Columnar Formats for Schemaless LSM-based Document Stores

Wail Y. Alkowaileet
University of California, Irvine
Irvine, CA
w.alkowaileet@uci.edu

Michael J. Carey
University of California, Irvine
Irvine, CA
mjcarey@ics.uci.edu

ABSTRACT

In the last decade, document store database systems have gained more traction for storing and querying large volumes of semi-structured data. However, the flexibility of the document stores' data models has limited their ability to store data in a column-major layout — making them less performant for analytical workloads than column store relational databases. In this paper, we propose several techniques based on piggy-backing on Log-Structured Merge (LSM) tree events and tailored to document stores to store data in a columnar layout. We first extend the Dremel format, a popular on-disk columnar format for semi-structured data, to comply with document stores' flexible data model. We then introduce a new columnar layout for organizing and storing data in LSM-based storage. We also highlight the potential of using query compilation techniques for document stores, where values' types are known only at runtime. We have implemented and evaluated our techniques to measure their impact on storage, data ingestion, and query performance in Apache AsterixDB. Our experiments show significant performance gains, improving the query execution time by orders of magnitude while minimally impacting ingestion performance.

PVLDB Reference Format:
Wail Y. Alkowaileet and Michael J. Carey. Columnar Formats for Schemaless LSM-based Document Stores. PVLDB, 15(10): 2085 - 2097, 2022.
doi:10.14778/3547305.3547314

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wailyk/column.

1 INTRODUCTION

In recent years, columnar storage systems have been widely adopted in data warehouses for analytical workloads, where typical queries access only a few fields of each tuple. By storing columns contiguously as opposed to rows, column store systems only need to read the columns involved in a query and the I/O cost becomes significantly smaller compared to reading whole tuples [42, 51]. As a result, open source and commercial relational column-store systems such as MonetDB [10, 54] (and the commercial version Actian Vector [1]), and C-Store [51] (commercialized as Vertica [17]) have gained more popularity as data warehouse solutions.

For nested data, Dremel [42] and its open source implementation Apache Parquet [6] offer a way to store homogeneous JSON-like data in a columnar format. Apache Parquet has become the de facto file format for popular big data systems such as Apache Spark and even for "smaller" data processing libraries like Python's Pandas. However, storing data in a column-oriented fashion for document store systems such as MongoDB [11], Couchbase Server [9] or, Apache AsterixDB [2, 24, 30] is more challenging, because: **(1)** Declaring a schema before loading or ingesting data is not required in document store systems. Thus, the number of columns and their types are determined upon data arrival. **(2)** Document store systems do not prohibit a field from having two or more different types, which adds another layer of complexity. Even though columnar systems are orders of magnitude more performant, many users have no choice but to use the slower yet flexible document stores.

In this paper, we show that users with such workloads can enjoy the performance gains from storing the data in a columnar format without sacrificing the flexibility of document stores. We achieved this by, first, proposing several extensions to the Dremel format to address its limitations to comply with document stores' flexible data model, which permits values with heterogeneous types and schema changes. Many prominent document stores, such as MongoDB and Couchbase Server, adopt Log-Structured Merge (LSM) trees [46] in their storage engines for their superior write performance. LSM lifecycle events (mainly the flush operations) allow transforming the ingested records upon writing them to disk. Thus, we use the techniques proposed in [22] to exploit the LSM flush operation to infer the schema and write the records (initially in row format) as columns using our extensions to the Dremel format.

We present a new model in our work here for storing columns in an LSM B^+-tree index. In this model, we stretch the B^+-tree leaf nodes to become mega nodes, where a leaf node occupies multiple pages. We refer to this model as the AsterixDB Mega-Attributes Across or AMAX for short. Despite its name, the AMAX layout is agnostic of the columns' structure and sees each column as a series of bytes; hence it should only require a few modifications to be adopted by other LSM-based document stores. In this paper, we evaluate the AMAX layout in terms of (1) ingestion performance and (2) query performance. In our extended version [23], we present and evaluate a Partitioned Attributes Across (PAX)-like format [21] called APAX, where each column occupies a contiguous region (called a minipage) within a B^+-tree's leaf page. We omit the details of the APAX format due to space limitations, and we refer interested readers to [23] for more information.

The goal of continuously reducing the I/O cost in disk-based databases is objectively justified (and is a focus of this paper). However, with the ever-growing advancements in storage technologies, the role of CPU cost becomes even more apparent. In our evaluation, we observe an interesting phenomenon in certain types of workloads, where we have been able to reduce the size of the data needed to process a query by several factors using the AMAX format as

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547314

compared to the vector-based format (a row-major format) from [22]. In some cases, the associated improvement to query execution time was negligible due to increased CPU cost. The dominant factor determining the CPU cost is the query execution model. Modern Database Management Systems (DBMSs) have moved away from using the traditional iterator model [34, 37] to use other execution models (such as the batch model [48] and the materialization model [41]) to minimize the CPU overhead. However, hand-written code outperforms all three models [54]. Thus, code generation and query compilation have become major contributors to the performance gains of many data systems [4, 43, 44, 49].

In this work, We shed light on the possibility of using query compilation techniques for document stores, where value types are not known until runtime. We utilize the Oracle Truffle framework (called Truffle hereafter) [53] to implement an internal language for processing data stored in a Java-based document store. Even though we only translate part of a query plan, our evaluations show a tremendous improvement over AsterixDB's existing model.

To show their benefits, we have implemented our proposed techniques to store document data in a columnar format and produce a compiled query plan in Apache AsterixDB. This enabled us to conduct an extensive evaluation of the AMAX format and present its tradeoffs for different datasets. We also show the impact of utilizing Truffle to generate and execute queries against different datasets stored as AMAX, as well as the original schemaless row format of AsterixDB and the recently proposed Vector-based format.

2 BACKGROUND

2.1 Apache AsterixDB

AsterixDB is a parallel semi-structured Big Data Management System (BDMS) that runs on large, shared-nothing, commodity computing clusters. To prepare the reader, here we give a brief overview of AsterixDB's storage engine [25], its compiler, Algebricks [29], and its query execution engine, Hyracks [28].

Storage Engine: AsterixDB stores its datasets' records in primary LSM B$^+$-tree indexes. Newly inserted records are hash-partitioned using their primary key(s) into distributed data partitions and inserted into the resulting partition's primary LSM in-memory component. When the in-memory component is full, the in-memory component's records are flushed into a new LSM on-disk component, as shown in Figure 1a. Upon completion, the newly flushed component is marked as valid by setting a validity bit on its metadata page and freeing the in-memory component to serve subsequent inserts. LSM on-disk components are immutable, and hence, updates and deletes are handled by inserting new entries. A delete operation adds an "anti-matter" entry to indicate that a record with a specified key has been deleted. An update simply adds a new record – including the updated value(s) – with the same key as the original one. The newly added record then replaces the older record. Hence, LSM-based document stores refer to this operation as "upsert" since in-place partial updates of values in a record are not supported. As on-disk components accumulate, the on-disk components are periodically merged into larger components in the background according to a configured merge policy [25, 38], which determines when and what to merge. Deleted and older versions of upserted records are garbage-collected during the merge operation.

In Figure 1b, during the merge of C0 and C1 (from Figure 1a) into a new disk-component [C0, C1], the record with id = 0 and its corresponding anti-matter annihilate each other. On completion, the older on-disk components are deleted and replaced by [C0, C1].

