

JSON Tiles: Fast Analytics on Semi-Structured Data

Dominik Durner
Technische Universität München
dominik.durner@tum.de

Viktor Leis
Friedrich-Schiller-Universität Jena
viktor.leis@uni-jena.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Developers often prefer flexibility over upfront schema design, making semi-structured data formats such as JSON increasingly popular. Large amounts of JSON data are therefore stored and analyzed by relational database systems. In existing systems, however, JSON’s lack of a fixed schema results in slow analytics. In this paper, we present *JSON tiles*, which, without losing the flexibility of JSON, enables relational systems to perform analytics on JSON data at native speed. JSON tiles automatically detects the most important keys and extracts them transparently – often achieving scan performance similar to columnar storage. At the same time, JSON tiles is capable of handling heterogeneous and changing data. Furthermore, we automatically collect statistics that enable the query optimizer to find good execution plans. Our experimental evaluation compares against state-of-the-art systems and research proposals and shows that our approach is both robust and efficient.

CCS CONCEPTS

• **Information systems** → **Semi-structured data; Data layout; Online analytical processing engines; Database query processing.**

KEYWORDS

Semi-structured data; JSON; JSONB; Storage; Analytics; OLAP; Scan

ACM Reference Format:

Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD ’21)*, June 20–25, 2021, Virtual Event, China. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3448016.3452809>

1 INTRODUCTION

A plethora of data is created every day and forecasts show that data volume will rapidly increase in the next years [43]. Much of this data is semi-structured, i.e., it combines the data content and the schema. The most common semi-structured format today is the JavaScript Object Notation (JSON), a human-readable plain text storage format that allows representing arbitrarily-complex hierarchies. Large JSON data sets are, for example, accumulated when logging software system events or collecting data through public web APIs, such as the JSON APIs of Facebook [24], Twitter [60], and Yelp [64]. Public JSON data sets are also used to enrich proprietary

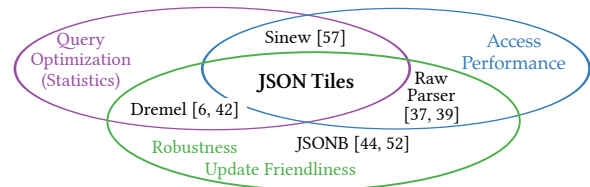


Figure 1: Classification of existing work.

data that is stored in relational systems. Analytics on large JSON data is valuable but expensive. Specialized tools for log file analysis, such as Splunk [55] exist, but lack the flexibility and functionality of general-purpose data management systems.

To speed up analytical processing of JSON data, a number of approaches have been proposed. Figure 1 classifies them with respect to access performance, robustness to heterogeneous data, and query optimization. SIMD-JSON [37] and Mison [39] allow parsing JSON with up to one GB/s per core. However, querying documents remains expensive because access to a single field requires a full parse over the data. Relational database systems store each JSON object as a string or use a per-object binary representation [52]. Both approaches are inefficient for analytical queries in comparison with relational column stores. Sinew [57] therefore extracts complete columns to speed up accesses. However, it can only produce good columnar extracts if the data mostly consists of the same static document structure. Sinew does not handle changing or heterogeneous data well and updates are expensive because new document structures change the global frequency of common keys. Reassembling shredded documents with different structures at a record level, as performed with Dremel [42] and implemented in Apache Parquet [6], results in additional work during query execution: many different optional fields have to be handled while evaluating the access automata. Processing Parquet files is CPU-bound even for purely relational files without optional fields [14].

This paper presents *JSON tiles*, a collection of algorithms and techniques that enable high-performance analytics on JSON data. The key idea behind JSON tiles is to automatically detect the implicit common structure across a collection of objects. Using this structural information, we infer types, materialize frequently occurring keys as relational columns, and collect query optimizer statistics – enabling performance close to that of native relational column stores. Infrequent keys and heterogeneous (outlier) objects are stored in an optimized binary format that allows fast access to individual keys. All these techniques are automatic and transparent, enabling fast analytics on JSON data without sacrificing the flexibility of the format.

We integrated JSON tiles into our RDBMS Umbra, which provides SQL, columnar storage, a fast query engine, and a cost-based query optimizer [34, 47]. Using JSON tiles, we leverage these mature technologies, which have been developed in a relational setting, for analytics on JSON data. This paper describes the deep integration

necessary and may therefore serve as a blueprint showing how to extend existing systems with high-performance JSON support.

2 DESIGN OVERVIEW

In principle, JSON objects can have very complex structures and each object can have a different implicit schema. However, in practice, JSON data is often machine-generated and has a fairly rigid and predictable structure. The key idea of our approach is to detect and leverage this implicit structure to speed up query processing.

2.1 Challenges

We first define three design goals before outlining how JSON tiles achieves these goals.

Access Performance: Accessing attributes of JSON documents requires document traversal. This traversal introduces a large overhead as every tuple requires a new lookup and all values are untyped. Accesses of relational columns, on the other hand, are cheap – in particular in column stores. This creates a big performance gap between JSON and relational attribute accesses. JSON tiles gains insights during data loading such that data can be stored in a columnar representation. This enables fast scans of JSON data.

Query Optimization: Traditional RDBMS collect statistics (such as histograms and distinct counts) on each column. As each JSON document is stored as one opaque tuple, the statistics are created based on full (textual) JSON representation. For example, this would likely result in the number of distinct values corresponding to the table’s cardinality. However, scan and join conditions usually access individual keys, and such statistics do not help in estimating selectivities.

For meaningful statistics, each document must be traversed and statistics on individual keys must be gathered. For example, join ordering uses distinct values of attributes to estimate the join cardinality. Without individual statistics, the optimizer relies on imprecise estimates. Thus, the query plan can be very inefficient [38] (e.g., because a bad join order is selected).

JSON tiles exploits the structural information gathered during loading to maintain data statistics. As the number of keys is unbounded, JSON tiles stores statistics on the frequent keys for precise estimates. This enables complex multi-table queries without having to manually transform the data to a relational schema first.

Robustness on Heterogeneous Data: The convenience of putting arbitrary documents into the database is often the primary reason for choosing semi-structured formats. Although the structure of objects is not arbitrary in practice, many data sets contain heterogeneous document types. Consequently, the storage engine needs to adapt to heterogeneous documents, changes of fields, and previously unseen data. For example, documents tend to grow over time as more and more fields are added to the original document type. Another important use case is the combination of log data from multiple sources. It is infeasible to define a global schema upfront for analytics on combined log data. As the analytics on JSON data was expensive in a general DBMS, log data is often analyzed by specialized providers such as Splunk [55].

JSON tiles handles different document types and copes with outliers through local computations. Further reordering helps in randomized insertions of heterogeneous documents.

2.2 Leveraging Implicit Document Structure

We explain the key ideas of JSON tiles using a running example that consists of real-world JSON documents from Twitter’s public API. Figure 2 shows a simplified example of 8 JSON documents representing information about tweets. Every document consists of an identifier, the tweet text, a create field, and a user object. As is common in many real-world data sets, the attributes of tweets changed over time. For example, Twitter introduced the famous hashtags after user feedback in 2007 and further attributes like reply (2007), retweet (2009), geo-tags (2010) were added over time [61].

Observations: As the example illustrates, the JSON documents in a collection often have the same set of keys and, therefore, have a similar implicit schema. Furthermore, the values for a key have matching types as well. In the example, the *identifier* attribute stores integers and the tweets (not shown in the example) would be textual strings. Another interesting type can be derived from *create* key. Although it is represented as a string because JSON does not specify a date data type, a query will most likely use it as date object. These observations lead to the insight that real-world semi-structured databases often effectively contain relational information.

Consequently, using the key structure and observed values, we can materialize the common structures as typed relational columns. However, detecting a single global relational schema, as proposed by Sinew [57], may be problematic. Simply materializing all keys as columns may lead to many null entries. Using some frequency cutoff, e.g., only extracting a particular column if at least 80% of all documents have that key, may prevent relevant columns from being extracted. In our example, the *replies* and *geodata* cannot be extracted by a global detection algorithm that extracts all keys that are represented by more than $\frac{2}{3}$ of all documents.

Our approach therefore breaks the input documents into multiple chunks – which we call *JSON tiles*. We search for local sub-structures within the smaller chunks to find more common patterns. We also automatically infer the data types and assume that values that look like a certain type will most likely be used as such. The small granularity of JSON tiles also enables parallelizing bulk loading as tiles can be constructed largely independently.

Outlier Handling: As JSON tiles collects document structures locally, it is likely that fewer document structures are observed in comparison to a global collection of structures. This already reduces the number of potentially materialized but unused columns, and thereby the number of null entries from absent fields. Because tiles are restricted in the number of tuples, a higher percentage of potential outliers, such as the missing geo-info, can be accepted. Hence, JSON tiles does not miss frequent keys and is able to adapt to changes of data objects and arrays, which results in a more robust system. New keys are added to the materialized parts, whereas removed keys are not extracted in future tiles.

