Partition, Don't Sort! Compression Boosters for Cloud Data Ingestion Pipelines

Patrick Hansert
patrick.hansert@cs.rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

Sebastian Michel
sebastian.michel@cs.rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

ABSTRACT

Data Lakes deployed in the cloud are a go-to solution for enterprise data storage. While the pay-as-you-go cost model allows flexible resource allocation and billing, it mandates an efficient use of resources like CPU hours, network traffic, and used storage. The distributed nature of cloud environments necessitates partitioning the data and processing these partitions separately. In this work, we put forward a practical solution to improve the efficiency of compression algorithms on Dremel-encoded data by clustering similarly structured nested data at ingestion time, such that compressible partitions can be created. We propose a clustering approach inspired by decision trees that outpaces even the naive partition-then-sort approach by up to factor 17.44 while also boosting the compression by up to factor 2. We further show that when sorting the individual buckets, a compression boost that is competitive with the well-established increasing-cardinality heuristic can be achieved, but at a lower ingestion time.

PVLDB Reference Format:
Patrick Hansert and Sebastian Michel. Partition, Don't Sort! Compression Boosters for Cloud Data Ingestion Pipelines. PVLDB, 17(11): 3456 - 3469, 2024.
doi:10.14778/3681954.3682013

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbislab/Partition-Dont-Sort.

1 INTRODUCTION

Data Lakes [7, 27] in the cloud have established themselves as the go-to solution for enterprise data storage. Data is read from distributed filesystems, processed in the cloud environment, and subsequently written back to cloud storage. Consequently, the development of efficient methods for both writing to and retrieving data from such storage systems has become increasingly crucial. In the pay-as-you-go model of cloud providers, billing is based on a diverse set of resources used, ranging from CPU hours to speed and usage of network storage, to in- and outbound network traffic costs. In general for cloud computing, but in particular, for disaggregated compute environments, data compression can directly reduce two prominent cost factors, data storage and retrieval bandwidth. For a database system, it is possible to freely choose how records are materialized regarding the order of rows and columns. This has been leveraged for decades to enhance compression [50]. The most prominent example is Run-Length Encoding (RLE) [53] where grouping identical values reduces the number of runs that need to be stored [39]. The same principles apply to other compression approaches as well [13, 40]. Unfortunately, the problem of determining the optimal row order is NP-complete [39].

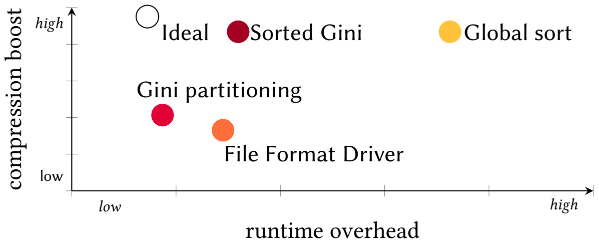


Figure 1: The Time vs. Efficiency Tradeoff

Traditionally, research on row reordering focuses on relational data. However, many datasets instead use nested formats like JSON due to their flexibility. In practice, columnar representations are often preferable since they are beneficial to analytical workloads. To transform nested data into a columnar representation in a lossless manner, Dremel encoding [43, 44] is employed. Its most widely used implementations are Apache Parquet [3] and ORC [32]. Columnar formats align well with compression techniques [1, 42]. For example, Parquet utilizes various methods including an RLE variant, dictionary encoding, and Snappy compression [25]. Notably, the columns encoding structural information are typically of low cardinality [64], highly correlated [29], and encoded using RLE. This renders them ideal candidates for enhancing compression efficiency.

In Big Data environments, such as Data Lakes, data is typically processed by distributed systems. That necessitates partitioning the data between multiple worker nodes. The key motivating point of this paper is that the potential of compression depends vastly on the way partitions are formed. The fastest conceivable partitioning strategy is to just focus on even sizes, e.g., using round-robin partitioning. However, it has no potential to form buckets that would be beneficial for RLE. It can even be actively harmful by distributing formerly long runs between multiple partitions. A drastically different approach is to globally sort data in a preliminary step, as suggested by Lemire et al. [40]. In contrast to round-robin partitioning, RLE inside the individual partitions is very effective.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682013

Unfortunately, it requires three passes over the dataset: one to determine the sort order, one to determine the ranges for partitioning, and a final one to perform the actual sorting. The aforementioned approaches can be seen as extreme points in the tradeoff between fast ingestion speeds required for archiving high-velocity data and high compression ratios beneficial to long-time storage. This tradeoff is visualized in Figure 1.

1.1 Problem Statement and Approach Sketch

We propose a clustering approach to partition data based on structural similarity. Given an input dataset D of structurally heterogeneous JSON documents, we strive to find a complete partitioning of D into **disjoint buckets** $b_1, b_2, \ldots$ such that

(1) the **storage requirement** of the individually encoded and compressed buckets is minimized,
(2) the **ingestion time**, i.e., the total time to decide on the partitioning and to perform partitioning and compression is minimized

As discussed above around the interpretation of Figure 1, these aims are connected and conflicting. To balance this tradeoff, we suggest a decision-tree-inspired clustering approach that groups structurally similar records from D based on the lowest expected number of run boundaries, estimated using Gini Impurity. This alone already reduces the overall size. Optionally, we apply the increasing-cardinality heuristic to order bucket contents after partitioning, but any alternative heuristic could be employed here.

Our approach also provides a query-agnostic way to improve data skipping if we posit that similarly structured records tend to be queried together. This assumption can be substantiated by considering that a selection predicate can typically only be satisfied if the value of the column in question is present. While specialized algorithms exist that provide more skipping opportunities for a known query workload, that information may not be available apriori in a data lake setting.

In the remaining paper, we use the terms bucket and partition interchangeably to refer to a batch of data deemed promising to store together in one Parquet file. This may differ from more specific concepts in frameworks such as Spark [4], Iceberg [5], or Delta [7].

1.2 Strengths and Limitations

Our approach is meant to work with structurally heterogeneous nested data that is stored in a schema-on-read manner. It capitalizes on how dremel-encoded formats like Parquet or ORC store the structure and exploits the fact that there typically are much fewer structural variants than records in a dataset. This makes our approach especially well-suited for use with data lakes.

However, it also makes the approach not applicable to data that has already been transformed, e.g., into a relational form for databases or a star schema for data warehousing. Such normalization steps eliminate the structure our approach works with. By extension, homogeneously structured data is also not a good candidate for our approach, even if it is nested.

Compression Boosters in general are computationally expensive and hence usually not suitable for frequently changing data. Instead, they are ideal for use cases like data lakes, where the dataset is stored for a longer time for later analysis. Our approach reduces the ingestion runtime overhead and thereby pays off earlier.

In general, there is also a tradeoff between optimizing the partitioning for compression efficiency or data skipping. These skipping-optimized partitions may once again split long runs and thereby sacrifice compression efficiency. However, this relationship is not as straightforward: On the one hand, our approach already provides query-agnostic data skipping means based on the structure. Additionally, better compression also improves query latency by reducing the transfer times between the distributed storage and the worker nodes. On the other hand, rows that qualify for the same set of queries must also have common columns and hence be similarly structured to some extent. Hence, a more thorough investigation of the interplay between these aspects is warranted. In practice, there may be a hybrid approach that optimizes the combined cost. However, such an approach is beyond the scope for this paper.

1.3 Contributions and Outline

For the above reasons, we investigate partitioning-based heuristics as compression boosters for cloud data ingestion pipelines. In our previous work [28], we have provided a preliminary investigation into the applicability of partitioning as a compression booster. In this paper, we go beyond that and focus on ingestion speed and correlated data. We contribute:

- A novel clustering heuristic for boosting compression of nested data based on its structure
- Based on it, a fast data ingestion pipeline
- An extensive evaluation of the involved parameters and tradeoffs on two real-world datasets and TPC-DS.

The remainder of this paper is structured as follows: In Section 2, we go over the necessary background for this work before reviewing preexisting research in Section 3. We give a birds-eye overview of our approach in Section 4 before we give more details about the statistical model we employ in Section 5 and the data structure we employ to manage it in Section 6. Section 7 gives details about the partitioning process. An analysis of our algorithm's runtime complexity is given in Section 8. We then evaluate our approach in Section 9 and draw our conclusion in Section 10.

2 PRELIMINARIES

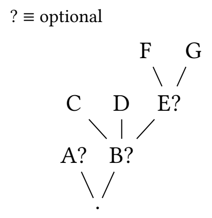
Run-Length Encoding. Run-length encoding (RLE) is a widely-known lightweight lossless compression scheme [53]. It works by storing the number of consecutive occurrences of an item rather than each item individually. Therefore, it is susceptible to changes in efficiency based on the order of the data, as detailed in Section 3.

For example, consider the integer sequence $(0, 0, 0, 0, 2, 0, 0, 0, 2, 2)$. Storing it as plain 4-byte integers takes 40 B. With RLE, we instead store a sequence of (value, frequency) pairs that report how often a value is repeated in the input. The values represented by one such tuple are called a *run*. In the example, the sequence becomes $((0, 4), (2, 1), (0, 3), (2, 2))$. As a result, the space consumption of our example is reduced to 32 B, i.e., a compression ratio of 1.25.

