 1000 and displayed them in Fig. 9 with a

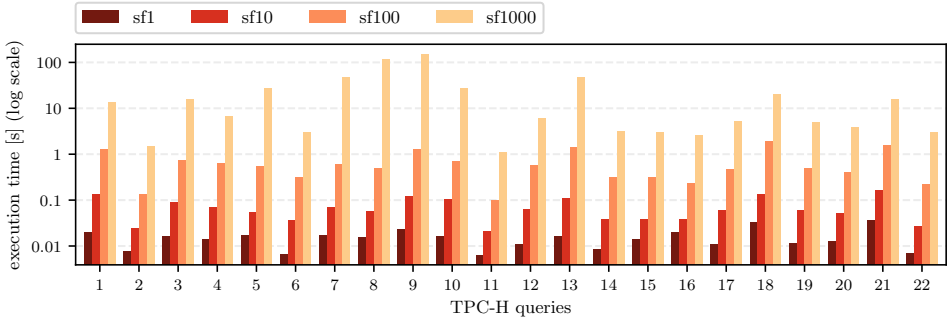


Fig. 9: Execution time of TPC-H queries with different scale factors.

logarithmic scale with base 10. Since the server in our experiments has 256 GiB of main memory, the *lineitem* Parquet files for scale factor 1000 do not fit into the buffer manager of Umbra. All queries are repeated ten times, meaning the data will be preloaded in the buffer manager if possible. The results show that our approach is stable and scales well on growing workloads. There are some queries where scale factor 1000 is more than 10 times slower than scale factor 100, because large chunks of the Parquet files have to be accessed for these queries, which do not fit into our buffer manager at once, so we have to evict pages. We also scanned all columns of the Parquet file *lineitem* and explicitly cleared the cache beforehand for all four scale factors. In this case, we observed perfect scalability.

4.3 System Comparison

We will now compare our implementation with the Parquet views of other systems. Apart from the two RDBMSs, DuckDB and Hyper, we also included a distributed query engine, namely Trino. For these measurements, we used the Trino CLI (v402) and the respective Python APIs for DuckDB (v0.4.0) and Hyper (v0.0.15530). To compare the systems, we measure the TPC-H benchmark with a scale factor of 10 and the JOB benchmark for this evaluation. We use the Parquet files generated by Spark, compressed and uncompressed, to evaluate the additional costs of the decompression. Fig. 10 shows the relative speedup of Umbra compared to the other three systems. We grouped the sets by the benchmarks into two separate graphs. We used a logarithmic scale for the speedup factor. We can see that our system outperforms all other systems for all queries. Moreover, our implementation is rarely less than twice as fast as the compared systems. We only display the speedups over the files generated by Spark since the way it distributes data is between the extremes of DuckDB and Arrow, but our system outperforms the other systems for all three writers. The geometric mean speedup for Umbra compared to DuckDB is $7.5\times$ on the TPC-H benchmark and $13\times$ for the JOB benchmark for the compressed versions. Compared to Hyper, Umbra is, on average, for both benchmarks $13\times$ faster. Trino is the slowest of all four systems. For the TPC-H benchmark, we measure a geometric mean slowdown of $21\times$ for the compressed

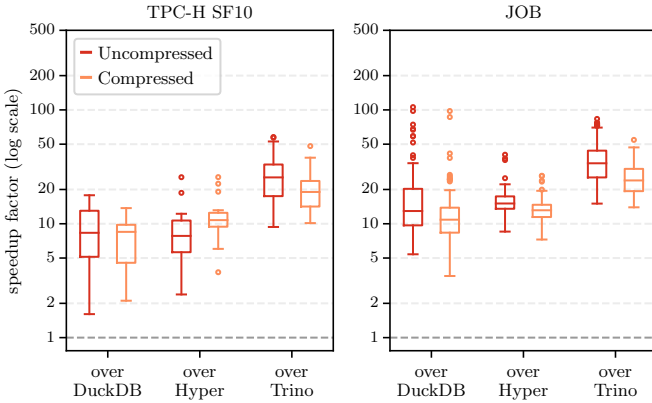


Fig. 10: Speedup of the Parquet scan of Umbra over the Parquet scans of DuckDB, Hyper, and Trino.