Column Extraction: Because data is materialized into a columnar format, no semi-structured access computations are necessary. The cost of accessing a column chunk is amortized by the number of tuples scanned. Therefore, our approach achieves high analytical columnar scan performance while being robust to heterogeneous data objects or combined log data documents from different sources. In the Twitter example, our approach is able to extract replies and geodata into column chunks of the second tile.

"id": Int	"create": Date	"text": Text	"user_id": Int	"replies": Int	"geo_lat": Float
1	3/06	a	1	1	
2	3/07	b	3	3	
3	6/07	c	5	5	
4	1/08	a	1	1	
5	1/10	b	7	3	1.9
6	1/11	c	1	2	3.5
7	1/12	d	3	0	2.7
8	1/13	x	3	1	3.5

Tile #1: below threshold; accesses use binary JSON data

Tile #2: ✗

Figure 2: Twitter Tweet JSON data extracted into two JSON tiles.

Statistics: As we collect the smaller JSON tiles, we trace whether keys have been materialized and compute query optimizer statistics. We store the statistics information in each JSON tile. The local statistics are then propagated to generate table statistics such that they give details about the input data. This information is used to find efficient query plans that minimize intermediate join results.

3 EXTRACTION

This section presents the fundamental ideas behind JSON tiles and the algorithms for constructing them.

3.1 JSON Tiles

Previous work by Tahara et al. [57] observed that documents in real-world data sets often have similar structure and they therefore propose extracting a single schema globally. However, such a global approach is not robust with respect to heterogeneous or changing data. Depending on the chosen extraction threshold, many keys will either fall below the threshold or the resulting relation will have many attributes with mostly null values. In both cases, performance will not be optimal for heterogeneous data sets.

We therefore propose to detect the implicit document structure at a fine granularity (hundreds or thousands of documents rather than globally). We split the input data into disjunct *JSON tiles*, for each of which we detect a local schema. This approach naturally exploits the spatial locality contained in many data sets and finds a sweet spot between fast scan performance and the reduction of uncommon patterns. Our experiments show that a tile size of 2^{10} - 2^{12} tuples works well across many workloads.

In the following, we show the extraction steps for JSON tiles. Tile #2 of Figure 2 acts as our running example. The tile size of the tweet data is 4 tuples and we use an extraction threshold of 60%.

(1) *Collect all key paths for each tuple:* A key path is the path of nested objects and arrays followed to the actual key-value pair. For an easier notation, we will use only the first letter of each key and encode the nesting with ‘.’. For instance, the tuple with id 5 has the key paths $\{i, c, t, u_i, r, g_l\}$. Tuples 7 and 8 have the same key paths, whereas tuple 6 lacks g_l .

(2) *Use the collected key paths as input for frequent itemset mining:* An itemset is frequent if it exceeds the extraction threshold. The extraction threshold is the frequency count, which counts how many tuples contain this itemset, divided by the overall number of tuples. The itemset miner finds subsets within the collected key paths that are frequent. In our example, the miner finds two frequent maximum subsets and their frequency: $(\{i, c, t, u_i, r\}, 4)$ and

```
{
  "id":1, "date": "1/11", type: "story", "score": 3, "desc": 2, "title": "...", "url": "..."}
{"id":2, "date": "1/12", type: "poll", "score": 5, "desc": 2, "title": "..."}
{"id":3, "date": "1/13", type: "pollop", "score": 6, "poll": 2, "title": "..."}
{"id":4, "date": "1/14", type: "story", "score": 1, "desc": 1, "title": "...", "url": "..."}
{"id":5, "date": "1/15", type: "comment", "parent": 4, "text": "..."}
{"id":6, "date": "1/16", type: "comment", "parent": 1, "text": "..."}
{"id":7, "date": "1/17", type: "pollop", "score": 3, "poll": 2, "title": "..."}
{"id":8, "date": "1/18", type: "comment", "parent": 1, "text": "..."}

```

Figure 3: News items [28] with different document types.

$(\{i, c, t, u_i, r, g_l\}, 3)$. They are maximum itemsets as each further subset of $(\{i, c, t, u_i, r\}, 4)$ has the same frequency. All details and constraints of the itemset mining algorithm are explained in Section 3.3.

(3) *Extract the union of the maximum itemsets:* JSON tiles iterates over the found itemsets and extracts the key paths as materialized relational columns. All key paths are materialized from the first maximum subset. As the first and second maximum subset overlap, only g_l is additionally materialized. This results in the final extraction of $\{i, c, t, u_i, r, g_l\}$ for Tile #2.

3.2 Tile Partitions and Tuple Reordering

A new tile is created whenever the number of newly-inserted tuples reaches the tile size. Consequently, the content of JSON tiles depends on the insertion order. For many applications, the insertion order already provides strong spatial locality and therefore high-quality JSON tiles. For instance, adding fields over time, as in the Twitter example, results in almost perfect tiles. However, workloads like the one shown in Figure 3, where each document is of a different type, have little spatial locality. Even fine-granular tiles would result in poor scan performance. In the following, we describe an approach that solves this issue by reordering tuples between neighboring tiles.

The goal of the reordering algorithm is to find frequent itemsets across multiple tiles. The tuples are then reordered such that the same frequent itemsets are clustered in a single tile. The neighboring tiles grouped together for reordering are denoted as a *partition*.

Reordering is illustrated in Figure 4, which uses a tile size of 5 tuples and shows 12 tiles that are split into partitions of size 4. Each tile mines frequent itemsets with a reduced threshold. In the example, every patterned rectangle represents a tuple and the pattern denotes the frequent itemset that describes the tuple best. If we assume that all of the different patterns have no key paths in common, no materialization would be possible without reordering. JSON tiles clusters tuples into the tiles such that every itemset cluster satisfies the original threshold. The tuples are then distributed accordingly.

Once the tile redistribution has been performed, most tiles are perfectly extractable in the example. Each tile has a frequent structure that is over the extraction threshold. However, some tiles contain tuples that cannot be materialized. In contrast, before reordering none of the patterns exceeded the threshold in any tile. Our experiments on multiple workloads show that a partition size of 8 tiles yields good results.

The full algorithm for reordering proceeds as follows:

(1) The frequent itemsets of each individual JSON tile are mined. As these itemsets are used for reordering, the threshold for being frequent is reduced to $\frac{\text{threshold}}{\text{partition size}}$.

(2) The itemsets of all tiles within one partition are exchanged. Itemsets with a frequency of more than $\text{threshold} * \text{tile size}$ survive.

(3) Every tuple in the partition is matched to the frequent itemset that describes it best. The algorithm picks the largest itemset that has the most items in common with the tuple. As the itemset mining needs to be limited (see Section 3.3), ties need to be resolved such that every tuple that encounters this tie will match the same itemset. For example, our JSON tiles implementation resolves ties by minimizing the sum of item ids for equal matches.

(4) While matching the tuples, a hash table aggregates the count of the itemsets for both the individual tiles and the partition. As each tuple is only matched to one itemset, the tuples are simply put into individual tiles such that the original extraction threshold is reached (if possible). This mapping is computed in a greedy fashion.

(5) With the current count of tuples matching the itemsets in the tile as well as those required to satisfy the mapping, the swap positions between tiles are computed. The algorithm iterates over all tuples. If the tuple is needed in the current tile, no swapping is performed. Otherwise, the tuple is swapped with another tuple that matches the need for this tile. For example, a tuple is of itemset type green and in tile 1. Further, tile 1 needs to be filled with type purple and tile 2 with type green. First, tile 2 will be searched for a matching tuple of type purple as this would benefit both tiles. If tile 2 does not have any tuple of type purple, which is directly visible from the aggregate map, the remaining tiles are searched.

(6) The last step simply computes the itemsets with the original threshold of the reordered tiles to find the final extraction columns. Even if tuples belong to different itemsets, they can share key paths.

As is indicated by Figure 4, the tile partitioning parallelizes well on larger data sets. Each thread is dedicated to a disjoint subset of the data (partition). No interaction is needed as the information is disjoint between different threads. During tile creation, no issues for concurrent scans arise, as the tile is visible to scanners only once it is fully created. Only if tuples are currently being swapped, concurrent readers need to block until the swapping is finished.

3.3 Frequent Itemset Mining

JSON tiles uses frequent structures to materialize columns and redistribute tuples between them. These structures are found by gathering information on all available key paths. The frequent itemset miner determines which items are common and therefore materialized. The knowledge of itemsets helps to find the best frequent representation so that similar tuples can be redistributed. Furthermore, reordering within a tile improves compression in systems that support run-length encoding.