A higher compression is achievable if the order of the items can be altered. If we sort the example sequence before compressing it, we can encode it as $((0, 7), (2, 3))$, i.e., using only 16 B with a compression ratio of 2.5. Thus, we have boosted the compression

```
{"B":{"C":5,"D":2}}
{"B":{"C":3,"D":7,"E":{"F":5,"G":2}}}
{"A":6}
{"B":{"C":8,"D":4}}
{"A":7}
```

(a) Example JSON Data



(b) Schema

A	B.C	B.D	B.E.F	B.E.G
0	1	1	1	1
0	1	1	2	2
1	0	0	0	0
0	1	1	1	1
1	0	0	0	0

(c) Definition Level Columns

Figure 2: Example for Dremel encoding

ratio by a factor of 2. If the sequence is a column in a database or a Dremel-encoded file, this becomes more complicated as we need to find an order that globally minimizes the size of all columns, a problem that is NP-complete [39].

Dremel Encoding. Originally introduced for Google's Dremel system [43, 44], this encoding provides a structure-preserving transformation to store nested data, such as JSON or Google's Protocol Buffers, in a columnar format. Since columnar formats are especially well suited for OLAP tasks and compression [1, 42], it has found widespread use in open source file formats such as Apache Parquet [3] and ORC [32], as well as many data analytics and processing tools such as Apache Spark [4].

Dremel-encoding works based on a tree-shaped schema where each node is marked as required, optional, or repeated. For simplicity, we omit the types of the leaf nodes. The schema for our running example can be seen in Figure 2b. At the top level, it shows two optional nodes A and B. While A is a leaf, there are three further nodes nested below B, namely C, D, and E. The latter is optional as well and has further nodes nested below it. The example schema does not have any repeated nodes. To transform records, such as the examples in Figure 2a, into columns, Dremel encdoing proceeds as follows for every root-to-leaf path p in the schema [43]:

Firstly, create one column to keep the actual values. Additionally, create one **definition level** column which stores the number of optional nodes in p that are present for the given record. Figure 2c shows these definition level columns of our example documents. Consider the path B.E.G in the example schema. It has two optional nodes, namely B and E. The first JSON document contains B but not E. Hence, the definition level for that path is 1. In the second document, both of these nodes are present and we end up with the maximum definition level of 2. Finally, neither of the nodes is present in the third document which leads to a definition level of 0. Repeated nodes are encoded using an additional *repetition level* column. However, since the order of items in an array may carry meaning, we cannot reorder their contents. Therefore, the specifics of the repetition level column are irrelevant to this paper, and we defer to the original article for an explanation.

In Apache Parquet [3], definition level columns are stored using RLE. Since they are also typically correlated depending on the data's schema [29], they are a good candidate for reordering techniques. For example, by just considering the definition level of B.C, we can improve the compression rate of all children of B. Since nested schemas are not very deep in practice [64], focusing on these columns first agrees with typical reordering techniques such as the increasing-cardinality heuristic.

Decision Trees and Hunt's Algorithm. Decision trees [48, 62] are a well-established and explainable model for data classification. Each inner node of a decision tree holds a split condition (e.g., salary ≤ 10 000) and data records are assigned to the leaves based on whether they fulfill these conditions. A key problem when constructing decision trees is to decide which splits to do, for which several so-called impurity measures exist. Intuitively, impurity is high if the data records that would be assigned to a child node after split belong to many classes and low if they belong to one or very few classes. Popular choices include the Shannon entropy and the Gini Impurity, the latter of which is defined as follows:

$$Gini_C(L) = 1 - \sum_{c \in C} P(c \mid L)^2 \quad (1)$$

Here, $P(c \mid L)$ is the probability of observing class c on a randomly chosen data record in set L. Since a split creates two or more child nodes, the quality of the entire split (all child nodes) is computed based on the so-called gain, which is essentially the difference in impurity of the parent node and the weighted sum of impurity values of the child nodes. The decision tree induction algorithm can then greedily choose the split with the highest gain.

3 RELATED WORK

Compression Boosters. The order in which rows and columns of database tables are stored can be altered without affecting the results of a query. This observation has been harnessed for decades to improve the efficiency of compression algorithms like RLE.

The problem has been proven NP-complete [39]; thus, in practice, heuristics are used. Pinar et al. [51] suggest using gray code orders to reorder the tuples, which they show to be optimal for bitmap indexes if all possible value combinations are present. Another common approach is the *increasing-cardinality heuristic*, which sorts the tuples lexicographically in the increasing order of their number of distinct values. Lemire and Kaser [39] show it to be 3-optimal on uniform distributions. Improvements over it include a tie-breaker based on column skewness [52], adapted traveling-salesperson heuristics [40], and soft-computing approaches [35]. Vo and Vo [63] have suggested sorting columns individually using their correlation to another column.

We have suggested a solely schema-based approach [29], and have preliminarily investigated partitioning rather than sorting as the means of rearranging runs [28]. Other partitioning-based approaches include vertical partitioning [12, 13], Bimax-Clustering [59], and disabeling RLE for columns far back in the sort-order [58].

Melnik et al. [44] have integrated an unpublished algorithm into the file format driver of their Dremel-encoded file format. Due to its placement in the file format driver, it can only affect the order after partitioning and thereby loses valuable optimization opportunities compared to our approach. While we cannot directly compare against their implementation, we will compare against an approach that partitions evenly and then locally employs the increasing-cardinality heuristic.

Space Filling Curves. Space-filling curves map d-dimensional space into one dimension. The most well-known such curves are the Hilbert [31], and Morton [46] curves. Aside from their long-established use for multi-dimensional indexes [55], they have also been applied as a reordering heuristic, especially in conjunction with differential encoding. The areas include medical images [41], point clouds [14], and data warehouses [18]. In general, Hilbert curves are preferred due to their superior locality properties [34]. For our use-case of reducing RLE runs, Lemire and Kaser [39] have shown that Hilbert curves are not competitive.

Ingestion Pipelines. Several pipelines have been suggested which aim to optimize how data is stored. Many, like the one introduced by Sun et al. [61], aim to reduce the query runtime by allowing the query processor to aggressively skip buckets that do not contain relevant data. This is especially interesting in disaggregated settings where the buckets must be transferred from the data node to the processing node. Data skipping techniques have been applied to and specialized for many settings, including non-rectangular partitions [36], correlated columns [60], joins [16], faster JSON querying [17], multidimensional layouts [15], and HTAP systems [2]. Amoeba [57] finds a good initial partitioning for data skipping without knowing a query workload and then specializes it in a cracking-like [33] manner. For querying arbitrarily partitioned Data Lake tables, Weintraub et al. [66] propose an approach that improves latency by calculating tight covering sets of files using an index. To allow effective data skipping for known query workloads, approaches like Iceberg [5] and the so-called liquid clustering in Delta Lake [7] allow the organization of data according to attribute values, e.g., used as folder names in HDFS. In contrast, our approach does not make any workload assumptions.

Orthogonal to data skipping, some authors also focus on the read and write costs. Bian et al. [10, 11] optimize the physical layout of wide tables in columnar files to reduce the read costs. Delta Lake [7] balances performance for incremental inserts by writing small files which are later compacted into larger files. Mukherjee et al. [47] minimize the monetary costs when placing data in a tiered architecture for given latency bounds.

Some systems have also been developed which consider the integration of reordering heuristics. As the simplest solution, Melnik et al. [44] suggest including a reordering heuristic in the file format driver. It has been suggested to create one partition per pattern [65], or to employ Locality Sensitive Hashing [23] to group similar records [20, 21]. Both approaches are designed for a single machine and would cause load-balancing issues in a distributed setting. Apaydin et al. [6] employ an extensible hashing scheme on the record's gray code order rank. In our distributed setting, the central entity governing such a scheme could quickly become the bottleneck. Lemire et al. [40] have argued that sorting and then partitioning the data is a good way to employ more computationally intensive heuristics. However, in our use case, sorting itself is the bottleneck, so this advice does not solve the underlying issue. Our solution to all these problems is to instead employ a statistical model to determine a good partitioning upfront. Then, we can ingest the data in a bulk-synchronous manner.

JSON Schema Extraction. Schema extraction strives to determine a human-understandable schema from a given dataset. Often, similar patterns as we use to boost the compression are analyzed. The granularity in which they do so varies. Some approaches treat all variants as distinct [8], maintain one version per object [54], or augment the schema with additional information such as occurrence counts [9], distinct value counts [45], or structural outliers [38]. Frozza et al. [19] present an approach that pre-groups the discovered schemas similar to the fingerprints we employ. More advanced techniques identify tagged unions [37], or disambiguate structs, maps, and lists [59]. Gallinucci et al. [22] employ a decision-tree-based approach to cluster schemas under two variants of entropy. Unlike us, they prioritize understandability, not compression.

4 APPROACH

Our main goal is to provide a fast partitioning heuristic that groups structurally similar rows together. This way, long runs remain in the same bucket and are still available to exploit for boosting compression. Such a grouping alone will already boost the compression because it decreases the entropy within each bucket. This enables compression algorithms to achieve a higher compression ratio. Afterward, the partitions can be reordered locally.

Clustering comes up as a natural solution for our goals. Since we need a means of identifying which row could be in which bucket for data skipping, the interpretability of the clusters matters. Therefore, we employ a variation of Hunt's algorithm. Further details on it can be found in Section 7. It recursively splits the source dataset based on candidate splits that relate to the dataset's properties. The best split is chosen based on the lowest information content, which we derive using the Gini Impurity as discussed further in Section 5. We chose this measure because it corresponds to the expected run count under run-length encoding. Finally, the induced tree can be used to partition the source data and later on for data skipping.

Given the potential size and width of datasets, it becomes infeasible to consider all possible values in all columns for splitting, especially if we were to rescan the source dataset for each candidate. Instead, we follow the suggestion of the increasing-cardinality heuristic and focus on splits for low-cardinality columns. In the case of Dremel-encoded data, the definition level columns are among the lowest cardinality columns because most real-world datasets are not deeply nested [64]. Therefore, we focus on these and hence on the structure of the data rather than its contents. To manage their correlation without rescanning, we devise a novel data structure coined **Fingerprint Set**, introduced in detail in Section 6.

