Overlay Bitmap Encoding for Efficient Consumption of Apache Parquet Files

Güneş Aluç
SAP Labs
Waterloo, ON, Canada
gunes.aluc@sap.com

Daniel J. Farrar
SAP Labs
Waterloo, ON, Canada
dan.farrar@sap.com

Manoj Kumar Sharma
SAP Labs
Waterloo, ON, Canada
manoj.sharma02@sap.com

Weibing Zhang*
University of Waterloo
Waterloo, ON, Canada
n23zhang@uwaterloo.ca

ABSTRACT

Apache Parquet has significantly transformed big data processing with its efficient columnar storage capabilities, incorporating techniques such as the Partition Attributes Across (PAX) layout and dictionary-based encoding to enhance data compression and query performance. Despite these advantages, Apache Parquet's limited indexing capabilities can lead to inefficiencies when users need to fetch only a subset of the data. This paper proposes an innovative extension to Apache Parquet by introducing a new encoding format where indexes form the foundational physical representation of a column. Specifically, we introduce *overlay bitmaps*, a data structure that organizes column chunks into sequences of bitmap parts, each representing distinct column values that are further compressed using run-length encoding. Our implementation within the Apache Parquet C++ libraries and integration into the SAP IQ relational engine demonstrates the dual use of overlay bitmaps as both a storage and indexing mechanism, significantly improving predicate evaluation speed and compression efficiency. Experimental evaluations highlight that overlay bitmap encoding surpasses traditional dictionary encoding in compression and reduces load execution times by up to two orders of magnitude by enabling efficient, row-level, predicate pushdown. This capability allows complex queries, which involve numerous joins and predicates, to be executed up to 10x faster.

PVLDB Reference Format:
Güneş Aluç, Daniel J. Farrar, Manoj Kumar Sharma, and Weibing Zhang. Overlay Bitmap Encoding for Efficient Consumption of Apache Parquet Files. PVLDB, 19(12): 3834 - 3846, 2026.
doi:10.14778/3827998.3827999

*This work was conducted during the author's internship at SAP Labs, Waterloo.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3827999

1 INTRODUCTION

Apache Parquet [49] has revolutionized the way big data are queried and processed. Apache Parquet offers a combination of (i) the columnar representation [2], (ii) the Partition Attributes Across (PAX) layout [4], (iii) the dictionary-based encoding [6] and (iv) lightweight indexes (e.g., zone maps [35], and existence and null bitmaps [19]) to compress and query tabular data. Apache Parquet forms the building blocks of Open Table Formats (OTFs) such as Delta Tables [11] and Apache Iceberg [43], and it has been successfully incorporated into the Apache SPARK [52] ecosystem as well as into many commercial and open-source database management systems [3, 14, 38, 42].

While Apache Parquet offers numerous advantages, it does not inherently support indexing beyond rudimentary data structures; therefore, in practice, one of three things happens: (i) selective queries often result in the unnecessary utilization of compute, and thus, monetary, resources [25, 40]; (ii) vendors implement proprietary engines to efficiently execute queries [15], but because of their proprietary nature, consumers face the risk of a vendor lock-in which defeats the purpose of an open format; (iii) data are indexed after being replicated in a database management system which leads to redundancy and/or issues with data freshness [39].

The problem arises because in Apache Parquet, indexing is treated as an afterthought. Instead, in this paper, we propose an extension to Apache Parquet in the form of a new encoding where indexes constitute *the* base physical representation of a column. More precisely, a table is logically partitioned into row groups and the columns within each row group are physically represented using a new data structure called the overlay bitmaps. An overlay bitmap is composed of a *sequence* of bitmap *parts* where there could be as many parts as there are distinct values in the row group of the column. Each bitmap part represents the row ID offset within the row group where the cell value of the column matches the distinct value associated with the bitmap part. The bitmap parts are encoded/decoded as a sequence, which means that the contents of a bitmap part during encoding/decoding depend on the contents of the previous parts in the sequence. The bitmap parts within the overlay bitmap are further compressed using run-length encoding [23]. This new representation serves not only as a base physical representation, for example, when an Apache Parquet column chunk needs to be scanned entirely, but also as an index that

significantly speeds up predicate evaluation (i.e., =, <, <=, >, >=, ≠), thus, addressing the originally stated problem.

To demonstrate these points, in a *development* branch, we have extended the Apache Parquet C++ libraries [7] that are currently used by SAP IQ's [3] load and unload engines, together with the necessary plumbing in SAP IQ to make use of these extensions. The same changes were applied also to the SAP HANA Cloud, Data Lake Relational Engine (HDLRE). With these modifications, users are able to unload SAP IQ database tables as Apache Parquet files where some or all column chunks are encoded using the new format. Likewise, users can load SAP IQ database tables from Apache Parquet files where some or all column chunks are encoded with the new format. Lastly, we have made modifications such that if users provide a set of predicates during the load operation, these predicates are pushed down to the Apache Parquet C++ libraries. The extensions we have implemented in these libraries are able to leverage the indexing capabilities of the overlay bitmap encoding to efficiently return the matching rows, thus, significantly speeding up the load and query execution times.

Through our proof-of-concept implementation, we have been able to demonstrate the following key points:

- Despite its dual nature (i.e., serving as both a columnar layout and an index), overlay bitmap encoding compresses better than dictionary encoding[1].
- Predicate evaluation can be pushed down to the Apache Parquet C++ library `LeafReader::NextBatch` function calls, where "found-set" bitmaps with row-level precision can be efficiently generated using overlay bitmaps. This allows values that do not match the predicates to be efficiently skipped during the decoding process.
- Experiments using the SAP IQ load engine show up to two orders of magnitude reduction in load execution times when predicate evaluation is pushed down to Apache Parquet using overlay bitmap encoding compared to loading the same files without any filters using dictionary encoding.
- Query evaluations in SAP IQ demonstrate up to an 10x speed-up for complex TPC-H queries compared to executing the same queries on dictionary-encoded files using SPARK SQL [10, 13].
- The time to *decode* Apache Parquet files encoded with overlay bitmap encoding (i.e., full scan) is comparable to those encoded with dictionary encoding, with up to 35 percent overhead.
- Overlay bitmap encoding may take up to twice as long to *encode* compared to dictionary encoding. This is acceptable due to the substantial benefits it offers during partial data loading (i.e., consider this as the indexing overhead).

The remainder of the paper is organized as follows. Section 2 gives an overview of the Apache Parquet file format, the SAP IQ relational engine and how the SAP IQ relational engine loads and unloads tables in the Apache Parquet file format. The section also provides a summary of related work. Section 3 describes the overlay bitmap encoding, in particular, the different flavors in which an overlay bitmap can be constructed and the various design trade-offs. Then, it introduces the algorithms that are used for encoding and decoding overlay bitmap parts, which is facilitated by a novel data structure called the *shift vector*. Section 4 presents the proposed file format changes to Apache Parquet. Section 5 discusses the changes that have been implemented in the Apache Parquet C++ libraries as well as in SAP IQ, and describes how predicate evaluation can be executed efficiently using the overlay bitmap encoding. Section 6 presents our experimental findings; and Section 7 discusses the strengths and weaknesses of our work as well as some future directions. Lastly, Section 8 offers some concluding remarks.

[1]By default, Apache Parquet relies on the `RLE_DICTIONARY` encoding, which combines dictionary encoding with run-length encoding. Throughout this paper, the term *dictionary encoding* refers to the default `RLE_DICTIONARY` encoding.

2 BACKGROUND AND RELATED WORK

The physical layout of a sample Apache Parquet file [7, 8, 39, 49]. begins and ends with a 4-byte magic number (`PAR1`), which serves as a signature to identify the file type. Between these magic numbers, the file is divided into two primary sections: data and metadata. The data section is organized into row groups, each containing a sequence of column chunks. Each column chunk holds all the data for a specific column within that row group, allowing for efficient columnar access and compression. A column chunk is made up of one or more data pages. While column chunks within the same row group may not have the same number of physical pages, they collectively represent the same number of rows. This organization enables Apache Parquet to minimize storage requirements and optimize query performance by accessing only the relevant data needed for a particular query.

The footer of the metadata section in an Apache Parquet file plays a crucial role in organizing and summarizing the structure of the data within the file. There are pointers that can be followed to easily locate the different data pages within a column chunk. Furthermore, this footer contains detailed metadata about each row group, including essential information such as the number of rows stored within each group. Additionally, within each row group's metadata, there are dedicated subsections for every column chunk. These subsections provide specific details about the column chunk, including the data type of the column, the encoding scheme used for storing the data, and the count of non-null values present in the column chunk. Furthermore, Apache Parquet's flexible design allows for an extensible list of key-value pairs within these subsections. This feature enables the storage of arbitrary metadata, offering users the flexibility to include additional context or information relevant to their specific use case.