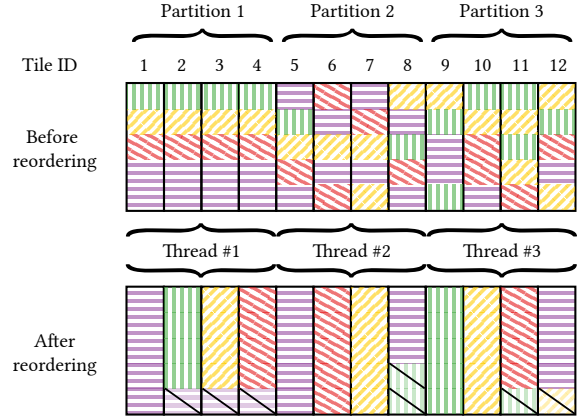


Figure 4: Reordering with partition size 4. Each tile has 5 tuples (vertical) and the extraction threshold is 60%.

To compute frequent itemsets, we rely on an efficient implementation of the *FPGrowth* algorithm [29]. In comparison to the classic *Apriori* [1] variant, *FPGrowth* does not need to generate candidate sets. *FPGrowth* creates a tree of frequent items and recursively iterates over the tree to generate output sets. We collect all keys from the documents and store them dictionary encoded. Dictionaries are created for every JSON tile and are used as the database to mine.

Unfortunately, the complexity of the result is a major problem of itemset mining. Since in the worst case the number of frequent itemsets is equal to the cardinality of the powerset of frequent items, we need to restrict the number of computed itemsets. Otherwise, itemset mining would be prohibitively expensive for tile creation.

As we only want to gracefully decrease precision, the algorithm computes itemsets until a budget is reached. Smaller itemsets are computed first as these are needed for larger ones. All frequent items are used to find potential itemsets that can be used for extraction. However, the number of elements (k) in the potential sets needs to be restricted. We denote a budget u as the upper bound of itemsets.

$$\sum_{i=1}^k \binom{n}{i} \leq u' \leq 2^n - 1, \text{ with } u' \leq u \quad (1)$$

We choose all 1 to k subsets of an n -ary set, resulting in the summation of the binomials. We compute k such that the number of generated subsets is limited to u' , which is always smaller than 2^n and u . Because k is dependent on the depth of the recursive mining of conditional pattern trees generated by *FPGrowth*, we bound the operations. As the recursion depth is restricted, the system is not overloaded during JSON tile materialization.

3.4 Value Types and Key Paths

In JSON, multiple values for the same key do not necessarily have the same primitive JSON type, e.g., some values are integer and some are float. If we decide to extract that key, we have to decide which data type to assign to the extracted column. At the same time, it must be ensured that the original type information is not lost and that JSON semantics is maintained.

To solve this problem, the tile extraction algorithm combines the key path with the primitive JSON type, i.e., each itemset entry is actually a pair and two key paths only match if their value types match as well. This way, if several options are available, extraction

will chose the most common type. Assume, for example, that the same key path contains integers as well as floats, and that the integers are extracted. This means that the float values cannot be stored in an extracted form and have to be stored in the binary JSON representation (cf. Section 5). On access, for example when summing up all values, we therefore traverse the binary representation when the extracted column value is null. This approach maintains JSON semantics for outliers, while providing fast scan performance for the majority of values.

3.5 Nested Objects and Arrays in JSON Tiles

A major feature of JSON is its capability to nest objects arbitrarily. Our extraction algorithm handles nesting by encoding it into the key path. During extraction, JSON tiles thereby do not have to distinguish between nested and non-nested objects. During the key path retrieval, the nesting level is computed as well as the followed keys. In the Twitter example (Figure 2), the nested key *lat* is extracted and encoded with its nested path (*geo*→*lat*).

Accessing nested column extracts require some care. For example, the access to 'key' → 'nestedKey' could first extract the object key and then use a regular JSON lookup or access *nestedKey* directly if available. Usually, a direct access to *nestedKey* is preferred. However, the database needs to know whether an access to key is needed as other expressions could use key as well.

To overcome this issue, JSON tiles recognizes during the scan operation whether the other levels of the key path are needed. We count how often each key is used as multiple expressions are able to share the same paths. If the path is used exclusively by one expression and the *nestedKey* is materialized, the intermediate access is removed. The final lookup is a simple access of this extracted key.

Another interesting challenge arises from heavily nested arrays. If the number of elements in an array is similar in all documents of a tile, JSON tiles is able to materialize all frequent elements. However, if the number varies, JSON tiles materializes only the leading elements that are frequent across all documents. For example, if every document contains an array with *x* elements but some documents have *x + c* array elements, only the first *x* elements are extracted.

This issue can be addressed by combining our approach with prior work. Deutsch et al. [19] distinguish between high-cardinality arrays and small-set arrays. The issue described only arises with high-cardinality arrays that contain many nested objects and differ significantly in the element count. Related work suggests extracting high-cardinality arrays into separate tables. The details on the orthogonal problem of detecting high-cardinality arrays are discussed in [19, 54]. After these arrays are determined, our JSON tiles extraction algorithm is used to automatically materialize additional tables from the detected arrays. In Section 6.3, we evaluate a combined approach in the presence of high-cardinality arrays.

4 INTEGRATION

JSON tiles touches many components of the DBMS. This section explains the adaptations that are necessary for a seamless integration.

4.1 Accessing JSON Attributes

In relational database systems, JSON data is usually stored in a single column of a table. Each value of this kind of JSON column

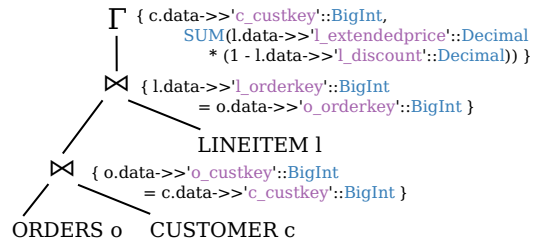


Figure 5: Join tree of simplified TPC-H query 10 before access expression push down.

holds a full JSON document. They are stored as JSON strings, which is a verified human-readable textual representation. JSON columns do not have any additional information on the structure of the contained documents. Some systems use a per-document optimized binary JSON format. This improves access performance by storing data in a binary representation that has minimal parsing overhead.

The most basic operation on JSON data is attribute access. In the examples throughout this paper, we use PostgreSQL-style access operators: the access as JSON type (→) and the access as Text (→>) expressions evaluate JSON queries [51]. These expressions are needed since the information is stored in nested objects and arrays. They return the value to the key (object) or slot (array).

For example, { "id": 0, "name": "JSON" } is a JSON object that holds two keys with one integer and one string value. Assuming that the user wants to access the id field, it can be requested as a value of type JSON with object-→ 'id' resulting in { 0 }. Note that the result type is not the integer itself. The access as JSON object function is necessary to access nested objects since access expressions can only be evaluated on JSON documents. The other option is to access the element as text with object-→> 'id', which returns the Text "0". Because the user usually intends to access the pure integer value, a cast from the string representation is needed. The expression is therefore rewritten to object-→> 'id' :: Int, which finally outputs the Integer 0. Umbra follows the PostgreSQL semantics of returning null if the requested key or any parent key is not present.

4.2 Push Down of Access Expressions

To utilize the scan performance of JSON tiles, changes to the query plan are necessary. The scan operator needs information on the keys that are accessed to decide whether extracts of tiles can be used. Previous work showed that the push down of accesses into the scan operator is crucial to heterogeneous data formats [33]. In the following, we describe the push down of JSON accesses and explain the steps to integrate JSON tiles into the query plan.

Figure 5 shows the query plan of a simplified version of TPC-H query 10 that uses JSON. In this example, the data is stored in a single JSON column (data). Each row is transformed into a JSON document such that every column name works as the key in the JSON objects. Because the operators above the scan need the JSON string for expression evaluation, each table scan operator has to produce the whole JSON string. Using the whole string when only parts are needed is inefficient.

As JSON tiles relies on the usage of extracted columns, the table scan operator needs to know which parts of the JSON data are accessed. If access expressions are evaluated further up the query plan, the table scan needs to provide the raw JSON data and cannot

utilize the materialized columns of JSON tiles. Thus, the access expression evaluation has to be pushed down into the scan operator.

We use placeholders for expressions and hand the computation of the access expressions over to the table scan operator. The result of an expression is then available at this location and directly usable by a parent operator. If a column extract of an expression is available, the data is read from the extract and the placeholder simply points to the materialized data. Otherwise, the raw JSON value is accessed.

4.3 Cast Rewriting

Because the return type of JSON accesses is Text, it is important to also push down the cast type information to the scan operator. Otherwise, the materialized types need to be transformed to Text first and later have to be transformed back to the cast requested type. This introduces a large query runtime overhead.

During the optimization phase of the RDBMS, cast rewriting reduces this overhead. The RDBMS checks whether the input expression of the cast is an access as Text lookup. If so, the cast result type determines which specialized access expression is used.

The RDBMS implements optimized access expressions for every data type that is defined for JSON documents (Section 5.1). Since these types are also used for the JSON tiles extraction, it is beneficial to rewrite these expensive casts. If the original type matches the cast result, we simply delete the cast operator and return the evaluated access expression directly. Otherwise, we reduce cast overhead as a better cast option is chosen. For example, a `BigInt` stored element key with the lookup `x->>'key': :Float` is rewritten to a `BigInt` lookup followed by a cheap cast to `Float`.

