Nested Parquet Is Flat, Why Not Use It? How To Scan Nested Data With On-the-Fly Key Generation and Joins

Alice Rey
Technische Universität München
Munich, Germany
rey@in.tum.de

Maximilian Rieger
Technische Universität München
Munich, Germany
max.rieger@tum.de

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

Abstract

Parquet is the most commonly used file format to store data in a columnar, binary structure. The format also supports storing nested data in this flattened columnar layout. However, many query engines either do not support nested data or process it with substantially worse performance than relational data.

In this work, we close this gap and present a new way to leverage relational query engines for nested data that is stored in this flat columnar file format. Specifically, we demonstrate how to process nested Parquet files much more efficiently.

Our approach does not store a copy of the data in an internal format but reads directly from the Parquet file. During query computation, the required flat columns are scanned independently and the nesting is reconstructed using joins with on-the-fly generated join keys. Our approach can be easily integrated into existing query engines to support querying nested Parquet files. Furthermore, we achieve orders of magnitude faster analytical query performance than existing solutions, which makes it a valuable addition.

CCS Concepts

• **Information systems** → **Database query processing**; **Hierarchical data models**; **Relational database model**.

Keywords

Parquet, Nested Data Processing, Database Systems

ACM Reference Format:
Alice Rey, Maximilian Rieger, and Thomas Neumann. 2025. Nested Parquet Is Flat, Why Not Use It? How To Scan Nested Data With On-the-Fly Key Generation and Joins. In *Proceedings of the 2025 International Conference on Management of Data (SIGMOD'25).* ACM, New York, NY, USA, Article 192, 14 pages. https://doi.org/10.1145/3725329

1 Introduction

In today's era of cloud object stores and data lakes, the volume of data is growing exponentially. Whether in web or scientific computing, data often contains nested objects or arrays. For example, logging data or structured documents use nested schemata [31, 32].

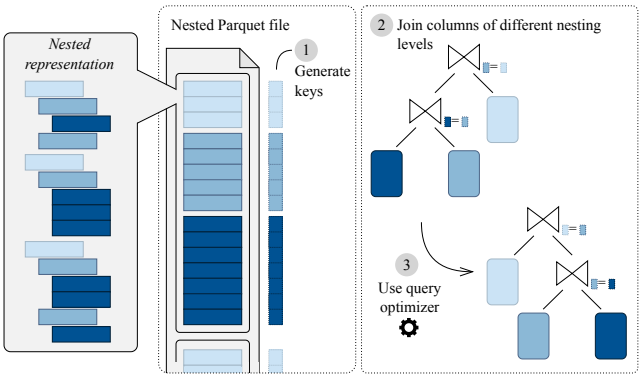


Figure 1: Idea of nested Parquet file processing: Abstract representation of a nested Parquet file and how it is efficiently processed with our approach 1 - 3 .

Examples of such formats include JSON and XML, which store nested data in a hierarchical, human-readable form.

Modern data lakes, however, predominantly use Parquet files, which store data in a columnar, binary structure [5, 14, 27]. The format leverages columnar storage for better query performance and compression techniques to improve the storage footprint and allow faster data scans. Handling semi-structured, nested data in Parquet is done using the record shredding and assembling algorithm, originally developed by Google for their distributed system Dremel [32].

Unlike human-readable file formats such as JSON and XML, Parquet does not use a nested representation internally to store nested data. Figure 1 (left) illustrates an abstract nested dataset in its nested form, as it would be structured in a JSON or XML file. Parquet, on the other hand, dissects nested records into columns as flattened chunks as depicted in the center of the figure.

Existing systems typically convert the flattened representation offered by the Dremel encoding into some internal nested formats, which complicates processing and increases complexity [9, 20]. In contrast, our approach allows reconstructing the nesting with on-the-fly generated join keys. We keep the data flattened for the entire query computation and join the required columns only as needed. Hence, the database system can scan the data without nested types and use the already existing efficient filtering and byte processing techniques of standard primitive types. When scanning, we do not persist any new data or auxiliary structures but process the original file on the fly each time. This approach obviates the need for complex nesting logic and is in addition trivially parallelizable on all nesting levels.

Authors' Contact Information: Alice Rey, Technische Universität München, Munich, Germany, rey@in.tum.de; Maximilian Rieger, Technische Universität München, Munich, Germany, max.rieger@tum.de; Thomas Neumann, Technische Universität München, Munich, Germany, neumann@in.tum.de.

SIGMOD'25, Berlin, Germany

Furthermore, our evaluation shows that this approach delivers significantly better performance compared to using the nested representations of various systems. We propose to derive surrogate keys to use joins between columns across different nesting levels to assemble tuples, achieving orders of magnitude speedup over the existing approaches. Compared to Trino [42], we achieve an average speedup of 20× using our approach, and for DuckDB [35] even 60×. Given the pivotal role of joins in database systems, their implementations are commonly heavily optimized and backed by decades of research [11, 12, 15, 16, 25]. We demonstrate that the systems we evaluated can process nested data much faster by using our join-based approach, compared to their existing nested algorithms.

Furthermore, modern query engines provide additional sophisticated techniques and optimizations to process Parquet files efficiently. Previous work leverages the capabilities of relational systems such as statistics and parallelization to speed up Parquet analytics by up to an order of magnitude [30, 36, 41, 46]. Our approach allows these techniques to be applied to nested Parquet files without intrusive extensive changes.

Figure 1 shows a high-level overview of how our approach works. First, we generate join keys for every nesting level (1). These keys act as pairs of foreign keys and primary keys that connect deeply nested columns to less deeply nested columns. We call these *surrogate keys* and *ancestor keys*. Ancestor keys reference the surrogate keys of less deep columns. Given a user query, we choose the required columns and build a join tree to join columns of different nesting levels (2). As join predicates, we use the newly generated key attributes. Thirdly, the database's query optimizer optimizes the query plan and chooses the optimal join ordering (3). If the query contains a selection predicate, the optimizer will also push down the predicate as much as possible.

Our approach reinterprets nested Parquet data as relational data. Similarly, significant work has been done on storing and querying different kinds of nested data in relational database systems [13, 19, 21, 26, 38, 44]. A relational schema has to be derived and then the data is loaded into the newly created relational schema, effectively converting the hierarchical format into a relational one. In contrast, we reinterpret the existing Parquet file on the fly without duplicating the data. Parquet already stores data in a columnar format, and we view the nested encoding as a predefined relational interpretation. In Section 7, we will further explore the parallels between our approach and the methods used for other nested file formats.

This work unlocks the good processing speed of modern query engines for nested Parquet files. In particular, we make the following contributions:

- A new, fast processing strategy for nested Parquet files that is easy to implement.
- An implementation of our approach that has been integrated into the state-of-the-art research database system Umbra [34].
- A thorough evaluation of our approach in three different systems as well as a performance comparison between the processing of nested and flat Parquet files, for which we publish all supplemental material[1].

[1]https://github.com/alicerey/nested-parquet

```
Users
 |-- UserId: int
 |-- Name: string
 |-- Followers: list(int)
 |-- Posts: list(struct)
      |-- Text: string
      |-- Reactions: list(struct)
           |-- UserId: int
           |-- Emoji: string
      |-- Comments: list(struct)
           |-- UserId: int
           |-- Text: string
           |-- Likes: list(int)
```

Figure 2: Nested dataset of a fictional social media platform.

The rest of this paper is structured as follows: Section 2 covers fundamentals such as the Parquet format and how nested data is encoded. In Section 3, we explain how we generate join keys on the fly for the different nesting levels. Next, Section 4 shows in detail how an initial join tree can be built on top of the different nesting levels to reconstruct the nesting. Section 5 concludes our method with information on how our approach can be integrated into existing systems and the operations performed during scanning. We provide a thorough performance evaluation of our approach in Section 6. Finally, we review existing work and draw conclusions in Section 7 and Section 8, respectively.

2 Background

In this section, we cover some background knowledge that we build upon in this work. First, we describe the Parquet format and current scanners on flat data. Then, we explain how nested Parquet files store their data using record shredding as introduced by Google Dremel [32]. Finally, we highlight the drawbacks of integrating existing strategies into query engines.

2.1 Parquet Format

The Parquet format was released in 2013 by Twitter and Cloudera and, since then, has become the most widely used open-source columnar file format in data lakes [5]. On a high level, Parquet files are partitioned horizontally into *row groups*. Within a row group, data is stored in a paged column store which allows fast processing. Pages can be encoded and compressed in various ways.

Existing Parquet writers make highly divergent choices for compression schemes as well as for row group and page sizes. State-of-the-art Parquet scanners, therefore, introduce elaborate techniques to guarantee robust and fast execution [29, 36]. First, they use very fine granular access for skipping filtered data as well as robust parallelization. By skipping data using existing statistics they can significantly speed up scan times. Second, they collect more statistics on the data during the first scans. This allows the query optimizer to make better choices, resulting in drastically faster processing. Using these techniques without complicated changes to the scanning logic is desirable.