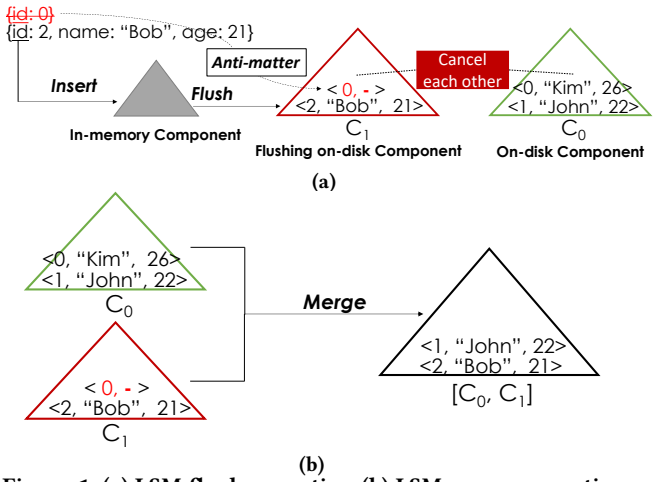


Figure 1: (a) LSM flush operation (b) LSM merge operation

Query Execution Model: To query stored data, a user can submit a query written in SQL++ [31, 47] to AsterixDB. The query is then translated into an optimized query plan, which is compiled into a Hyracks job. The compiled Hyracks job is then distributed to the query executors in all data partitions to run in parallel. Hyracks jobs consist of operators and connectors [28], where data flows between operators over connectors as a batch of tuples. Each batch of tuples received by an operator is processed and then materialized to the next operator's buffer as a new batch.

2.2 LSM-based Tuple Compaction Framework

The flexibility of document stores is targeted for applications where the schema can change without human intervention. However, document stores' flexibility is not free, as each record stores its schema instead of storing it in a centralized catalog. In a previous work [22], we presented a Tuple Compactor framework (implemented in Apache AsterixDB) that addresses this issue by exploiting LSM lifecycle events to infer the component's schema and compact its records using the inferred schema.

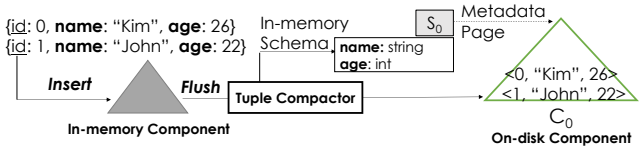


Figure 2: Schema inference workflow

To illustrate, when creating a dataset in AsterixDB, each partition starts with an empty dataset. During data ingestion, each partition inserts the received records into the in-memory component as in normal operation. When the memory component is full, the in-memory components' records are flushed into a new on-disk component, during which time the tuple compactor takes this opportunity to infer the schema and compact the flushed records. Figure 2 depicts the workflow of the tuple compactor along with the

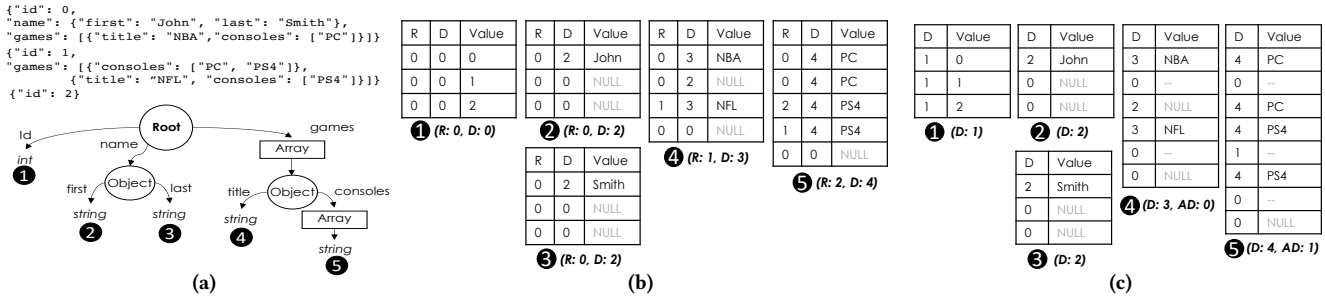


Figure 3: (a) Raw JSON records and their schema (b) Dremel columnar representation (c) Extended Dremel representation

inferred schema. In the figure, we see that the tuple compactor has inferred two fields, name and age, with the types string and integer, respectively, from the flushed records. Upon completing the flush operation, the inferred schema is persisted into the component's metadata page. Subsequent flushes follow the same workflow to build the schema for all of the ingested records. Among the flushed components, the schema of the latest flush is always a super-set of all previous schemas. Thus, we only persist the most recent component's schema into a merged component's metadata page.

Also in [22], we introduced the Vector-based format — a non-recursive, compaction-friendly, physical data format for storing semi-structured data. The vector-based format's main idea is to separate the data values from the records' metadata, which describes the record's structure. This separation enables the tuple compactor to efficiently process the record's metadata during the schema inference and record compaction. Additionally, being non-recursive, the vector-based format allows a record to be processed iteratively, which is more cache-friendly than AsterixDB's format [3].

3 A FLEXIBLE COLUMNAR FORMAT FOR NESTED SEMI-STRUCTURED DATA

Inferring the schema and compacting schemaless semi-structured records, using the tuple compactor framework [22], reduces their overall storage overhead and consequently improves query execution time. However, the compacted records are still in a row-major format, which is less than ideal for analytical workloads as compared to columnar formats. The Dremel format [42] allows for storing nested records in a columnar fashion, where atomic values of different records are stored contiguously in chunks. However, the Dremel (or Parquet) format still requires a fixed schema that describes all fields to be declared a priori, and all field values must conform to the declared fixed schema. One of the main reasons that document stores do not support storing data in a columnar format is the flexibility of their data model. In this section, we first present our extensions to the Dremel format and highlight the structural differences between the original and the extended Dremel formats. Next, we show how our extended Dremel format adapts to schema changes, such as adding new values and changing their types.

3.1 Extended Dremel Format:

To better explain our extensions to the Dremel format, we first highlight the structural differences between the original Dremel and our extended Dremel formats. Initially, we assume that the schema is known a priori for both formats. Later in Section 3.2, we detail how our extensions to the Dremel format allow for schema changes. For better illustration, Figure 3a shows an example of three JSON records about video gamers along with the structure of their declared schema. The schema's inner nodes represent the nested values (objects and arrays), whereas the leaf nodes represent the atomic values such as integers and strings. The circles (e.g., ❷) under the leaf nodes corresponds to column IDs, which link the schema to the column values in Figure 3b for the Dremel format and in Figure 3c for the extended format. The schema describes the JSON records' structure, where the root has three fields *id*, *name*, and *games* with the types integer, object, and array, respectively. The *name* object consists of $first$ and *last* name pairs, both of which are of type string. Next is the array of objects *games*, which stores information about the gamers' owned games, namely the games' $titles$ and the different versions of a game the gamers own for different *consoles*. Every value (nested or atomic) in our example is optional except for the record's key *id*. The optionality of all non-key values is synonymous with the schemaless document store model, which is the scope of this paper. We encourage interested readers to refer to [32, 42] for more details on the representation of non-optional values.