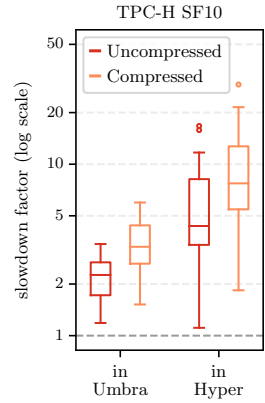


Fig. 11: Slowdown of Parquet files over database relations.

version. For the JOB benchmark, the compressed version is 26× faster in Umbra on average. The speedup for the uncompressed versions is mostly higher than for the compressed versions. Our system is significantly faster than the other systems because of our advanced statistics and sample computations. In addition, our approach allows parallelization beneath the row group level, which makes the approach more stable.

To factor out the impact of the query plan optimization, we reran the TPC-H benchmark for Hyper and Umbra and forced both systems to use the same query plans. To exclude the efficiency of other operators, we will execute the queries once on Parquet files and once on database relations in both database systems. We then compare the slowdown factors of the Parquet scans over their respective database relation versions. Fig. 11 visualizes the slowdown of Parquet files for the TPC-H benchmark for scale factors 1 and 10. We grouped them by the respective benchmark and used a logarithmic scale. We display the uncompressed version on the left and the compressed version on the right for each benchmark. On average, the uncompressed Parquet files have a slow down factor of 2.2 in Umbra. The compressed versions are, on average, 3.4× slower. Hyper is 10.2× slower than the corresponding database relation version for the compressed version. If we compare the execution times of the Hyper and Umbra Parquet scans with one another, we still achieve a speedup factor of 3.5× over Hyper. Compared to our black box evaluation, where we were 12× faster than Hyper, we can see that the statistics computation also added an average speedup of 3.5× to the overall query execution time.

4.4 Database Relation Comparison

The main goal of adding the Parquet view functionality to our database system is to offer a reasonable alternative to database relations. The main advantage of database relations is

that they have additional time to get to know the data during the initialization phase. For a fair comparison, we, therefore, have to consider the time it takes to execute the *CREATE TABLE* and *INSERT* statements. We measure how often a query has to be executed to make it reasonable to preload the data into the database instead of working on the Parquet files:

$$db_init + x \cdot db_exec > pq_first + (x - 1) \cdot pq_exec \quad (1)$$

We use the TPC-H benchmark with scale factor 10 and the uncompressed Parquet files for this evaluation. To get the number of required repetitions, we have to solve Inequality 1. As long as the left side of the inequation takes longer than the right side, preloading the data is not beneficial. The left side consists of the execution time of the given query (*db_exec*) multiplied by the number of repetitions (*x*) plus the time it takes to initialize all relations we require for the query (*db_init*). The right side contains the execution time of the Parquet version (*pq_exec*) times the number of repetitions (*x*) minus one. We add the first execution (*pq_first*) separately since it contains the statistics computation overhead. All queries must be executed at least 30 times to amortize the initialization cost. On average, the queries would need to be executed more than 200 times to pay off the initialization and loading steps. Our evaluation shows that our framework presents a reasonable alternative to database relations with comparable performance and without initial loading costs.

5 Related Work

Most related work towards integrating raw data into query processing focuses on CSV files. Mühlbauer et al. [Mü13] bulk load CSV files in parallel by splitting the file into chunks, and utilize vectorization methods to speed up the loading process. Similar to our framework, they allow the user to query CSV files without initial loading. Nevertheless, missing metadata and the row-wise format limit the performance of CSV scans. The NoDB system *PostgresRAW* presented by Alagiannis et al. [Al12] identified and resolved these performance bottlenecks. Their adaptive indexing strategy collects metadata about the CSV files that improve future queries, such as the positions of attributes in the CSV file referenced in previous queries. They also work with the built-in statistics routines of Postgres, which are very limited, to improve the selectivity estimates of their query plans. Similar to our approach, they will only compute statistics for columns required by the current query. Olma et al. [Ol17] partition the underlying CSV file logically depending on the user's access patterns. Per partition, they store index structures, statistics, access frequencies and the average query selectivity to tune the partitions and indices continuously. While we add new estimates over time when new columns are accessed, the LEO optimizer presented by Stillger et al. [St01] introduces a feedback loop to incrementally fix estimates that are off by comparing them to the actual results.