2.2 Nested Parquet Encoding

Parquet files use the Dremel encoding to encode nested data. Dremel is a record shredding and assembly algorithm that was presented in

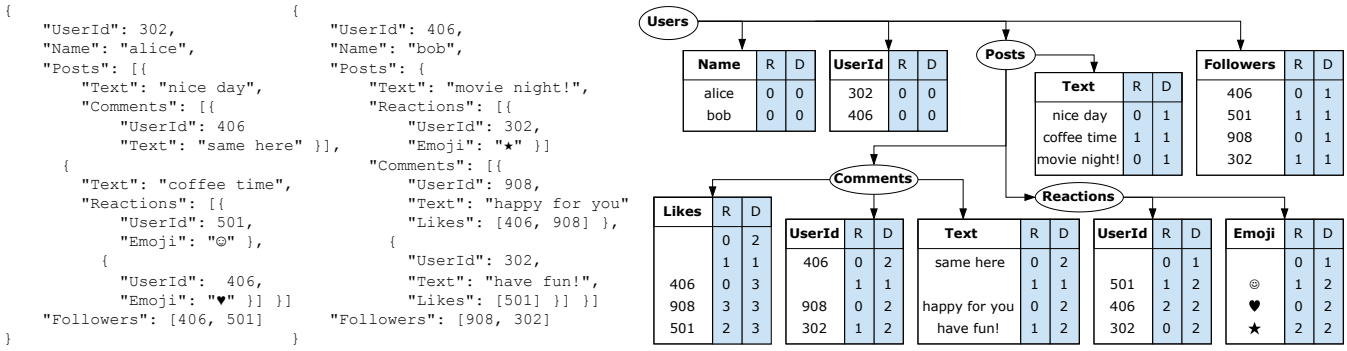


Figure 3: Example instantiation of the social media platform example from Figure 2. Represented as JSON Document on the left and on the right how a Parquet file conceptually stores it.

2010 by Melnik et al. [32]. The encoding consists of two main parts: Repetition and definition level fields for every atomic value. The repetition level tells us at which level the current entry is getting repeated and the definition level tells us at which level the current value is defined.

Figure 2 visualizes a nested document of a fictional social media platform which we will use as running example in this work. At the top level, it stores a list of users. Each user has an id and a name, which are stored as simple members. Additionally, each user has a list of followers and posts. While followers are organized as a simple list of integers representing the userids, posts form a further nested data type. A post contains text and can be reacted to and commented on. Comments can, in turn, be liked by other users. This results in a Parquet file with four levels of nesting where every atomic column is stored in a flattened representation, e.g., all user names will be stored in a single column, and the texts of all comments in the entire dataset will be stored in a single column. The Parquet file stores additional metadata for each value to later be able to reconstruct the nested records correctly.

Figure 3 visualizes how an instance of the schema in Figure 2 can be stored with Parquets nesting encoding. The left side visualizes the instantiation with JSON, and the right side uses record shredding to split the dataset into columns of primitive types. This examples contains entries for the two users "alice" and "bob". The visualization on the right is represented as a tree. All leaf nodes represent atomic values as stored in a Parquet file. The repetition levels (R) and definition levels (D) are encoded as metadata per column (light blue).

To reassemble the nested data structure, we need to scan the required leaves sequentially. Whenever the repetition level value (R) is zero, it denotes the start of a new entry at the highest level, which is e.g. a new user in our dataset. If the repetition level is one, it denotes the start of a new post or follower entry since these are our first-level nodes. We observe the largest variety of repetition level values in the Likes column. This column is nested inside three other nodes (Users, Posts, and Comments). Therefore, the maximum repetition level is three. If the repetition level equals three it is repeated on the innermost level, which means that the entry belongs to the same comment as the preceding one. In our case, this means that like 908 and like 406 belong to the same comment. If we start scanning the likes from the beginning, we see that the first two entries are empty. To understand that, we have to look at the definition level values. Since arrays can be empty, there might potentially be users without any posts, posts without comments, or comments without likes. This means that an entry in likes can be undefined at all levels. The definition level tells us at which level the value is still defined. The maximum definition level of the likes column therefore is three. For the first entry, the definition level is 2 which tells us that it is defined until level 2 (comments). Therefore, we have a post and a comment, but no like for the first post of the first user. The second entry is 1, which means the post is defined, but no comment exists for it. The last three like entries all have the maximum definition level which means that they are defined.

Google's original reassembly algorithm is based on a finite state machine (FSM) that allows reconnecting the layers[32]. The idea is that each state in the FSM represents one leaf node and the transitions are labeled by the repetition levels. The state machine is constructed based on the columns that are required and can be reduced to a subset of columns to speed up the reassembly. Consequently, this algorithm scans all nested columns at once, jumping back and forth between different nesting levels.

In our approach, we opt away from this approach and build a semantically equivalent algorithm.

2.3 Challenges of Existing Solutions

Query engines, especially relational databases, commonly work on data in a normalized relational format. Previous efforts to support nested Parquet to meet the increasing demand for analytics on data lakes immediately reconstruct the nesting from the flat layout into nested types. They make an effort to integrate nested types such as lists as new data types. This requires adding new logic to all relation operators. As described in Section 2.1, Parquet scanners themselves entail sophisticated functionality. Therefore, we argue for a decoupling of scanning and nesting logic.

Instead of scanning the whole data in one pass, we interpret the nested columns of the Parquet files as independent sets of flat columns and reconnect the sets later with joins. Hence, we can scan all the layers independently and parallelize over subsets of data more easily. Furthermore, as we treat each nesting level as simple columns, we also do not introduce any new data types.

Another advantage is that nested data appears like flat data to query optimizers, which can tremendously improve execution plans. We can build this approach on existing logic in query engines for query optimization and join ordering, taking advantage of decades of database research.

3 Nested Data Normalization

In this chapter, we will describe how a nested dataset can logically be split into multiple flat, or normalized, datasets while keeping the connection between the different nesting levels. On a high level, we first group the columns into nodes based on their repetition level. Each node represents a relation that only contains columns of basic types. We call this decomposition of nested data into relation-like nodes "normalization". In the second step, we will explain how we keep the connection between the different layers with generated keys. These keys are generated for every query on-the-fly and allow us to later join the base relations to answer queries that require columns from more than one node, like the number of posts per user in our example in Figure 2.

3.1 Logical Schema Normalization

In this section, we discuss how we can retrieve the different Parquet scan nodes that logically split and group the columns into basic relations. Figure 4 visualizes how nested datasets can be interpreted as normalized nodes of flat datasets that are connected with one-to-many relations using the schema of Figure 2. We use Google's Protocol Buffers [23] syntax to formalize the schema. All elements in nested schemata are classified as *required*, *optional*, or *repeated*. Required values appear exactly once, which is the same as *not null* in a relational schema. Optional values can appear once or not at all, which is like *nullable* in a relational system. Repeated values can occur not at all, once, or more than once. In a relational system, repeated values would be represented as some kind of array type or by introducing a new table. The repeated keyword is not trivially transferable to a normalized schema since it should only contain atomic values. Whenever a column is repeated, we assign this element and its potential child columns to a new relation. In other words: We logically normalize the schema by expanding all arrays into separate relations.

In our social media example, splitting the schema into relations based on the repeated keyword gives us the conceptual data model on the right side of the figure. All boxes are connected to their parent nodes with one-to-many relations. For the generated nodes, it does not matter whether the repeated column is of an primitive type or a group. Both, repeated groups such as posts, reactions, and comments as well as arrays with primitive types, such as followers and likes, are translated into separate nodes. The repeated groups will contain a column in the node for every contained flat member. For example, the reaction node will contain one column for the userid and one column for the emoji. The likes and followers nodes will only contain one column for the primitive integer type, which represents the userids.

Since we reinterpret the nested columns as logically split relations, we need surrogate keys to maintain the relationships between the logical relations which will be discussed in the following section.

```
message Users {
  required int64 UserId;
  required varchar Name;
  repeated int64 Followers;
  repeated group Posts: {
    required varchar Text;
    repeated group Reactions: {
      required int64 UserId;
      required varchar Emoji; }
    repeated group Comments: {
      required int64 UserId;
      required varchar Text;
      repeated int64 Likes; }}}
```

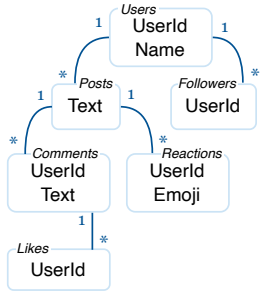


Figure 4: Conceptual grouping of nested columns into relations of simple domains. Social Media Example from Figure 2.

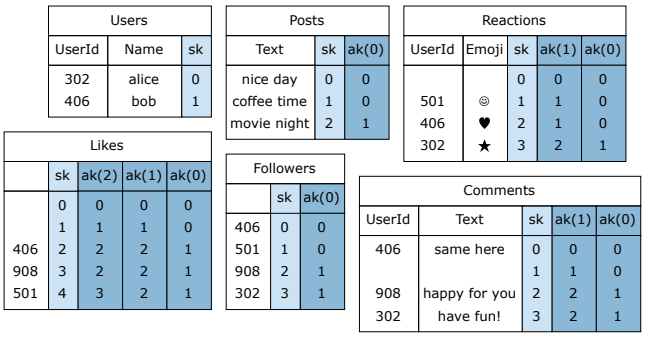


Figure 5: Reinterpretation of nested Parquet data as relations with on-the-fly generated surrogate keys (light blue) and ancestor keys (dark blue). This is only a logical reinterpretation, no data from the Parquet file is transferred into the internal database storage layer (Social Media Example from Figure 2).