As we will see in the experiments, our approach without sorting beats the partition-then-sort approach by a large margin in both compression and runtime. If we add sorting to our approach as a stand-in for a more elaborate heuristic, we obtain competitive compression ratios with a decreased ingestion time. Hence, the approach we will present in the following sections alleviates the tradeoff and can be an integral part of cloud data ingestion pipelines.

We implement our approach in Apache Spark [4] by gathering metadata, computing the partitioning scheme locally on the controller node, and writing the partitioned data using their DataFrame API. Our approach works in four steps.

(i) Generate the fingerprint set from the source data using the DataFrame API (Section 6.4) and transfer it to the controller node. Optionally, sampling can be employed to decrease the statistics gathering time.
(ii) Locally induce the decision tree (i.e., Hunt's algorithm).
(iii) Optionally, determine a sort order.
(iv) Use the resulting partitioning tree and order to materialize the dataset to be ingested.

For the final step, we have implemented a UDF that maps documents to their buckets' IDs. This allows us to read the input data using the DataFrame API and then distribute it to the worker nodes using hash partitioning by the bucket ID. Optionally, the workers sort the partitions to increase the compression ratio. Afterward, they materialize the buckets in the Parquet format.

5 MODELING RUNS UNDER PARTITIONING

In order to find an efficient partitioning scheme that minimizes the number of runs generated, we need to find a statistical model that accurately captures the dataset structure. To this end, we focus on the expected number of run *boundaries*, i.e., runs starting in a row. We derive this quantity for individual columns and then generalize it to entire datasets. Finally, we discuss handling correlation between columns by maintaining a collection of fingerprints per document in Section 5.1. Assuming independence between rows, we calculate the occurrence probability of a boundary based on the distribution of the values within a column and naturally obtain its **Gini Impurity**.

To see this, let V_C be the set of distinct values of a column C and $P(v \mid C)$ the probability that value $v \in V_C$ occurs in a row of C. We want to determine the expected number of boundaries $B(C)$ in rows of C. Let now $v_1, v_2 \in V_C$ be values of C in *consecutive* rows. A new run starts whenever $v_1 \neq v_2$. Hence, $B(C)$ can be characterized as $E(\mathbb{1}_{v_1 \neq v_2} \mid C)$. Under the assumption that data values inside a column are independent and with $\mathbb{1}_{v_1 \neq v_2} = 1 - \mathbb{1}_{v_1 = v_2}$, we obtain

$$E(\mathbb{1}_{v_1 \neq v_2} \mid C) = 1 - \sum_{v \in V_C} P(v_1 = v \mid C) \cdot P(v_2 = v \mid C) \qquad (2)$$

Since we assume column values are identically distributed, both these probabilities are equal to $P(v \mid C)$, and we finally obtain

$$B(C) = 1 - \sum_{v \in V_C} P(v \mid C)^2 \qquad (3)$$

Note that Equation 3 is identical to Equation 1 if we equate the distinct values of column C with the classes in a decision tree. Intuitively, this makes sense: In a decision tree, we try to separate records from different classes from each other. To optimize compression, we want to separate different values from each other to reduce the number of runs. This similarity inspires our approach: We induce a partitioning tree by selecting the split with the highest gain at each level.

Now, consider not only one column but n columns $C_1, \ldots, C_n$. As before, the goal is to quantify the expected number of boundaries. For a row, the total number of started runs is the sum of runs started in each column. While these are not necessarily independent, the linearity of the expected value also holds for dependent values, which leads to

Table 1: Transforming Fingerprints into a Fingerprint Set

(a) Fingerprints

A	B	E
0	1	0
0	1	1
1	0	0
0	1	0
1	0	0

$\Longrightarrow$

(b) Fingerprint Set

	A	B	E	#
	0	1	0	2
	0	1	1	1
	1	0	0	2
$\sum$	2	3	1	5

$$B(C_1, \ldots, C_n) = \sum_{i=1}^{n} B(C_i) = \sum_{i=1}^{n} Gini_{V_{C_i}}(C_i) \qquad (4)$$

We will use this equation as our **impurity measure**. Unfortunately, it does not directly yield a way to obtain the best split.

5.1 Handling Correlation Using Fingerprints

In practice, the assumption of independent columns is too strong, especially for definition level columns, as they are inherently dependent between siblings in the schema [29]. Thus, assuming independence can lead to splits that cause empty buckets. For instance, assume we have already split the example dataset from Figure 2a based on the presence of A and now consider splitting one of the resulting buckets further by path B.C. The presence of these two paths is not independent. A is absent whenever B.C is present and vice versa. Hence, the split on B.C would result in an empty bucket. The model cannot detect this if it assumes independence. This is particularly problematic if such a situation happens early (i.e., close to the root of the tree) as the model will suggest further splits of the already empty bucket, creating many more empty buckets.

To avoid such infeasible splits, we maintain a collection of fingerprints of the documents in the dataset. A *fingerprint* is a tuple that contains one Boolean value for each optional node in the schema. That value tells us whether the respective node is present in the document. For example, Table 1a shows the fingerprints for the documents in Figure 2a. In the first document, the node B is present while A and E are absent. Hence, its fingerprint is $[0, 1, 0]$ as shown in the first row of the table. The fingerprints for the other documents are constructed accordingly.

A collection of fingerprints contains all information necessary to calculate the number of boundaries and can be partitioned to obtain the fingerprints after a split. At the same time, it cannot lead to errors such as discussed above: We know a node can be both present and absent if the set contains at least one fingerprints for both variants. If a fingerprint is missing from the collection, e.g., due to sampling, it will introduce errors in the estimated sizes and amounts. At worst, a possible split will not be considered, but an impossible split will never be used.

6 THE FINGERPRINT SET

We now introduce the *Fingerprint Set*, the data structure we use to efficiently calculate the Gini Impurity after split operations. It maintains a collection of unique fingerprints and how often they

```scala
1 final case class FingerprintSet(
2     names: IndexedSeq[String],
3     fingerprints: Seq[(IndexedSeq[Boolean], Long)],
4     leaves: IndexedSeq[String],
5     leafMetadata : IndexedSeq[Option[ColumnMetadata]],
6     freqs: IndexedSeq[Long],
7     total: Long
8 ) // ...
```

Listing 1: Fingerprint Set Data Structure in Scala

are observed, as well as metadata to identify individual columns. The complete data structure can be found in Listing 1.

A fingerprint is an array with a Boolean value for each optional node in a document, which denotes whether it is present or absent. Alongside, we store the `names` array (Line 2) that contains the paths to each optional node such that the position matches the position of the corresponding Boolean in the fingerprint. We keep them, and hence also the entries in the fingerprints, in a fixed sorted order. By doing so, we can perform the mapping efficiently using binary search on names. This will allow us to check a node's presence in the fingerprints, e.g., for performing a split.

In Big Data, datasets contain too many rows to store a fingerprint for each one. Fortunately, in practice, there are much fewer distinct fingerprints than rows. Therefore, we store an array of tuples of unique fingerprints and their absolute frequency (`fingerprints`, line 3). For further speedup, we also maintain precomputed values. Firstly, we store the `total` (Line 7) number of fingerprints in the set, i.e., the sum of counts of each distinct fingerprint. Secondly, for each optional node, we store the number of rows where that node is present (`freqs`, Line 6). This avoids recalculateing them regularly by iterating through the list of fingerprints.

For example, consider the fingerprints of our running example in Table 1a. The Fingerprint Set containing all of them is visualized in Table 1b. Each row except for the header and the last row contains one fingerprint. Each time, the last column shows the number of occurrences of the fingerprint. The fingerprint $[0, 1, 0]$ occurs twice in our example, so the first row's last entry is 2. The last row of the table shows the precomputed fingerprint counts. For example, node `B` is present in the first and second fingerprints. They occur twice and once, respectively, so the entry in the last row is 3. Finally, the last entry in the last row contains the total number of fingerprints.

The fingerprint set also maintains metadata like the number of distinct values for the value columns. These are omitted in the aforementioned figure for clarity. Currently, we only use them as a weight for the present columns. In the future, this could also be used to implement value-based splits in the fingerprint set model if the correlation issues detailed in Section 5.1 are addressed. In a similar manner to the fingerprints, we store a sorted array of leaf paths (`leaves`, Line 4) and keep the metadata in an identically ordered array (`leafMetadata`, Line 5). With all this in place, we now look into using the fingerprint set for the operations needed to partition in the following sections.

6.1 Calculating Gini Impurity

Using a Fingerprint Set, we can now calculate the Gini Impurity of the bucket it represents in one traversal of the document schema. As part of the Gini Impurity computation, we need to determine the probability $P(A)$ that node A is present. To this end, we first look up the node's index in the `names` to then obtain its absolute frequency from `freqs` and divide it by the `total` number of fingerprints.