4.4 Storing the JSON Tiles Header

Because JSON tiles detects frequent document structures from the input seen locally, the extracted columns vary between different tiles. Thus, each tile needs its own header describing its seen and materialized data. For accessing materialized data, JSON tiles needs to store the extracted key paths and the corresponding value types. Since tiles vary in data and size, they are not directly stored in the fixed-sized part of the relation but in the variable-sized data. Only the pointer to the header is stored in the relation to map from tuples to the corresponding tile. Offsets into the variable-size data remain static as we either append the memory region or fill empty spaces.

In addition to the key path information and the type, the original JSON column is needed as the relation could contain multiple JSON columns. Moreover, JSON tiles stores the information on whether the key path is used with another type and whether null values are possible. The type information is necessary for correctness since the same key can have different values across the database. This is particularly interesting for JSON tiles, as null entries are often avoided due to a fine-grained JSON tiles size. For further optimizations, shown in Sections 4.6 and 4.8, the key paths that are not extracted are stored as well. Because the number of keys may be large, we store the key paths in a bloom filter [35].

4.5 Access Expression Evaluation

Information about the availability of an extracted column is only known during the table scan of each JSON tile. Because JSON tiles only materializes the frequent structures, not all keys are stored

as columns. Therefore, the access on the raw JSON data must be performed if no extract is found.

Accesses on JSON tiles use the information stored in the header of each tile to find the correct position of the requested data. The key path is stored as a string with information on the nesting depth (the number of nested levels) and the size of the string. We compute the matching of key paths in linear time, as the number of different key paths is limited within a single tile. Since it is expensive to calculate the availability of materialized columns per tuple, the calculations is performed once per tile. It is cached and reused for all the following tuples of the same tile.

If a materialized column is available, the header of JSON tiles is used to compute the access information. The matching column start position is computed by the position of the tile data and the offset into the matching column. The type information of the column is used to load the data and determine the best cast options. In Section 4.3, we show the rewriting of the cast expression that is used by both JSON tiles and our binary representation (Section 5.4).

To find the correct materialized column, our algorithm uses the key path and the requested types as inputs. If the types do not match, we test whether the types are both numerical values or the request type is a cast to Text. The former type suggests that it is easy to cast between the extracted and desired value. The only exceptions are values of type Date or Time. These are not allowed to be transformed to a textual representation. This restriction is explained further in Section 4.9.

4.6 Optimizer Integration

The query optimizer relies on cardinalities and selectivities to find a suitable query plan. In particular, join ordering relies on statistics to minimize intermediate join results. Without any tile statistics, the content of the JSON column is completely opaque to the database. Consequently, the optimizer has no information on how often a key path exists in a document and on the possible values. This can result in poor query plans and slow queries – in particular for complex, multi-table queries.

When constructing JSON tiles, we gather additional information for each tile. However, for join ordering the information needs to be available for the complete table. Thus, the information of the individual tiles is aggregated to leverage the data insights during query optimization. The additional tile information is used for optimizations as discussed in Section 4.8.

In the following, we describe the steps necessary to provide per-column statistics and estimators for JSON tiles. We use a fixed number of frequency counters and HyperLogLog [25] sketches for the extracted paths. The frequency counters are used to argue about the cardinality of the keys in the data. If, for example, a query requests `replies is not null` from the tweet data of Figure 2, only 5 out of 8 tuples match. Our JSON tiles implementation collects HyperLogLog sketches as these are the primary source of domain statistics in Umbra. The collection of regular histograms would work analogously. We suggest 64 sketches and 256 frequency counters as an upper bound on the statistics to restrict the maximum amount of memory used for query optimization.

During frequent itemset mining (Section 3.3), the frequency of all key paths within a tile is computed. The frequency of the key paths is used as the starting point for itemset mining. Each entry of the

database consists of the key and the number of occurrences in the tile. As described in Section 4.4, this database is also stored in the tile header. We update the relation-wide frequency counters (256 slots) if the key exists or replace empty slots as long as available. If all slots are utilized, we start replacing slots according to the most recent tile number that last updated that slot and the frequency count of the keys. Hence, new values can overwrite existing ones, however, the most frequent ones are always stored in the statistics.

To retrieve the cardinality of the keys, we simply use the frequency counters. If the key is not present in the frequency counters, we leverage the smallest available counter for this access. We argue that the missing counter will behave most similarly to the key with the minimal frequency of all retrieved counters. Although the smallest retrieved frequency is still an approximation, the results are significantly more accurate than using the global count of tuples.

Similar to the frequency counters, we collect statistics about the domain of the associated value of key paths with HyperLogLog sketches. When a tile is created, the inserted values are directly sampled. Hence, JSON tiles creates sketches without noticeable overhead. To aggregate the sketches at the relation level, we use the same replacement strategy as described with frequency counters. Note that HyperLogLog sketches are easy to combine.

During query optimization, the filter predicates on materialized JSON tiles leverage the distinct counts of the HyperLogLog sketches. With this information, better join orders are possible as the resulting join cardinality estimation is improved. Furthermore, different documents are sampled statically at query plan generation to find more accurate estimations. This improves the sketch estimates and creates new estimates if no HyperLogLog sketch is available.

4.7 Updates

As JSON tiles creates columnar chunks, we can simply update the values of the keys that were changed. As variable-length data is tracked in a separate memory region with offsets, value updates can be computed in place. If the new document does not contain some of the extracted keys, null values in the respective columns indicate the absence of these keys. Note that the tile header needs to add all new access paths to the bloom filter. Otherwise, queries that scan the data could incorrectly skip the changed tiles.

Tiles need to be recomputed only if many outlier documents are introduced. An outlier document is defined as one that does not overlap with the existing extracted keys. As the recomputation of the materialized JSON tiles are costly, the computation should only be triggered after the majority of the tuples do not match the current extracted JSON tiles schema.

4.8 Skip Tiles Without Matches

Because JSON tiles collects tuples locally, some tiles do not contain certain key paths. If an expression is searching for such a path, skipping these tiles seems valuable.

The simple skipping of tiles that do not provide a key path, similar to the previous work on efficient column stores [36, 53], leads to incorrect results. Accessing a value from a key that was not found returns a null value. Skipping null values results in incorrect results, for example, some aggregates count null values.

To overcome this issue, our system tracks the optimization path of skipping null values and whether null is evaluated as false. These

two properties can change at an operator and expression level. For example, an inner-join on top of the access expression has the property that null values are skipped as the join condition is evaluated as false. Another example for skipping null values are comparisons, e.g., the expression where $x \gg \text{'key'} : \text{Float} > 1$.

Thus, if the expression is not found and null values are skipped or evaluated as false, the whole JSON tile has no valuable information. These tiles are then skipped to improve performance.

4.9 Date and Time Extraction

Because the exact representation of values need to be restored during accesses, the extraction of Dates and Times from strings is complex. As many different formats exist, it is hard to guarantee the recreation of the original string. We use a hybrid method to store Dates and Times in which the access type is leveraged. If the database user casts the value into Date or Time, JSON tiles does not need to recreate the exact string representation. As a result, any correct internal representation can be used to satisfy the request. Therefore, JSON tiles extracts possible Date and Time values because these are probably accessed as such.

To find columns that store Dates and Times, we first sample on the potential column. If the string-encoded values match a Date or Time type, we extract these values encoded as SQL Timestamp. When the user casts the access to any Date- or Time-like type, the extracted Timestamp value is used to cast to the defined type. Otherwise, the string representation of the binary JSON is returned retaining the input format.

5 BINARY JSON FORMAT

JSON tiles extracts the frequent key paths of documents; however, some data sets contain outliers and infrequent keys that are not materialized. This section presents a new optimized binary format that allows fast access to individual keys of such infrequent objects. An optimized format is necessary as JSON is a human-readable data format. Each access results in an expensive parsing of the raw string. The goals for the binary format are fast lookups in objects and arrays, typed values, and few cache misses. The format must further conform to RFC 8259 [13], which defines the general JSON representation and needed value types.

Several binary JSON formats were developed to efficiently transfer data [26, 32, 49]. These formats, however, are not optimized for fast accesses and focus on (de-) serialization support. DBMS that use custom binary formats include PostgreSQL and MongoDB [45, 52]. Although the latter formats are better suited for query processing than exchange formats, they do not combine a logarithmic worst-case runtime for lookups with continuous memory accesses.

Our JSONB format is optimized to provide $O(\log(n))$ accesses to the correct key in objects and $O(1)$ accesses to array elements. Moreover, objects and arrays are forward iterateable such that all key-value pairs – even nested objects – can be accessed continuously without memory address jumps. This results in fewer cache misses for nested accesses. The physical types used in our binary JSON representation match the RFC requirement and are also used by JSON tiles as mentioned in Section 3.3. Hence, the cast rewriting presented in Section 4.3 is a universal access optimization.