The tables in Figure 3b and Figure 3c depict the columnar-striped representation of the records' atomic values from Figure 3a in both formats. For Dremel, each table consists of three columns: **R**, **D**, and **Value**, where **R** and **D** denote the Repetition-Level and Definition-Level of each Value as presented in [42]. The **Definition-levels** determine whether a value is present or NULL, whereas the **Repetition levels** determine the start and end of a repeated value (or array). The pairs (R:x, D:y), shown at the bottom of each table in Figure 3b, indicate the maximum value for the repetition and definition levels for each atomic value. For our extended Dremel format, we also use the definition level to determine the level at which the NULL value occurred. For repeated values, we use Array-Delimiter-Level (**AD**) to mark the end of a repeated value.

Non-repeated Values: To explain, let us take column ❷, which corresponds to $name.first$ as shown in Figure 3a. In Dremel, the column has a maximum repetition level of 0 indicating a non-repeated value (or not an array element), whereas the maximum definition level 2 is the level of the leaf node in the schema's tree ($root$ (0) → *name* (1) → $first$ (2)). In the first record in Figure 3b, the definition level for the value $name.first$ is 2, which indicates that the path $root \rightarrow name \rightarrow first$ is present and the gamer's $first$ name is "John". For the second and third records, the definition levels for ❷ are 0, indicating that only *root* (which is at level 0) is present in

both records but not the object *name* nor the atomic value $first$. Note that the value 'NULL' is indiciated by the definition level and not stored as a value – the shown 'NULL' values in Figure 3 are for illustration. For the extended Dremel format, we do not use repetition levels but use array delimiter levels instead. Since the column ❷ corresponds to a non-repeated value, there is no maximum **AD** as shown in Figure 3c. Otherwise, the definition levels in the extended Dremel format for the same column in the three records are similar to the original Dremel format. Another difference between the original Dremel format and our extended format appears in column ❶. In Dremel, the maximum definition level for the field *id* is 0 – shown in Figure 3b – as it is a non-optional field. However, in our extension, the maximum definition level (for the same column ❶ shown in Figure 3c) is 1 even though the field *id* is also not optional. As discussed later in Section 3.3, the *id* field is the primary key for the games dataset. Therefore, the definition levels for the primary keys are used to indicate 'anti-matter' tuples.

Repeated Values: For repeated values (array elements) such as column ❹ in our example, the repetition levels in Dremel determine the array starts and ends for each record. Note that for the repeated values for column ❹, the maximum repetition and definition levels are 1 and 3, respectively. The first record has only one value (0, 3, "NBA"), where the triplet (r, d, v) denotes its repetition level (r), definition level (d), and value (v), respectively. The repetition level 0 indicates that the value "NBA" is the record's first ❹ repeated value, and the definition level 3 indicates that the value is present. The following value (0, 2, NULL) corresponds to the second record, as the repetition level 0 indicates that the current value is, again, the first ❹ repeated value. However, the definition level 2 here indicates that the value is NULL, as it is less than the column's maximum definition level 3. Again, the 'NULL' value is not stored as a value but indicated by the definition level. The following value (1, 3, "NFL") is the second element of the same array, which is indicated by repetition level 1. Whenever a value's repetition level is greater than zero, we know that the value is another array element other than the first element. The last value (0, 0, NULL) indicates that the array *games* itself is NULL in the last record.

In Figure 3b, the values of column ❺ belong to the two nested arrays *games* and *consoles* in Figure 3a. Therefore, the maximum repetition level for column ❺ is 2. Like in column ❹, the value (0, 4, "PC") corresponds to the first record, as indicated by the repetition level 0. The definition level 4 here means that the value is present and the value is "PC". The following value (0, 4, "PC") is the first ❺ value for the second record, as indicated by its repetition level 0. The next value (2, 4, "PS4") has a repetition level 2, the maximum repetition level for the column ❺, which means it is the second value of the array *consoles*. The following value's (1, 4, "PS4") repetition level 1 means it still corresponds to the same record; however, the value marks the beginning of the record's second *consoles*' array, which has a single element "PS4". As in ❹, the last value (0, 0, NULL) again indicates that *games* itself is NULL in the last record.

In our example, we noticed that (i) the repetition levels of the column ❹ is a subset of the column ❺'s repetition levels (redundancy), as both share the same array ancestor *games*. The entire repetition levels of the column ❹ [0, 0, 1, 0] appear in the same order as the column ❺'s repetition levels [**0**, **0**, 2, **1**, **0**]. Also, we observed that (ii) all values with repetition levels greater than 0 must have definition levels greater or equal to the array's level. Recall that a value with a repetition level greater than 0 indicates another array element (i.e., not first). When the repetition level is greater than 0, it implies that an array exists and that its length is greater than one. Also recall that when the definition level is smaller than an array's level, it means that the array itself is NULL. As a consequence, having a repetition level greater than 0 and a definition level smaller than the array's level would be contradictory. It would mean the array exists and that its length is greater than one, but that the array itself is NULL. Given that, the number of bits used by Dremel for both the definition and repetition levels is more than what is needed to represent repeated values.

For these reasons, we will adopt a different approach for representing repeated values without repetition levels. Recall (ii), which says that the definition level of a non-first repeated value cannot be smaller than the array's definition level — thus, we can use such definition level values as delimiters instead of repetition levels. Figure 3c shows how repeated values are represented in the extended Dremel format. In our example, the definition levels of the values of columns ❹ and ❺ are subsets of the original Dremel definition levels. The additional definition levels act as array delimiters. To illustrate, column ❹'s maximum array delimiter level (**AD**) is 0. Thus, in the first two records, where the array *games* is not NULL, their repeated values are delimited by the definition level 0. The value that follows a delimiter indicates the start of the next array, and the value itself is the array's first value — except for the last repeated value, where the definition level 0 indicates the array *games* is NULL in the last record. Note that the last value's definition level of 0 cannot be a delimiter since it is the first value after the preceding delimiter.

In the case of nested arrays, as in the column ❺, the maximum **AD** is 1, which indicates that the two delimiter values 0 and 1 are for the outer (*games*) and inner (*consoles*) arrays, respectively. The first value in column ❺ is present, as indicated by the definition level 3, and its value is "PC". The following value is a delimiter of the outer array *games*, indicated by the definition level 0. We omit the definition level 1, the delimiter for the inner array *consoles*, since the delimiter 0 also encompasses the inner delimiter 1. The next two values are the first and second array elements of the second record's array *consoles*. The following delimiter of 1 here indicates the end of the first *consoles* array ["PS4", "PC"], and the next value marks the start of the second *consoles* array ["PS4"] in the same record. The next delimiter 0 indicates the end of the repeated values in the second record. Like in column ❹, the following definition level of 0 implies that the *games* array is NULL in the last record.

3.2 Schema Changes:

For LSM-based document stores, one could use the approach proposed in [22] to obtain the schema and use it to columnize the values. However, a major challenge for supporting columnar formats in document stores is handling their potentially heterogeneous values. For example, the two records `{"id": 1, "age": 25}` and `{"id": 2, "age": "old"}` are valid records and both could be stored in a document store. Limiting the support for storing data in a columnar format to datasets with with fixed schemas and homogeneous values may suffice for many cases, as evidently shown by the popularity of Parquet. However, those limitations prevent

document stores from storing their data as columns. In this section, we address the two main challenges: (i) handling schema changes and (ii) handling data with heterogeneous types.