The total sum of the Gini Impurities of all columns consists of the value columns and the definition level columns. For the former, we iterate through all columns with known metadata in `leafMetadata`. We estimate the Gini Impurity for column C, assuming uniform distribution, from the number of distinct values $|V_C|$:

$$B(C) = \frac{|V_C| - 1}{|V_C|} \tag{5}$$

For definition level columns, we iterate over all paths in `leaves` and compute the absolute probability of each prefix $n_0, \ldots, n_i$ of column C. Here, n_i is the full path of C and n_0 is the empty prefix which has a presence probability of 1. Since the definition level is d if and only if n_d is present but n_{d+1} is absent, we can obtain the probability of each definition level from the difference of the presence probabilities. Using Equation 1, we arrive at the following formula for the Gini Impurity of a definition level column:

$$B(C) = 1 - P(n_i)^2 + \sum_{k=0}^{i-1} (P(n_k) - P(n_{k+1}))^2 \tag{6}$$

Note that the presence probability of each node is required once for each of its descendants which are leaves. Here, the preprocessing pays off. Without it, we would have to iterate the `fingerprints` sequence each time instead of simply looking up the frequency in `freqs`. Using Equations 5 and 6, we can now calculate the total number of boundaries as the sum of the Gini Impurities of all columns as per Equation 4. The result will be used in Section 7 to pick the best candidate split.

6.2 Determining a Sort Order

Reordering and partitioning are not exclusive. Any reordering approach can further decrease the size of the buckets resulting from our partitioning.

One of the most commonly recommended reordering heuristics is the increasing-cardinality heuristic. It sorts the rows lexicographically and considers the columns in the order of their increasing number of distinct values. Applying this heuristic alone typically requires one pass over the input data to determine estimates of the cardinalities. However, the fingerprint set already contains all the information needed. It explicitly stores the cardinalities for all value columns. For definition level columns, the cardinality can be derived from the fingerprints. For a given root-to-leaf path, determine all its prefixes, query the fingerprint set for their presence probability, and then count the number of non-zero distinct probabilities. The pseudocode for this operation can be found in Algorithm 1. Now that we have all column cardinalities, we can determine a column order for lexicographical sorting.

6.3 Split Operations

Recalculating the model from the source dataset for each candidate split incurs a too-costly performance penalty. Instead, we obtain the model after splits by partitioning the fingerprints. The pseudocode for the procedure can be found in Algorithm 2.

```
CARDINALITIES(p, fp)
input: the root-to-leaf path p, the fingerprint set fp
result: the cardinality
R := ∅
for each prefix pref of p
    prob = presence probability of pref in fp
    if prob > 0
        $R := R \cup \{prob\}$
return $|R|$
```

Algorithm 1: Cardinalities using Fingerprint Sets

Assume we want to calculate a split based on whether the node with path n is present or absent. First, we need to identify its associated index i in the fingerprint using binary search on the `names` field. Then, we iterate through the `fingerprints` sequence and build two new sequences from it: one containing all tuples where the fingerprint's i-th value is `true` and one with all others. These become the new fingerprint sets of the buckets after the split. We assume independence between the column values and the structure, so the leaf metadata does not need to be touched.

Finally, only `freqs` and `total` need to be recalculated for both newly created sets. Instead of recreating both from scratch, we only sum the values for the present side. Then, the values for the absent side can be calculated as the difference between the old value and the value for the present side. This roughly halves the number of additions we need to perform.

Table 2 shows the fingerprint sets that result from splitting our running example from Table 1b by the presence of node `B`. All fingerprints where it was marked as present were moved to the set in Table 2a, the others in Table 2b. Note how this has automatically captured the fact that `A` is only present when `B` is not.

6.4 Generating a Fingerprint Set

This section discusses how to initially obtain a Fingerprint Set in one scan of the source dataset. We first extract all optional nodes from the schema and sort them by their paths as described in Section 6. In Spark, the fingerprint can then be derived as the array of whether these paths are not null in a document. Using this, we can obtain

```
SPLIT(fp,a)
input: the fingerprint set to split fp, the path to split by p
result: the fingerprint sets where p is present/absent
▹ split fingerprints
i := index of p in fp.names
(pres,abs) := partition fp.fingerprints by index i
▹ precalculate frequencies
pTotal := $\sum_{f \in pres} f.count$
pFreqs := recalculate freqs from pres
aTotal := fp.total - pTotal
aFreqs := fp.freqs - pFreqs
▹ build sets
fpPres := build FingerprintSet from pres, pTotal, pFreqs
fpAbs := build FingerprintSet from abs aTotal, aFreqs
return (fpPres, fpAbs)
```

Algorithm 2: Splitting a Fingerprint Set

Table 2: The Fingerprint Set from Table 1b partitioned by `B`

(a) `B` Present

	A	B	E	#
	0	1	0	2
	0	1	1	1
∑	0	3	1	3

(b) `B` Absent

	A	B	E	#
	1	0	0	2
∑	2	0	0	2

the respective counts as a simple query for the count grouped by the fingerprint. If we wanted to employ sampling, we could include a sampling operator before the grouping operation.

Additionally, we need to collect the distinct values for each value column, i.e., each leaf in the schema. Again, we extract these paths and sort them as described before. We then use Spark's [4] implementation of the HyperLogLog++ algorithm [30] to obtain the cardinality estimates. To prevent a second scan over the data, we use Spark's `observe` function to collect the aggregates as the grouping of the fingerprint counts consumes them.

As we will see in Section 9, gathering statistics contributes majorly to the runtime overhead of our approach. To counter this, we can employ **sampling** on our first pass over the input and, thus, approximate the fingerprint set from a subset of the data. The second pass, which transfers the data to the storage remains unaffected.

However, traditional randomized sampling adds every document to the sample with a probability p. Since this is also true for the last document, it still requires parsing the entire file. Hence, it does not deliver the desired speedup on read- or parsing-intensive data sources like JSON documents. Instead, we sample by only reading the first k documents in every JSON file. Thereby, we can stop parsing after k documents. The risk is that these initial records may not be representative of the entire dataset, e.g., if the dataset comes presorted in some manner.

7 FINDING A PARTITIONING SCHEME

We model the partitioning scheme as a tree structure and then use Hunt's algorithm to derive such a tree from the previously generated fingerprint data. Similar to prior work [28], we represent a partitioning scheme as a non-symmetrical tree and materialize it alongside the data. Each of its leaves corresponds to a collection of Parquet files in the underlying storage, which it references using a path. In our case, we employ HDFS, but the same principles apply to, e.g., cloud storage. Inner nodes represent split operations. The most relevant version represents a presence-based split. It stores a path from the schema's root to an optional node in the schema. All documents for which the path is present will reside in a bucket in its left subtree. Conversely, all documents where the path is absent end up in the right subtree. This structure allows us to efficiently identify relevant buckets given a query that uses the path in one of its predicates. For more details on read operations, see [28].

Additionally, we support another type of inner node called an n-way node. It can have arbitrarily many children and imposes no further restrictions on the contents or structure of its subtrees. We use it to represent random or round-robin partitioning. For our use-case, its children will only ever be buckets.

We adapt Hunt's algorithm for inducing decision trees to construct a partitioning tree from our model. Initially, we can consider the input dataset to be one large bucket. Since it contains the entire dataset, we can obtain its fingerprint set using the procedure presented in Section 6.4. From here on, we recursively split buckets by considering all optional schema nodes as candidate splits. For each candidate split, we split the bucket's fingerprint set using the procedure outlined in Section 6.3. Then, we estimate the number of boundaries using the Gini Impurity as outlined in Section 6.1 and pick the split that results in the lowest estimate. Thereby, we essentially treat each fingerprint as one class.

However, it is not desirable to continue the process until a clean separation is achieved. In many cases, fingerprints only encompass a single document. For such small buckets, the overhead of storing the Parquet metadata alone makes them unattractive. While experimenting with purity-based termination criteria, we found that the largest bucket dominates the runtime for data ingestion. The purity does not give us any guarantee for its size. Hence, we instead terminate the recursion based on the size of the buckets. To this end, our algorithm has two parameters. First, the **MAX parameter** controls the maximum size of a bucket in terms of documents. For easier comparison to sorting approaches, we express it in terms of buckets: When MAX is set to 100, we continue splitting until a bucket is at most as large as a bucket when evenly partitioning into 100 buckets, i.e., at most 1 % of the total documents.

Second, we introduce the **MIN parameter** which controls the minimum size of buckets. We express it as a percentage of MAX When MIN is set to 50 %, the smallest allowed bucket is half as large as the maximal allowed bucket. The higher MIN is, the more evenly the buckets will be sized. If a candidate split would result in a bucket smaller than the minimum size based on the fingerprint set, we do not consider it for the split. If there are no valid split candidates for a bucket b that is larger than the maximum size, we instead perform a random n-way split. We choose n such that it is the smallest value that satisfies the MAX boundary, i.e., $n = \left\lceil |b| \cdot \frac{\text{MAX}}{total} \right\rceil$.

The changed termination criterion essentially transforms the algorithm into a divisive clustering algorithm. As opposed to other clustering algorithms such as k-means or LSH, we can control the size of the clusters. Besides, retaining the partitioning tree enables us to do data skipping as it encodes how the documents were matched to the clusters.

8 COMPLEXITY

In this section, we analyze the complexity of our approach. Let n be the number of rows, s be the number of nodes in the schema, h the height of the schema, and f the number of distinct fingerprints the data contains. In the following, we will look at each of the steps as listed in Section 4.

In step one, we initialize the fingerprint set. The dominant operation in this step is grouping the fingerprints. This can be done by sorting in $O(n \log n)$ comparisons, each of which requires $O(s)$ time. Hence, the initial fingerprint set can be constructed in one scan with $O(sn \log n)$ time.