3.2 Generate surrogate and ancestor keys

To keep track of the one-to-many relationships between the different relations, we work with generated pairs of primary keys and foreign keys which we generate on-the-fly. We treat these keys like additional columns inside the relations. Figure 5 visualizes the resulting logical relations with the additional key columns for the running example. An example of a one-to-many relationship is the relationship between users and their posts. Each user can be associated with many posts, so we add an additional primary key column to the user relation. The posts relation gets an additional foreign key column that references the primary key column of the user. We call the generated primary key column *surrogate key* and the foreign key *ancestor key* since the posts relation is a child node of the users relation.

As primary keys, we use the row numbers of the users. Generating foreign keys that match the generated primary keys of the parent nodes is more complex. For every post, we need to get the row number of the corresponding user. We retain this information using the repetition levels that are part of the nested Parquet encoding. The repetition level is used to encode at which level the current entry is repeated. Values can be repeated multiple times along their path. For example, the comments in the running example are contained in two nested repeated groups: Users and posts. To compute the surrogate key, i.e., the row number of the corresponding parent

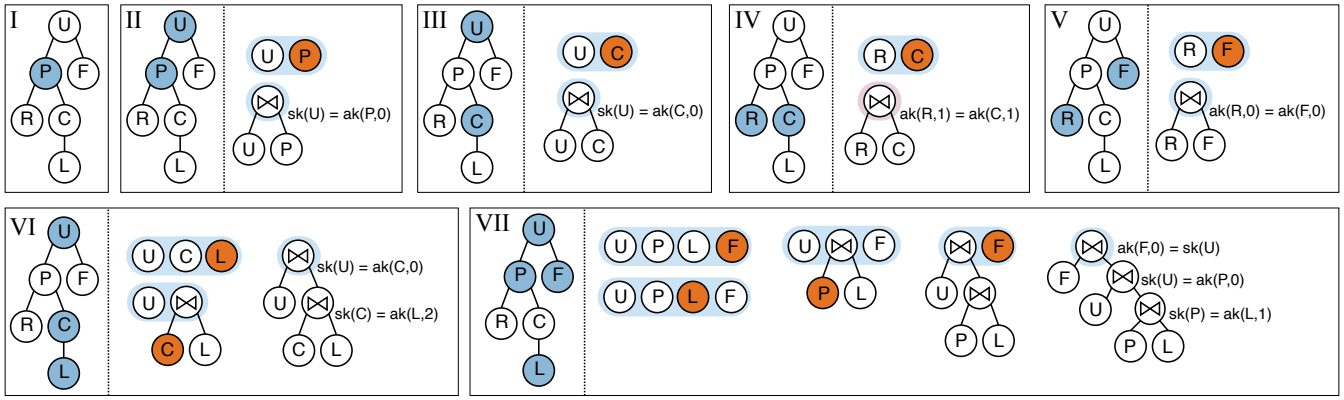


Figure 6: Seven examples of building initial join trees. Examples of join tree construction. The required nodes are blue, the light blue list contains the required nodes in preorder traversal order, and the currently processed node is highlighted in orange.

node, we have to scan the repetition level values of all previous rows until the required entry. Whenever the repetition level value is less than or equal to the nesting level of the parent, we increment the parent surrogate key. We call the corresponding surrogate keys of ancestor nodes *ancestor keys*. Nodes can be nested in multiple repeated groups, therefore we create ancestor key columns for every nesting level. This allows us to skip intermediate nodes later and directly join two nodes that are not neighbors. Figure 5 visualizes the relational interpretation of the nested Parquet data along with the generated surrogate and ancestor keys of the running example. In practice, these columns will only be generated on-the-fly during query execution.

The posts and followers relations only have one ancestor at level 0, so they just have one foreign key column. The likes relation has two ancestors: Posts at level 1 and users at level 0. Therefore, it has two foreign key columns referencing the nodes at the corresponding levels. The comments relation has three ancestors and therefore three foreign key columns. The keys for different values allow us to directly connect two base relations that are not directly connected with a one-to-many relation. If we for example search for all comments that user "bob" ever received, we can filter the comments relations for rows where ak(0) equals the row number of "bob". In Section 4, we explain in more detail how this can be integrated and how the query engine chooses the correct pairs of surrogate and ancestor keys for the join predicates.

```
ancestorKey = -1
for i in range(rowCount):
  ancestorKey += rows[i].repetitionLevel <= ancestorLevel
  ancestorKeyBuffer[i] = ancestorKey
```

Listing 1: Simplified pseudocode for computing ancestor keys.

The pseudocode in Listing 1 shows how ancestor keys are computed. Given a node and an ancestor for whom we need to generate foreign keys, we start by retrieving the nesting level of the required ancestor. Then, we loop over all rows in our node and check for every tuple if the repetition level is less than or equal to the ancestors' nesting depth. Whenever this is the case, we increment the current ancestor key index (`ancestorKey`). This is a simplified realization of the computation which can be further optimized for parallelized execution of the scan (c.f. Section 5).

4 Expressing Queries as Joins

After explaining the theoretical aspects of how nested Parquet files can be normalized, we now describe how the normalized data can be queried in a typical relational query engine. We integrate our approach into a fully fledged database system, but the techniques that we describe in the following are applicable to any query engine. This general applicability is also demonstrated later, in the evaluation in Section 6. There, we simulated the integration of our approach into other systems by manually transforming Parquet files accordingly and measured the performance impact.

In this section, we show how the initial join tree can be constructed in linear time, in a single pass.

Given a SQL query, there is a set of required columns. Our approach internally joins the Parquet nodes to provide a normalized result. In this section, we describe how the initial join tree is built. We refer to it as the "initial" join tree since afterward, a query optimizer can take the join tree as input and optimize it based on the available statistics.

To make the explanation more visual, we will use the examples in Figure 6 to guide us through the different cases based on the social media example from Figure 2. We abstracted it with nodes (c.f. Figure 4) and used the first letters for their naming: U for the users, P for the posts, R for the reactions, C for the comments, L for the likes, and F for the followers.

For every example, we highlight the required nodes in blue. In the end, our join tree will only contain the highlighted nodes. In the following, we list a possible query that would have resulted in the given highlighting for every example:

I Get all posts about coffee.
II Get the users with the longest posts.
III Get the users with the most comments.
IV Get comments from users who also reacted to the post.
V Get followers that write the most comments.
VI Get users that commented and liked their own posts.

VII Get followers of Alice that liked any comment of her posts about movies.

The first step in the join tree construction is to evaluate which nodes contain columns that are required in the query. We already did that for our examples in Figure 6 by highlighting the required nodes in blue.

I. For the first example, we require all texts of posts, so only node P. Therefore, we do not need any join and do not build a join tree for this case. This showcases how beneficial it is to stick with the columnar representation as long as possible. Since we do not store the data in an explicitly nested representation, we will only scan the data we need and do not have to access all entries of U to get all entries of P.

Examples II to V only require two nodes. These examples demonstrate the four different join types that can occur during the join tree construction:

(a) Join with parent node.
(b) Join with an ancestor node.
(c) Join with a sibling node.
(d) Join with a descendant of a sibling node.

Examples VI and VII require more than two nodes. These examples show how our approach generalizes to more complex scenarios with more than two nodes. In example II, we need to join the users with the posts which corresponds to a join between a parent and a direct child node. To do that we take the surrogate key from the users node U and the ancestor key from the posts node P. Example III requires information about users and the comments that they got, so we are joining a node with an ancestor node that is not a direct parent. In this case, we again take the surrogate key from the node higher up in the path U and from the comments node C, we choose the ancestor key that references the correct level, in this case, we need the *ak* that references level 0.

Example IV joins the reactions R with the comments C, which are two sibling nodes. At first, we have to identify their lowest common ancestor, which is the posts node P in this case. Then, we take the ancestor keys from the required nodes R and C that reference node P. We do not need to include the posts node P itself in the join tree since the ancestor keys from nodes R and C reference the same column in P (sk(P)) and therefore we can directly compare the ancestor keys from R and C. Example V shows that we might also need to join two nodes that are not direct siblings but are themselves on different levels in the tree. In our example query, with nodes R and F, we are joining the followers with the reactions. In the end, this does not change anything for the tree construction: We approach it the same way we approached example IV. We identify their lowest common ancestor and then choose the ancestor keys corresponding to the level of the lowest common ancestor. In this case, this is the root node U on level 0.