In our previous work [22], we introduced union types in our inferred schemas to represent values with heterogeneous types. Figure 4 depicts an example of two variant records with their inferred schema. The inferred schema shows that the records have different types for the same value *name*, which could be a string or an object. Thus, we infer *name*'s type as a union of string and object. In the schema, we observe that union nodes resemble a special case of object nodes, where the keys of a union nodes' children are their types. For example, the union node of the field *name*, in the schema shown in Figure 4, has two children, where the key "string" corresponds to the left child, and the key "object" corresponds to the right child.

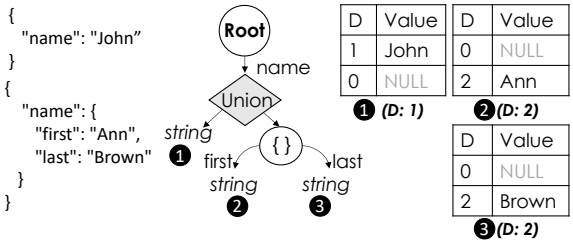


Figure 4: Example of heterogeneous values and their schema

Based on this observation, we can columnize unions' atomic values by treating them as object values with one modification. Observe that an actual value can only be of a single type in any given record, and, hence, only a single value can be present, so the other atomic values associated with the union should be NULLs. To better illustrate, consider an example where the records are inserted one after another, and the schema changes accordingly. Columnizing the records' values can be performed while inferring the schema in a single pass, as in the compaction process in [22]. After inserting the first record of Figure 4, we infer that field *name* is of type string, and thus we write the string value "John" with definition level 1 as shown in column ❶ in Figure 4. In the following record, the field *name* is an object consisting of $first$ and *last* fields. Therefore, we change the field *name*'s type from string to a union type of string and object as was shown in Figure 4. Since the second record is the first to introduce the field *name* as an object, we can write NULLs in the newly inferred columns ❷ and ❸ for all previous records. Then, we write the values "Ann" and "Brown", with definition levels 2 in ❷ and ❸, respectively. Recall that only a single value can be present in a union type; therefore, we write a NULL in column ❶. After injecting the union node in the path root → union → string, we do not change the definition level of column ❶ from 1 to 2 for two reasons. First, union nodes are logical guides and do not appear physically in the actual records. Therefore, we can ignore the union node as being part of a path when setting the definition levels even for the two newly inferred columns ❷ and ❸. The second reason is more technical — changing the definition levels for all previous records is not practical, as we might need to apply the change to millions of records, were it even possible due to the immutable nature of LSM.

When accessing a value of a union type, we need to see which value is present (not NULL) by checking the values of the union type one by one. If none of the values of the union type is present, we can conclude that the requested value is NULL. In the example shown in Figure 4, accessing *name* goes as follows: First, we inspect column ❶, which corresponds to the string child of the union. If we get a NULL from ❶, we need to proceed to the following type, an object with two fields, $first$ ❷ and *last* ❸. In this case, we need to inspect one of the values' definition levels, say column ❷. If the definition level is 0, we can conclude that the value *name* is NULL, as the string and the object values of the union are both NULLs. However, if the definition level is 1, we know that the parent object is present, but the $first$ string value is NULL. Thus, the result of accessing the field *name* is an object in this case. Inspecting all the values of a union type is not needed when the requested path is a child of a nested type. For instance, when a user requests the value *name.last*, processing column ❸ is sufficient to determine whether the value is present or not. Thus, accessing the value *name.last* are NULL in the first record and "Brown" in the second record.

3.3 LSM Anti-matter

In Section 2.1, we explained the process of deleting records in an LSM-based storage engine using anti-matter entries. Anti-matter entries are special records that contain the key of the deleted record. To support deletes, we need to represent anti-matter tuples using the proposed columnar representation. Figure 5 shows the columnar representation of the component C_1 from Figure 1a. The definition level for the primary key *id* does not indicate whether the value is NULL or present; instead, it indicates whether the primary key value corresponds to a record or to anti-matter. When the definition level of a primary key value is 0, it represents an anti-matter entry, which indicates that a record with the same primary key is deleted (Section 2.1). When the definition level is 1, we know that it is a non-anti-matter record. Figure 5 also shows that the anti-matter has an entry for the column *id* but none of the others.

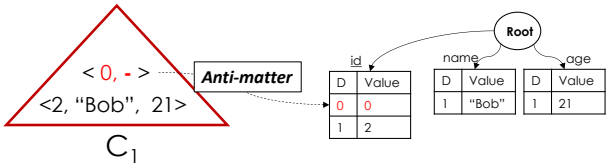


Figure 5: Representing anti-matter tuples

3.4 Record Assembly:

When accessing a nested value such as the nested value *name* in Figure 3b in our approach, all of its atomic values (i.e., first and last) are stitched together to form an object (e.g., `{"first": "John", "last": "Smith"}`) using the same record assembly automaton used in [42]. Also, we use the same Dremel algorithm to assemble repeated values (arrays). However, a difference is that we use delimiters to transition the state when constructing the arrays instead of the repetition levels as in Dremel.

4 COLUMNAR FORMATS IN LSM INDEXES

A major feature of representing records' values as contiguous columns, as in our extended Dremel format, is that it allows us to encode and possibly compress the values of each column according to its type to reduce the overall storage footprint. The

immutability of LSM-based storage engines makes them especially good candidates for storing encoded values, as in-place updates are not permitted. In this work, we propose the AsterixDB Mega Attributes Across *(AMAX)* layout for storing the columns in LSM-based document stores. We have implemented and evaluated the AMAX layout in Apaches AsterixDB, hence the name. In the following sections, we first briefly explain the supported techniques used to encode the column values. Then, we detail the structure of the AMAX layout. Next, we describe the lifecycle of writing and reading the columns, and finally, we cover challenges related to answering queries in AMAX with secondary indexes.

4.1 Encoding

Apache Parquet offers a rich set of encoding algorithms [12] for different value types, including bit-packing, run-length encoding, delta encoding, and delta strings. In this work, we use all of Parquet's encoding algorithms except for dictionary encoding, which requires additional pages to store the dictionary entries. (We leave potential support for dictionary encoding for future work.)

4.2 *AMAX* Layout

The PAX (and APAX [23] for that matter) layout stores different columns within a page, and hence in this layout, one needs to read entire pages, regardless of which columns are needed to answer a query. In AMAX, we instead stretch LSM B$^+$-tree leaf nodes to become mega leaf nodes, where each one can occupy more than one physical data page. Figure 6 illustrates the structure of the AMAX pages in a B$^+$-tree, where a mega leaf node consists of multiple (6 in this case) physical pages. Each mega leaf node in the AMAX layout starts with Page 0, which consists of three segments. The first segment stores the page header, which contains meta-information such as the number of columns and the relative pointers to each megapage. The second segment stores fixed-length prefixes of the minimum and maximum values for each column. Each minimum and maximum prefix pair occupy 16 bytes (8-bytes each), and they are used to filter out entire AMAX pages that do not satisfy a query predicate (e.g., *age* > 20). Lastly, in the third segment, most of the space in Page 0 is used to store the encoded primary key(s) values.