For the partition tree induction, we first look at the runtime of testing an individual candidate. We can calculate the split in $O(fs)$ time. On the resulting sets, the gini impurity is calculated using a $O(\log s)$ lookup for all $O(h)$ prefixes of the $O(s)$ definition level columns. Hence, the overall time for one candidate is in $O(fs + sh \log s)$. We need to test $O(s)$ candidates, which can be done in parallel. Since all candidate splits are binary and the algorithm creates at most $l = \frac{\text{MAX}}{\text{MIN}}$ leafs, we arrive at a total runtime of $O\left(l\left(fs^2 + s^2 h \log s\right)\right)$ for step two.

If sorting is applied, we extract column cardinalities from a fingerprint set. For each of the $O(s)$ root-to-leaf paths, and their $O(h)$ prefixes, we retrieve their probabilities in $O(\log s)$ time. This is more expensive than sorting, so we end up with $O(sh \log s)$ time.

In the final step, we perform a second and final scan of the data. For each of the n rows, we need to look up which partition it belongs to in the partition tree. That incurs a worst-case overhead of $O(l)$. Naturally, the optional sorting step also induces a time cost of $O(n \log n)$ per partition.

In practice, $f \ll n$ since every row can contribute at most one fingerprint. Further, we can assume that the dataset is large, i.e., $s \ll n$, and hence $h \ll n$. Therefore, steps 1 and 4 dominate the overall ingestion time because they directly depend on n. This is confirmed by our experiments in Section 9.1.

Table 3: Dataset Metadata

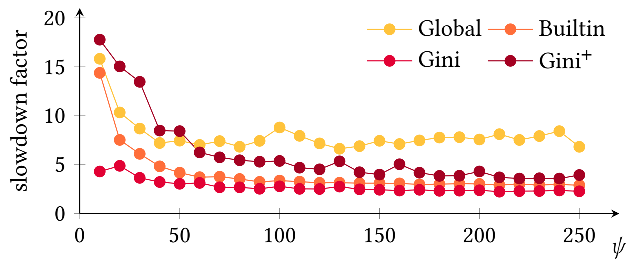
	Twitter	GitHub	TPC-DS
Records	45.95 M	29.63 M	188.544 M
Columns	371	611	1524
Fingerprints	177.9 K	3338	> 1.4 M
JSON SIZE (GiB)	108.1	112.1	681.6
Parquet Size (GiB)	26.24	29.22	89.99
Ingestion Time (min)	5.4	5.61	102.2

9 EVALUATION

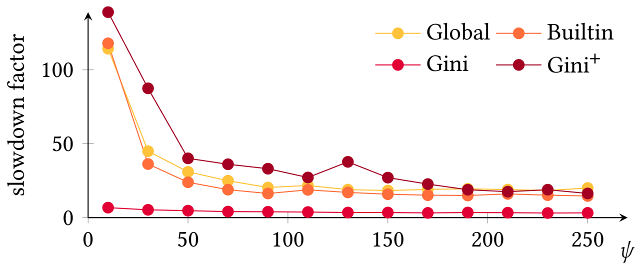
We conduct our experiments in a dockerized Spark cluster hosted on 8 servers, each with Intel® Xeon® E5-2603 v4 CPUs and 126 GiB of RAM. The data resides on HDD with a read rate of about 180 MB/s.

Each physical server hosts two docker containers: one spark worker as a compute node and one HDFS data node as a storage node. The two containers are unaware they are co-located and can only communicate via the 10 Gbit/s network. Two servers additionally run the HDFS namenode and the Spark controller, respectively. HDFS is configured to use a blocksize of 128 MiB and threefold replication. The input data resides on network-attached storage which is accessible by all workers at the slower speed of 1 Gbit/s.

Datasets. We evaluate our results on two real-world JSON datasets. First, a collection of tweets scraped from the service formerly known as **Twitter** in the first seven days of May 2017. Second, a collection of **GitHub** API events [24] from January to March 2015 that was downloaded from GH Archive [26]. An overview of the metadata can be found in Table 3. Both datasets are around the same size in GiB, but the GitHub dataset is wider and in turn, has fewer documents. It also has fewer distinct fingerprints and thus offers fewer opportunities for compression to our approach. Additionally, its contents naturally are in a better-than-random order for compression which diminishes the performance of all approaches. Hence, it serves as an example of a dataset that is hard to boost.



(a) Twitter



(b) GitHub

Figure 3: Overall Ingestion Time

We further evaluate our approach on the **TPC-DS** benchmark [49]. It consists of a normalized multiple-star schema and is typically used to evaluate relational data warehouses. As our approach works with heterogeneous JSON data, we have adapted the data in a manner similar to Durner et al. [17]. First, we have stripped the unique prefixes from the column names. We then transformed each row of the fact tables into a JSON object whose keys are the former column names. The referenced dimension entries were transformed analogously and recursively merged into the object as nested objects. Finally, we extended the fact objects by a new key which identifies the source table and then shuffled all objects into a single dataset. The resulting JSON collection for scale factor 10 is by far our most challenging dataset and almost exhausts our cluster's memory. Its characteristics can be found in Table 3.

Competitors. In our evaluation, we compare four different approaches. We refer to our approach as **Gini**, and **Gini**$^+$ when optional sorting is enabled. The competitor coined **Global** follows the suggestion by Lemire et al. [40] and creates a global sort order by sampling the dataset and then creating evenly sized range-based partitions, which it eventually sorts. Sorting is done using Spark's sort functionality. For the other end of the investigated spectrum, we follow the suggestion by Melnik et al. [44] and mimic the behavior of a compression booster integrated into a file format driver. To this end, we partition the data evenly and then locally sort each bucket. We refer to this baseline as **Builtin**.

In all cases, sorting is done using the increasing-cardinality heuristic for value and definition level columns. As such, the two baseline approaches also include a statistics-gathering phase, which estimates all column cardinalities using HyperLogLog++ [30].

Comparing Partition Sizes. The runtime of sorting and materializing buckets both depend on the partition sizes. Since partitions are also the unit of distribution in Spark, the determining factor of our runtime is the size of the largest partition. Each of the approaches has a parameter that controls this property. When comparing them in the following, we will always consider alternatives where these *partition size indicators* are set to the same value and refer to it as ψ. Since each approach works with approximately even-sized buckets, the bucket count is indicative of the sizes, albeit not determining it precisely. For Builtin, we can directly set the number of partitions to ψ. We set the `spark.sql.shuffle.partitions` parameter to ψ for Global, but we have found that Spark does not precisely follow the set value. For our approaches, we set MAX to ψ as it determines the maximum size of buckets.

Unless otherwise stated, we set ψ to 150, MIN to 50 %, and disable sampling. For TPC-DS, we only run $\psi = 250$ and aggressively sample the first 5000 rows per input file. To warm up the caches, we run Spark's schema detection multiple times before the experiments.

9.1 Ingestion Time

In this experiment, we want to evaluate the time it takes to import new data into the system. To this end, we have run all four approaches and measured the time between the DataFrame creation and the completion of the materialization The numbers reported always refer to the median of 10 executions for Twitter. Due to the generally longer execution times on GitHub, we only report the average between two executions on it. We have supplied a predetermined schema to Spark to avoid running its schema detection.

End-to-end Runtime. All Compression Boosters inherently invest runtime when the data is ingested to achieve a better compression ratio. As such, we measure the ingestion performance as the relative increase in ingestion time compared to using no such booster. These slowdown factors on Twitter for ψ between 10 and 250 can be found in Figure 3. As can be seen, the slowdown always diminishes as ψ increases as expected. Our Gini approach is the fastest for all values because it does not rely on sorting. On Twitter, we obtain a 3.34 times speedup over Builtin for $\psi = 10$ and factors between 1.14 and 1.67 for all other values. Compared to Global, Gini is considerably faster, and the speedup factors range between 2.12 and 3.68. On GitHub, the slowdown of all sorting-based approaches is worse by an order of magnitude. Yet, the purely partitioning-based Gini approach still performs very well and hence obtains even more drastic speedups. For $\psi = 10$ it is 17.44 times faster than Builtin and 16.89 times faster than Global. For higher values of ψ the speedup ranges between 4.12 and 8.50.

If we add sorting, our Gini$^+$ approach starts as the slowest but outpaces Global as ψ increases. On Twitter, this happens very quickly and we then observe a speedup of up to factor 2.35. On GitHub, we only outpace Global at $\psi = 190$ for a maximum speedup of factor 1.22. However, all sorting-based approaches are prohibitively slow on that dataset with at least a 14.67-fold slowdown. A potential reason for the slow start is the assignment of partitions to Spark workers. Our approach generates more partitions than the given ψ. If a node gets predominantly large partitions, this straggler's longer runtime determines the overall performance. On GitHub, this effect is emphasized due to the overall poor performance of

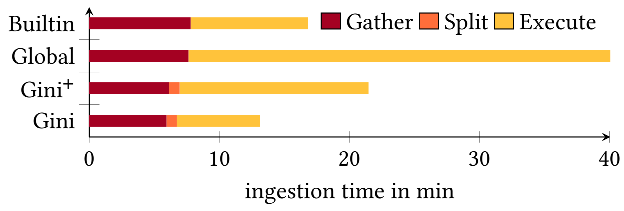


Figure 4: Ingestion Time Composition on Twitter for $\psi = 150$

sorting. We will discuss this further in Section 9.3. This aside, our approach is meant for cases where the ingestion process is highly parallelized, so the smaller cases are less relevant.

Since the TPC-DS dataset is extremely wide, it puts a heavy strain on our cluster and we frequently see tasks fail due to memory or garbage collection issues. This especially affects the Gini$^+$ approach which requires the memory for both the very large fingerprint set and the data to sort. As a result, it ends up slower than all other approaches with a slowdown of factor 8.57. As expected, Gini beats all other approaches with a slowdown of only factor 3.3. This includes the Builtin approach which is slowed down by factor 4.36.