SAP IQ is a disk-based, columnar, fully relational database management system specifically designed for handling complex online analytical processing (OLAP) tasks [3]. The cloud version of SAP IQ, also known as the SAP HANA Cloud, Data Lake Relational Engine (HDLRE) has recently gone through a significant transformation [3] whereby its storage subsystem has been reengineered to operate directly on object stores such as AWS S3 [1]. Support for loading and unloading Apache Parquet files has been added to both [42].

Techniques for executing queries on Apache Parquet files can be grouped into three main categories. The first category includes conventional database engines, which allow data to be imported from Apache Parquet files [49] or Delta tables/shares [11, 37] into their native formats. This approach leverages the ability of the database

engine to build indexes and gather statistics on the imported data, benefiting from decades of advancements in relational database technology to execute queries efficiently [20]. However, a significant drawback of this method is the potential for redundant network transfer costs and issues arising from data duplication, especially when dealing with large Apache Parquet files and selective queries. Most vendors, including SAP IQ [3, 42], Teradata [41], Vertica [48], and Amazon Redshift [14], support this type of workflow [46].

The second category, known as SQL-on-Files systems [40], extends database engine technology to execute queries directly on Apache Parquet files without data replication. These systems address the challenges posed by the absence of robust statistics and indexes within Apache Parquet [44]. One approach employs adaptive query processing techniques [22] that learn data distribution and statistics dynamically [29]. Another strategy utilizes existing metadata in Apache Parquet files, such as min-max zone maps and the physical file layout, which can be quickly read and parsed to accelerate query execution [39]. Additionally, some techniques focus on caching portions of Parquet files to expedite repeated queries [39, 40]. Our research complements these methodologies by enabling predicates to be pushed down to Apache Parquet.

The third category involves native Lakehouse architectures [12, 31, 46], where query execution is performed as close as possible to the data's storage location. These architectures typically integrate advanced techniques such as vectorized execution [45], on-the-fly query plan compilation [33], and the use of modern hardware or techniques like GPUs [24] and Low Level Virtual Machines (LLVMs) [50]. These techniques are embedded directly into the distributed query execution pipeline [9, 18, 21, 27, 51], often near the leaf level, where execution is single-threaded. Examples of systems employing these architectures include the Apache Arrow DataFusion [28], the Photon engine [15], Presto [47], and Velox [36]. By introducing a new encoding type at the lowest level in Apache Parquet, our work can enhance these native systems by enabling row-level push-down of predicate evaluation.

The research by Li et al. [30] introduces a technique for enhancing predicate evaluation in Apache Parquet by utilizing hardware-level bit manipulation instruction sets (BMIs). In their method, filter operations are executed by assessing all decoded values against the filter predicate, creating a bitmap that indicates whether each selected value satisfies the predicate. Although this process resembles the "found set" bitmap discussed in our paper, there is a fundamental difference between their approach and ours. Our work introduces a new Apache Parquet encoding that functions both as a columnar representation and as a bitmap-based index. In contrast, Li et al.'s methodology involves first decoding column chunk values that were originally encoded using dictionary encoding, and then evaluating predicates at runtime using BMI accelerators. Overlay bitmaps could potentially be improved also using hardware-level acceleration techniques, which is future work.

3 OVERLAY BITMAP ENCODING

To compute the overlay bitmap encoding for a column chunk, such as the example shown in Table 1a, we begin by identifying the distinct values present in the column chunk [5]. For each unique value, we create a bitmap part where a bit is set at a specific offset if the column value at that offset matches the distinct value being represented. For instance, consider the bitmap part for "alice" in Table 1b: bits are set at offsets 1 and 10 because the column chunk contains the value "alice" at those row positions (as shown in Table 1a). All other bits are set to zero in the bitmap part for "alice". Note that both the bitmap and the row offsets use a 1-based index.

Consider the overlay bitmap part for "bob". The order in which the overlay bitmap parts are constructed plays a crucial role. In the scan order example shown in Table 1b, the bitmap part for "alice" is generated before the one for "bob" because "alice" is encountered first while scanning the column chunk. Consequently, for "bob", bits at offsets 1 and 10 are left blank since they have already been set for "alice" – explicitly setting the bits to zero would have been redundant. This convention continues for all subsequent bitmap parts. Since "bob" appears at row positions 2 and 4, those corresponding bits are set in the bitmap part for "bob" and left blank in the subsequent parts. The process proceeds similarly for "heidi", "david", "eve", "frank", and "grace", as depicted in Table 1b.

In the overlay bitmap encoding process, spaces can be removed from bitmap parts without losing information. When a blank space is removed, all subsequent bits shift one position to the left, as illustrated in Table 1c. This technique preserves the integrity of the data as long as the bitmap parts are decoded in the same sequence they were originally encoded. For instance, to reconstruct the original bitmap part for "heidi" (refer to Table 2a), we begin with the leftmost set bit and shift all bits underneath it by one position to the right, thus, effectively reinserting the previously removed space. If multiple bits are set in the same leftmost position, the highest positioned bit that is 1 is used as reference (i.e., the bitmap part that appears earlier in the sequence). This procedure is repeated until all set bits are traversed, as further detailed in Tables 2b–2e.

In Table 1c, the removal of blank spaces results in a significant reduction in the total number of bits in the overlay bitmap. Specifically, it reduces by 2 bits for "bob", 4 bits for "heidi", 6 bits for "david", 12 bits for "eve", 13 bits for "frank", and 15 bits for "grace", resulting in a total reduction of 52 bits out of an original 112 bits. This marks the initial phase of compression in the overlay bitmap encoding process. A more effective compression can be achieved by encoding the overlay bitmap parts in order of decreasing density, as demonstrated in Table 1d. This approach optimizes the compression because any bit set in a higher-level overlay bitmap part causes corresponding bits in lower levels to remain blank, allowing them to be removed entirely. Thus, ordering the parts by decreasing number of bits set maximizes compression. Indeed, in Table 1d, a total of 65 bits have been removed, surpassing the 52 bits reduction.

The encoding method demonstrated in Table 1d enables efficient querying for equality predicates across row groups. For instance, if we want to find the row offsets where the column values are equal to "alice", we only need to decode the sequence of bitmap parts for "david" and "alice". There is no need to decode the remaining bitmap parts in the overlay bitmap encoding. Once decoded, the bitmap for "alice" is represented as 1000000001000000, indicating that the values in rows at offsets 1 and 10 satisfy the predicate.

The encoding method described in Table 1d is not optimized for efficiently handling comparison predicates like $<$ or $\leq$. If you need to efficiently execute these types of comparisons, consider using the encoding outlined in Table 1e. In this setup, the densest

Table 1: Example Construction of Overlay Bitmap Encodings

(a) Columnar Values

	Value		Value
1	alice	9	frank
2	bob	10	alice
3	heidi	11	grace
4	bob	12	heidi
5	david	13	david
6	david	14	david
7	eve	15	frank
8	david	16	david

(b) Bitmaps in Scan Order

	1234567890123456
alice	1000000001000000
bob	1010000000000000
heidi	1000000010000
david	1101001101
eve	1000
frank	101
grace	1

(c) Encoding in Scan Order

	1234567890123456
alice	1000000001000000
bob	1010000000000000
heidi	1000000010000
david	1101001101
eve	1000
frank	101
grace	1

(d) Density-Based Encoding

	1234567890123456
david	0000110100001101
alice	1000001000
bob	10100000
frank	001001
heidi	1001
eve	10
grace	1

(e) Preserving Sort Order

	1234567890123456
*david	0000110100001101
alice	1000001000
bob	10100000
eve	010000
frank	01001
grace	010
heidi	11

Table 2: Example Decoding of Overlay Bitmap Parts in a Lossless Fashion

(a) Original Encoding

	1234567890123456
alice	1000000001000000
bob	1010000000000000
heidi	1000000010000

(b) Step-I

	1234567890123456
alice	**1**000000001000000
bob	1010000000000000
heidi	1000000010000

(c) Step-II

	1234567890123456
alice	**1**000000001000000
bob	**1**010000000000000
heidi	1000000010000

(d) Step-III

	1234567890123456
alice	**1**000000001000000
bob	**1**0**1**00000000000
heidi	1 00000010000

(e) Step-IV

	1234567890123456
alice	**1**000000001000000
bob	**1**0**1**00000 000000
heidi	1 00000 010000

bitmap part, "david", is encoded first, followed by the other bitmap parts arranged in lexicographic order. Choosing to encode the most dense bitmap part first enhances compression, while sorting the remaining parts allows for efficient evaluation of $<$ or $\leq$ predicates. The out-of-order bitmap parts should be flagged so that they can be identified during decoding.

To determine the row offsets where column values are less than or equal to "carol" using the encoding in Table 1e, we begin by decoding the "david" bitmap part, noting that it is out of order. Since "david" is not less than or equal to "carol" lexicographically, we proceed to the next parts, "alice" and then "bob", both of which meet the $\leq$ "carol" condition. We stop processing before reaching "eve", as it exceeds the predicate. Decoding "alice" yields the bitmap 1000000001000000, while "bob" results in 0101000000000000. By computing the bitwise OR between these two bitmaps, we get 1101000001000000, indicating that rows at offsets 1, 2, 4, and 10 satisfy the $\leq$ "carol" predicate.