Each megapage of the AMAX layout corresponds to a single column. Figure 6 shows the representation for the column stored in *Megapage* 1. Each column stores (1) the **size** of the column's values in bytes, (2) the **count** of values in the column, (3) the **definition levels** as in our Dremel extensions, followed by (4) the **values** in the column. The megapages are ordered by their size, from largest to smallest. In other words, a mega leaf stores the largest megapage's physical pages contiguously first on disk, followed by the physical pages of the second-largest megapage, and so on. This ordering of megapages allows for better utilization of the empty space of the physical pages. For example, after writing *Megapage* 1, the physical *Page* 3 in Figure 6 is mostly empty, and thus, we allow *Megapage* 2 to share the same physical *Page* 3 with *Megapage* 1. After writing *Megapage* 2, note that *Page* 4 is not full. A tuning parameter (called the $empty - page - tolerance$) allows the AMAX page writer to tolerate a certain percentage of a physical page to be empty if the next column to be written does not fit in the given empty space. Tolerating smaller empty spaces can help to minimize the number of pages to be read from when retrieving a column's values. The content of a megapage is encoded, and specific readers and decoders (determined by the schema) are used for interpreting their content.

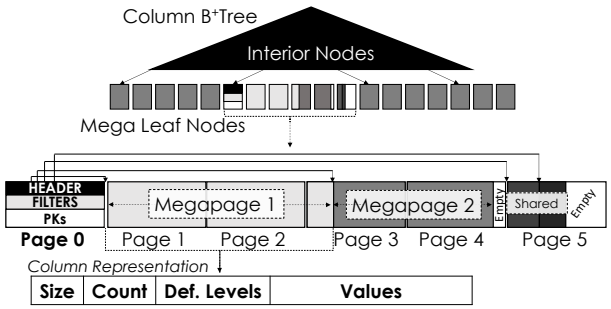


Figure 6: AMAX multi-page layout in a B-$^+$tree

4.3 Writing AMAX Pages

As in [22] and summarized in Section 2.2, we exploit LSM-lifecycle events to infer the schema and split the records in row-major format into columns. During data ingestion, we first insert records into the in-memory LSM component in our vector-based format. Once it is full, the records of the in-memory component are flushed into a new on-disk component, during which time we infer the schema of the flushed records and also split their values into columns. Finally, the inferred schema (e.g., Figure 3a) is persisted into the flushed component's metadata page as in the tuple compactor (Section 2.2).

AMAX columns can occupy one or more physical pages (megapages), while the smaller columns may share a single physical page. Initially, we do not know which columns might span into multiple physical pages, so we write the values of each column into a fixed-size temporary buffer first. Once the buffer is full, we confiscate (or acquire) a page from the system's buffer cache, which then replaces the temporary buffer for writing the columns' values. Instead of allocating a memory budget for writing columns, we use the system's shared buffer cache as a temporary buffer provider. (Allocating a dedicated memory budget for writing columns might be wasteful, especially for cases where writes are not continuous, e.g., when loading a dataset once.) As the column size increases, we confiscate more pages from the buffer cache to accommodate the written values of that column, and those physical pages form a megapage. Once done, we write the megapages to disk.

Page 0 of the AMAX could also, in theory, grow to occupy multiple physical pages. However, we do not permit that, as the number of keys could then grow into hundreds of thousands. Consequently, point-lookups would perform poorly, as we need to decode and search for the required key; this could negatively impact both the ingestion rate and the performance of queries with secondary indexes, as we discuss later in Section 4.6. Therefore, we limit the number of records stored in an AMAX page to 15K by default. This specific limit was determined empirically, where we found that a limit of 15K was not too large to affect point-lookups operations yet not too small to impact scan-only workloads negatively. However, one can tune this parameter per their workload needs. For example, increasing the limit for scan-only workloads, where no secondary indexes are declared, would improve query execution time. Tuning the limit parameter in AMAX is synonymous with tuning Parquet's row-group and data page sizes [19] – for example, a smaller data page size is more suitable for single row lookups.

4.4 Reading AMAX Pages

When a user submits a query, the compiler optimizes the query and generates a runtime job that will access the appropriate collections (or datasets in AsterixDB's terminology) and project the required attributes from the resulting records. The generated job is then distributed to all partitions for parallel execution. Before execution, each partition consults the inferred schema to determine the columns needed (i.e., AMAX megapages) for executing the query. Then, only the physical pages that correspond to the columns needed by the query are read. For each requested column, an iterator goes over the columns' values. If a query contains a filtering predicate (e.g., `WHERE age > 20`), the prefixes are also used to skip reading the entirety of the requested columns of a mega leaf page that would not satisfy the query predicate.

When reading from an LSM index, the system reads one tuple at a time from each component, and tuples with the same keys (e.g., anti-matters and actual tuples) are reconciled. Thus, deleted and upserted records are ignored and will not appear in the final result of the query. When reading AMAX pages, we need to (i) perform the same reconciliation process as in the row-major layout. Also, we need to (ii) process the in-memory component's records, which are still in a row-major layout. To address those two requirements, we implement an abstracted view of a "tuple", whether in row-major or column-major format, resulting from reading an LSM component.

Reconciling tuples in a row-major layout is performed simply by ignoring the current (deleted or upserted) tuple and going to the next one using the tuple's offset stored on the slotted page. Doing the same in the AMAX format means advancing the relevant columns' iterators by one step. Doing so eagerly would be inefficient, as (i) we would need to touch multiple regions of the memory, resulting in many cache misses, and (ii) we would need to decode the values each time we advance a column iterator, which could be a wasted effort as we illustrate next. Let us consider the following query:

```
SELECT name, salary FROM Employee WHERE age>30
```

Suppose that we have three records with primary keys 1, 2, and 5 stored in an on-disk component in a columnar layout. Also, suppose that the in-memory component has three records with the same primary keys, i.e., 1, 2, and 5. In this example, the records of the in-memory component will override the records of the on-disk component. If we advanced every column's iterator eagerly (namely the *name*, *age*, and *salary* columns' iterators) in order to get the next tuple, the decoding of the columns' values would be a wasted effort. For that reason, we only decode the primary key values during the reconciliation process, and we count the number of ignored records. Once actually accessed, we advance each column's iterator by the number of ignored records at once, ensuring that the process of advancing the iterator is performed in batches per column. As a consequence, none of the AMAX columns would be decoded in our example as none were accessed.

4.5 Impact of LSM Merge Operations:

From time to time, an LSM merge operation is scheduled to compact the on-disk components. In the AMAX layout, we need to read the columns' values from different components and write them again into a newly created merged component. The order in which the columns' values are written is determined by the records' keys from each component, and the column values that correspond to the smallest keys are written first. Similar to the issue discussed in Section 4.4, eagerly reading the columns' values in each component would result in touching different regions in memory, which would not be cache-friendly. To remedy this issue, we employ what we call a vertical merge. In the vertical merge, we first merge the primary keys resulting from the different components, and we record the sequence of the components' IDs in memory. Then, we merge the values of each column one-at-a-time from the different components based on the order of the recorded component ID sequence from merging the keys. This vertical merge of the columns ensures that only one column is merged at a time. Thus, the number of memory regions that we need to read from is equal to the number of merging components instead of the number of columns times the number of components. This approach also allows us to only read one megapage at a time from each component instead of all megapages (which could otherwise pressure the buffer cache).