Overall, we see that our Gini approach significantly outperforms the baseline sorting approaches in terms of ingestion time. In many cases, this still holds if sorting is applied afterward.

Runtime Composition. To further analyze the composition of the approaches' runtimes, we have measured the runtime of the steps from Section 4 individually. The results are summarized in Figure 4. The numbers shown reflect each step's median runtime. We have omitted the runtime of determining a sort order in the plot as it is below 50 ms for all approaches.

As can be seen, the time needed to determine the split is insignificant compared to the overall runtime thanks to the algorithmic optimizations we applied. Gathering statistics takes constant time regardless of the desired partition counts. It takes about as long for the baseline approaches as for ours. The majority of the time is spent in the execution step. In cases with intensive sorting overhead, such as Global in the figure, it is so dominant that even the contribution of the gathering step becomes insignificant. However, for the Gini approach or higher values of ψ, it can make up as little as 45 % of the total time. This validates our application of sampling to speed up the gathering step in Section 6.4.

9.2 Compression Ratio

In this experiment, we investigate the achieved compression boost, i.e., the compression ratio compared to the compressed size without applying a booster. We use the same experimental setup as for the runtimes in Section 9.1.

The results on the Twitter dataset can be seen in Figure 5. There, we can see the filesize of the Builtin approach increases with ψ as expected. For GitHub, not using it at all would yield better results. This is because the records in that dataset are already in an order which benefits compression. If we randomize their order, the size without booster increases to 37.13 GiB—1.27 times more. On Twitter, the filesize resulting from our Gini approach decreases with ψ because more splits can be performed. For small values our approach starts worse than Builtin by a factor of 1.12, but for high values ends out better by a factor of up to 1.16. This again underlines that our algorithm is ideal for highly parallelized use cases. On GitHub, the Gini approach can hardly improve over the already highly correlated input. Note that this corresponds to a boost factor of 1.33 relative to the randomized size of the dataset—much closer to the improvements we see on Twitter. However, it also does not worsen it like the Builtin approach. We attribute the marginal improvements to the much lower number of distinct fingerprints in the GitHub dataset.

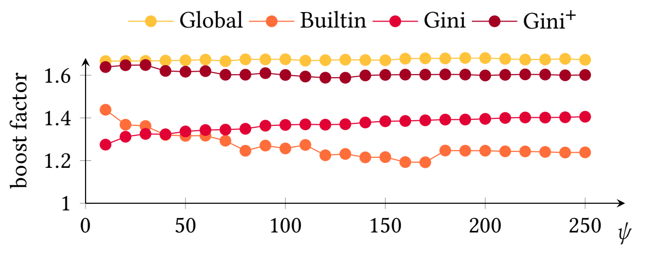
If sorting is considered, our Gini$^+$ approach yields compression very close to the Global approach. We are always within 5 % of it on Twitter. As ψ increases, more fingerprints occur more often than the maximum size of a partition. Therefore, our splitting algorithm has to resort to n-way splits more often. That again splits long runs between buckets and hence deteriorates the compression rate. Since GitHub has fewer fingerprints, the effect is more pronounced in that dataset. A potential solution to this would be to incorporate the values of further low-cardinality columns into the fingerprint at the expense of a combinatorial explosion of the number of fingerprints. We leave an investigation of this effect to future work.

On TPC-DS, the highest boost factors are achieved overall. As expected, Builtin is the worst with only 1.56. Our Gini approach obtains a boost of factor 1.95 and thereby improves over Builtin by factor 1.25. The best approach is Global as expected with a boost of 2.74, and Gini$^+$ comes in second with a boost of 2.22. Overall, these results confirm our expectations.

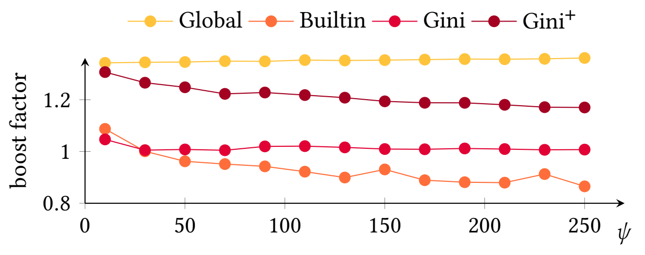
9.3 Algorithm Parameters

Influence of the MIN Parameter. In this experiment, we evaluate the impact of the MIN parameter on the ingestion time and the resulting compressed size. To this end, we have fixed MAX to 50, 150, or 250 while varying MIN from 10 % to 90 % on the Twitter dataset. The results show that the boost factor increases as MIN decreases. Lower values for MIN allow for the creation of smaller partitions. Thereby, more efficient splits can be chosen that would otherwise be eliminated due to the size requirement. However, the change is only marginal. Decreasing MIN by 10 percentage points improves the boost by between 0.6 and 0.8 percentage points depending on ψ for Gini. With Gini$^+$, we see a slightly higher change of between 0.8 and 1 percentage point.

The respective slowdowns do not reveal a consistent change in ingestion time as the MIN parameter changes. However, we notice that runtimes for low MIN values vary more than for high values and the variant with sorting varies more than the one without. We again attribute this to the assignment of partitions to worker nodes. Lowering MIN from 90 % to 10 % results in a 30 % increase of the partition count and leads to more uneven sizes. If a worker ends up with mostly above-average-sized partitions, it will take longer to complete its workload and hence slow down the overall computation. Since sorting is a superlinear operation, it exaggerates this issue. This is evident from the fact that the variance of the sorted approach is consistently higher than its unsorted counterpart. If this is indeed the issue, a more sophisticated worker assignment algorithm may offer great benefits. The fingerprint sets our algorithm creates contain ample information to infer the partition sizes and hence to estimate the runtime cost of each partition.

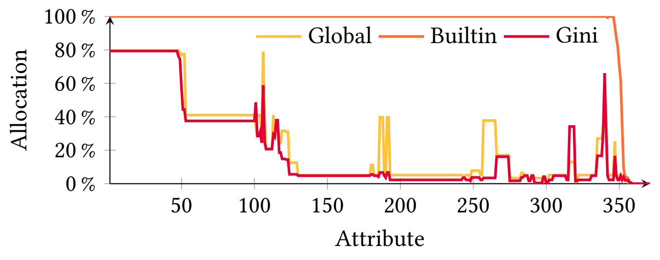


(a) Twitter

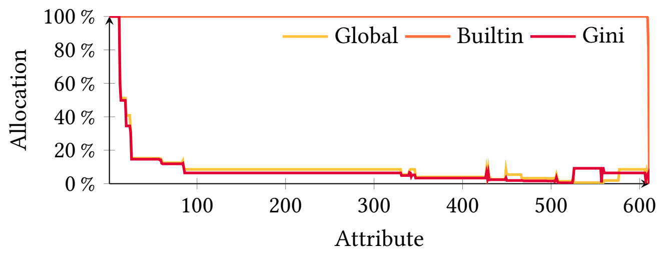


(b) GitHub

Figure 5: Achieved Compression Boost



(a) Twitter



(b) GitHub

Figure 6: Attribute Allocation for Attributes sorted by Occurrence Frequency in the Dataset

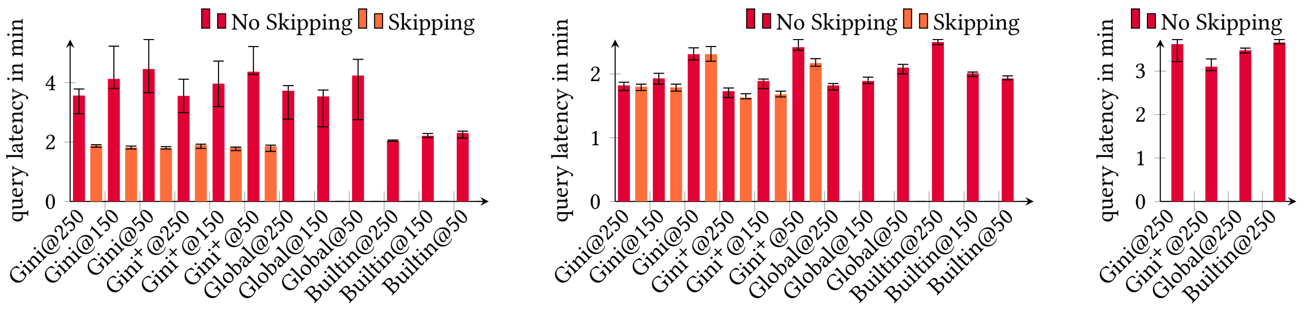
Sampling. As we have seen in Section 9.1, gathering statistics can make up a significant portion of the ingestion runtime. In this experiment, we want to determine the extent to which we can alleviate this using the sampling approach detailed in Section 6.4. To this end, we sample between the first 2000 and 26000 documents of each source JSON file of the Twitter dataset. Note that the upper bound corresponds to reading the entire dataset. In the unsorted case, the ingestion time grows linearly with the sample size by 0.2 min per 1000 rows. In total, this amounts to is a significant decrease of factor 1.37. In the sorted case, the trend shows a similar improvement in absolute terms. As we have seen in Figure 4, the gathering step makes up a lower proportion of the total runtime when sorting is enabled. Therefore, the lower relative improvement of 1.04 was as expected.

We also observe a non-linear effect on the sorted case. We again attribute this to the assignment of partitions to workers. As the sampling rate changes, so does the induced partitioning tree and with it the worker assignment. It is worth reiterating that all repetitions with the same sample size produce the same partitioning tree because we deterministically sample the first k records from every input file. We have also measured the corresponding boost factors and do not see a significant impact on the compression ratio. For both approaches, we do not see a more than 0.6 % increase in the size from sampling. As a result, we recommend employing aggressive sampling to decrease the statistics gathering time.