These concepts can be readily adapted to handle other types of comparison operators. For efficiently evaluating predicates such as $>$ or $\geq$, we consider creating an overlay bitmap encoding in reverse lexicographic order. To support the full range of comparison operators, we could maintain two separate overlay bitmap encodings: one sorted in forward lexicographic order and the other in reverse (in Section 4, we explain how to use the overlay bitmap encoding as a secondary index). This twofold encoding approach allows for quick and effective evaluation of both ascending and descending comparisons.

It is important to note that in all the encoding variations presented from Table 1c to Table 1e, there is a distinct property in the bitmap parts. The overlay bitmap parts at the upper levels tend to have more regions with contiguous zeros, while those at the lower levels are more likely to have regions with contiguous ones. This characteristic can be leveraged to apply run-length encoding, as described by *Golomb* in 1966 [23], allowing for further compression of the overlay bitmap parts.

Aside from the obvious compression benefits, using run-length encoding for overlay bitmap parts offers the advantage of more efficient scanning for set bits. In cases where only M out of N bits are set — and M is much smaller than N — the process of locating these set bits approaches a computational complexity of $O(M)$. This is significantly more efficient than the straightforward method of scanning each bit individually, which has $O(N)$ complexity.

Shift Vectors: The techniques described so far emphasize the removal of blank spaces from bitmap parts during encoding and the insertion of blank spaces during decoding. These processes require fast positional access to individual bits to evaluate their values. However, common data structures do not efficiently support both deletion/insertion and fast positional access.

For instance, standard vectors offer $O(1)$ time complexity for positional access, but inserting or deleting elements involves an $O(N)$ time penalty [26]. Sets allow for efficient insertions and deletions with a complexity of $O(\log(N))$, yet accessing an element by position requires $O(N)$ time, as it necessitates traversing elements up to the desired position. Hash tables also offer efficient insertions and deletions with $O(1)$ complexity, but they do not support positional access. Linked lists can handle insertions and deletions in constant time, $O(1)$, but require $O(N)$ time for positional access due to the need to traverse elements sequentially. As a result, these data structures are not suitable for efficiently encoding and decoding overlay bitmap parts.

To address this problem, we introduce an auxiliary data structure called a *shift vector* that offers vector-like properties with $O(\log(N))$ positional access without having the need to shift elements every time elements are deleted from (or inserted into) the shift vector. We achieve these properties by building a segment tree on top of the physical vector (array) positions that maps logical positions to physical array offsets. The leaf nodes of the tree contain physical array offsets while the non-leaf nodes contain, for each child node, information about the number of array positions that are currently occupied within each sub-tree rooted at that child node. The tree can be a binary tree or have a fixed fan-out. Furthermore, the leaf nodes can be integers but also bitmaps. The tree is a complete tree and because it also has a fixed fan-out (2 or more children), then the tree itself can be represented as a vector

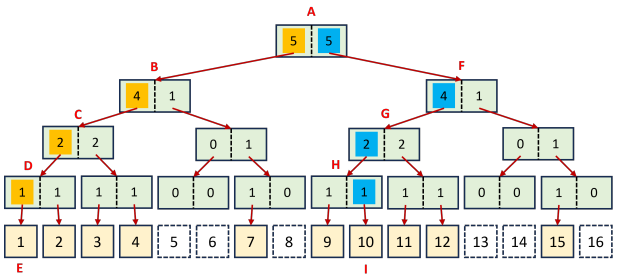


Figure 1: A shift vector to efficiently process overlay bitmaps

without having the need to maintain links. For the sake of presentation, we use links in the shift vector examples in the paper even though we use a more optimized version in our actual prototype implementation.

When decoding the parts of an overlay bitmap, we use a shift vector to manage the relationship between the logical bit positions in an overlay bitmap part and the physical row offsets these bits represent. The logical positions are those depicted in an overlay bitmap part after blank spaces have been removed during encoding.

For example, consider decoding the overlay bitmap shown in Table 1e. Figure 1 illustrates the state of the shift vector after fully decoding the bitmap part for "david". In this depiction, the leaf nodes of the tree represent physical row offsets, while the non-leaf nodes indicate the counts of leaf nodes within the subtree accessible from those nodes.

While traversing the bits for the "david" bitmap part, we encounter logical positions 5, 6, 8, 13, 14, and 16. These positions correspond to the same physical row offsets because "david" is the first bitmap part in the encoding, where no blank spaces were initially present. As these logical positions are processed, they are removed from the shift vector. This leads to the shift vector configuration depicted in Figure 1.

Next, we begin processing the bitmap part for "alice". The first bit that is set is found at logical position 1. To find its corresponding physical row offset, we consult the shift vector. This process starts at the root node of the tree, labeled A. The root node displays two counts: 5 for its left subtree and 5 for its right subtree. Since the position we are pursuing (1) is less than or equal to the left subtree's count (5), we move down to the left child, node B. Node B has counts of 4 and 1 for its left and right subtrees, respectively. The logical position we seek (1) is less than or equal to the count for the left subtree (4), prompting us to descend to the left child, node C. This path continues downward through nodes C, D, and finally E. Node E is a leaf node and contains the physical offset 1, indicating that the value "alice" is located in the first row offset. Finally, before moving to the next set bit in the "alice" bitmap part, we must remove the leaf node E from the shift vector. This requires adjusting the relevant counts in nodes D, C, B, and A as we trace back up from node E to the root.

The next bit set in the "alice" bitmap part is at logical position 7. When looking up this position in the shift vector, we must account for the fact that one preceding position has already been removed. Therefore, we actually need to look up position $(7 - 1 = 6)$. At node A, the current counts are 4 and 5 for its left and right subtrees, respectively. Since 6 is greater than 4, we descend into the right subtree, node F, and adjust our lookup position to $(6 - 4 = 2)$.

At node F, the position 2 is less than or equal to the count of its left subtree (4), so we move to the left child, node G. Similarly, at node G, 2 is less than or equal to its left subtree's count (2), prompting us to descend to node H. Here, 2 is greater than the count of H's left subtree (1), so we move to the right child, node I. Node I is a leaf and contains the physical row offset 10, indicating that the value "alice" appears at this offset. With no further bits set in the "alice" bitmap part, we proceed to process the next bitmap part in the sequence, after removing node I and adjusting the counts at nodes H, G, F and A in the shift vector.

Algorithm 1 Decoding Overlay Bitmap Parts

```
1:  if parts.empty() then
2:      return nullptr;
3:  end if
4:  nBits ← parts[0].getNBits();
5:  rv ← allocateResultVector(nBits);
6:  sv ← allocateAndInitializeShiftVector(nBits);
7:  for i = 0; i < parts.size(); i++ do
8:      nDeleted, cursor ← 0;
9:      while true do
10:         cursor ← advanceCursor(cursor, parts[i]);
11:         if cursor == EOF then
12:             break;
13:         end if
14:         pos ← cursor − nDeleted;
15:         offset ← sv.getExpandedPositionAndDelete(pos) − 1;
16:         rv[offset] = parts[i].getToken();
17:         nDeleted++;
18:     end while
19: end for
20: return rv;
```

Algorithm 1 provides a formal description of the process for decoding overlay bitmap parts. The `advanceCursor` function identifies the next set bit in a given bitmap part or returns `EOF` if all the set bits have been processed. As previously discussed, with run-length encoding, the time complexity for finding the next set bit in a bitmap part is approximately $O(M)$, where M is the number of bits set in that bitmap part.

Each invocation of the `getExpandedPositionAndDelete` function has a computational complexity of $O(\log(N))$, where N is the number of leaf nodes in the shift vector, equivalent to the row group size. Given that a total of N bits are set across all bitmap parts combined (with each bit being exclusive), the overall computational complexity of the decoding process amounts to $O(N \log(N))$, where N represents the row group size.

