Bridging the Gap between Data Lakes and RDBMSs

Efficient Query Processing with Parquet

Alice Rey

Supervised by Prof. Dr. Thomas Neumann
Technical University of Munich, Boltzmannstraße 3, 85748 Garching, Germany, rey@in.tum.de

Abstract
In the age of massive data, databases are getting less convenient for data exploration tasks due to the costly loading phase. Still, the highly optimized query engines of database systems are greatly beneficial for the performance of data analysis tasks. With our research, we want to bridge this gap and provide paramount analytical performance without the need of static data loading. Our approach enables the integration of Parquet files — one of the most used columnar file format in the data lake context — into the data processing pipeline of a database system in a convenient way. We allow end-users to benefit from the database system performance without a costly and time-consuming loading phase.

Keywords
Parquet, Data Lakes, Dremel, Database Systems

1. Introduction

Data is ever-growing. By now, cloud-based systems and data lakes are a popular choice for storing and processing data. Loading massive amounts of data into traditional database systems can be considered time-consuming and unnecessary overhead. Still, big data workloads can benefit from the performance that well-engineered database systems can provide. One of the most common file formats to store vast amounts of data is Parquet [1]. This binary file format stores data in a columnarized way, which makes it similar to processing data stored in a relational database system with a columnar storage layout.

In our first work [2], we presented our approach for integrating Parquet files into the data processing pipeline of relational database systems without the costly loading phase. Our framework allows queries to be executed directly on Parquet files with stable parallelization. In addition, we store statistics about the data to speed up future queries. This work showed that the presented techniques lead to promising results that outperform other data engineering tools and database systems. With our approach, the observed performance is close to cases where the relational database system directly manages and stores the data.

Since data is not always stored in relational database systems that encourage the end user to keep the data in third normal form, other techniques are used to represent relations between datasets. One commonly used technique is to model one-to-one and one-to-many relationships with nesting. Many-to-many relations can only be projected to a nested data structure by introducing a lot of duplicated data.

Inside Parquet files, the Dremel [3] encoding facilitates nesting. The benefit of this encoding is that even nested data can be stored in a columnar format, which enables optimized storage, better compression, and more efficient access. If the users need to reconstruct the data, they must decode the repetition levels stored per column, which encode the nesting.

Until now, we have focused on supporting non-nested data. Scanning Parquet files is already a very complex task, even without the addition of nesting. Adding nesting to the scanner makes dealing with the complexity of the Parquet format even more complicated and an immense task. In addition, we want to benefit from the strengths of the database system which is processing non-nested data in third normal form. Therefore, we opted for a different direction: Our plan is to work with flattened data inside the Parquet scanner and thereby keep the complexity and implementation effort as minimal as possible. The nesting is added in a later stage.

The rest of this paper is structured as follows: First, we will cover related work about file-based and nested data processing in Section 2. Afterward, in Section 3, we will summarize our current work. We start with our first paper, which dealt with the seamless integration of Parquet files into database engines. Then, we will introduce our plan for Dremel-encoded nested data on a high level. Then, in Section 4, we will discuss our plans for future work and areas where we still see potential for improvements. Lastly, Section 5 concludes the paper.

Published in the Proceedings of the Workshops of the EDBT/ICDT 2024 Joint Conference (March 25-28, 2024), Paestum, Italy

2. Related Work

We identified two main areas of related work that are relevant for our work: (i) Processing raw data in RDBMSs and (ii) processing nested data. In the following, we categorize the collected related work by these two areas.

2.1. Processing of Raw Data

There exists a multitude of work on processing CSV files. For example, Mühlbauer et al. [4] present techniques to directly query the row-wise non-binary format in an efficient way by reading the file in chunks with vectorization methods. Furthermore, Alagiannis et al. [5] present approaches to compensate for the missing metadata inside CSV files by utilizing the statistics routines of Postgres to optimize their selectivity estimates. Similarly to their CSV scanner, we collect statistics and metadata tailored to the Parquet file format.