Due to restructuring, some of the unimportant properties of the original document get lost, such as whitespace information or the order of keys in the input. We argue, aligned with the guidelines of using binary JSON in PostgreSQL, that the gains in query performance outweigh the ability to recreate the exact syntax of the input [52]. Apart from the syntactic restriction, our JSONB document is round-trip safe. Thus, all other properties and the exact value representation can be reconstructed.

5.1 JSONB Storage Format

To provide interoperability and correctness, our binary JSON format conforms to RFC 8259 [13]. It defines objects, arrays, numbers, strings, and three literals. Our binary format uses the following data types to represent the type definitions. Each type has an 8-bit header with the type identifier and additional information.

Numeric Integers use the SQL type `BigInt`. We store small values (< 23) in the integer header, otherwise, we calculate the number of bytes needed to represent the integer and store the amount in the header. The size-optimal integer follows the header.

Numeric Floats store the remaining numeric values, using the SQL Float datatype represented with IEEE 754 double precision. RFC 8259 relies on double-precision floats for the remaining numerics as they are “generally available”, “widely used”, and “good interoperability can be achieved” [13]. We further optimize for smaller precision levels (half-floats, and single-precision floats) if the conversion from double-precision floats is lossless.

Literals use a special header representing the value.

Objects contain all key-value pairs including nested objects and arrays. Objects need an object header, followed by an integer representing the number of elements in that object. The integer uses the minimum number of bytes as defined for Numeric Integers. This is followed by an offset into the object for every element. Each offset points to the end of the corresponding element. The offset key follows the payload in every element slot. Note that nested objects and arrays are also stored in the payload such that we can iterate over the object without memory address jumps. Keys are further sorted to accelerate lookups and guarantee $O(\log(n))$ accesses since we can use binary search to find the correct entry. The representation is illustrated in Figure 6.

Arrays are stored similar to objects but do not use keys.

Strings are stored with the possibility to use unicode characters. The exact representation matches the RFC 8259 definition.

5.2 Detection of Numerics in Strings

As RFC 8259 does not specify any precision for JSON numbers, strings are usually used to preserve the exact representation. For example, monetary values should not be represented as floating-point values. As a result, a decimal-valued price is usually stored as string. We auto-detect numeric values hidden in strings and extend our binary JSON format with an additional numeric string type.

During the transformation, we check whether the complete string, except the start and end quotes, can be represented as an SQL Numeric. We first test whether the input string is a valid numerical value (digits, point, etc.). If so, round-trip safety is guaranteed since the exact representation can always be reconstructed with the help of scale and precision. Because the original input must be a string,

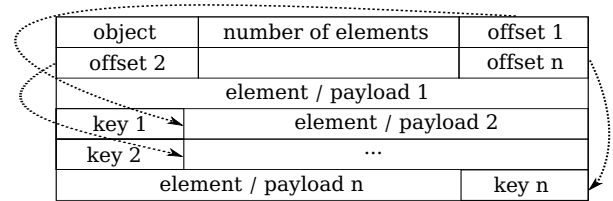


Figure 6: JSONB representation of an object.

the start and end quotes are simply added to the Numeric output. The key motivation is that strings that are representable as Numeric will probably be used as numeric values. Query execution benefits by performing a smaller number of expensive casts from strings.

5.3 Two-Pass Transformation Algorithm

Our JSONB format stores nested objects within the parent object. This allows for continuous accesses without memory address jumps. Continuously stored data increases locality and reduces cache misses. However, this makes object creation harder as the object size depends on the size of its nested objects and arrays. For example, the object with a nesting level of 0 only knows its size after the sizes of the inner objects have been computed. The simple approach of on-the-fly resizing is not feasible as resizing is expensive and needs to be performed for every inner object. Our compressed storage algorithm for floats and integers even aggravates this issue.

To overcome the problem of resizing, we propose a two-pass algorithm. In the first iteration, we check for validation errors and calculate the required memory for every JSONB type. This is possible because we remember the computed nesting level and perform a depth-first calculation. Note that depth-first is the order as defined in the input of JSON documents. Nested objects are textually represented within the parent object. Hence, we can simply forward iterate over the input. In the second iteration, we use the information of the first pass to allocate the right amount of memory and transform the data without further checks. In total, we iterate twice over the input data. However, this is usually not a performance issue because most JSON objects fit into the CPU cache.

5.4 Accessing Elements

Since JSON documents consist of objects and arrays that contain the information, the user typically looks up only specific parts of them. Umbra uses the access as `JSONB ->` and the access as `Text ->>` expressions to lookup the values of objects and arrays.

The access expression is implemented in two phases. First, a lookup into the object or array is executed. Object keys can be accessed in $O(\log(n))$ since keys are sorted and binary search is used to perform the positioning. Because arrays are stored sequentially, we can access the element in $O(1)$. The second phase extracts the found value. The default extracted SQL types are `JSONB (->)` and `Text (->>)`. As a result, the storage of the right type would reduce the performance if the access needs to cast to `Text`.

The database user usually casts the access result to the desired type, e.g., `x->>'key'::Integer`. Our system analyzes the cast and, if possible, directly returns the correct result type instead of the string representation. Otherwise, it parses the value as `Text` and performs the cast afterwards.

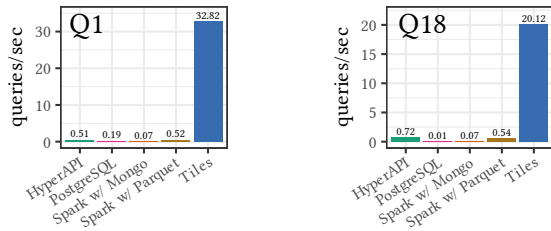


Figure 7: External competitors with all 32 threads.

Table 1: Execution times for all TPC-H queries in seconds.

	PG.	Spark		Hyper	Umbra			
		Mongo	Parquet		JSON	JSONB	Sinew	Tiles
1	5.276	14.297	1.939	1.950	1.725	0.178	0.122	0.030
2	> 100	23.383	2.735	1.370	1.608	0.584	0.637	0.035
3	17.905	15.892	1.288	0.560	0.675	0.280	0.259	0.030
4	3.013	10.439	1.755	0.539	0.692	0.227	0.228	0.026
5	87.468	22.659	2.072	> 100	1.340	0.372	0.326	0.045
6	1.259	14.896	0.690	0.244	0.254	0.119	0.085	0.010
7	> 100	21.035	2.554	3.111	1.177	0.429	0.351	0.103
8	> 100	26.608	1.814	1.156	1.469	0.474	0.416	0.062
9	> 100	23.688	3.939	1.728	2.576	0.395	0.370	0.153
10	> 100	21.967	2.003	0.984	1.362	0.388	0.294	0.067
11	> 100	23.444	0.809	0.829	1.070	0.344	0.353	0.068
12	1.493	18.783	1.316	0.419	0.450	0.286	0.289	0.061
13	5.570	10.597	2.146	0.683	0.665	0.149	0.291	0.044
14	1.502	9.552	0.734	0.343	0.392	0.171	0.142	0.017
15	9.105	19.024	1.306	0.339	0.399	0.211	0.185	0.018
16	4.220	15.119	2.693	0.898	0.629	0.201	0.273	0.048
17	> 100	16.379	1.381	0.605	0.567	0.173	0.091	0.026
18	86.167	14.861	1.849	1.388	0.949	0.260	0.179	0.050
19	1.290	33.885	0.970	0.363	1.834	0.213	0.170	0.057
20	> 100	20.234	1.613	0.787	0.974	0.355	0.348	0.042
21	12.372	39.236	3.517	1.415	1.787	0.615	0.479	0.103
22	2.060	11.306	3.135	0.529	0.566	0.172	0.180	0.016

6 EXPERIMENTAL EVALUATION

We integrated JSON tiles into our high-performance relational database system Umbra that supports SQL, columnar storage, and efficient memory management [23, 47]. We compare it with the following industrial-strength database systems: *PostgreSQL* (12.4) with its binary JSONB format, *Tableau Hyper* (0.11556) with its JSON format (no binary JSON available), *Apache Spark* (3.0) with *Apache Parquet* (Dremel), and *Apache Spark with MongoDB* (3.6). Because MongoDB does not support joins, Spark is used to schedule the queries. The mandatory sampling of the MongoDB data is not accounted.

Besides this system-wide comparison, we also integrated a number of prior JSON handling proposals into our system (sharing the optimizer and query engine): human-readable *JSON* format, our binary *JSONB* representation as described in Section 5, *Sinew*, which extracts the whole table with the original proposed 60% table-frequency using our JSONB format, and *JSON tiles* with JSONB. Unless otherwise noted, we use the tile size 2^{10} , partition size 8, and extraction threshold 60%.

All experiments were performed on an AMD Ryzen Threadripper 1950X (16 cores, 32 threads) with 64 GB of main memory. The system runs Ubuntu 20.04 and uses a Samsung 850 Pro SSD (2TB).