Algorithm 2 provides a detailed approach for efficiently encoding overlay bitmap parts within a column chunk using a shift vector. In the initial steps of the algorithm (lines 3–7), the input is transformed into a map of vectors. Each vector represents a unique value in the column chunk and records the 1-based row offsets where this value is present. Subsequently, line 8 arranges these unique values in a preferred sequence, which might be the scan order as shown in Table 1c, a density-based order as illustrated in Table 1d, or a sort order like in Table 1e. This arrangement is based on the strategies discussed earlier in the section. Finally, lines 9–19 involve the construction of overlay bitmap parts according to the predetermined sequence. During this process, the shift vector plays a crucial role in reverse lookup (imagine that the links are reversed), determining

Algorithm 2 Encoding Overlay Bitmap Parts

```
1:  ob ← createEmptyOverlayBitmap();
2:  sv ← allocateAndInitializeShiftVector(chunk.size());
3:  hashtable<string, vector> positionMap ← ∅;
4:  for i = 0; i < chunk.size(); i++ do
5:      itr ← positionMap.insert(chunk[i], {}).first;
6:      (*itr).push_back(i + 1);
7:  end for
8:  vector<string> order ← organizeParts(positionMap);
9:  for i = 0; i < order.size(); i++ do
10:     nDeleted ← 0;
11:     obp ← createEmptyOverlayBitmapPart();
12:     itr ← positionMap[order[i]].begin();
13:     for ; itr ≠ positionMap[order[i]].end(); itr++ do
14:         offset ← sv.getReducedPositionAndDelete(*itr);
15:         obp.setBit(offset + nDeleted);
16:         nDeleted++;
17:     end for
18:     ob.push_back(obp);
19: end for
20: return ob;
```

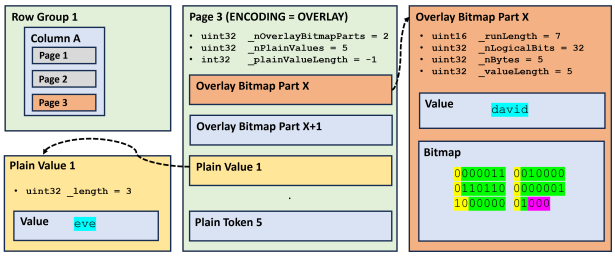


Figure 2: Data pages serialized using the `OVERLAY` encoding

the logical position associated with a given physical row offset in the current overlay bitmap part being constructed.

Algorithm 2 has a computational complexity of $O(N \log(N))$ where N is the number of values in the column chunk. In the algorithm described, the computational complexity for executing lines 4 through 7 is $O(N)$, and for line 8, it is $O(N \log(N))$. Notably, lines 9 through 19 also have a complexity of $O(N \log(N))$. The loop in these lines executes $O(N)$ times, and each invocation of the `getReducedPositionAndDelete` function incurs an overhead of $O(\log(N))$, as previously detailed. It is important to emphasize that line 15, which involves setting bits in the overlay bitmap part in a streaming fashion—that is, without ever setting a lower-order bit—results in a computational overhead of $O(1)$. This efficient overhead is achievable when the overlay bitmap part is encoded using run-length encoding, adding to the overall efficiency of the process.

4 FILE FORMAT CHANGES

We propose two types of enhancements to the Apache Parquet file format, which can coexist seamlessly. The first enhancement is backward-compatible while the second enhancement is both backward and forward-compatible.

Backward Compatibility: The first enhancement introduces a new encoding called `OVERLAY`. This encoding expands the current set of encodings supported by Apache Parquet, such as `PLAIN` and `RLE_DICTIONARY`. Built on the *overlay bitmap encoding* discussed in Section 3, it serves both as a foundational columnar representation and as an efficient indexing tool for evaluating predicates (=, <, ≤, >, ≥, and ≠).

Additionally, we propose a hybrid layout as part of this enhancement, combining the strengths of the discussed methods in Section 3 with the straightforwardness of plain encoding. This results in a compact representation that maintains efficient predicate evaluation, even for column chunks containing a large number of distinct values.

Importantly, these format extensions are backward-compatible. This means that implementations of Apache Parquet readers and writers designed to handle the new encoding will also be able to process Apache Parquet files in the older format using existing encodings. However, the extension is not forward-compatible, indicating that older implementations will not be able to process Apache Parquet files featuring column chunks encoded with the `OVERLAY` encoding (i.e., they will return an error).

Consider the physical data layout shown in Figure 2. In "Row Group 1", the column chunk for "Column A" is encoded using the `OVERLAY` encoding and is serialized across three physical pages. Each page includes details about the number of overlay bitmap parts serialized on that page (`_nOverlayBitmapParts`) and the number of plain values (`_nPlainValues`). All overlay bitmap parts are serialized before the plain values, so only one page might have both `_nOverlayBitmapParts` and `_nPlainValues` with positive values. Furthermore, overlay bitmap parts are serialized in the order they are encoded in the overlay bitmap.

When encoding a column chunk as an overlay bitmap, only the top-K most frequently appearing values in the column chunk are considered, where K is a user-defined threshold. Regardless of the top-K threshold, values that appear only once (singletons) are not included in the overlay bitmap, as encoding them would waste space. After removing all values represented by the overlay bitmap, the remaining plain tokens are serialized in the order they initially appear in the column chunk.

Details about the organization of overlay bitmap parts during encoding—such as whether scan order, density-based ordering, or sort order was used—are recorded in the first page of the column chunk in the row group. This information can be dynamically processed when an overlay bitmap is deserialized and decoded.

Figure 2 illustrates how an overlay bitmap part is serialized in detail. Each serialized overlay bitmap part includes the following components: a 2-byte unsigned integer that indicates the run-length used during encoding; a 4-byte unsigned integer representing the number of logical bits covered by the overlay bitmap part (this number differs from the actual serialized physical bits due to run-length encoding and padding); a 4-byte unsigned integer indicating the total number of bytes occupied by the bitmap's payload (B); a 4-byte unsigned integer specifying the size in bytes of the value represented by the overlay bitmap (S); a byte array of size S that contains the value; and a byte array of size B that holds the actual bitmap.

The value payload in Figure 2 contains the value associated with the overlay bitmap part, and has exactly $\_valueLength = 5$ bytes. The bitmap payload is organized in the following way: the data are divided into segments of $\_runLength = 7$ bits each, with the exception of the last segment, which may contain fewer bits. If a segment begins with a zero bit, it indicates that run-length encoding is not used, and the subsequent $7 - 1 = 6$ bits are copied directly during deserialization. In the example from Figure 2, the first four

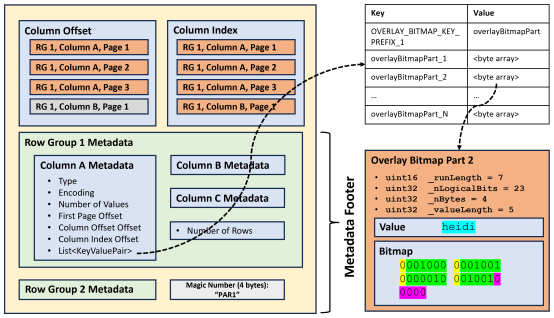


Figure 3: `OVERLAY` encoding used as a secondary index

segments start with a zero bit, meaning the first $4 \times 6 = 24$ bits in the decoded bitmap should be: `000011 010000 110110 000001`.

If a segment starts with a one bit, as with the fifth segment, it signifies that run-length encoding is employed. The next bit being zero indicates the run is filled with zero bits (it would be filled with ones otherwise). The remaining `00000` in the section indicates that $0 + \_runLength = 7$ consecutive zeros follow, resulting in a total of $24 + 7 = 31$ bits.

Finally, the sixth and last segment begins with a zero, showing that run-length encoding is not used, so the remaining bits must be copied directly. Given that $\_nLogicalBits = 32$ and we've already processed 31 bits, we need to copy just one more bit, which is set to one (highlighted in green in Figure 2). The following three bits, highlighted in purple, are padding bits that bring the total bitmap payload to 5 bytes or $5 \times 8 = 40$ physical bits. The final decoded bitmap is: `000011 010000 110110 000001 000000 01`.

The encoding method for plain values in a column chunk that are not covered by any overlay bitmap parts varies according to the column's data type. For fixed-length columns, such as `INT32`, `INT64`, and `FIXED_LEN_BYTE_ARRAY`, the byte length is stored in the *_plainValueLength* field on the first page containing plain values (as shown in Figure 2). In contrast, for variable-length columns like `BYTE_ARRAY`, the *_plainValueLength* field is set to -1. Instead, the *_length* field within each plain value specifies the actual length of that value's payload.

Forward Compatibility: To solve the forward-compatibility issue, our second enhancement utilizes Apache Parquet's ability to store additional key-value pairs (as metadata) to manage overlay bitmaps linked to column chunks. This extension lets each overlay bitmap serve as a secondary index atop the existing base columnar layout, which can still utilize traditional encodings or the `OVERLAY` encoding if preferred. We represent the overlay bitmap as key-value pairs using the same format introduced in the first extension (Figure 3). Instead of integrating overlay bitmap parts into data pages, we record the serialized byte stream as values associated with special or reserved keys.

These second extensions are forward-compatible, allowing existing Apache Parquet readers and writers to handle files created with these enhancements: while predicate evaluation won't be improved on older systems (as the overlay bitmaps would be ignored), older systems would seamlessly process new files without errors.

With these metadata extensions, users can create multiple secondary indexes using the `OVERLAY` encoding. As illustrated in Figure 3, the reserved key `OVERLAY_BITMAP_KEY_PREFIX_1` marks the key prefix used for the $1 \ldots N$ overlay bitmap parts. To create M secondary indexes with the `OVERLAY` encoding, users can specify values for each `OVERLAY_BITMAP_KEY_PREFIX_{i}` where $i \in \{1 \ldots M\}$. This capability is useful for handling a wide range of predicate types. For instance, a user might create one index with overlay bitmaps arranged in forward sort order to efficiently evaluate the $<$ and $\leq$ comparators. Simultaneously, another index could be organized in reverse sort order to optimize the execution of the $>$ and $\geq$ comparators.