Durner et al. [DLN21] deal with JSON files, where the main challenge is the lack of a schema and the possibility that the schema can change over time. Li et al. [Li20] build a storage engine based on the Apache Arrow file format. Their work focuses on improving the export

costs for data science and machine learning engines. Vogel et al. [Vo20] use the Parquet file format in their storage layer. Their main goal is to spread the data column-wise across a tierless device pool, depending on its usage. They split the Parquet files onto different devices to optimize the throughput. Idreos et al. [Id11], Abouzied et al. [AAS13], and Chenk et al. [CR14] incrementally improve query execution time on RAW files by loading more and more parts of the data into the database while executing queries. SAHARA[Br22], a table partitioning advisor, collects data access statistics and chooses a partitioning layout based on these statistics. Our synopses come in handy if the SAHARA optimizer is used since the data will be sorted by columns that are frequently used in the filter predicates.

The idea of synopses was used in the literature under different names. Moerkotte et al. [Mo98] called them small materialized aggregates (SMAs). They are used to store *min*, *max*, *count*, and *sum* aggregates on page-level granularity. Lang et al. [La16] extended these to PSMA (positional SMAs) with a lightweight index structure to narrow down scan ranges further. E3 (Eagle-Eyed Elephant) is the term Eltabakh et al. [El13] used to describe a set of techniques, namely range indices, inverted indices, materialized views, and a caching algorithm to prevent reading unnecessary data. Ziauddin et al. [Zi17] store minimum and maximum values of one or more columns over contiguous blocks called zone maps. Presto, introduced by Sethi et al. [Se19], is a distributed query engine that also uses the optional min/max statistics of Parquet files to skip pages and row groups if possible. The system has custom implementations, for example, for joins that work with the dictionary-encoded data to build hash tables, which is an interesting opportunity for future work. Armbrust et al. and Brehm et al. describe in their work [Ar20; Be22] how they process Parquet files. They can skip unneeded data more aggressively due to the clustering of Delta Lake. In addition, instead of collecting statistics to make future executions faster, Photon, their vectorized query engine for lakehouse environments, supports batch-level adaptivity and can switch execution kernels based on the metadata collected from previous batches.

6 Conclusion

Parquet files are great candidates for integration into the query processing of a database engine. In this paper, we introduced a framework that solves the challenges of this file format. We presented different techniques for making the performance of analytical workloads on Parquet files more convenient. We demonstrated that we achieve comparable execution times to database relations. Computing statistics while accessing the columns of a Parquet file for the first time introduces a small overhead, but the query plan optimizer benefits a lot from these. Furthermore, the synopses computation introduces another significant performance boost for Parquet files stored remotely. In addition, our approach can compete with existing Parquet scans and outperforms them in all presented scenarios.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286). 

References

- [AAS13] Abouzied, A.; Abadi, D. J.; Silberschatz, A.: Invisible loading: access-driven data transfer from raw files into database systems. In: EDBT. ACM, pp. 1–10, 2013.
- [AGN15] Abedjan, Z.; Golab, L.; Naumann, F.: Profiling relational data: a survey. VLDB J. 24/4, pp. 557–581, 2015.
- [Al12] Alagiannis, I.; Borovica, R.; Branco, M.; Idreos, S.; Ailamaki, A.: NoDB: efficient query execution on raw data files. In: SIGMOD Conference. ACM, pp. 241–252, 2012.
- [An94] Andersson, M.: Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. In: ER. Vol. 881. Lecture Notes in Computer Science, Springer, pp. 403–419, 1994.
- [Ar20] Armbrust, M.; Das, T.; Paranjpye, S.; Xin, R.; Zhu, S.; Ghodsi, A.; Yavuz, B.; Murthy, M.; Torres, J.; Sun, L.; Boncz, P. A.; Mokhtar, M.; Hovell, H. V.; Ionescu, A.; Luszczak, A.; Switakowski, M.; Ueshin, T.; Li, X.; Szafranski, M.; Senster, P.; Zaharia, M.: Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. Proc. VLDB Endow. 13/12, pp. 3411–3424, 2020.
- [ASF13] Apache Software Foundation: Apache Parquet, 2013, URL: <https://parquet.apache.org>, visited on: 12/15/2022.
- [ASF14] Apache Software Foundation: Apache Spark, 2014, URL: <https://spark.apache.org/>, visited on: 12/15/2022.
- [ASF16] Apache Software Foundation: Apache Arrow, 2016, URL: <https://arrow.apache.org/>, visited on: 12/15/2022.
- [Be22] Behm, A.; Palkar, S.; Agarwal, U.; Armstrong, T.; Cashman, D.; Dave, A.; Greenstein, T.; Hovsepian, S.; Johnson, R.; Krishnan, A. S.; Leventis, P.; Luszczak, A.; Menon, P.; Mokhtar, M.; Pang, G.; Paranjpye, S.; Rahn, G.; Samwel, B.; van Bussel, T.; Hovell, H. V.; Xue, M.; Xin, R.; Zaharia, M.: Photon: A Fast Query Engine for Lakehouse Systems. In: SIGMOD Conference. ACM, pp. 2326–2339, 2022.
- [Br22] Brendle, M.; Weber, N.; Valiyev, M.; May, N.; Schulze, R.; Böhm, A.; Mörkotte, G.; Grossniklaus, M.: SAHARA: Memory Footprint Reduction of Cloud Databases with Automated Table Partitioning. In: EDBT. OpenProceedings.org, 1:13–1:26, 2022.
- [BZN05] Boncz, P. A.; Zukowski, M.; Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: CIDR. www.cidrdb.org, pp. 225–237, 2005.
- [CLP22] Crotty, A.; Leis, V.; Pavlo, A.: Are You Sure You Want to Use MMAP in Your Database Management System? In: CIDR. www.cidrdb.org, 2022.
- [CR14] Cheng, Y.; Rusu, F.: Parallel in-situ data processing with speculative loading. In: SIGMOD Conference. ACM, pp. 1287–1298, 2014.