Before we explain the last two examples, we introduce our general approach to join tree construction, which we show in Listing 2. The basic intuition is to start with the nodes at the bottom and move up the tree until we reach the top-most node (`limit` in line 10). For example VI, this means that we will at first join nodes C and L and then we will add node U with another join. The resulting joins are of the same form as the joins from examples II and III. To state this

```
1  def storeJoin(join):
2     current = predecessor
3     ancestor = current.parent
4     current.replaceWith(join)
5
6  def buildJoinTree(current, limit):
7     predecessor = current.predecessor
8     ancestor = current.parent
9
10    while current > limit:
11       if (predecessor == ancestor):
12          join = joinSkAk(predecessor, current)
13          storeJoin(join)
14       elif (predecessor < ancestor):
15          ancestor = ancestor.parent
16       else: # predecessor > ancestor
17          if # only one element from left sibling:
18             join = joinAkAk(predecessor, current)
19             storeJoin(join)
20          else:
21             subtree = buildJoinTree(current.predecessor,
22                       ancestor)
23             if finalNode(subtree) == ancestor:
24                join = joinSkAk(subtree, current)
25                storeJoin(join)
26             else:
27                join = joinAkAk(subtree, current)
28                storeJoin(join)
29    return join
30
31 buildJoinTree(requiredNodes.last, requiredNodes.first)
```

Listing 2: Pseudocode for computing the initial join tree.

intuition more precisely: We sort the tree in preorder traversal order, then we add all required nodes into a list based on the ordering, and then we scan the nodes from back to front to get the join tree. For each node we visit in the list, we compare its predecessor with the parent of the current node, and based on the result, we decide what kind of join we have to construct. The predecessor can either be equal to (line 11), less than (line 14) or greater than (line 16) the parent, which are the three cases visualized in the pseudocode in Listing 2.

II. In the second example, we build the chain with the nodes U and P and start at the back with node P. Comparing its predecessor U with the parent of P, which is also U, gives us equality. Therefore we know that the join we need is a join between the surrogate key (sk) of the parent and the ancestor key (ak) of the child node. In the pseudocode this kind of join is abbreviated with a function call to `joinSkAk` (line 13).

III. The third example starts at node C. In this case, the predecessor U is less than the parent P (line 14). In this case, we will move up to the next ancestor which is in this case the grandparent of node C. In the pseudocode, we will enter the while loop once again and now we meet the first condition and therefore again construct a join between a surrogate and an ancestor key.

IV. - V. The fourth example demonstrates the third case since the predecessor of C, which is R, is greater than the parent of C, which is P. This tells us that the next required node is located in a sibling subtree from the current node. The fourth example only requires one node from the sibling subtree, which fulfills the first subcondition

(line 17). We will construct a join between two ancestor keys and therefore call the `joinAkAk` function. The fifth example does the same as the fourth. Again, the predecessor of F, which is R in this example, is greater than the parent of F.

VI. Example VI works with three nodes. We start by building the list of required nodes where all nodes are added in preorder traversal order. Then, we start at the back with node L. The predecessor of L equals the parent of L, so we call `joinSkAk`. Then, we will replace the chain nodes C and L with the join node we just built (line 2). Next, we move to the preceding element in the chain C now represented by the newly generated join. We compare C's predecessor P with its parent U. P is greater than U, so we increment the parent distance and continue with the grandparent which is U. Now, we again construct a join between a surrogate and ancestor key.

VII. For the last example, we start with node F and compare the predecessor L with the parent U. L is greater than U, but this time we see that the next predecessor P is still greater than U, so we know that we need more than one element from the left sibling. In this case, we will construct a subtree for the left sibling and then return to the node F again when we reach the ancestor of the current node. In our algorithm, we do this with a recursive call to the build function (line 21). If we reach the parent of node F (`limit` in line 10), we stop and return the constructed subtree. In our case, the recursive call will start with node L. The predecessor P to parent comparison C tells us that we need to move up to the grandparent where we call the `joinSkAk` function. Then, we continue with node P and again call `joinSkAk` since U is the predecessor and parent of P. Now, since we reached the parent of F, we return to the outer call and continue with node F again. The constructed subtree with the final node U is the new predecessor representing U. Since U is the parent of node F, we call once again the `joinAkAk` function. This approach allows us to scan the list of required nodes only once and therefore finishes in linear time. If node U is not required, we would perform a join between two ancestor keys by calling `joinAkAk` with the final node of the subtree.

5 Implementation

In this section, we describe the modifications required for existing database systems to support nested Parquet files and how they can be smoothly integrated with the existing code base.

Changes in Our Implementation. We built our nested Parquet support on top of the existing Parquet file scanner in Umbra [34]. Inside the Parquet Scanner we add the computation of ancestor and/or surrogate keys. These generated keys are treated by our Parquet Scanner like standard columns in most parts, but have a specialized way to retrieve them. Since generating the surrogate and ancestor keys is cheap and only introduces minor overhead, which we show in Section 6.6, we do not store them in the database, but instead recompute them for every query. Our SQL dialect provides all nested columns by their names. During the semantic analysis, we then generate the required joins as described in Section 4.

Required Changes for Other Systems. To integrate nested Parquet files with our approach, existing systems have to be modified and extended in three places. First, the database system needs to support a nested SQL syntax, if it does not support it already for other nested data formats. Second, during the semantic analysis phase the nested SQL syntax must be translated into appropriate relational algebra, incorporating the necessary joins. The database engine then perceives the Parquet file as a collection of relational tables. Third, the Parquet scanner must be extended to compute ancestor and/or surrogate keys when needed. Importantly, the main path of the Parquet scanner implementation remains untouched, ensuring that the performance of Parquet files without nested columns is unaffected.

Optimizing Computation of Ancestor Keys. For ancestor keys, we scan the repetition level of some column of the requested node. Since it does not matter which column we choose, we piggyback the scanning of the repetition level on the scan of the first required column that is needed anyway. For every ancestor key we need, we track the nesting level of the respective ancestor. Whenever the repetition level value is smaller or equal to this nesting level, we increment the ancestor key.

```
1  def computeAncestorKey(rowGroup,page,offset,blockSize):
2     ancestorKey = -1
3     for (r = 0; r < rowGroup; r++)
4        ancestorKey += numValuesParentColumnChunk(r)
5     for (p = 0; p < page; p++)
6        ancestorKey += getAncestorKeyJumps(p)
7     for (o = 0; o < offset; o++)
8        ancestorKey += repLevel <= parentLevel
9     for (i = 0; i < blockSize; i++)
10       ancestorKey += repetitionLevel <= parentLevel
11       ancestorKeyBuffer[i] = ancestorKey
```

Listing 3: Pseudocode for efficiently computing ancestor keys

In Listing 3 we formalized the computation of the ancestor keys in pseudocode. The function takes as input parameters the index of the row group, the index of the page, as well as the offset in the page and the size of the block we want to process. In the first step, we need to get the ancestor key of the first row in this row group. Afterwards, we can incrementally determine the following keys. To get to the first ancestor key, we do not have to scan all previous row groups, but instead, we can base the computation on information from the Parquet file metadata: Row groups are used in Parquet files to split the data into horizontal partitions. For each row group, the Parquet metadata footer tracks the number of rows per column inside the row group. This information is important especially for nested data since the total number of rows we have for a repeated column is different from the number of rows of an atomic column. The first ancestor key of a certain row group equals the row number of the first parent row in this row group. To get the row number of the first parent element, we can sum up the row counts of all preceding row groups (line 4). This works since all nested data is stored along with their respective ancestor rows in the same row group.

To reach the correct page inside the required row group, we cannot use any metadata of the parent level since the spread of rows over pages does not have to align with the nested rows. This means that if the parent level stores 1,000 rows on the first page, it does not imply that the nested column chunk stores all nested rows belonging to these first 1000 rows on a single page as well. Therefore, the repetition levels of all previous pages leading to the required page have to be scanned (line 6). This information cannot be obtained from the metadata footer. Inside the required page, we compute the number of increments until we reach the offset

(line 8) and then we start gathering the ancestor keys for the block and store them in a preallocated array (line 11). This approach parallelizes trivially over row groups since we can easily compute the starting ancestor key per row group. For more fine-grained parallelization below row group level, which we do in our multi-threaded implementation, we compute the increments per page only once, and all following threads can reuse the result.

Leveraging the Query Optimizer. While the changes to the scanner were only minor, there are a lot of benefits we get from extending and relying on an existing scanner implementation. An efficient scanner implementation can evaluate predicates already during the scanning process and therefore can largely reduce the amount of data we have to read for selective queries.

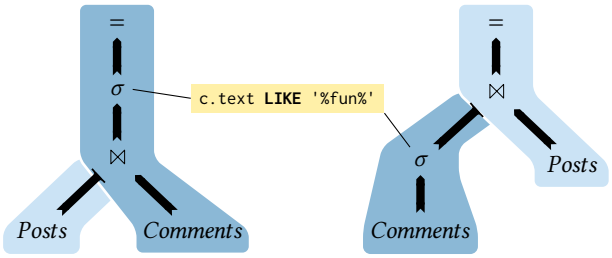
The join tree can be fed into the standard query optimizer which is then able to treat filter predicates over an array like predicates over atomic values. If possible, the query optimizer can push the predicates down to the scanner which is then able to apply early predicate evaluation as usual. In Figure 7, we visualize the query plans for an example query on the artificial social media dataset to emphasize the impact that the query optimizer can have. The query returns all posts that have a comment including the word "fun". Figure 7a shows the unoptimized query plan that the initial join tree construction will generate. At first, the posts and comments nodes will be joined, and afterward, the predicate is evaluated. Due to the simple nature of the constructed query plans, established techniques, such as predicate push down can be applied: The query optimizer will push the selection down to the comments scan. Assuming that the cardinality of comments that contain the word "fun" is lower than the total number of posts that exist in the dataset, the query optimizer can also switch the probe and build sides of the join operator. All of this works because our approach produces relational joins over relational data, meaning every RDBMS can take advantage of it.