5 IMPLEMENTATION DETAILS

To enable encoding and decoding with the newly proposed overlay bitmap encoding, we have made several enhancements to the Apache C++ libraries [7]. We have added a new value, `OVERLAY`, to the `parquet::Encoding::type` enumeration. Additionally, two classes have been implemented: `OverlayEncoder` inherits from `EncoderImpl` and `TypedEncoder`; and `OverlayDecoder` inherits from `DecoderImpl` and `TypedDecoder`. We have also updated the `MakeEncoder` function that returns a `std::unique_ptr<Encoder>` to accommodate the new encoding type. With these enhancements, users can establish custom `parquet::WriterProperties` like this:

```
writerProps = parquet::WriterProperties::Builder()
 .version(parquet::ParquetVersion::type::PARQUET_2_0)
 ->compression(this->GetCompressionCodec())
 ->encoding(parquet::Encoding::OVERLAY)
 ->build();
```

Consequently, any Apache Parquet writer initialized with these settings will produce column chunks using the overlay bitmap encoding. Similarly, Apache Parquet readers will automatically recognize when a column chunk is encoded using this new method and will accordingly utilize the `OverlayDecoder` class.

In our proof-of-concept implementation within SAP IQ, we have enhanced the load and unload engines to leverage these new extensions. To unload a database table into an Apache Parquet file, users can execute a statement similar to the following:

```
unload select * from lineitem into file '<filename>'
 format parquet compression 'GZIP'
 row group size 16384 max parallel degree 4;
```

If the temporary database option "Use_Overlay_Encoding" is enabled before running the unload statement, SAP IQ creates a `WriterProperties` object with the `OVERLAY` encoding specified. It then constructs the appropriate `arrow::FileWriter` objects for use during the unload process. Additionally, SAP IQ supports the parallel unloading of a table. In this mode, work units—each responsible for writing a row chunk—can be distributed across multiple threads, enhancing efficiency and performance. Users have the option to specify additional compression methods, such as GZIP or SNAPPY, as shown in the example above. They can also control the size of the row groups being generated, allowing for more tailored performance and storage optimization. In our evaluations, we experiment with several of these parameters to assess their impact.

To load data into a table from a set of Apache Parquet files, users can execute a statement like this:

```
load table region (r_regionkey, r_name, r_comment)
 from '<filename1>', '<filename2>' format parquet escapes off;
```

The load engine automatically creates `ParquetFileReader` objects and manages parallelism according to the `-iqnumbercpus` server

Algorithm 3 Extending the Apache Parquet C++ Libraries to Support Predicate Evaluation Using the `OVERLAY` Encoding

```
1:  function ReadNext(RecordBatch* out)
2:      foundSet ← initFoundSet(size);
3:      for i = 0; i<readers.size(); i++ do
4:          if predicateNotApplicable(columns[i]) then
5:              continue;
6:          end if
7:          readers[i].nextBatch(size, columns[i], &foundSet);
8:      end for
9:      rewind();
10:     size ← min(size, foundSet.count());
11:     for i = 0; i<readers.size(); i++ do
12:         readers[i].useFoundSet(foundSet);
13:         readers[i].nextBatch(size, columns[i]);
14:     end for
15:     table ← Table(schema, columns);
16:     tableBatchReader ← TableBatchReader(table);
17:     tableBatchReader.readNext(out);
18: end function
```

startup option. Work units responsible for reading row chunks utilize the `RowGroupRecordBatchReader` class to fetch the next set of rows from the row group, which in turn calls the appropriate leaf readers for each column chunk. These readers automatically identify the encoding used and, if necessary, call the appropriate `OverlayDecoder` classes to decode the data efficiently. This essentially means that the load engine can successfully process Apache Parquet files in which different column chunks use different encodings.

In our prototype, users can choose to select only a subset of rows from Apache Parquet files when all column chunks are encoded with the `OVERLAY` encoding. To achieve this, they can configure the temporary database option `Overlay_Filters` with a string containing a list of predicates before executing the load statement. For example:

```
set temporary option Overlay_Filters =
 'l_shipdate<="1996-11-20" AND
  l_commitdate="1996-11-10" AND
  l_receiptdate="1996-11-25"';
```

To support this feature, we have extended the `ReadNext` method in the `RowGroupRecordBatchReader` class within the Apache Parquet C++ libraries. In Algorithm 3, lines 11–17 (excluding line 12) illustrate the existing behavior, while lines 2–10 plus line 12 represent our modifications.

In the existing implementation, the `ReadNext` method operates by iterating through each column and invoking the `nextBatch` function on the leaf reader corresponding to each column. These functions eventually call the appropriate decoder classes, including the newly added `OverlayDecoder`, to produce vectors of values. These values are then combined into a table snippet, as shown in lines 15–17, and returned to the caller, which, in our case, is SAP IQ's load engine.

To facilitate predicate evaluation within the leaf readers, we use a two-stage approach. In the first stage (lines 2–8), for each column to which any given predicate applies, we execute an overloaded version of the `nextBatch` function on the leaf readers. These calls are translated into overloaded versions of the `Decode` functions in the `OverlayDecoder` classes. Instead of producing a vector of values, these functions create a bitmap called the `foundSet`, which starts as a bitmap of all 1s (line 2), suggesting that all rows initially match (meaning no filtering is applied). As additional columns are processed, the `foundSet` is progressively adjusted to reflect the rows matching the applicable predicates. Once the `foundSet` is fully computed, we rewind all iterators, reset heap pointers, and then proceed to the second stage. As future work, we plan to explore the ordering of predicates during evaluation, based on selectivity estimation. This strategy has the potential to enhance performance; however, it is a research topic on its own.

In the second stage, we set a special flag on the leaf reader object to specify that only the rows corresponding to the offsets indicated by the `foundSet` bitmap should be returned when the `nextBatch` function is called (line 12). This directive is propagated down to the `OverlayDecoder` class and is utilized when executing Algorithm 1.

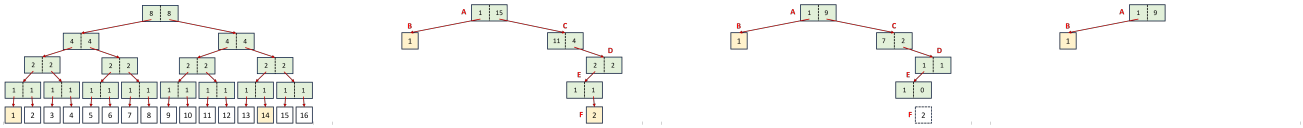
Pruned Shift Vectors: To efficiently decode overlay bitmap parts in the presence of a `foundSet` bitmap, we construct a pruned shift vector. Consider the overlay bitmap encoding in Table 1e. Assume the `foundSet` is `10000000 00000100`, meaning only the first and fourteenth bits are set. Figure 4a shows the original shift vector, with the first and fourteenth leaf nodes highlighted. As the first step of pruning, we remove all remaining leaf nodes. It's important to note that eliminating redundant nodes results in the adjustment of physical offsets within the remaining leaf nodes. We begin to percolate counts up within non-leaf nodes of the same subtree, provided this does not change the logical position of existing leaf nodes and reduces the number of levels within the subtree. Finally, in Figure 4b, we shift counts from one subtree to an adjacent subtree when it doesn't affect the logical position of any leaf nodes.

Let's demonstrate how Algorithm 1 operates on the pruned shift vector (Figure 4b). We begin with the first overlay bitmap part corresponding to "david" (Table 1e). When using the `getExpandedPositionAndDelete` function with logical offsets 5, 6, 8, and 13, we find that only nodes A and C are traversed. It is apparent that the physical offsets (i.e., leaf nodes) we seek have been pruned, which is confirmed by node C lacking a left subtree. Upon removing these four logical positions, the tree is adjusted such that the right counter in node A becomes 11, and the left counter in node C becomes 7.

To locate the physical offset corresponding to the logical position 14 in the current overlay bitmap part, we traverse the tree through nodes A, C, D, E, and finally reach the leaf node F. The value stored in this leaf node indicates that "david" should appear at the second offset in the output vector. We then remove this node from the tree, adjusting the counts as we move back up to the root. After this adjustment, the right counter in node A becomes 10, the right counter in node C becomes 3, the left counter in node D becomes 1, and the right counter in node E becomes 0. Finally, we address the bit at logical position 16. We traverse down the path A, C, D, and stop, realizing that node D's right subtree does not exist. Upon updating the counts, we obtain the shift vector depicted in Figure 4c. Because the right subtree in the remaining shift vector lacks any reachable leaf nodes, it can be entirely pruned by moving the counts up to the root. This results in the shift vector shown in Figure 4d.