Apache Arrow, an in-memory columnar format, was used as a foundation for the storage layer of a DBMS in the work of Li et al. [6]. They focused on good OLTP performance by utilizing a relaxed Apache Arrow format while supporting a fast export to data science engines. Liu et al. [7] provide an overview of how columnar formats can be utilized as a base for DBMSs. They compare three major formats, Parquet, Arrow, and ORC, under different aspects like compression performance, transcoding throughput, and their performance for different database operations like projection or filtering. Zeng et al. [8] compare the layout of Parquet and ORC under different aspects. They do not find a clear winner but focus more on how future formats can learn from their findings. Both papers, highlight Parquet as one of the standards for columnar data formats.

2.2. Processing of Nested Data

Durner et al. [9] present with their work on JSON tiles an approach to efficiently process JSON files. They also face the challenge of nested data and tackle it by extracting common schema parts and materializing them into a columnar format.

Wang et al. [10] present Steed, a native database system designed for tree-structured data. They built a row and a columnar data layout to store JSON data. For the column data, they utilize the Dremel schema. They suggest optimizing for simple accesses with at most one array along the access path, so fewer nodes must be visited.

Trance, presented by Smith et al. [11], is a framework that transfers queries that run on nested collections to a set of relational queries that can run on flattened collections. They introduce a “fully-flattened” data representation, whereas the Dremel encoding stores the data “semi-flattened”.

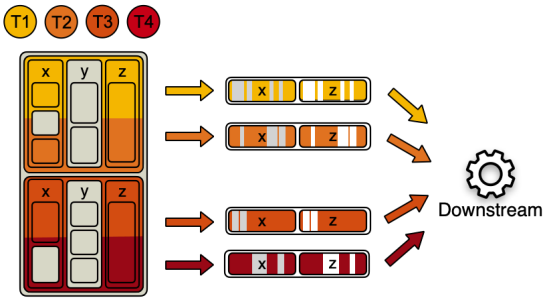

Figure 1: Parallelized Processing of Parquet files with early predicate evaluation.

3. Current Work

In this section, we start by discussing the work we have already completed. Afterwards, we explain our current work. As a first step, we focused on efficiently scanning Parquet files without any nested types. In the second step, which we are working on, we look into adding support for nested types. This support will be added orthogonally so that we are not forced to rewrite our scanner entirely and keep the complexity of nested types outside of the scanner implementation.

3.1. Efficient Processing of Parquet Files

The main challenge of building an efficient Parquet Scanner is to handle the broad variety of potential input file structures. Parquet files are first split horizontally into row groups with an arbitrary number of rows. Each row group is then stored column-wise in so-called column chunks. The data is split per column chunk into one or more pages with an arbitrary number of values per page using one of the many different supported encodings and compressions. Such files can be created by different Parquet writers with different settings, leading to arbitrarily different files. In our example in Figure 1, the Parquet file contains three columns (x, y, and z) and is split into two row groups. The column chunks inside the row groups are split into one to three pages.

Robust parallelization. We aim for optimal performance independent of how the utilized Parquet writer distributed the data over the Parquet levels (Row Groups and Pages) and for which granularity statistics were collected and stored in the Parquet file. Some Parquet file writers that write significantly more rows into one row group than what we consider a good number of rows that should be processed as one batch [12, 13].Hence, we deal with this situation by parallelizing below row group level which we visualize in Figure 1 with the threads `T1` - `T4`, where each row group is processed by two threads in parallel. While scanning a single batch, we scan as little data as possible. First, we only scan the columns that are

required for producing the query result, columns x and z in our example. This optimization can be easily realized since Parquet stores the data in a columnar format. Hence, columns can be accessed independently, which fits quite nicely into a columnar engine like Umbra [13].

Multi-level pruning. Suppose we have to evaluate selection predicates on specific columns; we can use these predicates in two phases to minimize the data we have to access. First, we use them to exclude files, row groups, and pages as early as possible. In our example in Figure 1, we have a selection predicate for column x with whom we can exclude in each row group one entire page.