Another issue when merging columns is the CPU cost of decoding and encoding the columns' values, especially for datasets with large numbers of columns. In our initial experiments, this CPU cost became more apparent during concurrent merges, peaking at 800% on an 8-core machine, which could render the system unusable for users who want to query their data while LSM operations are occurring. The potential resource saturation resulting from concurrent merges in LSM-based storage engines is well-known [20], and limiting the number of concurrent merges can remedy this issue. In AsterixDB, the number of configured partitions determines the number of CPU threads that serve a query. Therefore, we limit the number of concurrent merges to half the number of partitions by default to free some cores to serve users' queries. Limiting the number of concurrent merges may stall writes and negatively impact the ingestion rate [40], but writing the records in a columnar format can reduce the overall storage footprint, which means less I/O. We believe an extensive evaluation, as in [40], should be conducted to measure those tradeoffs; however, it is beyond the scope of this work, so we leave it for future work.

4.6 Point Lookups and Secondary Indexes

In LSM-based key-value stores, one can blindly insert new records into the in-memory component without checking if a record with the same key exists (to ensure the uniqueness of the primary keys), as records with identical keys are reconciled at the query time. However, that mechanism only applies to a primary index and not to its associated secondary indexes. For a secondary index, in addition to adding the new entry, we also need to clean out the old entry (if any). Thus, a point lookup is needed to fetch the old value from the primary index to clean the old values by adding appropriate anti-matter entries in each secondary index. Consequently, during data ingestion with secondary indexes, point lookups must be performed for each newly inserted record to check if a record with an identical key exists. If so, its old values are retrieved to maintain secondary indexes' correctness.

Performing point lookups against datasets stored in the AMAX layout is more expensive than in its row-major counterparts, as we need to decode primary keys and search for the requested value in the AMAX layout. To alleviate the cost of point lookups for datasets

in the AMAX layout, we use a "primary key index", an additional secondary index that stores only primary keys, to first see if a record with an identical key exists [38, 39]. If the primary key index does not yield any keys, we can skip accessing the primary index, as the newly inserted key does not correspond to an older record.

When answering queries (e.g., range queries), the appropriate secondary index is first searched, yielding the primary keys of records that satisfy the query predicate. Then, the resulting primary keys are sorted in ascending order. Finally, point lookups are performed using the sorted primary keys to retrieve the records that satisfy the query predicate. Luo et al.'s generalized approach [38] exploits the ordered keys to perform these point lookups in batches while preserving the state of the LSM cursor to reduce the cost of subsequent point lookups. This approach allows us to read the columns' values in a single pass by accessing the values of the first record with the smallest key followed by the record with the second smallest key, etc., without the need to start over each time.

5 CODE GENERATION

In our early evaluation, we saw that the CPU cost of AsterixDB's query execution engine eclipsed the I/O savings when querying some datasets stored using the AMAX format. To understand the CPU cost, let us consider the query shown in Figure 7, which counts the number of each game's *title* owned by different *gamers* for the dataset shown in Figure 3. When a user submits a query, the query optimizer applies a set of optimization rules to the query to produce an optimized query plan, as shown in Figure 7. After profiling such a query, we found that (i) the materialization cost between operators (e.g., *SCAN* → *ASSIGN*) and (ii) the cost for operators such as the *UNNEST* operator, which flattens and "joins" every element of the array *games* in the tuple with the tuple itself, are high whether the data was stored as rows or columns. For datasets stored in the AMAX format, the values are reassembled so that Hyracks operators can operate on row-major tuples. We observed that (iii) reassembling the values (e.g., the array *games*) eagerly incurred an additional CPU overhead — making querying records in the AMAX format sometimes slower than records in a row format.

In [44], Neumann discussed the CPU costs associated with different execution models and proposed a solution that fuses the work of multiple operators and replaces them with a single function call by generating and compiling a code segment that performs the work of the fused operators. For instance, Figure 7 shows that the operators *SCAN*, *ASSIGN*, *UNNEST*, and *PROJECT* are replaced with generated code, which eliminates the materialization costs of the replaced operators – addressing the cost (i) mentioned earlier.

Before showing how the proposed solution in [44] can be used to address costs (ii) and (iii), let us first show the process of generating code for a query. The generated code, shown in Figure 7, is produced by applying a rule which traverses the optimized plan, during which each operator contributes part of the generated code [44]. The code begins with the function *run*'s header, which takes two parameters *c* and *r*, where *c* is a tuple **c**ursor over the *gamers* dataset and *r* is a field **r**eader for the *games*[∗].*title* values. The reader is preconfigured to accept a tuple as an input and produce the value of a requested field. Figure 7 also shows which operator in the plan (color-coded) contributes which part of the code. First, we see the generated code produced from the *SCAN* operator, which loops through the *gamers'* tuples. Next, the *ASSIGN* operator contributes the code for using the reader *r* to get the field *games* value from the tuple. The *UNNEST* operator contributes the *while* loop to produce the items of the array *games*. Finally, the code only writes the values of *title* of *games*, as they are the only projected values from the *PROJECT* operator.

Regarding the costs (ii) and (iii) in the original plan, the *UNNEST* operator joins the original tuple with each element of the array *games*. Then, the *PROJECT* operator projects only the value *title* from the unnested array *games* and removes the joined original tuple. In comparison, the generated code outputs only the *title's* values of the array's *games* – eliminating this unnecessary join, which addresses the cost (ii). Additionally, the reader *r* in Figure 7 only accesses the values in the column *title* (by calling *r.getValue*()), which are of the scalar type string (Figure 3). Consequently, in the AMAX format, the generated code avoids the cost of reassembling the array *games*, which addresses (iii).

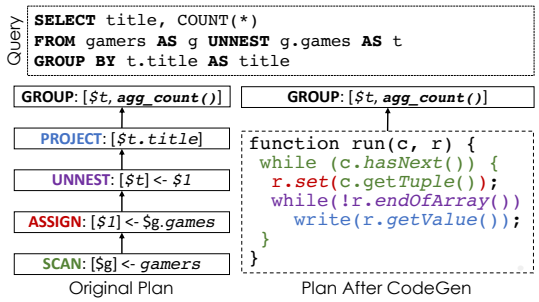


Figure 7: Code generation workflow

In relational databases, the schema provides the columns' types, which simplifies generating a code for expressions like $X + Y$ since the types of both X and Y are known at compile-time (e.g., adding two integers). However, in document stores, the types of X and Y are known at runtime, and they may change from one tuple to another – making the code generation process trickier. Thus, we selected Truffle [53], a framework for implementing dynamically typed languages (e.g., Python), to produce Truffle Abstract Syntax Trees (AST) for a part of the query plan. Each node of the AST describes a language operation, such as a numerical expression (e.g., arithmetic addition) or a control flow statement (e.g., loop statement). Also, we specify the expected behavior (called specialization in Truffle) for each expression given its inputs. For example, we specify that the output for adding two integers is an integer, while an integer with a double is double. After observing a few values by executing the AST in an interpreted mode, Truffle optimizes the AST and generates a bytecode to run it in the Java Virtual Machine (JVM), where the generated bytecode is optimized further to machine code

In our work to date, the code generation is a proof-of-concept and only supports "pipelining" operators [44]. In our example, the *GROUP* operator is a "pipeline-breaker", as it requires, for instance, performing external sorting (sort-group-by) to compute the groups and their aggregate counts. We plan to expand our proof-of-concept to include such operators in the future.