We then proceed to the next overlay bitmap part in Table 1e, which is "alice". In this part, the first and seventh logical bits are set. We locate the first bit on the pruned shift vector in Figure 4d and identify the corresponding physical offset as 1 in the leaf node. This indicates that "alice" should occupy the first position in the output vector. After deleting the leaf node and adjusting the counts in the tree, we find there are no longer any reachable leaf nodes, meaning we can stop decoding. The output vector, therefore, contains the values "alice" and "david", respectively.

Figure 4: Pruning shift vectors to efficiently decode values matching a found set



(a) Original Shift Vector **(b) Pruned Shift Vector** **(c) After decoding "david"** **(d) After further pruning**

In the first stage of the approach described above, overlay bitmaps are utilized to efficiently compute the `foundSet` bitmap without the need to decode all bitmap parts. In the second stage, the `foundSet` bitmap aids in pruning the shift vector, allowing for efficient decoding of the overlay bitmap parts. Experiments conducted with the SAP IQ engine demonstrate significant improvements in load and query execution times when predicate evaluation is pushed down to Apache Parquet using the overlay bitmap encoding, which are detailed in the next section.

6 EVALUATION

In this section, we present the results of our experimental evaluation. The experiments were conducted on an SAP Converged Cloud virtual machine (g_c32_m128) equipped with an Intel®Xeon®Platinum 8260 2.40GHz CPU featuring 32 cores, 128 GiB of VMware Virtual RAM, and a 1 TiB virtual disk with ample free space at the time of testing. The operating system used for our experiments was SUSE Linux Enterprise (SLE) 15 Service Pack 5 (SP5). Our prototype was developed on a fork of the latest development branch of SAP IQ, which aligns with the HDLRE 2025-QRC4 release, though our version includes additional modifications. The internal architecture of SAP IQ utilizes the Apache Arrow 0.17.1 libraries, which we extended to incorporate support for the OVERLAY encoding. The experiments were executed with SAP IQ configured with the following parameters: iqmc=512, iqtc=512, iqnumbercpus=4.

The first experiment aims to evaluate the compressibility of the overlay encoding and analyze its impact on load (decode) and unload (encode) performance. To achieve this, we used the TPC-H benchmark dataset, available in CSV format. Initially, the entire dataset was loaded into SAP IQ tables using SAP IQ's existing load functionality. Subsequently, we unloaded these tables as Apache Parquet files, employing the newly implemented overlay encoding for all columns. Following this, the tables were truncated and reloaded from these generated Apache Parquet files. The 22 benchmark queries were executed both before and after reloading the tables, and the results were compared to verify the correctness of our implementation as well as to ensure that the data were not getting corrupted during the unload nor load processes.

To support filters with both equality (=) and less than or equal to (≤) comparators, we applied the encoding scheme described in Table 1e where values are sorted in ascending order. Equally, we could have supported the (≥) comparator by sorting in reverse order. The overlay encoding was configured to create row groups of size 16,384, and to consider up to 128 of the most frequent tokens within each row group as candidates for the overlay bitmaps, while encoding less frequent tokens as plain. It is important to remind the reader that in our approach, singleton tokens (those appearing only once within a row group) are always encoded as plain, irrespective of the maximum number of tokens allowed as overlay bitmaps. We conducted the experiments across multiple settings: (i) at scale factors 1, 10, and 100; (ii) additionally using DICTIONARY encoding; and (iii) with secondary compression using either the SNAPPY or GZIP algorithms. Apache Parquet pages are further compressed because encoding and compression solve different problems; encoding optimizes for the logical structure of the data, while compression targets redundancy at the raw byte level. By compressing these plain and encoded sections further, we ensure efficient storage and optimize I/O, and demonstrate that our overlay encoding approach can seamlessly integrate with standard industry practices. Unless explicitly stated otherwise, we use the default threshold for dictionary encoding, which stores up to 1 MiB worth of tokens per row group before reverting to plain.

Figure 5a shows, for the entire dataset, the total size of the Apache Parquet files produced using different encodings combined with gzip compression. As expected, the dictionary encoding offers effective compression. However, the overlay encoding achieves even better compression, which is noteworthy because it serves both as a fundamental columnar representation—similar to what the dictionary encoding does—and as an index, which is something that dictionary encoding cannot provide off the shelf.

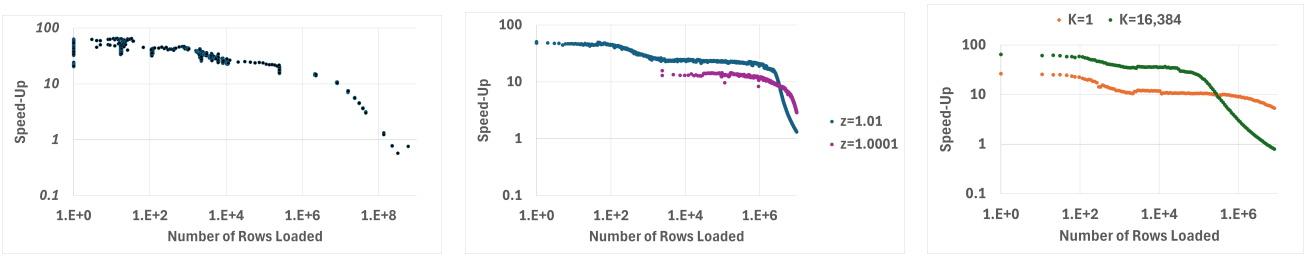
Figure 5b illustrates the impact of the additional compression used on top of the encoding. For clarity, we focus on the LINEITEM table at scale factor 100, the largest table in the dataset, although our findings apply across all tables and scale factors. When using both the snappy and gzip algorithms, overlay encoding achieves better compression than dictionary encoding. This flexibility allows users to choose between these compression algorithms to balance compressibility with decode speed (with snappy being faster).

Figures 5c and 5d show the end-to-end load and unload times for the lineitem table using different encodings and scale factors. The load performance with dictionary encoding is similar to that from csv files. Meanwhile, the load performance with overlay encoding is quite comparable to dictionary encoding, with up to 35 percent increase in execution time. During loading, maximum I/O reaches 108 MiB/second, and with Max_Query_Parallelism set to 256 threads, the memory overhead is 1.4 GiB. Our technique ensures a low and constant memory overhead of up to 5.6 MiB per thread, as overlay data structures are created for each row group and removed after processing. Although unloading with overlay encoding is about twice as slow as with dictionary encoding, this extra time is justified: the overlay encoding allows for filtered loads, which can significantly reduce load times by up to two orders of magnitude, as we will discuss as part of our second experiment.

Figure 5: Comparison of Compressibility, Load and Unload Times Using the Entire or Parts of the TPC-H Dataset

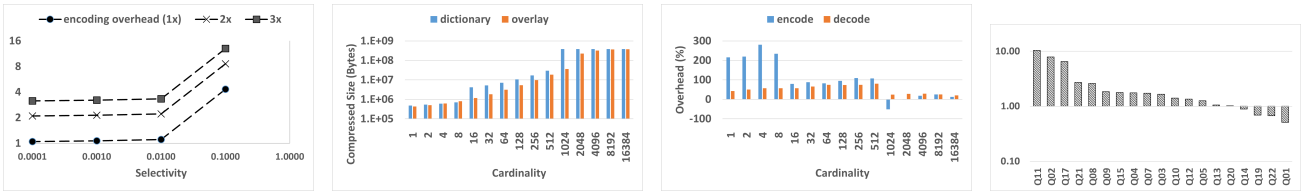
(a) TPC-H Compressibility (b) Lineitem Compressibility (c) Lineitem Load Times (d) Lineitem Unload Times

Figure 6: Filtered Loading Using OVERLAY Encoding at Various Selectivities



(a) TPC-H Lineitem Table at Scale Factor 100 (b) Skewed Datasets (K=128) (c) Skewed Datasets (z=1.01)

Figure 7: Additional Evaluations



(a) Break-Even Analysis (b) Compression (c) Encode/Decode Overhead (d) Query Speed-Up

The second experiment builds on the first by adding a step where the lineitem table is partially loaded from the same set of Apache Parquet files. This time, however, we use a filtered load expression with the previously mentioned database option, *Overlay_Filters*. We execute thousands of filter expressions on the columns *l_shipdate*, *l_commitdate*, and *l_receiptdate*. While our prototype exploits existing zone maps wherever applicable, the dataset is not explicitly clustered along any of these columns, which makes them categorically inapplicable for this experiment.

After the execution of each filtered load statement, we verify the table contents for correctness before truncating it and moving on to the next statement. We generate filters using both equality (=) and less than or equal to (≤) comparators, ensuring a balanced representation of filters across one, two, and three columns. During our experiments, we cover the entire selectivity spectrum, relying solely on ≤ comparators for non-selective statements, as filters with = comparators are not effective in this selectivity space due to the uniform distribution of values in the TPC-H benchmark [16]. The experiment is conducted at scale factors 1, 10, and 100.

Figure 6a illustrates the speed-up achieved using filtered load statements compared to loading the entire dataset encoded with the DICTIONARY encoding at TPC-H scale factor 100. Across all three scale factors, a speed-up of up to two orders of magnitude is observed for selective statements. In the chart, the rightmost point represents the time needed to load the entire lineitem table without any filters using the OVERLAY encoding, resulting in speed-ups of less than 1—acknowledging that full scan with overlay encoding can be slower than dictionary encoding.