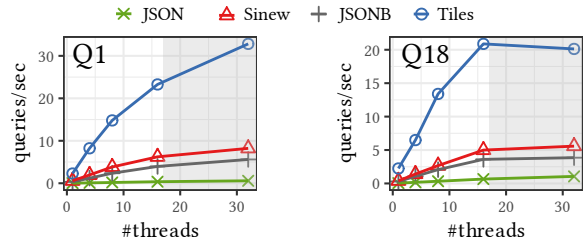


Figure 8: Scalability of internal competitors.

6.1 Combined TPC-H JSON

Our initial experiments are based on TPC-H. As this benchmark is based on a relational schema, we first explain the steps necessary to convert the data to JSON. Queries are modified similarly to the example query shown in Section 4.2. We modify TPC-H such that every row of each table is represented as a JSON object with the column names as the keys of the object. Thus, each JSON document contains the schema of the table and the values of one row. To simulate a combined log data workload with different JSON documents, we combine the different structures of these multiple relations into a single one. Although the documents are adapted in JSONized TPC-H, the queries return the same result as queries executed on the original TPC-H relations. Data loading is performed in parallel and uses all cores which leads to an imperfect insertion order.

In the following, we focus on chokepoints for TPC-H, which have been elaborated by previous work [11, 21]. Therefore, the results of the queries Q1 (expression calculation & aggregation), Q3 (join & aggregation), and Q18 (join) are shown in detail. The execution times of all TPC-H queries are shown in Table 1.

Query 1 only accesses items of the original lineitem table and performs low-cardinality aggregations with expensive expression calculations. As visually illustrated in Figure 7, our approach is an order of magnitude faster than Spark with Parquet and Hyper, which are both able to leverage a large fraction of the available cores. In comparison to other approaches within Umbra, visualized in Figure 8, we are able to speed up the computation by a factor of 3. As Query 1 relies on date expressions, our date and time optimization helps to significantly outperform Sinew. Umbra scales with a rising number of cores despite executing a single pipeline.

On the other hand, Query 18 joins multiple original tables with groups, and is therefore a chokepoint for join and high-cardinality aggregation performance. PostgreSQL uses a sub-optimal join order which results in very low performance. Although the lineitem columns accessed by the query are extracted with Sinew, the query performance is 4× slower than JSON tiles. This is a result of the missing information on cardinalities and the non-materialized tuples of customer and order data. Query 3 contains an expensive aggregation and performs joins. Our approach dominates all others since the optimal join order is computed and all lineitem fields are materialized.

6.2 Combined Yelp

To confirm the findings of the TPC-H benchmark, we test additional queries on the real-world Yelp data set (~9 GB) [64]. We define five queries on top of the data to gather interesting business insights [22]. Table 2 shows the results for all Yelp queries. For example, Yelp

Table 2: Execution times for all Yelp queries in seconds.

PG.	Spark		Hyper	Umbra				
	Mongo	Parquet		JSON	JSONB	Sinew	Tiles	
1	15.883	9.211	1.114	1.892	6.068	0.487	0.366	0.293
2	5.121	8.582	1.868	0.454	0.813	0.191	0.163	0.044
3	> 100	> 100	> 100	> 100	3.262	0.444	0.302	0.145
4	10.961	4.774	0.188	0.296	0.843	0.105	0.013	0.013
5	49.033	8.521	1.499	1.095	2.698	0.273	0.160	0.088

Table 3: Execution times for all Twitter queries in seconds.

	Spark		Hyper	Umbra				
	Mongo	Parquet		JSON	JSONB	Sinew	Tiles	Tiles-*
1	17.226	3.246	65.381	8.319	0.419	0.255	0.116	0.116
2	5.517	1.100	1.262	4.510	0.181	0.191	0.091	0.091
3	1.881	1.336	> 100	> 100	0.191	0.204	0.215	0.017
4	28.860	4.139	1.401	23.749	0.229	0.212	0.206	0.022
5	17.095	2.542	1.603	2.802	0.164	0.049	0.057	0.058

Query 4 counts the number of reviews in groups of stars. Because the number of reviews is large, Sinew also materializes all fields needed for this query. The performance of our approach and Sinew is very similar in this example, which results from the extraction of the star rating. Although this is one of the best cases for Sinew, our approach is able to slightly increase the performance, which highlights the small static overhead per JSON tile. JSON tiles has a higher throughput due to the skipping defined in Section 4.8.

6.3 Twitter

As we use Twitter as our running example, we benchmark multiple queries on an excerpt of tweets from June 1, 2020 (~31 GB) [22, 58]. Tiles-* combines JSON tiles with extracting high-cardinality arrays as discussed in Section 3.5. We extract high-cardinality arrays (hashtags, mentions) and store them in an additional JSON tiles relation. Queries join these relations with the original Twitter table.

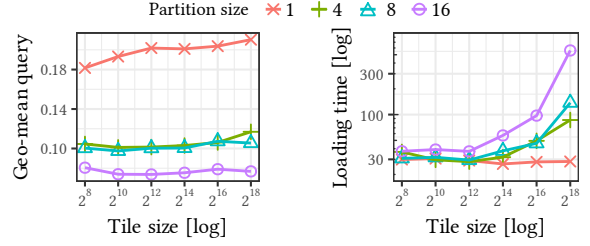
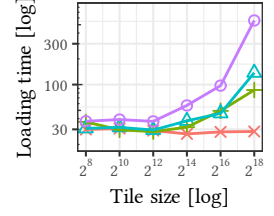
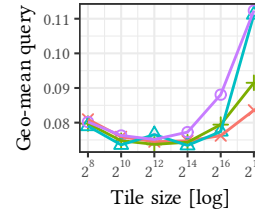
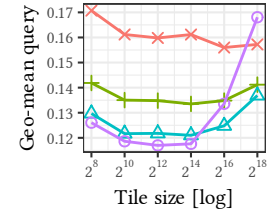
Query 1 selects the tweets of the most influential users of the day. Although the user object is mandatory in tweets and extracted by both Tiles and Sinew, we are able to outperform the competitors. The deleted tweets of each user are aggregated with query 2. Deletions use a different JSON structure that is not frequent globally. This structure is reordered and can be materialized in some tiles.

Query 3 selects tweets that mention @ladygaga (user_mentions array), and query 4 selects tweets that include the hashtag #COVID (hashtags array). As both rely on the extraction of high-cardinality arrays, only a subset of the items is materialized within JSON tiles. JSON Tiles-* outperforms all competitors by joining the matching high-cardinality arrays with the base Twitter data.

Table 4: Geo-mean of Twitter.

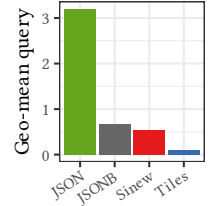
	JSON	JSONB	Sinew	Tiles	Tiles-*
Twitter	11.803	0.258	0.239	0.122	0.054
Changing	11.683	0.236	0.182	0.115	0.054

Table 4 shows the geo-mean runtime on a data set that changes its tweet structure as described in Section 2.2 [22, 61]. As the changes in the JSON structure reduce the number of matches, most systems have an improved geometric mean. JSON tiles can easily adopt to unseen access keys and does not introduce null values if an access path is absent.

**Figure 10: Geometric Mean of shuffled TPC-H.****Figure 11: Loading Time of shuffled TPC-H.****Figure 12: Yelp Geo-mean.****Figure 13: Tweet Geo-mean.**

6.4 Shuffled TPC-H

To demonstrate the robustness of our novel partitioning algorithm, we manually shuffled the TPC-H table before loading. Thus, during the insertion no local tuple patterns are retained. JSON tiles with a partitioning of 8 and a tile size of 2^{10} is able to reduce the query runtime significantly. Figure 9 shows the geometric mean of the shuffled TPC-H benchmark. The JSON string representation has poor performance due to the parsing needed for every document. Although JSONB and Sinew are able to significantly increase performance, JSON tiles can further improve on these results by a factor of 4x.

**Figure 9: Shuffled TPC-H Geo-mean.**

6.5 Tile and Partition Size

The choice of the tile and partition size has impact on the insertion time and materialization quality. The following experiments show how to find robust values for these settings.

Figures 10, 12, and 13 show the different choices and the resulting geometric mean for the respective workloads. The more partitions are enabled, the better the reordering. Even in naturally ordered data sets (e.g., Yelp and sequential TPC-H), the parallel insertion into our database (32 threads) creates outliers and imperfect data. Hence, the reordering is also beneficial there. Considering the insertion performance, Figure 11 highlights that a tile size of less than 2^{14} and a partition size of less than or equal to 8 do not introduce any overhead. Thus, we recommend tile size 2^{10} and partition size 8.

6.6 Optimizations for JSON Tiles

For the TPC-H and Yelp workloads, Figure 14 shows the impact of the optimizations discussed in Section 4.8 and 4.9. The skipping of tiles without matches is an optimization that helps to speed computations if the number of different JSON document types is higher.