Pages of Parquet files have optional fields for the page's current minimum and maximum value, called *ColumnIndex* [7]. Since these are generated independent of the nesting depth of the column chunk, the query engine can — as usual — use these for pruning by determining if the page can contain any matches. This logic is already implemented and since we move the nesting out of the Scanner logic, we can evaluate those predicates as usual on the atomic values without the scanner being aware that the column is in fact, an array.

Optimizing Data Locality with Merge Joins. Surrogate keys are row numbers and therefore sorted by design. Nested elements are sorted by their parent elements which makes them also sorted by the ancestor key columns. For our Parquet node joins this means that the join partners are sorted by the join keys. However, many query engines would choose a hash join by default since they are unaware of the sorting, which reduces data locality.

Intuitively, the best approach to join two lists already sorted by the join key would be a merge join. Extensive work on comparing hash joins to sort-merge joins [11, 28] shows sort-merge joins are usually slower than hash joins due to their expensive sort phase. However, in our case, we can skip the sort phase and only do the merging step which makes merge joins a strong contestant.

Unlike the classical sort-merge join, where the left and right sides are scanned in parallel, our approach has to generate the join keys



(a) Unoptimized Query Plan (b) Optimized Query Plan

```
SELECT "posts.text"
FROM 'socialmedia.parquet'
WHERE "posts.comments.text" LIKE '%fun%';
```

Figure 7: Query plan optimization for an example query over the nested social media Parquet file. Posts and Comments are perceived as separate relational tables.

at the same time. For an efficient multi-threaded implementation, we materialize the left input with the generated join keys first and then scan the right side. We materialize each sorted chunk from the left and store its key range in an index. For each sorted chunk from the right, we find the matching region to its first tuple in this index. We then skip to the correct offset and sequentially scan both chunks in parallel to find matching tuples. When we reach the end of a left-side chunk, we move to the succeeding chunk, continuing until the right-side chunk is fully processed.

Using this approach, our merge-join implementation achieves slightly better performance than hash joins for small build sides and outperforms hash joins when the hash table exceeds the CPU cache capacity, as visualized in Figure 8. We demonstrate the relative speedup of merge joins over hash joins for two different dataset sizes: one that fits into the CPU caches and one that exceeds them. For the smaller dataset, the performance is similar, but for the larger dataset, merge joins are more efficient. However, the difference is not as large as one might expect. This is because hash joins also benefit from the sorted inputs. In our case, when probing with the elements of an array, cache misses only occur whenever the first child of a parent is probed. Subsequent elements read the same cached hash table entry, minimizing cache misses.

In Figure 8 we vary the array sizes from 1 to 20 to validate this effect. Especially in the out-of-cache case, hash joins perform worse for very small arrays. But the gap narrows substantially for larger arrays. While hash joins are not prohibitively slower, especially with large arrays, we chose to use merge joins in our implementation as they consistently offer slightly better performance. However, this optimization is not crucial and one could achieve most of the speedup of our approach without implementing a new join type.

6 Evaluation

For the evaluation, we integrated our approach into Umbra, a fully fledged database system which supports processing Parquet files without nested data. We compare our join-based approach to DuckDB [35], Trino [42], and AsterixDB [3], all of which natively support nested Parquet files. The difference between their approaches and our is that they interpret the nested Parquet files as nested data and base their computations on nested scans over

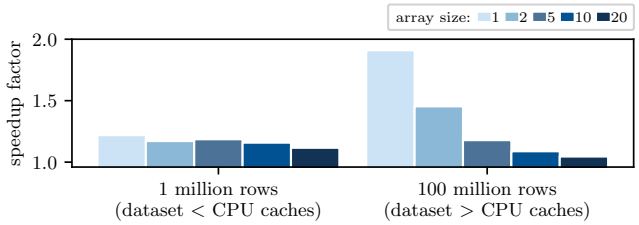


Figure 8: Speedup of merge joins compared to hash joins. The Parquet file contains an integer column and an integer array column. In the query, the integer column is joined with the elements of the corresponding integer array. The colors represent the length of the integer array.

Dataset	File Size	Nesting Depth	Node Count	Tuples at Root	Max Tuple Count
Synthetic	80 MB	1 - 6	1 - 6	10 M - 10	10 M
DBLP	516 MB	1	9	6 M	13 M
TPC-H SF 1	290 MB	5	5	5	6 M
TPC-H SF 10	2.6 GB	3	3	1.5 M	60 M
Twitter	6.4 GB	3	153	7 M	13 M
XMark	555 MB	3	38	1	600 K

Table 1: Properties of utilized datasets

the data. This is the bottleneck of the built-in approaches. When nested columns are retrieved with their outer layers in one pass, this limits the degree of parallelization for inner layers. The processing in parallel blocks is fixed by the blocksize that is chosen for the outermost required layer. Not only in terms of parallelization, but also for predicate pushdown or grouping, we have a lot more freedom by processing all levels separately and joining them as late as possible. We also generate manually split files to demonstrate how well these other query engines could perform if they used our approach.

All experiments were conducted multi-threaded 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 stored on a local Samsung 850 Pro SSD with 2 TB of storage space. All queries where executed 10 times, with the fastest execution time recorded.

For the experiments, we use five datasets. Table 1 lists the different specifics of all datasets that we found useful for interpreting the results. For the nesting depth, we only take those nested columns into consideration that are repeated. The node count indicates the number of normalized Parquet nodes when splitting nested Parquet files with our approach. To show the range of tuple counts at different levels, we list the tuple count of the root node as well as the tuple count of the node with the most elements. All nodes will at least have the same number of tuples as level 0.

6.1 Impact of Nesting Depth

In the first step, we want to compare the robustness of the different systems to see if they maintain a stable performance independent of the nesting level. We work with a synthetic dataset that has varying numbers of nesting levels. At each level, we store only one integer column with randomly generated values. To ensure the same result set size across all datasets, we maintain a constant number of tuples at the deepest nesting level. For the base dataset with no nested column, we have 10 million integers. For the dataset with one level of nesting, we store 1 million tuples, each containing an integer and an integer array with ten elements. This results in a total of 10 million integers across all nested arrays. For every additional nesting level, we use the same approach: We divide the number of tuples at the top level by ten and add a new nesting depth by creating an additional array of length ten for every entry at the currently deepest level.

We ran four different queries on the dataset. The first query accesses the innermost attribute, which always requires a scan of 10 million rows. In the second query, we retrieve two attributes from neighboring levels, resulting in one join operator in our query tree that joins a dataset of 1 million tuples with a set of 10 million tuples. In our third query, we again retrieve two attributes, but this time we increase the nesting level distance between them. For the final query, we do a full table scan by querying the attributes from all levels, leading to one additional join for every added nesting level. To keep the output size small, we added aggregates in all queries on top of the scans.

Figure 9 shows the query throughput for all systems visualizing both the absolute throughput and the relative slowdown as column nesting depth increases. For all queries, our implementation is significantly faster than the others. In the first graph, we query a column from seven different levels, where each result set contains the same number of tuples. Since the column values are stored identically for all seven levels, the execution time should be constant. In our system, we see that our approach allows us to directly access the required column without dealing with any sort of nesting and therefore we have no performance differences. Both DuckDB and Trino have additional internal computations that grow with increasing nesting depth. DuckDB has a significant performance drop comparing no nesting and only one nesting level of 50×. Afterwards, the performance decreases linearly. For query 2, the normalized performance numbers are less significant but still our approach overall shows a more stable performance whilst for the other systems there is a performance drop.

In query 3, the execution time of our system is a bit worse for the first two cases whereas for the rest it is again near constant. The reason for it is that we need more tuples on the root level to produce the same number of output tuples. Therefore, the build side is bigger and needs a bit more maintenance. The other systems again experience a performance drop. For AsterixDB, we cancelled the execution of the last case because it did not terminate under 15 minutes. For the last query, which essentially is a full table scan, all systems show a decline in performance. This behavior is expected, as each new column from an additional level introduces another join operation to the query plan. One might assume that, in this scenario, the traditional nested scans would outperform our join-based approach. However, despite the increased complexity from adding joins for every level, our relative throughput is still more stable and, in terms of absolute numbers, remains significantly faster. Even for the last case where the dataset contains six nesting levels for which we need five joins, we are 10× faster than Trino and even 45× faster than DuckDB. In AsterixDB, we again cancelled the last case because it did not terminate under 15 minutes.