Executing a predicate requires scanning at least one of the sixteen columns of the lineitem table to determine the matching rows, which introduces some overhead to the execution of filtered load statements. This becomes evident with some of the non-selective statements, shown by data points just to the left of the rightmost one that have a speed-up of less than 1. In these cases, the end-to-end execution of the filtered load statement takes longer than scanning the entire dataset without any filters.

Based on the same reasoning, we would expect the speed-up for the most selective statements to not exceed 16, yet this is clearly not the case. In some instances, executing a filtered load statement is 86 times faster than using a non-filtered approach. Several factors contribute to this observation, but the key point is that for selective statements, the layered design of overlay bitmaps allows us to halt

the processing of pages beyond the overlay bitmap parts relevant to the predicate's execution.

Next, we calculate how many queries are needed in a workload to offset the indexing overhead introduced by overlay encoding, considering queries at different selectivity levels. For these calculations, we utilize the speed-up figures from Figure 6a: specifically, 1.3, 10, 15, and 22 for queries with selectivities of 0.1, 0.01, 0.001, and 0.0001, respectively. We evaluate three scenarios where the *extra* encoding overhead is 1, 2, and 3 times the time taken to encode using dictionary encoding. Additionally, we assume that zone maps cannot effectively prune pages for these queries; therefore, without row-level indexing, the dataset needs to be scanned almost entirely. As shown in Figure 7a, in the worst-case scenario, approximately a dozen queries at the 0.1 selectivity level are sufficient to justify the additional overhead of overlay encoding, which is quite promising.

In the third experiment, we study the impact of skew. To do this, we synthetically create a table with 16 columns and 10 million rows. Each column contains up to 16,384 distinct values (INT32), generated randomly following a Zipfian distribution. In the experiment, we adjust the Zipfian coefficient (z), which determines the relative frequency of items in the sequence—the higher the (z) value, the more pronounced the skew in the dataset. We generate and execute filtered load statements across the entire selectivity spectrum using $\leq$ comparators similarly to the approach used earlier.

Figure 6b shows the speed-up achieved using filtered load statements compared to loading the same dataset encoded with DICTIONARY encoding, following the same evaluation method as in Figure 6a. For selective statements (with a selectivity of ≤ 0.1), we observe greater speed-up as skew increases, which is expected. In overlay encoding, the top-K most frequent tokens in a row-group are represented using layered overlay bitmap parts, while the rest are encoded as plain tokens. For this selectivity space, evaluating filters on the overlay bitmap parts is more efficient than on the plain array of tokens, as overlay bitmaps act as indexes; whereas, for plain tokens, the found set must be determined by scanning the entire array. Similarly, when a found set is given, retrieving matching values directly from overlay bitmap parts is more efficient than scanning the plain token array. As skew increases in the dataset, a larger proportion of rows can be encoded within the overlay bitmap parts, enhancing performance. This benefit is complemented by better compression when more rows are encoded using overlay bitmap parts. In our experiments, we observed a 13 percent reduction in dataset size between (z=1.01) and (z=1.0001).

Figure 6c shows the results of repeating the experiment for a highly skewed dataset (z=1.01) with varying top-K thresholds. Here, K indicates the number of distinct tokens in the row group encoded with overlay bitmap parts instead of as plain. In a row group of size 16,384, setting K to this value allows all tokens, except singletons, to be encoded with overlay bitmaps. We observe that increasing the top-K threshold boosts speed-up for selective load statements but has the opposite effect for non-selective ones. We also report that increasing K from 1 to 16,384 results in about 9 percent reduction in the total size of the files produced.

Algorithm 1 has a computational complexity of $O(N \log(N))$, while decoding plain tokens has $O(N)$. Thus, encoding most tokens with overlay bitmap parts can complicate decoding when all or most need processing with non-selective statements using the $\leq$ comparator. This supports our choice to encode only the top-K frequent tokens as overlay bitmaps, similar to how dictionary encoding defaults to plain when the dictionary grows too large. Although not shown, we found that the ratio of K to row group size has impact on performance while row group size alone has negligible impact. Theoretically, increasing the row group size from 16,384 to 1 million rows increases the number of computations by 43% for a table with one billion rows. However, this also reduces the overhead associated with creating and destroying overlay data structures for each row group. In practice, we have observed negligible differences in performance across various row group sizes, provided the value of K remains constant.

In the fourth experiment, we evaluate how increasing cardinality affects both overlay and dictionary encoding. We conduct experiments across various dictionary configurations by adjusting the threshold for reverting to plain encoding and test K values of 128, 512 and 1024 which control the top most frequent tokens encoded as overlay. We create a char(32) column with uniformly random strings and manage the number of distinct strings up to 16,384, matching the row group size used. We generate a total of one million rows. Figure 7b illustrates the compression achieved with both dictionary and overlay encoding at K=1024. For a fair comparison, we configure the dictionary encoding to revert to plain encoding after roughly the same number of distinct tokens. In both cases, the row groups undergo additional gzip compression.

The results show that dictionary encoding operates in a step-wise manner because it encodes tokens at the byte level. In contrast, overlay encoding outperforms dictionary encoding, delivering a smoother increase in size as cardinality rises, even when the number of distinct tokens surpasses K=1024. On the other hand, dictionary encoding exhibits a sudden increase in size once the threshold is exceeded. This occurs because, in dictionary encoding, once the threshold is met, all tokens are encoded as plain, whereas in overlay encoding, the top-K most frequent tokens continue to be efficiently encoded. Figure 7c presents the percentage of additional overhead associated with overlay encoding and decoding compared to dictionary encoding. The chart shows that while the measured overhead fluctuates due to the step-wise nature of dictionary encoding, it remains well within the ranges used in our break-even analyses (see Figure 7a) even for high-cardinality columns. These results also apply to other K values with corresponding dictionary thresholds, differing mainly at the inflection points. Compared to K=128, overlay encoding is 2% and decoding is 5% slower at K=1024, and the overheads with respect to dictionary encoding remain within reported ranges. Due to space limitations, these charts are omitted.

In the fifth experiment, we evaluate queries using the overlay encoding. We start by manually scripting the execution of three TPC-H queries in SAP IQ, combining filtered load statements with applicable predicates, followed by the execution of the original query. We utilize generative AI [17, 34] to script most of the remaining TPC-H queries, excluding those that we currently do not support due to complex expressions such as "T_a.col1 $\neq$ T_b.col2." We then meticulously review and correct any AI-induced errors and confirm the accuracy of the query results. To optimize the data loaded into temporary tables during query execution, we implement semi-join filters using bit vectors. These filters are generated

during the execution of a filtered load statement and are employed inside Apache Parquet to streamline subsequent load statements.

We compare our prototype with the latest version of the SPARK SQL engine [10, 13], configuring both SAP IQ and SPARK to utilize a maximum of 16 vCPUs and 8 GiB of memory. Figure 7d illustrates the speed-up achieved by executing queries in SAP IQ using overlay bitmap encoding, relative to their execution in SPARK SQL over dictionary-encoded files. It is important to note that the execution times for SAP IQ are comprehensive; they include the creation of temporary tables, the time taken to partially load these tables, and the final execution of each query.

We make several key observations. For approximately three quarters of the queries, SAP IQ outperforms SPARK, which is encouraging. As queries increase in complexity—particularly those involving joins across multiple tables and multiple predicates—and become more selective, SAP IQ's performance advantage becomes more pronounced. This highlights the benefits of overlay indexing and its potential to fill a gap in Apache Parquet's capability to execute complex queries efficiently. For queries that are either not selective or involve complex aggregates with group-by operations, SPARK SQL proves to be faster. However, we believe this gap can be closed through two future enhancements: (i) implementing filtered load as a data-flow operator (DFO) within the SAP IQ engine to eliminate the cost of materialization, and (ii) by supporting pre-aggregation and group-by directly within Apache Parquet, similar to our support for executing predicates.

In summary, our experiments demonstrate that overlay encoding successfully meets its objectives: providing efficient indexing capabilities in Apache Parquet with less storage overhead than dictionary encoding, offering comparable scan times when no filters are used while maintaining an acceptable overhead for unload (decode) times. We demonstrate that our approach is resilient to skew, even benefiting from it, and can effectively manage high-cardinality columns. Our break-even analysis indicates that approximately a dozen less-selective queries are needed to justify the additional indexing overhead associated with overlay encoding. Moreover, we emphasize that overlay encoding does not need to be applied to all columns or row groups despite us doing so in our experiments.