Since min/max statistics are specified as optional fields in the Parquet format, we can not rely on them. Therefore, we added fallback min/max statistics, which we call *synopses*. To keep the number of synopses fixed, we store them on row group level and group multiple row groups together if the Parquet file contains many row groups.

The second phase, where we use the predicates to minimize the scanned data, is during the actual data loading. We start by loading the columns that are restricted by the predicates and evaluate them with vectorized functions. In Figure1, we do that for column x. Then, we only access the rest of the columns, in our example column y, if the predicate holds for the specific row.

On-the-fly statistics computation. Even with those parallelization and pruning techniques, we observed significant performance differences for more complex queries that are ran on Parquet files instead of their native database relation equivalents. Due to some Parquet files lacking statistics entirely and only basic min/max statistics being available in the rest of the Parquet files, we decided to compute our own set of statistics for each column inside a Parquet file whenever they are accessed for the first time. We compute HyperLogLog sketches [14] and keep a random data sample. We keep the overhead of the statistics computation low by not accessing the data unnecessarily. We only write such statistics when scanning the data for query evaluation. Starting with the second time the file is accessed, our query optimizer can work with these statistics to generate query plans that are as optimized as for standard database tables.

Based on the statistics and information from the query plan, we also try to estimate the primary keys of each file which help us decide if we have a primary key/foreign key join. Based on the usage of the columns and distinct value estimates, we check if single columns and pairs of columns could be primary key candidates.

Our experiments show that combining all these techniques enables us to process Parquet files very efficiently and achieve similar performance compared to data stored in traditional database relations. In addition, we can outperform other existing systems that support querying Parquet files directly.

3.2. Dedremelize Parquet Files

We consider the handling of nested data an orthogonal problem to an efficient Parquet scanner. Therefore, we investigate that topic based on the assumption that the underlying database system already supports scanning Parquet files with basic types.

Relational database systems are optimized for non-nested data, specifically in third normal form. The Dremel encoding allows Parquet files to store nested data in a columnar way. Basic fields are stored in their own columns, and the so-called repetition levels tell us later how the nested structure can be rebuilt using an automaton presented in the Dremel encoding paper [3]. Interestingly, elements from the same nesting level with the same parent have the same repetition level [15]. Based on this finding, we can group columns by this condition (grouped column sets) and end up with a set of tables in the third normal form.

If we postpone the reconstruction of the nesting to a later stage, we can split the scan of nested data into multiple scans that can be performed individually on the grouped column sets from above. Based on the repetition levels, we can generate what we call *surrogate keys* that allow us to later join the different levels back together. We will publish more details on that in the future, addressing all the requirements and showing performance results.

Benchmarks for nested data. Finding appropriate benchmarks is very challenging since nested data can be very versatile. The number of nesting levels and the children-to-parent ratio can vary significantly. We identified two resources for benchmarks: Firstly, there exists work on benchmarking big data systems using a modified version of the TPC-H benchmark that nests `lineitems` into the corresponding `orders` [16] and even the `orders` into the corresponding `customers` [11]. The downside of nesting the TPC-H dataset is that one of the benchmark's key challenges, the join performance, is at an advantage since the data is already grouped by the primary key/foreign key join predicates.

Second, there exists work on benchmarking other nested file formats like JSON and XML [9, 17, 18]. The downside of these benchmarks is that they were designed to highlight the performance bottlenecks that those file formats introduce, like the unknown schema and the reading from a non-binary file format. In addition, they were not designed to be used for big data workloads but for data sizes that are still manageable in a human-readable format. Scaling those benchmarks up to achieve big data scale is not always possible. DeepBench [19] is an extensible and scalable benchmark for JSON data focusing on different nesting levels and array types which could be extended to Parquet since it is targeting similar bottlenecks. We will evaluate these benchmarks and mi-

crobenchmarks in the future to get a better understanding of their capabilities and limitations.