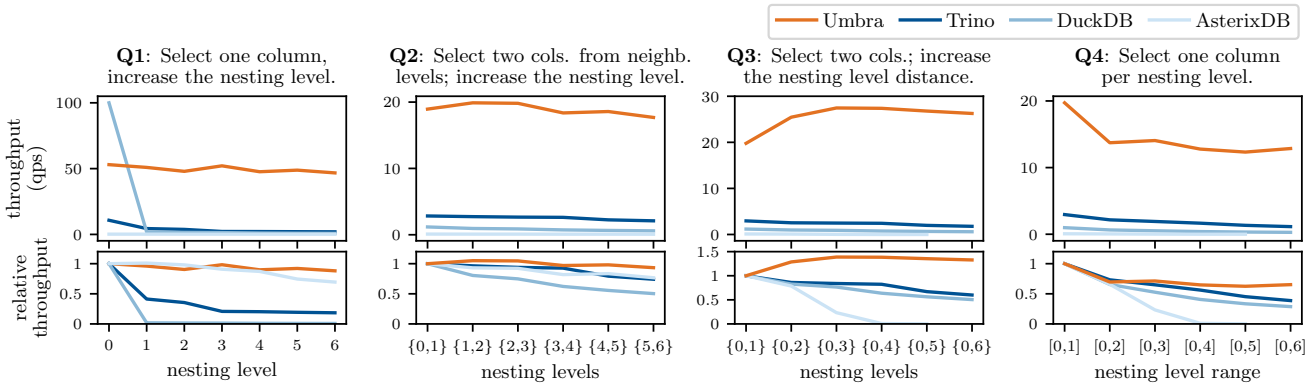


Figure 9: Throughput of queries on synthetic datasets with varying nesting levels for four different queries. The top graph visualizes the throughput in queries per second. The graph on the bottom shows the relative throughput of the queries (normalized to the first/least nested version).

6.2 Evaluation Datasets

In addition to the benchmark on the synthetic dataset, we also did benchmarks with four more realistic datasets:

The **TPC-H** benchmark includes a dataset with several flat relations. For our use case, we worked with two versions of the dataset. We used the dataset with scale factor one to nest five of their relations that are connected with one-to many relations into one nested relation: We nested lineitems into orders, orders into customers, customers into nations, and nations into regions. For the scale factor ten, we only nest three of their relations: lineitems into orders and orders into customers. From the TPC-H queries, we selected those that only run on the nested relations. Queries 1 and 6 only read from the lineitem relation. The others originally join multiple relations together. We removed these join predicates since those implicitly hold due to the nested representation. Queries 4 and 12 contain one join, query 3 contains two joins and query 10 contains three joins, which is why we only use it for the SF 1 version.

The **Twitter** benchmark, developed as part of the JSON tiles paper [21], uses a dataset of 7 million tweets from June 1, 2020 [10]. We converted the dataset to the Parquet format and evaluated the five queries on it. Three queries access a single node, while the other two access two nodes each, resulting in one join.

The **XMark** benchmark [37] was built to evaluate XML engines with queries written in XQuery. We generated the dataset in XML with the scale factor 10 and transformed it to Parquet. From its 20 XQuery queries, we used all except 6 queries that focus on XML specifics. Most of the queries only query attributes of one level. Only queries 2, 3, and 4 query attributes from two levels.

The nested **DBLP** dataset was generated to evaluate JSON and XML DODBMSs [43]. Since the dataset only contains arrays with basic unnested types, the maximum nesting depth is one. Most of the queries only differ in the filter predicate but not in the used fields and nesting level. The benchmark consists of nine queries. Five queries only query the root node. The other four queries group publications by authors and therefore require two levels.

6.3 Impact of Optimizations

In Section 3 and Section 5, we discuss several optimizations for our nested Parquet scanner. To evaluate the impact of these optimizations individually, we added them one by one to our baseline and evaluated all benchmarks for each configuration. The first and most impactful optimization is enabling the query optimizer. Since we abstracted the nesting from the query engine, this optimization is effectively "free". The query optimizer treats the nested data as joins and applies existing optimization techniques without requiring additional adaptations. Another optimization adds the ability to compute ancestor keys for arbitrary ancestors. This optimization does not add a significant improvement for our workload, because most of the benchmarks do not query nodes that are not direct ancestors. For the five queries in our benchmark that benefit from this optimization, the speedup is significant. We recommend to use this optimization, because it is rather straightforward to implement and offers significant benefits for level-skipping queries. In our last optimization we replace hash joins with merge joins. This change results in only minor improvement, consistent with our findings from the microbenchmarks in Figure 8, especially as array sizes increase. Since arrays in our workloads are mostly longer than one or two elements, the benefits of this optimization are minimal.

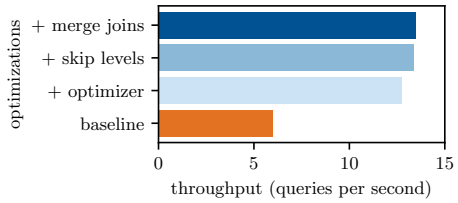


Figure 10: Performance improvements of optimizations on the entire benchmark set from Table 1. Query optimizer, skip levels, merge joins instead of hash joins.

6.4 System Comparison

To evaluate how our approach performs on real-world data, we ran the queries of the different datasets on all four systems. Figure 11 shows the speedup of our implementation compared to the native implementations of DuckDB, Trino, and AsterixDB. For all datasets, our approach achieves significant speedups of 10× to 100× for nearly all queries. On the TPC-H dataset with scale factor one, all queries show speedups over 100× compared to DuckDB. For Trino, the speedups range from 10× to 100×. In AsterixDB, only two queries

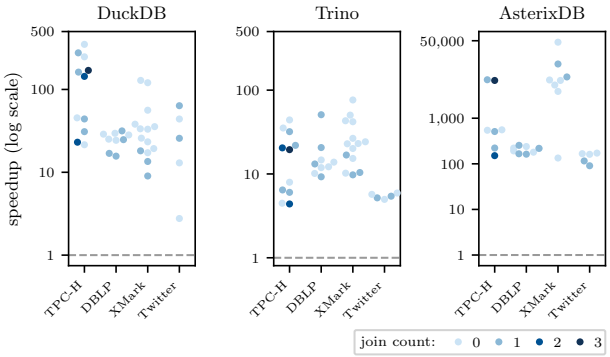


Figure 11: Speedup of our approach implemented in our own system over DuckDB and Trino. The colors depict the number of generated joins in the query.

completed within 15 minutes, both with speedups exceeding 1000×. At scale factor ten, the speedups are smaller due to the reduced number of nesting levels in the dataset (see Section 6.2). Our system is at least 10× faster than DuckDB and Trino and 100× faster for AsterixDB for most of the DBLP queries. The XMark benchmark contains the most queries out of all four benchmarks. Our system outperforms DuckDB and Trino with a speedup range between 10× and 100× for most of the queries. For AsterixDB, its even between 100× and 50,000×. The Twitter dataset has the biggest file size. For Trino, we get speedups around 8× for all queries. For DuckDB all except one query are more than 10× faster in Umbra. The exception, query 2, operates on columns from the top-level therefore no joins or complex nesting techniques are required which would differentiate the executions. In AsterixDB, the Twitter queries are around 100× slower than in Umbra.

6.5 Our Approach in Other Systems

After comparing the built-in approaches of our system with Trino, DuckDB, and AsterixDB, we now take a look at how the other systems would perform if they used our explicit nesting approach. To evaluate our approach in other systems, we split the nested Parquet files into flattened files for every normalized node that our algorithm would generate. In addition, we added an additional column to each flattened file for the surrogate key as well as additional columns to store ancestor keys for all ancestors up to the root node. Then, we compare the built-in versions to scan nested Parquet files to join queries on our manually flattened files. In Figure 12, we formalized one of the queries of the Twitter benchmark in the SQL syntax of DuckDB. The goal of the query is to find all users who used the hashtag "COVID" in one of their tweets. We can use the built-in functions `contains` and `transform` to scan the hashtags arrays. The second version is the manual version using our manually flattened files in DuckDB. To join the hashtags with the corresponding users, we retrieved the correct flattened Parquet files that contain the users and the hashtags and join them based on the equality of the user's surrogate key and the hashtag's parent key, which we highlighted in yellow.

In Figure 13, we visualized the speedup of our approach over the built-in versions. For all systems, our approach is faster than

```
-- DuckDB --
SELECT DISTINCT user.ID
FROM 'twitter.parquet' t
WHERE list_contains(list_transform(
        entities.hashtags, x -> x.text), 'COVID');

-- Manual Version in DuckDB --
SELECT DISTINCT "user.id"
FROM 'twitter_0.parquet' t0, 'twitter_3.parquet' t3
WHERE t0._sk = t3._ak0 AND
        "entities.hashtags.text" = 'COVID';
```

Figure 12: Query 4 over Twitter dataset formalized in DuckDB: Find users that used the hashtag "COVID".