6 EXPERIMENTS

In this section, we evaluate an implementation of the techniques proposed here in Apache AsterixDB. In our experiments, we first

evaluate on-disk storage size after data ingestion to measure the potential storage savings from storing data as columns — giving the different characteristics of different datasets. Second, we measure the data ingestion rate to evaluate the cost of inferring the schema and columnizing the records. Additionally, we evaluate the ingestion performance for an update-intensive workload to measure the impact of maintaining secondary indexes. Finally, we evaluate and analyze the impact of our proposed techniques on analytical query performance, which is the main objective to improve in this work. We evaluate the performance for storing and querying records in different layouts, namely: (i) AsterixDB's schemaless record format (Open), (ii) the Vector-Based (VB) format proposed in [22], and (iii) AMAX. Again, Open and VB are both row-major formats, whereas AMAX is a columnar format.

Experiment Setup We conducted our experiments using a single machine with an 8-core (Intel i9-9900K) processor and 32GB of main memory. The machine is equipped with a 1TB NVMe SSD storage device (Samsung 970 EVO) capable of delivering up to 3400 MB/s for sequential reads and 2500 MB/s for sequential writes. We used AsterixDB v9.6.0 to implement and evaluate our proposed techniques. We configured AsterixDB with a single node and eight partitions. The eight partitions share 16GB of total allocated memory, and from this, we allocated 10GB for the system's buffer cache and 2GB for the in-memory component budget. The remaining 4GB is allocated for use as temporary buffers for query operations such as sorting and grouping as well as transforming records into *AMAX* layout during data ingestion. Additionally, we used 128KB for the on-disk data page size and 64KB for in-memory pages. Throughout our experiments, we used AsterixDB's page-level compression with the Snappy [15] compression scheme to reduce the storage footprint for all formats.

6.1 Datasets

In our evaluation, we used five different datasets (real, scaled, and synthetic) that differ in terms of their records' structures, sizes, and value types. Table 1 lists and summarizes the characteristics of the five datasets. In Table 1, *# of Columns* refers to the number of inferred columns for records in the *AMAX* layout.

The *cell* dataset (provided by a telecom company) contains information about the cellphone activities of anonymized users, such as the call duration and the cell tower used in the call. The *cell* dataset is the only dataset we used that does not contain nested values (i.e., its data is in first-normal form or 1NF), and its scalar values' types are a mix of strings, doubles, and integers. The *sensors* dataset contains primarily numerical values that describe the sensors' connectivity and battery statuses along with their daily captured readings. In contrast, the *wos* dataset, as well as *tweet_1* and *tweet_2*, consist mostly of string values. The *wos* dataset, an acronym for Web of Science [7], encompasses meta-information about published scientific articles (such as authors, abstracts, and funding) from 1980 to 2014. The original dataset is in XML and we converted it to JSON using an XML-to-JSON converter [18]. After the conversion, the resulting JSON documents contain some fields (resulting from XML elements) with heterogeneous types, specifically a union of an object and an array of objects. Thus, we used the *wos* dataset to evaluate our extensions to the Dremel format to store heterogeneous values in *AMAX*. Lastly, we obtained the *tweet_1* and *tweet_2* datasets using the Twitter API [16], where we collected the tweets in *tweet_1* from September 2020 to January 2021. The *tweet_2* dataset is a sample of tweets (~ 20GB) that we collected back in 2016, predating Twitter's increasing the character limit from 140 to 280. We replicated the *tweet_1* dataset to have around 200GB worth of tweets in total. Note that the records of *tweet_1* and *tweet_2* differ in terms of their sizes and the numbers of columns that they have, as shown in Table 1.

We used the *tweet_2* dataset for evaluating the impact of declaring secondary indexes for an update-intensive workload as we detail later in Section 4.6. Additionally, we evaluated the impact of answering queries using the created secondary indexes. We created two indexes in this experiment. The first index is on the tweet *timestamp* values, a set of synthetic and monotonically-increasing values that mimics the times when users posted their tweets. We also created a primary key index to reduce the cost of point lookups, as discussed in Section 4.6. We chose *tweet_2* for this experiment since it has a moderate number of columns, which directly impacts the ingestion performance, as we discuss later in Section 6.3.

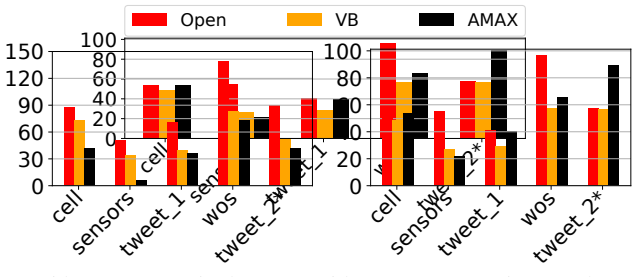
Table 1: Datasets summary

	cell	*sensors*	*tweet_1*	*wos*	*tweet_2*
Type	Real	Synthetic	Real	Real	Scaled
Size (GB)	172	212	210	277	200
# of Records	1.43B	40M	17M	48M	77.2M
Avg. Rec. Size	141B	3.8KB	5.3KB	6.2KB	2.7KB
# of Columns	7	16	933	296	275
Dominant Type	Mix	Integer	String	String	String

6.2 Storage Size

In this experiment, we first evaluated the on-disk storage size after ingesting the five datasets: *cell*, *sensors*, *tweet_1*, *wos*, and *tweet_2*. Figure 8a shows the total on-disk size after ingesting the five datasets using the three layouts: *Open*, *VB*, and *AMAX*. For the *tweet_2* dataset, the presented total size includes the sizes for storing the two declared secondary indexes (namely the *timestamp* index and the primary key index).

In the *cell* dataset, which is the only dataset in 1NF, Figure 8a shows that the records in the two row-major layouts *Open* and *VB* took roughly the same space; the *VB* layout took slightly less space (~ 17% smaller) due to compaction [22]. However, records in the *AMAX* layout were 42% and 51% and smaller compared to the records in the *Open* and *VB*, respectively. The storage overhead reductions in the *AMAX* layout are due to (i) storing no additional information (e.g., field names), compared to *Open*, and (ii) the values being encoded, which is not possible in the row-major layouts.



(a) Storage size (GB) **(b) Ingestion time (Minutes)**

Figure 8: Storage size and ingestion time

The impact of encoding in the *AMAX* layout is most apparent for storing the *sensors* dataset, where the values' are primarily numeric. Figure 8a shows that the *Open* and *VB* layouts took 8.5X and 5.6X more space as compared to the *AMAX* layout, respectively. This clearly shows that the encoding of numerical values in the same domain is superior to page-level compression alone, making the columnar layout more suitable for numerical data.

In contrast to the *sensors* dataset, the Twitter dataset *tweet_1* contains more textual values than numerical ones. Thus, the storage savings from the *AMAX* layout (as shown in Figure 8a) is negligible compared to the *VB* layout, as encoding large textual values is relatively less effective compared to numerical values. The *tweet_1* records took more space in the *Open* layout than other layouts due to the *Open* layout's recursive structure (as detailed in [22]), where nested values use 4-byte relative pointers for each nesting level. Additionally, the *Open* layout records embed the field names for each value, which takes more space than the other layouts.

Storing the *wos* dataset using the three layouts shows a similar trend, shown in Figure 8a, as in the *tweet_1* dataset, even though the number of columns in the *wos* dataset is not as excessive, as shown in Table 1. However, the average size of a record in the *wos* dataset is larger than the average record size in the $tweet_1$ dataset. The reason is that some of the values in the *wos* dataset are relatively larger than the *tweet_1* values. For example, the abstract text of a publication can consist of multiple paragraphs.