4. Future Work

Our current work already discusses how Parquet scanning capabilities can be integrated into database engines. In the real world, extensions for Parquet files and other columnar storage formats make file-based storage formats even more powerful. Well-known extensions are Iceberg [20] and Deltalake [21]. They allow schema evolution, support updates as deltas or full rewrites, and even support time travel to earlier versions. In the future, we plan to investigate how the aforementioned extensions align with the capabilities of a database engine and how these can be integrated seamlessly into existing systems.

5. Conclusion

We presented our investigations on how the gap between highly efficient relational database systems and data lake file formats can be bridged. We started with a Parquet scanner that shows how these files can be seamlessly integrated into the stack of a relational database engine. Our current work focuses on supporting nested data, a common pattern in big data workloads. The goal is to keep the implementation effort and complexity as low as possible. In addition, we want to abstract as much logic away from the core database engine as possible. The first measurements show promising results, proving that our vision of a clean, easy-to-implement addition does not contradict competitive performance.

References

[1] Apache Software Foundation, Apache parquet, 2013. URL: https://parquet.apache.org.

[2] A. Rey, M. Freitag, T. Neumann, Seamless integration of parquet files into data processing, in: BTW, volume P-331 of *LNI*, 2023, pp. 235–258.

[3] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, et al., Dremel: Interactive analysis of web-scale datasets, VLDB 3 (2010) 330–339.

[4] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, T. Neumann, Instant loading for main memory databases, VLDB 6 (2013) 1702–1713.

[5] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, A. Ailamaki, Nodb: efficient query execution on raw data files, in: SIGMOD Conference, 2012, pp. 241–252.

[6] T. Li, M. Butrovich, A. Ngom, W. S. Lim, W. McKinney, A. Pavlo, Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats, VLDB 14 (2020) 534–546.

[7] C. Liu, A. Pavlenko, M. Interlandi, B. Haynes, A deep dive into common open formats for analytical dbmss, VLDB 16 (2023) 3044–3056.

[8] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, H. Zhang, An empirical evaluation of columnar storage formats, VLDB 17 (2023) 148–161.

[9] D. Durner, V. Leis, T. Neumann, JSON tiles: Fast analytics on semi-structured data, in: SIGMOD Conference, 2021, pp. 445–458.

[10] Z. Wang, S. Chen, Exploiting common patterns for tree-structured data, in: SIGMOD Conference, 2017, pp. 883–896.

[11] J. Smith, M. Benedikt, M. Nikolic, A. Shaikhha, Scalable querying of nested data, VLDB 14 (2020) 445–457.

[12] V. Leis, P. A. Boncz, A. Kemper, T. Neumann, Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age, in: SIGMOD Conference, 2014, pp. 743–754.

[13] T. Neumann, M. J. Freitag, Umbra: A disk-based system with in-memory performance, in: CIDR, 2020.

[14] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, et al., Dremel: A decade of interactive SQL analysis at web scale, VLDB 13 (2020) 3461–3472.

[16] P. Pirzadeh, M. J. Carey, T. Westmann, A performance study of big data analytics platforms, in: IEEE BigData, 2017, pp. 2911–2920.

[17] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse, Xmark: A benchmark for XML data management, in: VLDB, 2002, pp. 974–985.

[18] C. Truica, E. S. Apostol, J. Darmont, T. B. Pedersen, The forgotten document-oriented database management systems: An overview and benchmark of native XML dodbmses in comparison with JSON dodbmses, Big Data Res. 25 (2021) 100205.

[19] S. Belloni, D. Ritter, M. Schröder, N. Rörup, Deepbench: Benchmarking JSON document stores, in: DBTest@SIGMOD, 2022, pp. 1–9.

[20] Apache Software Foundation, Apache iceberg, 2013. URL: https://iceberg.apache.org.

[21] M. Armbrust, T. Das, S. Paranjpye, R. Xin, S. Zhu, et al., Delta lake: High-performance ACID table storage over cloud object stores, VLDB 13 (2020) 3411–3424.