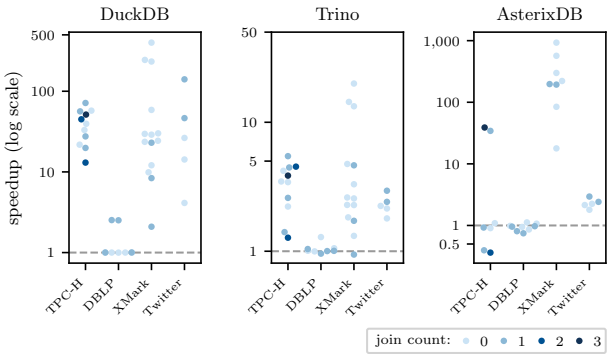


Figure 13: Speedup of manually joined normalized datasets over built-in versions of DuckDB, Trino, and AsterixDB. The colors depict the number of generated joins in the query.

the built-in versions for almost all queries. Note, that performance changes can origin from avoiding nested data, but also from changes in the structure of the Parquet file. Overall, DuckDB benefits the most from our approach, but all systems can achieve great speedups with our approach. We color-coded the number of joins that are required for the manual versions. The dark blue ones mark queries where we introduced three joins. Most of the queries are very simple and therefore require no or only one join.

The only queries where we are not able to outperform the built-in versions of DuckDB and Trino, are the DBLP queries. The DBLP dataset has only one nesting level and therefore has the lowest complexity. It is expected that in this case the difference between the two approaches is the least visible.

In AsterixDB, the DBLP and TPC-H queries for scale factor 10 are generally slower using our join-based approach compared to AsterixDB's built-in methods. However, for TPC-H scale factor 1, only two built-in queries terminated in under 15 minutes; the others only terminated for the normalized approach. In contrast to Trino and DuckDB, AsterixDB is not a relational database management system and therefore not optimized for relational data processing.

Memory Consumption. In Figure 14, we visualize the memory consumption to see if there is a trade-off between performance and memory usage. We reran all queries from Figure 13. For all four systems, the memory usage stays low for our join-based approach. For Trino, DuckDB and AsterixDB we also measured the memory consumption when using their built-in approaches. For all queries, all three systems have a similar memory usage. For DuckDB and

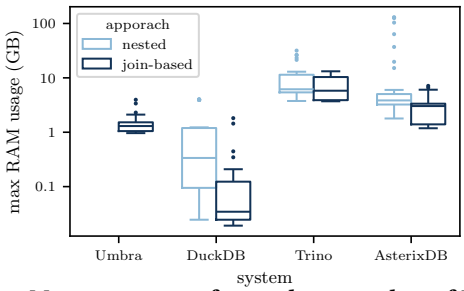


Figure 14: Memory usage of nested approaches of DuckDB, Trino, and AsterixDB and (manually) joined-based approaches of Umbra, DuckDB, Trino, and AsterixDB.

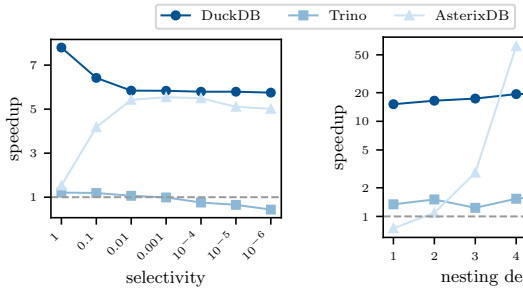


Figure 15: Speedup of manually joined approach with growing selectivity.

Figure 16: Speedup of manually joined approach with growing number of joins.

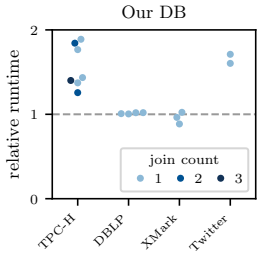


Figure 17: Overhead of key computations over normalized datasets in Umbra.

	scale factor	
Query	1	10
3	1.576	1.565
4	2.846	2.076
6	1.943	1.547

Table 2: Slowdown factor of ancestor key computation in lineitem scan for TPC-H query 3, 4, and 12.

AsterixDB, the memory usage of the join-based approach is even better than for the built-in approach. The outliers of Trino's nested approach are the TPC-H queries for scale factor 10 where 30 GB of memory are required to process a file that is smaller than 3 GB. For AsterixDB, the outliers stem from the TPC-H queries with scale factor 1 where we only have five rows on the outer level and 6 million rows on the innermost level. This imbalance seems to produce large intermediate results for the AsterixDB engine.

Challenging Queries. We performed two microbenchmarks to evaluate if built-in approaches can outperform our approach. Figure 15 shows the speedup of our approach over the built-in methods for different selectivities using the dataset from Figure 9, with one nesting level and a predicate on the outer level. Nested approaches can skip inner level scans when the outer level predicate is false, but since we scan both levels separately, we cannot skip inner-level values. Still, our approach benefits from selective predicates by reducing the join build side. In Trino, the built-in approach outperforms ours for extremely selective queries, though the performance difference is minor. For DuckDB and AsterixDB, our approach remains faster, even with very selective queries. The reason for this is that to correctly retrieve the nesting, the repetition levels of all rows still need to be scanned. Our approach could be further optimized for selective queries by using "sideways information passing", where higher-level scans inform lower-level scans about qualifying row groups or ancestor key values. The granularity of this information should depend on the selectivity of the predicate.

In Figure 16, we visualize the impact of nesting depth on the speedup of our approach over the built-in approaches. Every additional nesting level, adds an additional join operator to our plan. For all three systems we can see that the speedup of our approach does not get smaller with growing nesting depth, for DuckDB and AsterixDB it even gets bigger. The reason for this behavior is that built-in approaches need to also deal with more complexity when the nesting depth grows, which seems similar to the complexity of adding another join for Trino and DuckDB.

6.6 Impact of Key Generation

In the previous subsection, we discussed the potential of our approach in other systems. The only aspect we do not consider is the overhead of computing the surrogate and parent surrogate keys required for joining multiple levels in a query. To demonstrate that this computation has only a minor impact on the overall execution time, we also run the manual normalization version in our system and compare the results with the execution times when we use our built-in approach. This comparison only includes queries that contain at least one join.

In Figure 17, we visualize the relative runtime of our approach compared to the baseline where we join on manually rewritten Parquet files. For all datasets, we can see that the slowdown is minor. Compared to the speedup our approach has over the built-in approaches of other systems, it still pays off massively. The colors denote the number of joins that were generated and with every join, additional keys have to be generated. The DBLP and XMark datasets are the smallest and their queries are equally or even slightly faster than the built-in version. For the Twitter dataset and the TPC-H dataset the overhead is more noticeable. Specifically, for the TPC-H benchmark, the queries experience a slowdown of 1.2× at scale factor 1 and 1.8× at scale factor 10. This increased slow down originates from differences in how the data is stored inside the Parquet files, rather than from the key computation. At scale factor 10, the normalized dataset uses 10× smaller pages than the nested dataset, enabling more effective parallelization.

To demonstrate the scalability of the ancestor key computation, we present the execution times for the largest table scan for queries 3, 4, and 12 at scale factor 1 and 10 in Table 2. These queries retrieve various columns from the lineitem relation with the ancestor key computation introducing varying slowdowns depending on the queried columns. For instance, query 3 involves retrieving two more expensive columns compared to query 4. Therefore, the impact of the additional ancestor key column is stronger for query 4. Despite this variability, the slowdown factor does not increase with scale; it even slightly decreases between the two scale factors.

7 Related Work

We identified two relevant areas of related work. We start with examining related work focusing on processing of nested data in RDBMSs. Then, we discuss how other binary, columnar formats handle nested data.

Processing nested data in RDBMSs There is a plethora of work on how nesting in relational formats can be expressed in RDBMSs. Especially for XML processing and later for JSON files as well. This work focuses on converting nested data into a relational format for storage in the database system. In contrast, our approach directly processes nested Parquet files without an initial ETL phase. Therefore we cannot choose how the data should be mapped into the relational representation.

Still, we want to discuss their approaches since often they transform the nested file formats into flattened relational representations that are similar to how the data is stored in Parquet files. Existing work on storing XML in database systems mostly stems from the late 90s and early 2000s and can be split into two categories. Either it is mapped with schema-oblivious or schema-based approaches [13, 26]. The schema-oblivious approaches create relations based on the tree structure[19, 22, 24]. Since the schema is always defined in Parquet files, we compare our work only to existing schema-based approaches.

Work that matches ours closest inlines child elements with their parents if they can occur at most once and create new relations for repeated child element types [19, 38]. Their mapping is equivalent to how Parquet files handles nested data. However, unlike our approach, they must store a copy of the data in these relations explicitly. To keep the amount of generated data low, they only store references to direct parent nodes, whereas we compute ancestor keys dynamically for the required ancestor nodes, allowing us to skip unnecessary intermediate nodes.

There is further work on optimizing the storage layout of XML in database relations like using data and usage statistics to optimize the generated relational schema [17], as well as constraint-preserving approaches [18, 40]. Parquet files are predominantly used in data lakes, where direct querying of data without a costly initial loading phase is essential. Therefore, we chose not to change the layout of the data stored in the Parquet file, making these approaches inapplicable to our context.

Durner et al. [21] focus on processing JSON files. Unlike our work, they cannot efficiently process the files in situ. They materialize common schema parts into multiple chunks to be able to query the data efficiently which they call *JSON tiles*. Their columnar format has no explicit normalization process for repeated/array fields. Instead, they extract every array index into a separate column.