- [De13] Dem, J. L.: Announcing Parquet 1.0: Columnar Storage for Hadoop, 2013, URL: https://blog.twitter.com/engineering/en_us/a/2013/announcing-parquet-10-columnar-storage-for-hadoop, visited on: 12/15/2022.
- [DLN21] Durner, D.; Leis, V.; Neumann, T.: JSON Tiles: Fast Analytics on Semi-Structured Data. In: SIGMOD Conference. ACM, pp. 445–458, 2021.
- [El13] Eltabakh, M. Y.; Özcan, F.; Sismanis, Y.; Haas, P. J.; Pirahesh, H.; Vondrák, J.: Eagle-eyed elephant: split-oriented indexing in Hadoop. In: EDBT. ACM, pp. 89–100, 2013.
- [Fl07] Flajolet, P.; Fusy, É.; Gandouet, O.; Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In: Discrete Mathematics and Theoretical Computer Science. Discrete Mathematics and Theoretical Computer Science, pp. 137–156, 2007.
- [FN19] Freitag, M.; Neumann, T.: Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In: CIDR. www.cidrdb.org, 2019.
- [GG11] Google: Snappy, a fast compressor/decompressor, 2011, URL: <https://github.com/google/snappy>, visited on: 12/15/2022.
- [Id11] Idreos, S.; Alagiannis, I.; Johnson, R.; Ailamaki, A.: Here are my Data Files. Here are my Queries. Where are my Results? In: CIDR. www.cidrdb.org, pp. 57–68, 2011.
- [JN20] Jiang, L.; Naumann, F.: Holistic primary key and foreign key detection. *J. Intell. Inf. Syst.* 54/3, pp. 439–461, 2020.
- [Ka14] Karpathiotakis, M.; Branco, M.; Alagiannis, I.; Ailamaki, A.: Adaptive Query Processing on RAW Data. *Proc. VLDB Endow.* 7/12, pp. 1119–1130, 2014.
- [KN11] Kemper, A.; Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE. IEEE Computer Society, pp. 195–206, 2011.
- [La16] Lang, H.; Mühlbauer, T.; Funke, F.; Boncz, P. A.; and Alfons Kemper, T. N.: Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In: SIGMOD Conference. ACM, pp. 311–326, 2016.
- [Le14] Leis, V.; Boncz, P. A.; Kemper, A.; Neumann, T.: Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD Conference. ACM, pp. 743–754, 2014.
- [Le15] Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P. A.; Kemper, A.; Neumann, T.: How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9/3, pp. 204–215, 2015.
- [Le18] Leis, V.; Haubenschild, M.; Kemper, A.; Neumann, T.: LeanStore: In-Memory Data Management beyond Main Memory. In: ICDE. IEEE Computer Society, pp. 185–196, 2018.