7 DISCUSSION AND FUTURE WORK

The *overlay bitmap encoding* presents a complementary alternative to Apache Parquet's default encoding by building on the same principles that have fueled Apache Parquet's success [18, 27, 39, 49, 52] while offering additional benefits. We demonstrate that the proposed encoding effectively fills a gap in Apache Parquet that existing page-level indexes, such as min-max zone maps and column indexes, cannot fully address. The key difference between our approach and these existing indexes lies in the granularity of their operation. Overlay encoding offers row-level indexing, while existing indexes provide page-level indexing, and these two are orthogonal. This fine-grained indexing allows for the progressive filtering of rows during the execution of complex queries involving multiple predicates and joins across several tables. In contrast, page-level indexes require data to be clustered or organized to function efficiently. Although overlay indexing offers superior pruning capabilities for complex queries, it does not replace existing indexes in Apache Parquet. For instance, min-max zone maps can still be used to initially prune irrelevant pages, after which overlay indexes can be applied for more granular filtering. This combined approach is precisely how our prototype operates.

The system's compressibility leverages overlaying bitmap parts, which naturally integrates with run-length encoding to optimize storage. Combining overlay and plain encoding balances compressibility with indexing capabilities, ensuring versatility across data types. Importantly, evaluating predicates does not require decoding all bitmap parts; relevant rows can be efficiently decoded using pruned shift vectors, enhancing speed and reducing overhead.

We aspire to have overlay encoding integrated into the Apache Parquet standard file format. While this is an extensive process requiring formal discussions and stakeholder alignment, which we plan to pursue, our immediate focus is on implementing the proposed encoding to accelerate internal processes and pipelines while maintaining backward and forward compatibility, hence, our design decision to support two different types of extensions. This approach allows our customers to use data products [32] stored as Delta tables [11] backed by Apache Parquet files across various platforms, including SAP HANA Cloud [3, 40] and external ecosystems like Apache SPARK [9, 10, 13, 52], without compromising the openness of the file format. We aim to provide these tools as both open-source and proprietary additions to our internal products.

Looking ahead, we have several more concrete future work items planned as well. We aim to fully integrate support for overlay encoding into SAP IQ's query engine. In particular, we plan to implement the filtered load operation as a distinct DFO, which will enable us to avoid the materialization of intermediate results. Additionally, we plan to incorporate pre-aggregation directly within Apache Parquet, similar to how we execute predicates on overlay bitmaps. We are also considering integrating these techniques with Apache SPARK to further extend their applicability. We also see the potential to enhance overlay bitmaps by developing intra-bitmap parallelism, which involves processing contiguous bitmap parts within the same overlay bitmap through a fusion operation. This approach, paired with a method to mitigate I/O overhead by strategically grouping and adding metadata indexing, could significantly improve performance by potentially another order of magnitude.

8 CONCLUSIONS

The introduction of overlay bitmap encoding offers a compelling enhancement to Apache Parquet's existing capabilities [49], building upon its foundational principles while introducing new benefits. Overlay bitmap encoding offers advantages that traditional dictionary encoding cannot and achieves exceptional compressibility. By leveraging bitmap representations of row offsets for distinct values, overlay bitmap encoding provides precise row-level indexing, leading to efficient predicate evaluation push-down. This enables the retrieval of only relevant rows from a column chunk without false positives, enhancing load performance by up to two orders of magnitude during load and up to 10x during query execution. Our prototype implementation within the Apache Parquet C++ libraries and the SAP IQ engine showcases its practical benefits, with potential for broader application through adaptations in Java-based environments and Apache Spark integrations.

REFERENCES

[1] 2025. Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/. [Accessed: 2025-11-07].

[2] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1664–1665.

[3] Mohammed Abouzour, Güneş Aluç, Ivan T Bowman, Xi Deng, Nandan Marathe, Sagar Ranadive, Muhammed Sharique, and John C Smirnios. 2021. Bringing cloud-native storage to SAP IQ. In *Proceedings of the 2021 International Conference on Management of Data*. 2410–2422.

[4] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[5] Güneş Aluç. 2026. Overlay bitmaps for indexing and querying data. https://patents.google.com/patent/US12602363B1/en

[6] Gennady Antoshenkov. 1997. Dictionary-based order-preserving string compression. *The VLDB Journal* 6, 1 (1997), 26–39.

[7] Apache Software Foundation. 2025. Apache Parquet C++. https://github.com/apache/parquet-cpp. Accessed: November 16, 2025.

[8] Apache Software Foundation. 2025. Apache Parquet Java. https://github.com/apache/parquet-java. Accessed: November 16, 2025.

[9] Apache Software Foundation. 2025. Apache Spark. https://github.com/apache/spark. Accessed: November 16, 2025.

[10] Apache Software Foundation. 2025. Apache Spark SQL. https://spark.apache.org/sql/. Accessed: November 16, 2025.

[11] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[12] Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, et al. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[13] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1383–1394.

[14] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[15] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*. 2326–2339.

[16] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 61–76.

[17] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[18] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. 2019. Apache Hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*. 1773–1786.

[19] Chee-Yong Chan and Yannis E Ioannidis. 1998. Bitmap index design and evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of data*. 355–366.

[20] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 34–43.

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[22] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. 2007. Adaptive query processing. *Foundations and Trends® in Databases* 1, 1 (2007), 1–140.

[23] Solomon Golomb. 1966. Run-length encodings (corresp.). *IEEE Transactions on Information Theory* 12, 3 (1966), 399–401.

[24] Sasun Hambardzumyan, Abhinav Tuli, Levon Ghukasyan, Fariz Rahman, Hrant Topchyan, David Isayan, Mark McQuade, Mikayel Harutyunyan, Tatevik Hakobyan, Ivo Stranic, et al. 2022. Deep lake: A lakehouse for deep learning. *arXiv preprint arXiv:2209.10785* (2022).

[25] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.

[26] Donald E Knuth. 1997. *The Art of Computer Programming: Fundamental Algorithms, Volume 1.* Addison-Wesley Professional.

[27] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop.. In *Cidr*, Vol. 1. Asilomar, CA, 9.

[28] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data*. 5–17.

[29] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. 2018. Rios: Runtime integrated optimizer for spark. In *Proceedings of the ACM Symposium on Cloud Computing*. 275–287.

[30] Yinan Li, Jianan Lu, and Badrish Chandramouli. 2023. Selection Pushdown in Column Stores using Bit Manipulation Instructions. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[31] Szoke Mark-Andor. 2026. Modern alternatives to hive: A systematic review and single-node benchmark of SQL-on-Hadoop and lakehouse engines. *Information Systems* 136 (2026), 102635.

[32] Jürg Meierhofer, Thilo Stadelmann, and Mark Cieliebak. 2019. Data products. In *Applied data science: Lessons learned for the data-driven business*. Springer, 47–61.

[33] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[34] OpenAI. 2023. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2026-05-11.

[35] Oracle. 2023. Using Zone Maps. https://docs.oracle.com/en/database/oracle/oracle-database/21/dwhsg/using-zone-maps.html. [Accessed: 2025-11-07].

[36] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[37] Krishna Puttaswamy, Abhijit Chakankar, Tao Tao, Zaheera Valani, Ramesh Chandra, William Chau, Mengxi Chen, Akram Chetibi, Tianyi Huang, Jonathan Keller, et al. 2025. Delta Sharing: An Open Protocol for Cross-Platform Data Sharing. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5197–5209.

[38] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[39] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *BTW 2023*. Gesellschaft für Informatik eV, 235–258.

[40] Daniel Ritter, Mihnea Andrei, Sukhyeun Cho, Maik Görgens, Taehyung Lee, Norman May, Amit Pathak, and Paul R Willems. 2025. The HANA Native Query Engine for Lakehouse Systems. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4831–4845.

[41] Sandeep Singh Sandha, Wellington Cabrera, Mohammed Al-Kateb, Sanjay Nair, and Mani Srivastava. 2019. In-database distributed machine learning: demonstration using Teradata SQL engine. *Proceedings of the VLDB Endowment* 12, 12 (2019).

[42] SAP. 2023. Loading Parquet Files. https://help.sap.com/docs/hana-cloud-data-lake/administration-guide-for-data-lake-relational-engine-sap-hana-db-managed/loading-parquet-files. [Accessed: 2025-11-08].

[43] Tomer Shiran, Jason Hughes, and Alex Merced. 2024. *Apache Iceberg: The Definitive Guide: Data Lakehouse Functionality, Performance, and Scalability on the Data Lake.* O'Reilly Media, Inc.

[44] Mohamed A Soliman. 2018. Query Optimization Challenges for SQL-on-Hadoop. In *Encyclopedia of Big Data Technologies*. Springer, 1–10.

[45] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 33–40.

[46] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *ACM Sigmod Record* 53, 2 (2024), 21–37.

[47] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A decade of SQL analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[48] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon mode: Bringing the vertica columnar database to the cloud. In *Proceedings of the 2018 International Conference on Management of Data*. 797–809.

[49] Deepak Vohra. 2016. Apache Parquet. In *Practical Hadoop Ecosystem: a Definitive Guide to Hadoop-Related Frameworks and Tools*. Springer, 325–335.

[50] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.* 37, 1 (2014), 31–37.

[51] Tom White. 2012. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.".

[52] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.