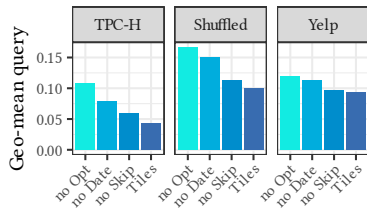


Figure 14: Geometric means of different optimization levels.

6.7 Micro Benchmark

The following micro benchmark demonstrates that our approach has only minimal static overhead for each JSON tile while gaining robustness. JSON tiles is able to achieve an order of magnitude improvements in comparison to only using binary JSON documents.

To explain the overhead behavior, we choose a query that is executed optimally by both the regular relational system and Sinew. The query simply sums up the linenumber field. Sinew extracts the column perfectly just like JSON tiles. The performance is shown in Figure 15, the corresponding low-level CPU performance counters are shown in Table 5. The benchmarks labeled with “Comb.” use the combined TPC-H, whereas the others use the original lineitem table. Note that the relational approach cannot use combined TPC-H.

First, the materialization of JSON tiles leads to significant improvements over using the raw or binary optimized JSON formats. The performance of both extraction algorithms is similar to a pure relational TPC-H workload if the original lineitem table is used. The imperfect combined data consists of outliers and different structures because of the parallel data loading. The performance is reduced for the extraction algorithms, however, it is still an order of magnitude faster than when only JSONB is used. The relational table needs 32 instructions per tuple, Sinew 65, and JSON tiles 70. As this is the perfect extraction workload for Sinew, it is expected that the increased robustness requires some additional computations.

Table 5: Low-level performance counters for the summation query on lineitem; normalized per tuple computed.

System	Cycles	Instr.	Branch-	L1-Miss	Sec/All
Relational	17.01	31.58	0.00	0.02	0.001613
Tiles	39.33	69.82	0.02	0.18	0.002494
Sinew	32.12	65.08	0.01	0.10	0.002050
Sinew Comb.	39.07	71.73	0.03	0.10	0.003450
Tiles Comb.	50.15	74.20	0.04	0.14	0.004462

6.8 Data Loading and Storage Consumption

As our approach preprocesses the data during insertion, we measure the time needed to load the data sets. Figure 17 shows the loading times of all systems. The fastest system for TPC-H and Yelp is Hyper, which just stores the raw JSON string in the database and uses almost-instant data file loading [46].

Focusing on the overhead of JSON tiles, only a small reduction compared to the raw JSON and binary JSON insertion times are noticeable. The performance drop by Sinew results from the single-threaded frequency algorithm and the materialization of the detected columns. For a fair comparison, Sinew eagerly extracts the data after the insertion.

Because many real-world workloads use queries that are constrained by date ranges, the extraction of date and time is beneficial. As Figure 14 shows, the optimizations improve the performance considerably.

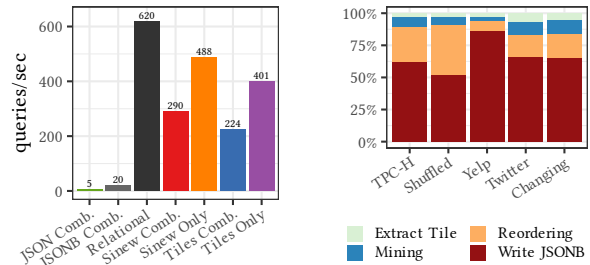


Figure 15: Throughput of the summation query. Figure 16: Insertion time breakdown (32 threads).

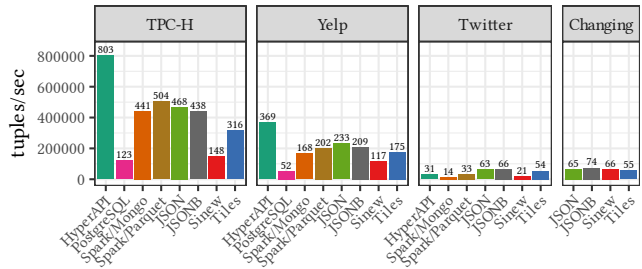


Figure 17: Parallel loading (numbers in 1000 tuples/sec).

Figure 16 breaks down the time needed for the different steps to create JSON tiles. Most of the insertion time is spent for storing the binary JSON data. Note that the expensive creation of the binary JSON is not measured as these steps are further up the pipeline. Although the JSON tiles operations require computation time, the overall loading times indicate that these computations do not change the insertion speed significantly. For example, the shuffled TPC-H spends a crucial amount of time on reordering, however, Figure 11 shows that this does not result in slower overall insertion times. Also, the insertion times do not change between partition sizes for small tile sizes.

Table 6: Size in MB (% of JSONB).

	JSON	JSONB	+Tiles	+LZ4-Tiles
TPC-H	3092	2766	665 (24%)	317 (11%)
Yelp	8657	7809	718 (9%)	237 (3%)
Twitter	31271	24106	706 (3%)	247 (1%)

We measured the size needed to store JSON tiles in Table 6 to analyze the storage requirements. In our current implementation,

JSON tiles are materialized in addition to the original JSONB data. All benefits of JSON tiles come with only a moderate size overhead. As TPC-H consists of few strings and many extractable columns, the overhead is the highest there. Only 3% overhead results in significantly improved performance for Twitter. Because the data of JSON tiles are stored in columnar format, we can achieve strong compression ratios. For example, LZ4 compression on JSON tiles can further reduce storage consumption by a factor of 2-3x.

6.9 JSON Binary Formats with Nesting

As some documents cannot be extracted, we rely on a high-performance binary JSON representation. We compare our binary format, referred to as JSONB below, to the BSON implementation from MongoDB’s open source C++ driver [45], and the JsonCons

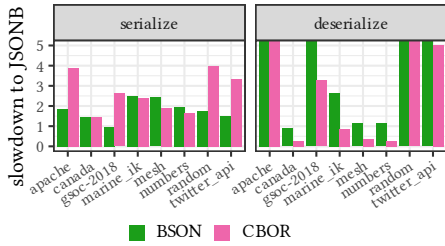


Figure 18: (De-) Serialization performance slowdown compared to our JSONB format.

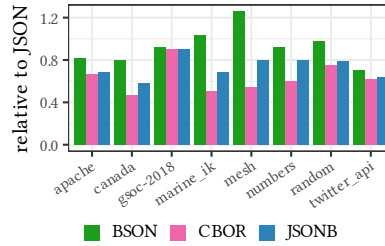


Figure 19: Storage size consumption compared to JSON string.

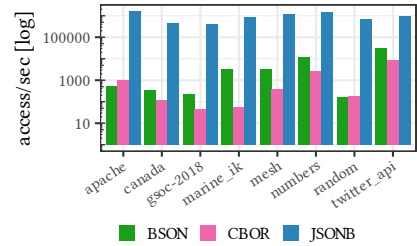


Figure 20: Performance of random accesses with different nesting levels.

C++ CBOR implementation [49]. To demonstrate a wide variety of complex and nested JSON documents, we use standardized JSON files from the SIMD-JSON repository [37].

First, we analyze the serialization and deserialization performance of the different JSON formats. Figure 18 shows the slowdown of the other two approaches compared to our JSONB implementation. JSONB is the fastest format in all serialization workloads and only three deserialization workloads are beneficial for CBOR.

The normalized costs for storing the JSON documents in a binary format are shown in Figure 19. CBOR’s space requirements are the lowest as this is used mostly to exchange messages. In comparison to MongoDB’s BSON, our representation uses less disk space.

Our binary representation has the best lookup performance, which is shown in Figure 20. Accessing keys within a document requires the object to be extracted in CBOR. This reduces the access performance significantly. Our $O(\log(n))$ object key lookup is superior to the linear-time algorithm of BSON. Thus, JSONB achieves large performance gains for random accesses.

7 RELATED WORK

Due to the increasing importance of semi-structured data, many systems have been developed to handle different data documents. In the following, we differentiate between database systems and raw data processing systems.

Database Systems with JSON Support: With the rising usage of JSON, relational database systems integrate storage solutions for these data formats. One common idea is to store and index the data such that consequent accesses can be evaluated efficiently [3, 15, 18, 33, 40, 41, 56, 62]. Sinew [57] extends PostgreSQL with the approach of extracting data from the whole table, which incurs robustness problems for changing or combined data. This reduces query performance as only a certain number of keys are extracted [51, 57]. Our system focuses on the efficient and robust storage of JSON data to satisfy multiple user queries thereafter.

Proteus [33] builds indexes on top of JSON data to speed up accesses. Recache accelerates processing of heterogeneous formats by caching accesses of the data according to the query workload [8]. Other systems, such as Apache Spark [7] or Hive [59], use different storage plugins for heterogeneous data. Apache Parquet [6] and Avro [5] are common formats for storing JSON data. Although these plugins are quite robust, e.g., record shredding of Dremel [42], the performance of Spark on combined data is severely reduced.

NoSQL systems such as MongoDB [44], Couchbase [17], and DocumentDB [4] store semi-structured documents directly. However,

their feature set for querying is limited and analytical (columnar) accesses are slow as these systems are optimized for point accesses.