9.4 Query Processing Performance

Data Skipping. One of the main advantages of using our decision-tree-based approach over other clustering methods and our baselines is that the assignment of rows to buckets is explainable. The inner nodes of our partition tree retain the split conditions, i.e., presence information, which enables the use of data-skipping when reading the data. In this section, we want to evaluate the extent to which we can eliminate buckets from query processing.

We have materialized one partitioning for each of the approaches with $\psi = 150$. For each bucket, we determined which of the attributes are present in at least one row in the bucket. Assuming perfect information, this means we would have to download and process that bucket for a query involving this attribute. Figure 6 shows the percentage of rows we have to read this way for each attribute. The resulting measure is similar to the Attribute Allocation introduced by Shanbhag et al. [57]. The attributes on the x-axis have been sorted by frequency of occurrence in the dataset. Note that Gini$^+$ only differs from Gini in the order of rows within a bucket and therefore performs identically to the unsorted variant. As can be seen in the figure, the Builtin approach performs worst by a huge margin. Blindly assigning rows to partitions spreads each structural variant over all buckets and hence forces us to almost always read all of them. Global performs better, but the further back an attribute occurs in the sort order, the more its occurrences are spread over all buckets. As a consequence, our Gini approach achieves a tighter fit which outperforms sorting for all but 16 columns (4.31 %) on Twitter. There, we achieve an up to 11 times improvement in the rows that need to be processed over Global and 253 times over Builtin. Conversely, in the bad cases, we are at most 3.37 times worse. On average, we save 3.35 % additional rows. On GitHub, we perform slightly worse. We still universally outperform the Builtin approach by up to factor 188. Compared to Global, our approaches are better for all but 51 columns (8.35 %). In the best case, we improve by factor 2.83. In the worst case, our approach needs to read 14 times



(a) Twitter with BETZE **(b) GitHub with BETZE** **(c) TPC-DS**

Figure 7: Total Query Latencies on the Partition Schemes

as much data. Overall, we save 0.9 % rows on average. All in all, our approach does not sacrifice skipping opportunities to achieve its ingestion performance. On the contrary, it improves the skippable data drastically in many cases, especially compared to the Builtin approach. The common structure in the partitions is sufficient to skip more than 80 % of rows for many columns.

Data Analysis Workloads. To evaluate the query performance of our partitioning scheme, we have employed the BETZE benchmark generator [56]. It simulates the behavior of a data analyst during data exploration. We have generated two runs for each real-world dataset on the intermediate preset with aggregation enabled using a 5 % sample resulting in 20 queries for each dataset. Due to differences in the data model between BETZE and Spark, we have disabled the `is_string` predicate. Where applicable, we have employed data skipping [28]. Note that it only considers the column chosen for the split and not all information from the fingerprints that remain in the final buckets. The implementation of a more precise skipping technique is beyond the scope of this paper and is left for future work.

Figures 7a and b show the total runtimes for each approach both with and without data skipping. Each measurement is repeated 10 times and the median value is reported, with error bars indicating the minimum and maximum values. As can be seen, data skipping is very effective at reducing the query times on Twitter: Without it, our approaches result in runtimes similar to Global. With skipping, our approach becomes between 1.88 and 2.48 times faster and thereby beats all other approaches. On GitHub, the employed skipping technique cannot pick up the predicates of the least selective queries and hence only yields smaller improvements of up to factor 1.13. Except for low choices of ψ, we still outpace the baselines up to 1.53-fold. The query times without skipping increase significantly for lower values of ψ. This aligns with the decrease of the ingestion time and the increase of the Gini approach's boost factor we have seen in the previous experiments. Therefore, we conclude that we can tune our algorithm's parameters for ingestion time and compression rather than query time without sacrificing performance later on. The good performance of the Builtin approach on Twitter comes as a surprise to us. It beats the query times of the Global approach by a large margin despite producing larger files that occupy more HDFS blocks and present fewer data-skipping opportunities. However, we notice significantly higher errors for the other non-skipping approaches. We believe this is a property of the dataset and the queries generated for it because GitHub does not show this behavior.

TPC-DS Workload. We have also evaluated the query latency on the resulting layouts of our approaches on TPC-DS. Since our data transformation eliminates all dimension tables, some of the benchmark's queries are not immediately applicable anymore. We have adapted and run the first 10 queries which fit the new representation and visualize the results in Figure 7c, i.e., Queries 1-5, 7, 11-13, and 15. Unfortunately, the partition tree has not picked up any predicates directly used by the queries, so we only report the runtimes without skipping. However, the Parquet-level skipping in conjunction with the changed file sizes still makes Gini$^+$ the fastest approach. Gini's runtime is competitive with the other approaches. It beats Builtin and in some cases even Global.

10 CONCLUSION

In this paper, we presented a clustering approach for integrating compression boosters into cloud data ingestion pipelines which partitions the data based on the Gini Impurity of the documents' structure. We showed that it outpaces even the fast Builtin approach by up to factor 17.44 while at the same time also providing up to 1.95-fold lower filesize. If the created buckets are further sorted, we obtain a compression boost competitive with the well-established increasing-cardinality heuristic applied on the entire dataset, but at lower ingestion times. Additionally, the partitioning tree generated by our approach can be employed for data skipping when querying the data later on. There, we achieved an up to 11 times tighter fit than the increasing-cardinality heuristic. For many columns, more than 80 % of rows can be skipped based on the structure which improved the query latency by up to factor 2.48. That makes our approach the best choice for structurally heterogeneous datasets and emphasizes the need to already consider compression boosters in the ingestion pipeline and not only in its last step as part of the file format driver.

ACKNOWLEDGMENTS

This work has been partially funded by the German Federal Ministry of Food and Agriculture under grant number 28DE113C18 (DigiVine). The responsibility for the content of this publication lies with the authors.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 671–682. https://doi.org/10.1145/1142473.1142548

[2] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 700–714. https://doi.org/10.1145/3514221.3517834

[3] Apache Software Foundation. 2013. Apache Parquet. https://parquet.apache.org/ (Last accessed: July 9, 2024)

[4] Apache Software Foundation. 2014. Apache Spark. https://spark.apache.org/ (Last accessed: July 9, 2024)

[5] Apache Software Foundation. 2017. Apache Iceberg. https://iceberg.apache.org/ (Last accessed: July 9, 2024)

[6] Tan Apaydin, Hakan Ferhatosmanoglu, Guadalupe Canahuate, and Ali Saman Tosun. 2008. Dynamic data organization for bitmap indices. In *3rd International ICST Conference on Scalable Information Systems, INFOSCALE 2008, Vico Equense, Italy, June 4-6, 2008*, Ronny Lempel, Raffaele Perego, and Fabrizio Silvestri (Eds.). ICST / ACM, 30. https://doi.org/10.4108/ICST.INFOSCALE2008.3554

[7] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[8] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 222–233. https://doi.org/10.5441/002/edbt.2017.21

[9] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting types for massive JSON datasets. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, Tiark Rompf and Alexander Alexandrov (Eds.). ACM, 9:1–9:12. https://doi.org/10.1145/3122831.3122837

[10] Haoqiong Bian, Youxian Tao, Guodong Jin, Yueguo Chen, Xiongpai Qin, and Xiaoyong Du. 2018. Rainbow: Adaptive Layout Optimization for Wide Tables. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1657–1660. https://doi.org/10.1109/ICDE.2018.00200

[11] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. 2017. Wide Table Layout Optimization based on Column Ordering and Duplication. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 299–314. https://doi.org/10.1145/3035918.3035930

[12] Adam L. Buchsbaum, Donald F. Caldwell, Kenneth Ward Church, Glenn S. Fowler, and S. Muthukrishnan. 2000. Engineering the compression of massive tables: an experimental approach. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, David B. Shmoys (Ed.). ACM/SIAM, 175–184. http://dl.acm.org/citation.cfm?id=338219.338249

[13] Adam L. Buchsbaum, Glenn S. Fowler, and Raffaele Giancarlo. 2003. Improving table compression with combinatorial optimization. *J. ACM* 50, 6 (2003), 825–851. https://doi.org/10.1145/950620.950622

[14] Jiafeng Chen, Lu Yu, and Wenyi Wang. 2022. Hilbert Space Filling Curve Based Scan-Order for Point Cloud Attribute Compression. *IEEE Transactions on Image Processing* 31 (2022), 4609–4621. https://doi.org/10.1109/tip.2022.3186532

[15] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated multidimensional data layouts in Amazon Redshift. In *SIGMOD 2024*. https://doi.org/10.1145/3626246.3653379

[16] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 418–431. https://doi.org/10.1145/3448016.3457270

[17] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 445–458. https://doi.org/10.1145/3448016.3452809

[18] Todd Eavis and David Cueva. 2007. *A Hilbert Space Compression Architecture for Data Warehouse Environments*. Springer Berlin Heidelberg, 1–12. https://doi.org/10.1007/978-3-540-74553-2_1

[19] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. 2018. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*. IEEE, 356–363. https://doi.org/10.1109/IRI.2018.00060

[20] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. 2010. Net-Fli: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *Proc. VLDB Endow.* 3, 2 (2010), 1382–1393. https://doi.org/10.14778/1920841.1921011

[21] Francesco Fusco, Michail Vlachos, and Marc Ph. Stoecklin. 2012. Real-time creation of bitmap indexes on streaming network data. *VLDB J.* 21, 3 (2012), 287–307. https://doi.org/10.1007/s00778-011-0242-x