- [Li20] Li, T.; Butrovich, M.; Ngom, A.; Lim, W. S.; McKinney, W.; Pavlo, A.: Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14/4, pp. 534–546, 2020.
- [Me10] Melnik, S.; Gubarev, A.; Long, J. J.; Romer, G.; Shivakumar, S.; Tolton, M.; Vassilakis, T.: Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3/1, pp. 330–339, 2010.
- [MK17] Motl, J.; Kordík, P.: Foreign Key Constraint Identification in Relational Databases. In: *ITAT*. Vol. 1885. CEUR Workshop Proceedings, CEUR-WS.org, pp. 106–111, 2017.
- [MNS09] Moerkotte, G.; Neumann, T.; Steidl, G.: Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2/1, pp. 982–993, 2009.
- [Mo98] Moerkotte, G.: Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In: *VLDB*. Morgan Kaufmann, pp. 476–487, 1998.
- [Mü13] Mühlbauer, T.; Rödiger, W.; Seilbeck, R.; Reiser, A.; Kemper, A.; Neumann, T.: Instant Loading for Main Memory Databases. *Proc. VLDB Endow.* 6/14, pp. 1702–1713, 2013.
- [Ne11] Neumann, T.: Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4/9, pp. 539–550, 2011.
- [NF20] Neumann, T.; Freitag, M.: Umbra: A Disk-Based System with In-Memory Performance. In: *CIDR*. www.cidrdb.org, 2020.
- [OI17] Olma, M.; Karpathiotakis, M.; Alagiannis, I.; Athanassoulis, M.; Ailamaki, A.: Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing. *Proc. VLDB Endow.* 10/10, pp. 1106–1117, 2017.
- [PN17] Papenbrock, T.; Naumann, F.: Data-driven Schema Normalization. In: *EDBT*. OpenProceedings.org, pp. 342–353, 2017.
- [RM19] Raasveldt, M.; Mühleisen, H.: DuckDB: an Embeddable Analytical Database. In: *SIGMOD Conference*. ACM, pp. 1981–1984, 2019.
- [SE19] Skyscanner Engineering: Building a self-served ETL pipeline for third-party data ingestion, 2019, URL: <https://medium.com/@SkyscannerEng/building-a-self-served-etl-pipeline-for-third-party-data-ingestion-3959eab6840b>, visited on: 12/15/2022.
- [Se19] Sethi, R.; Traverso, M.; Sundstrom, D.; Phillips, D.; Xie, W.; Sun, Y.; Yegitbasi, N.; Jin, H.; Hwang, E.; Shingte, N.; Berner, C.: Presto: SQL on Everything. In: *ICDE*. IEEE, pp. 1802–1813, 2019.
- [Sh99] Shanmugasundaram, J.; Tufte, K.; Zhang, C.; He, G.; DeWitt, D. J.; Naughton, J. F.: Relational Databases for Querying XML Documents: Limitations and Opportunities. In: *VLDB*. Morgan Kaufmann, pp. 302–314, 1999.

- [St01] Stillger, M.; Lohman, G. M.; Markl, V.; Kandil, M.: LEO - DB2's LEarning Optimizer. In: VLDB. Morgan Kaufmann, pp. 19–28, 2001.
- [TSF20] Trino Software Foundation: Trino, 2020, URL: <https://trino.io/>, visited on: 12/15/2022.
- [Vo18] Vogelsgesang, A.; Haubenschild, M.; Finis, J.; Kemper, A.; Leis, V.; Mühlbauer, T.; Neumann, T.; Then, M.: Get Real: How Benchmarks Fail to Represent the Real World. In: DBTest@SIGMOD. ACM, 1:1–1:6, 2018.
- [Vo20] Vogel, L.; van Renen, A.; Imamura, S.; Leis, V.; Neumann, T.; Kemper, A.: Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. Proc. VLDB Endow. 13/11, pp. 2662–2675, 2020.
- [WG17] Weeks, D.; Gianos, T.: Petabytes Scale Analytics Infrastructure @Netflix, 2017, URL: <https://www.infoq.com/presentations/netflix-big-data-infrastructure/>, visited on: 12/15/2022.
- [Zi17] Ziauddin, M.; Witkowski, A.; Kim, Y. J.; Lahorani, J.; Potapov, D.; Krishna, M.: Dimensions Based Data Clustering and Zone Maps. Proc. VLDB Endow. 10/12, pp. 1622–1633, 2017.