Steed, presented by Wang et al. [45], is an analytical database system designed for tree-structured data formats with both a row and columnar data layout. For the columnar data, they utilize the Dremel schema. They optimize for simple accesses with at most one repeated node along the path. For those cases, they choose a flat in-memory data layout which allows them to skip the FSM algorithm. Still, if more than one node is repeated, they use the baseline FSM algorithm of the Dremel encoding.

Nesting in binary columnar formats. Apart from Parquet, Apache ORC [4] and Apache Arrow [6] are the two other major binary columnar file formats. While Arrow is an in-memory format, ORC has similar objectives as Parquet. Still, Parquet is the most commonly used binary columnar format [14, 27].

ORC, like Parquet, is considered an on-disk file format. In contrast to Parquet, it encodes nesting by storing the number of repetitions per record or parent tuple. The encoding of the ORC format as repetition counts is easily compressible and therefore a good fit for on-disk file formats. Melnik et al. measured for Google datasets that the ORC format produces on average 13 % smaller files compared to Parquet [33]. On the other hand, the ORC format cannot be processed as efficiently as Parquet because it requires the reading of all ancestors of the accessed level. In contrast, for Parquet, it is sufficient to read the accessed level directly.

The in-memory format Arrow encodes nested data structures by storing offsets to child element arrays. While this allows for faster direct access, it makes compression more challenging, aligning it well with in-memory processing purposes. However, similar to the ORC format, reconstructing nested structures still requires reading multiple levels. In contrast, Parquet enables skipping levels during FSM construction based on the repetition level encoding, which we also support with the on the fly ancestor key construction.

The Arrow library also provides support for reading Parquet files by transforming data into its own in-memory representation. This Parquet reader offers useful features, such as only reading specific columns and defining filter predicates that are evaluated while reading the data. However, these features are limited to top-level fields and cannot directly access subfields within nested structures [8].

In a follow-up work of the original Dremel paper [32], Afrati et al. look at filter and aggregate queries on Dremel encoding and how these can be optimized [1]. While we work with what they call a "fully flattening" approach where normalized nodes with the same parent are joined with all tuples of neighboring nodes (cross join), they work with a semi-flattening approach where every tuple only appears once in the result set. Our approach could be extended to achieve the same result by introducing an additional outer join checking the row numbers for equality.

Alkowaileet et al. [2] present an extension of the Dremel encoding that allows changes to the schema and even columns with two or more different types. They use LSM-based document stores to store the data. Their extension to the Dremel encoding also uses one column to encode definition levels and delimiters for repeated values. Nevertheless, our approach would still be applicable.

Smith et al. [39] present *Trance* which is a framework to process nested data in spark. They store the nested collections in their system as flat collections and use relational queries to query the flat representation. Inner collections are stored separately, which is similar to how we interpret nested data inside Parquet files. While we directly query the data inside the Parquet file by just reinterpreting it, they do not talk about a specific data format but instead store them explicitly in their own system.

8 Conclusion

In this work, we present a new algorithm for nested Parquet files that scans data orders of magnitude faster than the established algorithm that is based on a finite state machine. We demonstrate that

existing query engines can scan Parquet files much faster when using manually normalized files. Furthermore, we fully implemented our approach in an existing state-of-the-art database system. This enables our system to scan nested Parquet files almost as fast as non-nested data. It is easy to integrate into existing systems for two reasons. First, the algorithm's implementation is orthogonal to the scanning logic of non-nested Parquet files. Second, it leverages existing primitives of query engines such as joins and query optimizers. In conclusion, we show how systems can achieve drastic speedups when scanning nested Parquet files.

References

[1] Foto N. Afrati, Dan Delorey, Mosha Pasumansky, and Jeffrey D. Ullman. 2014. Storing and Querying Tree-Structured Records in Dremel. *Proc. VLDB Endow.* 7, 12 (2014), 1131–1142.

[2] Wail Y. Alkowaileet and Michael J. Carey. 2022. Columnar Formats for Schemaless LSM-based Document Stores. *Proc. VLDB Endow.* 15, 10 (2022), 2085–2097.

[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (2014), 1905–1916.

[4] Apache Software Foundation. 2013. *Apache ORC*. Retrieved 2024-05-16 from https://orc.apache.org/

[5] Apache Software Foundation. 2013. *Apache Parquet*. Retrieved 2024-05-16 from https://parquet.apache.org

[6] Apache Software Foundation. 2016. *Apache Arrow*. Retrieved 2024-05-16 from https://arrow.apache.org/

[7] Apache Software Foundation. 2017. *ColumnIndex Layout to Support Page Skipping*. Retrieved 2024-05-18 from https://github.com/apache/parquet-format/blob/master/PageIndex.md

[8] Apache Software Foundation. 2022. *Can not refer to field in a list of structs*. Retrieved 2024-10-02 from https://issues.apache.org/jira/browse/ARROW-17540

[9] Apache Software Foundation. 2024. *BuildArray implementation of Arrow*. Retrieved 2024-09-30 from https://github.com/apache/arrow/blob/maint-17.0.0/cpp/src/parquet/arrow/reader.cc#L589

[10] Archive Team. 2020. *The Twitter Stream Grab - 2020.06*. Retrieved 2020-10-12 from https://archive.org/details/archiveteam-twitter-stream-2020-06

[11] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (2013), 85–96.

[12] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD Conference*. ACM, 168–180.

[13] Denilson Barbosa, Philip Bohannon, Juliana Freire, Carl-Christian Kanne, Ioana Manolescu, Vasilis Vassalos, and Masatoshi Yoshikawa. 2009. XML Storage. In *Encyclopedia of Database Systems*. Springer US, 3627–3634.

[14] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD Conference*. ACM, 2326–2339.

[15] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *DaMoN*. ACM.

[16] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*. ACM, 37–48.

[17] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. 2002. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *ICDE*. IEEE Computer Society, 64–75.

[18] Yi Chen, Susan B Davidson, and Yifeng Zheng. 2002. Constraint preserving XML storage in relations. WebDB.

[19] Alin Deutsch, Mary F. Fernández, and Dan Suciu. 1999. Storing Semistructured Data with STORED. In *SIGMOD Conference*. ACM Press, 431–442.

[20] DuckDB. 2024. *ListColumnReader implementation of DuckDB*. Retrieved 2024-09-30 from https://github.com/duckdb/duckdb/blob/665e3547d184f0b6e912681a612d5802ae12d205/extension/parquet/column_reader.cpp#L829

[21] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *SIGMOD Conference*. 445–458.

[22] Daniela Florescu and Donald Kossmann. 1999. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.* 22, 3 (1999), 27–34.

[23] Google LLC. 2008. *Protocol Buffers: Developer Guide*. Retrieved 2024-05-18 from https://protobuf.dev/overview/

[24] Torsten Grust, Maurice van Keulen, and Jens Teubner. 2004. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.* 29 (2004), 91–131.

[25] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.

[26] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F Naughton. 2003. XML-to-SQL query translation literature: The state of the art and open problems. In *Database and XML Technologies: First International XML Database Symposium, XSym 2003, Berlin, Germany, September 8, 2003, Proceedings 1*. Springer, 1–18.

[27] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[28] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively Parallel NUMA-aware Hash Joins. In *IMDM at VLDB*. 1–12.

[29] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proc. ACM Manag. Data* 1, 2 (2023), 178:1–178:26.

[30] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (2023), 3044–3056.

[31] Zhenxiao Luo. 2017. *Engineering Data Analytics with Presto and Apache Parquet at Uber*. Retrieved 2024-05-27 from https://www.uber.com/en-DE/blog/presto/

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339.

[33] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.

[34] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[35] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.

[36] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *BTW (LNI, Vol. P-331)*. 235–258.

[37] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. 2002. XMark: A Benchmark for XML Data Management. In *VLDB*. Morgan Kaufmann, 974–985.

[38] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*. Morgan Kaufmann, 302–314.

[39] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. *VLDB* 14, 3 (2020), 445–457.

[40] Hongwei Sun, Shusheng Zhang, Jingtao Zhou, and Jing Wang. 2002. Constraints-Preserving Mapping Algorithm from XML-Schema to Relational Schema. In *EDCIS (Lecture Notes in Computer Science, Vol. 2480)*. Springer, 193–207.

[41] Raphael Taylor-Davies and Andrew Lamb. 2022. *Querying Parquet with Millisecond Latency*. Retrieved 2024-05-30 from https://www.influxdata.com/blog/querying-parquet-millisecond-latency/

[42] Trino Software Foundation. 2020. *Trino*. Retrieved 2024-05-19 from https://trino.io

[43] Ciprian-Octavian Truica, Elena Simona Apostol, Jérôme Darmont, and Torben Bach Pedersen. 2021. 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.

[44] Alexander Ulrich and Torsten Grust. 2015. The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *SIGMOD Conference*. ACM, 1421–1426.

[45] Zhiyi Wang and Shimin Chen. 2017. Exploiting Common Patterns for Tree-Structured Data. In *SIGMOD Conference*. 883–896.

[46] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161.

Received October 2024; revised January 2025; accepted February 2025