[22] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. 2018. Schema profiling of document-oriented databases. *Inf. Syst.* 75 (2018), 13–25. https://doi.org/10.1016/j.is.2018.02.007

[23] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 518–529. http://www.vldb.org/conf/1999/P49.pdf

[24] GitHub Inc. 2024. Github developer webhooks. https://developer.github.com/webhooks/ (Last accessed: July 9, 2024)

[25] Google Inc. 2011. *Snappy Compressed Format Description.* Retrieved February 29, 2023 from https://github.com/google/snappy/blob/main/format_description.txt

[26] Ilya Grigorik. 2023. GH Archive. https://www.gharchive.org/

[27] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Managing Google's data lake: an overview of the Goods system. *IEEE Data Eng. Bull.* 39, 3 (2016), 5–14. http://sites.computer.org/debull/A16sept/p5.pdf

[28] Patrick Hansert and Sebastian Michel. 2022. Ameliorating data compression and query performance through cracked Parquet. In *BiDEDE '22: Proceedings of The International Workshop on Big Data in Emergent Distributed Environments, in conjunction with the 2022 ACM SIGMOD/PODS Conference in Philadelphia, PA, USA, June 12, 2022*, Sven Groppe, Le Gruenwald, and Ching-Hsien Hsu (Eds.). ACM, 6:1–6:7. https://doi.org/10.1145/3530050.3532923

[29] Patrick Hansert and Sebastian Michel. 2023. Schema-based Column Reordering for Dremel-encoded Data. In *BiDEDE '23: Proceedings of The International Workshop on Big Data in Emergent Distributed Environments, in conjunction with the 2023 ACM SIGMOD/PODS Conference in Seattle, WA, USA, June 18, 2023*, Sven Groppe, Le Gruenwald, and Ching-Hsien Hsu (Eds.). ACM, 6:1–6:7. https://doi.org/10.1145/3579142.3594286

[30] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, Giovanna Guerrini and Norman W. Paton (Eds.). ACM, 683–692. https://doi.org/10.1145/2452376.2452456

[31] David Hilbert. 1891. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Math. Ann.* 38, 3 (Sept. 1891), 459–460. https://doi.org/10.1007/bf01199431

[32] Yin Huai, Siyuan Ma, Rubao Lee, Owen O'Malley, and Xiaodong Zhang. 2013. Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters. *Proc. VLDB Endow.* 6, 14 (2013), 1750–1761. https://doi.org/10.14778/2556549.2556559

[33] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[34] H. V. Jagadish. 1990. Linear Clustering of Objects with Multiple Atributes. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 332–342. https://doi.org/10.1145/93597.98742

[35] Jane Jovanovski, Nino Arsov, Evgenija Stevanoska, Maja Siljanoska Simons, and Goran Velinov. 2019. A meta-heuristic approach for RLE compression in a column store table. *Soft Comput.* 23, 12 (2019), 4255–4276. https://doi.org/10.1007/s00500-018-3081-5

[36] Donghe Kang, Ruochen Jiang, and Spyros Blanas. 2021. Jigsaw: A Data Storage and Query Processing Engine for Irregular Table Partitioning. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 898–911. https://doi.org/10.1145/3448016.3457547

[37] Stefan Klessinger, Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2022. Extracting JSON Schemas with tagged unions. In *Proceedings of the First International Workshop on Data Ecosystems co-located with 48th International Conference on Very Large Databases (VLDB 2022), Sydney, Australia, September 5, 2022 (CEUR Workshop Proceedings)*, Cinzia Cappiello, Sandra Geisler, and Maria-Esther Vidal

(Eds.), Vol. 3306. CEUR-WS.org, 27–40. https://ceur-ws.org/Vol-3306/paper4.pdf

[38] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings (LNI)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.), Vol. P-241. GI, 425–444. https://dl.gi.de/20.500.12116/2420

[39] Daniel Lemire and Owen Kaser. 2011. Reordering columns for smaller indexes. *Inf. Sci.* 181, 12 (2011), 2550–2570. https://doi.org/10.1016/j.ins.2011.02.002

[40] Daniel Lemire, Owen Kaser, and Eduardo Gutarra. 2012. Reordering rows for better compression: Beyond the lexicographic order. *ACM Trans. Database Syst.* 37, 3 (2012), 20:1–20:29. https://doi.org/10.1145/2338626.2338633

[41] Jan-Yie Liang, Chih-Sheng Chen, Chua-Huang Huang, and Li Liu. 2008. Lossless compression of medical images using Hilbert space-filling curves. *Computerized Medical Imaging and Graphics* 32, 3 (April 2008), 174–182. https://doi.org/10.1016/j.compmedimag.2007.11.002

[42] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex - Designed For Analytics. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 1227–1230. https://doi.org/10.1016/B978-012088469-8.50111-X

[43] 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. https://doi.org/10.14778/1920841.1920886

[44] 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. https://doi.org/10.14778/3415478.3415568

[45] Michael J. Mior. 2023. JSONoid: Monoid-based Enrichment for Configurable and Scalable Data-Driven Schema Discovery. *CoRR* abs/2307.03113 (2023). https://doi.org/10.48550/arXiv.2307.03113 arXiv:2307.03113

[46] Guy M Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing*. resreport. https://dominoweb.draco.res.ibm.com/0dabf9473b9c86d48525779800566a39.html (Last accessed: July 9, 2024)

[47] Koyel Mukherjee, Raunak Shah, Shiv Saini, Karanpreet Singh, Khushi, Harsh Kesarwani, Kavya Barnwal, and Ayush Chauhan. 2023. Towards Optimizing Storage Costs on the Cloud. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE. https://doi.org/10.1109/icde55515.2023.00223

[48] Sreerama K. Murthy. 1998. Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. *Data Min. Knowl. Discov.* 2, 4 (1998), 345–389. https://doi.org/10.1023/A:1009744630224

[49] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1049–1058. http://dl.acm.org/citation.cfm?id=1164217

[50] Frank Olken and Doron Rotem. 1986. Rearranging Data to Maximize the Efficiency of Compression. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, Avi Silberschatz (Ed.). ACM, 78–90. https://doi.org/10.1145/6012.15407

[51] Ali Pinar, Tao Tao, and Hakan Ferhatosmanoglu. 2005. Compressing Bitmap Indices by Data Reorganization. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 310–321. https://doi.org/10.1109/ICDE.2005.35

[52] Elaheh Pourabbas, Arie Shoshani, and Kesheng Wu. 2012. Minimizing Index Size by Reordering Rows and Columns. In *Scientific and Statistical Database Management - 24th International Conference, SSDBM 2012, Chania, Crete, Greece, June 25-27, 2012. Proceedings (Lecture Notes in Computer Science)*, Anastasia Ailamaki and Shawn Bowers (Eds.), Vol. 7338. Springer, 467–484. https://doi.org/10.1007/978-3-642-31235-9_{3}{1}

[53] A.H. Robinson and C. Cherry. 1967. Results of a prototype television bandwidth compression scheme. *Proc. IEEE* 55, 3 (1967), 356–364. https://doi.org/10.1109/proc.1967.5493

[54] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. 2015. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Conceptual Modeling*. Springer International Publishing, 467–480. https://doi.org/10.1007/978-3-319-25264-3_35

[55] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures.* Academic Press.

[56] Nico Schäfer and Sebastian Michel. 2022. BETZE: Benchmarking Data Exploration Tools with (Almost) Zero Effort. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2385–2398. https://doi.org/10.1109/ICDE53745.2022.00224

[57] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. 2017. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 229—-241. https://doi.org/10.1145/3127479.3131613

[58] Jia Shi. 2020. Column Partition and Permutation for Run Length Encoding in Columnar Databases. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2873–2874. https://doi.org/10.1145/3318464.3384413

[59] William Spoth, Oliver Kennedy, Ying Lu, Beda Christoph Hammerschmidt, and Zhen Hua Liu. 2021. Reducing Ambiguity in Json Schema Discovery. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1732–1744. https://doi.org/10.1145/3448016.3452801

[60] Sivaprasad Sudhir, Wenbo Tao, Nikolay Pavlovich Laptev, Cyrille Habis, Michael J. Cafarella, and Samuel Madden. 2023. Pando: Enhanced Data Skipping with Logical Data Partitioning. *Proc. VLDB Endow.* 16, 9 (2023), 2316–2329. https://doi.org/10.14778/3598581.3598601

[61] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-oriented Partitioning for Columnar Layouts. *Proc. VLDB Endow.* 10, 4 (2016), 421–432. https://doi.org/10.14778/3025111.3025123

[62] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. 2018. *Introduction to Data Mining (2nd Edition)* (2nd ed.). Pearson.

[63] Binh Dao Vo and Kiem-Phong Vo. 2007. Compressing table data with column dependency. *Theor. Comput. Sci.* 387, 3 (2007), 273–283. https://doi.org/10.1016/j.tcs.2007.07.016

[64] Zhiyi Wang and Shimin Chen. 2017. Exploiting Common Patterns for Tree-Structured Data. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 883—-896. https://doi.org/10.1145/3035918.3035956

[65] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. 2023. LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 452–468. https://doi.org/10.1145/3552326.3567484

[66] Grisha Weintraub, Ehud Gudes, Shlomi Dolev, and Jeffrey D. Ullman. 2023. Optimizing Cloud Data Lake Queries With a Balanced Coverage Plan. *IEEE Transactions on Cloud Computing* PP (2023), 1–16. Issue 99. https://doi.org/10.1109/TCC.2023.3339208