Raw Data Processing: Another approach of accessing JSON files is to query raw files without explicitly loading them. After defining the queries and providing the raw files, the system should return the results without any loading delay [30]. Modern database systems try to saturate the wire speed to keep the loading gap small. Raw systems have reduced performance on multiple queries as the data is not stored as efficiently as possible [46].

Other approaches, e.g., NoDB [2], use in-situ raw accesses [10, 16, 50] to query the raw files directly. This requires the data to be parsed quickly. For both structured and semi-structured data, parsers such as FAD.js, Mison or SIMD-JSON use modern CPU properties for fast reads [12, 27, 37, 39]. Raw filters are used to speed up the parsing and reduce the amount of data ingested into the database [48, 63].

JSON Schema Retrieval: Inspired by the usage of *JSON Schema* [31], which is a work-in-progress description language for JSON, recent theoretical work [9, 20] has studied schema inference for JSON data. Although these approaches can describe the inherent JSON schema accurately, the computation of the schema file is expensive as all optional and required schema fields have to be enumerated. Different JSON documents in large-scale data sets can further decrease the performance, as the existence of many optional fields makes it harder to choose the right fields to extract.

8 CONCLUSION

We presented JSON tiles, a collection of algorithms and techniques for deeply integrating high-performance JSON support into relational database systems. High scan performance is achieved by extracting the frequent parts of the data into chunks of materialized JSON data. During the materialization we collect statistics about the data so that the query optimizer of the RDBMS is able to find good query plans. The materialized chunks are robust to heterogeneous data as we find globally and locally frequent structures. We further infer data types from the textual representation. If attributes cannot be extracted, we use an optimized binary format for JSON so that object lookups are in logarithmic time of the keys within an object. The experimental evaluation shows that our approach is an order of magnitude faster on imperfect and combined workloads, without adding any significant overhead to perfectly-structured data.

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

REFERENCES

- [1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*. 487–499.
- [2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*. 241–252.
- [3] Wail Y. Alkawaileet, Sattam Alsubaiee, and Michael J. Carey. 2020. An LSM-based Tuple Compaction Framework for Apache AsterixDB. *PVLDB* 13, 9 (2020), 1388–1400.
- [4] Amazon Web Services. 2020. Amazon DocumentDB. <https://aws.amazon.com/documentdb>. accessed: 2020-01-03.
- [5] Apache Software Foundation. 2020. Apache Avro. <https://avro.apache.org/>. accessed: 2020-01-04.
- [6] Apache Software Foundation. 2020. Apache Parquet. <https://parquet.apache.org/>. accessed: 2020-01-03.
- [7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.
- [8] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data. *PVLDB* 11, 3 (2017), 324–337.
- [9] Mohamed Amine Baazizi, Housseem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *EDBT*. 222–233.
- [10] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. 2014. Parallel data analysis directly on scientific file formats. In *SIGMOD*. 385–396.
- [11] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*. 61–76.
- [12] Daniele Bonetta and Matthias Brantner. 2017. FAD.js: Fast JSON Data Access Using JIT-based Speculative Optimizations. *PVLDB* 10, 12 (2017), 1778–1789.
- [13] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259.
- [14] Luca Canali. 2017. Performance Analysis of a CPU-Intensive Workload in Apache Spark. <https://externaltable.blogspot.com/2017/09/performance-analysis-of-cpu-intensive.html>. Results presented at Spark Summit.
- [15] Craig Chasseur, Yanan Li, and Jignesh M. Patel. 2013. Enabling JSON Document Stores in Relational Systems. In *WebDB*. 1–6.
- [16] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *SIGMOD*. 1287–1298.
- [17] Couchbase. 2019. Couchbase Under the Hood: An Architectural Overview. <https://resources.couchbase.com/c/server-arc-overview>.
- [18] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.
- [19] Alin Deutsch, Mary F. Fernández, and Dan Suciu. 1999. Storing Semistructured Data with STORED. In *SIGMOD*. 431–442.
- [20] Michael DiScala and Daniel J. Abadi. 2016. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *SIGMOD*. 295–310.
- [21] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *PVLDB* 13, 8 (2020), 1206–1220.
- [22] Dominik Durner. 2019. JSON queries. <https://github.com/durner/json-queries>.
- [23] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *DaMoN*. ACM, 21:1–21:3.
- [24] Facebook. 2020. Using the Graph API. <https://developers.facebook.com/docs/graph-api/using-graph-api>. accessed: 2020-01-04.
- [25] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. 137–156.
- [26] Sadayuki Furuhashi. 2020. MessagePack. <https://msgpack.org/>. accessed: 2020-11-07.
- [27] Chang Ge, Yanan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative Distributed CSV Data Parsing for Big Data Analytics. In *SIGMOD*. 883–899.
- [28] HackerNews. 2020. HackerNews Items API. <https://github.com/HackerNews/API/>. accessed: 2020-01-07.
- [29] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *SIGMOD*. 1–12.
- [30] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. 2011. Here are my Data Files. Here are my Queries. Where are my Results?. In *CIDR*. 57–68.
- [31] JSON Schema. 2020. Specification of the new draft. <https://json-schema.org/specification.html>. accessed: 2020-01-04.
- [32] Riyad Kalla. 2020. UBJSON. <https://ubjson.org/>. accessed: 2020-11-07.
- [33] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB* 9, 12 (2016), 972–983.
- [34] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbr. *The VLDB Journal* (2021).
- [35] Adam Kirsch and Michael Mitzenmacher. 2008. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures & Algorithms* 33, 2 (2008), 187–218.
- [36] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.
- [37] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.
- [38] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.
- [39] Yanan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *PVLDB* 10, 10 (2017), 1118–1129.
- [40] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -. In *CIDR*.
- [41] Zhen Hua Liu, Boda Christoph Hammerschmidt, and Doug McMahon. 2014. JSON data management: supporting schema-less development in RDBMS. In *SIGMOD*. 1247–1258.
- [42] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010), 330–339.
- [43] Tova Milo. 2019. Getting Rid of Data. <https://vldb2019.github.io/files/VLDB19-keynote-2-slides.pdf>. VLDB Keynote.
- [44] MongoDB, Inc. 2019. MongoDB Architecture Guide. <https://www.mongodb.com/collateral/mongodb-architecture-guide>.
- [45] MongoDB, Inc. 2020. Mongo CXX Driver. <https://github.com/mongodb/mongo-cxx-driver/tree/r3.5.1>.
- [46] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant Loading for Main Memory Databases. *PVLDB* 6, 14 (2013), 1702–1713.
- [47] Thomas Neumann and Michael Freitag. 2020. Umbr: A Disk-Based System with In-Memory Performance. In *CIDR*.
- [48] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *PVLDB* 11, 11 (2018), 1576–1589.
- [49] Daniel Parker. 2019. JsonCons. <https://github.com/danielaparker/jsoncons>.
- [50] Christina Pavlopoulou, E. Preston Carman Jr., Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. 2018. A Parallel and Scalable Processor for JSON Data. In *EDBT*. 576–587.
- [51] PostgreSQL Global Development Group. 2020. JSON Functions and Operators. <https://www.postgresql.org/docs/11/functions-json.html>. accessed: 2020-01-03.
- [52] PostgreSQL Global Development Group. 2020. JSON Types. <https://www.postgresql.org/docs/11/datatype-json.html>. accessed: 2020-01-03.
- [53] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091.
- [54] 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*. 302–314.
- [55] Splunk Inc. 2020. The Data-to-Everything Platform. <https://www.splunk.com/>. accessed: 2020-01-17.
- [56] Tableau. 2020. Tableau Hyper API. https://help.tableau.com/current/api/hyper_api. accessed: 2020-01-04.
- [57] Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. 2014. Sinew: a SQL system for multi-structured data. In *SIGMOD*. 815–826.
- [58] Archive Team. 2020. The Twitter Stream Grab - 2020.06. <https://archive.org/details/archiveteam-twitter-stream-2020-06>. accessed: 2020-10-12.
- [59] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.
- [60] Twitter. 2020. Developer Guide. <https://developer.twitter.com>. accessed: 2020-01-03.
- [61] Twitter. 2020. Tweet Timeline. <https://developer.twitter.com/en/docs/tweets/data-dictionary/guides/tweet-timeline>. accessed: 2020-01-07.

- [62] Zhiyi Wang, Dongyan Zhou, and Shimin Chen. 2017. STEED: An Analytical Database System for TrEE-structured Data. *PVLDB* 10, 12 (2017), 1897–1900.
- [63] Dong Xie, Badrish Chandramouli, Yinan Li, and Donald Kossmann. 2019. Fish-Store: Faster Ingestion with Subset Hashing. In *SIGMOD*. 1711–1728.
- [64] Yelp. 2019. Yelp Dataset Challenge. <https://www.yelp.com/dataset/challenge>. accessed: 2019-11-20.