For the last dataset, *tweet_2*, the total storage size includes the sizes of the two declared indexes. Secondary indexes are agnostic of the records' layout in the primary index, and their sizes are the same for all three layouts. Hence, the differences between the sizes for the different layouts for *tweet_2*, shown in Figure 8a, correspond to the layouts' characteristics. For instance, the sizes of the records in the *VB* and *AMAX* layouts are comparable, with *AMAX* being slightly smaller. However, the *Open* layout took more space for the reasons explained earlier.

6.3 Ingestion Performance

We next evaluated the ingestion performance for the three different layouts using AsterixDB's data feeds. We first evaluated the **insert-only** ingestion performance of the *cell*, *sensors*, *tweet_1*, and *wos* datasets without updates. In the second experiment, we evaluated the ingestion performance of an **update-intensive** workload with secondary indexes using the *tweet_2* dataset. The latter experiment focuses on measuring the impact of the point lookups that are needed to maintain the correctness of the secondary indexes for upserts and deletes; hence, this experiment stress-tests the AMAX format for handling update-intensive workloads.

We configured AsterixDB to use a tiering merge policy with a size ratio of 1.2 throughout the experiments. This policy merges a sequence of components when the total size of the younger components is 1.2 times larger than that of the oldest component in the sequence. To measure the ingestion rate accurately, we used the fair merge scheduler as recommended in [40], where the components are merged on a first-come, first-served basis. We set the maximum tolerable number of components to 5, after which a merge operation is triggered. We limited the number of concurrent merges to reduce CPU and memory consumption (Section 4.5) while merging *AMAX* components. Additionally, we limited the number of primary keys in AMAX's Page 0 to 15K for the reasons discussed in Section 4.3.

Insert-only: The *cell* dataset is the smallest in terms of the average record size and the dataset's overall size, as shown in Table 1. However, it also has the most records (1.43 billion). We see in Figure 8b that the ingestion rate is about the same for the three layouts, as writing to the transaction log buffer was a major bottleneck while ingesting the high cardinality *cell* dataset. For this particular dataset, we changed AsterixDB's configuration (as detailed in our extended version of this paper [23]) to alleviate this contention.

In the *sensors* dataset, in contrast to the *cell* dataset, the ingestion rate varied among the different layouts as shown in Figure 8b. Ingesting *Open* records took more time than records in the other layouts due to the record construction cost of the *Open* layout [22]. Recall that the records of the in-memory components are in the *VB* format for *AMAX* (as discussed in Section 4.3), and during the flush operation, the records are transformed into a columnar layout. Thus, the lower construction cost of the *VB* records [22] contributed to the higher ingestion rate of the *AMAX* layout. We also observed that the cost of transforming the records into a column-major layout during the flush operation and the impact of decoding and encoding the values during the merge operation were negligible.

For the *tweet_1* and *wos* datasets, the cost of transforming the records into columns became more apparent due to the higher number of columns in those two datasets. Figure 8b shows the ingestion time for the *tweet_1* dataset. For the *AMAX* pages, most of the time here was spent performing LSM merges, as we need to fetch all 933 columns for each merge operation. The ingestion performance using the *AMAX* layout was similar to the row-major layout (*Open*) and only 25% slower than the *VB* layout.

The *wos* dataset is less extreme in terms of the number of columns compared to the *tweet_1* dataset; however, its data contains large textual values (e.g., abstracts). As in the *sensors* dataset, the lower per-record construction cost of the *VB* layout was the main contributor to the performance gains (shown in Figure 8b) for the *AMAX* layout. Additionally, the records in the *Open* layout took more space to store, which means that the I/O cost of the LSM flush and merge operations was higher compared to the other layouts. The ingestion performance of the *AMAX* layout was comparable and slightly slower than the *VB*.

Update-intensive: Our evaluation of the ingestion performance for insert-only workloads using the different datasets showed that the ingestion rate using columnar layouts, in general, was faster or comparable to the row-major layout *Open*. We now discuss the performance for an update-intensive workload with secondary indexes using the dataset *tweet_2*. In this experiment, we randomly updated 50% of the previously ingested records either by upserting or deleting them. The updates followed a uniform distribution where all records were updated equally. Prior to ingesting the data, we created two indexes: the first was a primary key index to minimize the cost of point lookups of non-existent (new) keys. The second index was on the *timestamp* values. Figure 8b shows *tweet_2* the ingestion time for the different layouts.

The ingestion time for records in the *AMAX* was ~ 35% slower than the *Open* layout. Updating a record requires accessing the primary index to fetch the old *timestamp* value to delete it from the *timestamp* secondary index before inserting the updated value.

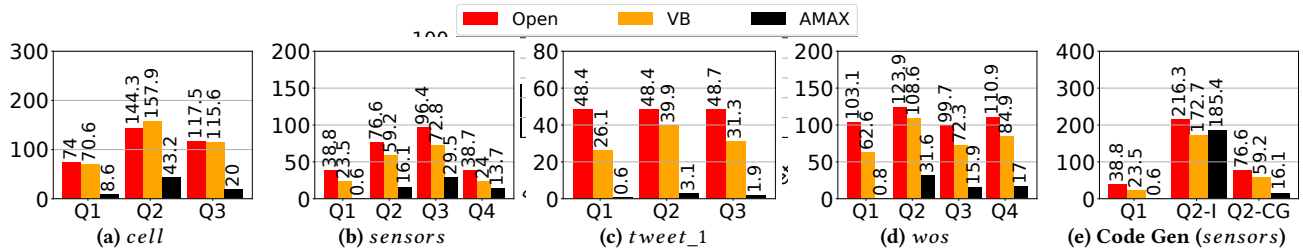


Figure 9: Query execution times (Seconds)

Recall that the cost of searching for a value in a columnar layout is higher, as the values need to be decoded before performing the search. Thus, with 50% of the records being updated, the cost of updating old *timestamp* values for the columnar layout became higher than for the row-major layouts. This was true even though we only needed to read the pages corresponding to the *timestamp* in the *AMAX* layout (i.e., less I/O cost) — decoding large numbers of *timestamp* values stored in *AMAX* megapages (a CPU cost) for each update is high. We will soon (Section 6.4) discuss the benefit of secondary indexes when answering queries; however, the cost of maintaining the correctness of secondary indexes is higher for columnar layouts. Thus, one should consider how often the index will be utilized.

6.4 Query Performance

Next, we evaluated the performance of executing different analytical queries against the ingested datasets. We first evaluated scan queries (i.e., without secondary indexes) with the code generation technique against the *cell*, *sensors*, *tweet_1*, and *wos* datasets. Table 2 summarizes the queries used for each dataset. Q1, which counts the number of records – `SELECT COUNT(*)` – is executed against all four datasets to measure the I/O cost of scanning records in the different layouts. We executed each query six times and reported the average execution time for the last five. Then, we show the benefit of the code generation model (Section 5) over AsterixDB's original query execution model using the *sensors* dataset. In the next experiment, we evaluated the performance of different queries against the *tweet_2* dataset using the created secondary indexes on the *AMAX* layout with and without indexes.

Scan-based Queries: Figure 9 shows the execution times for all queries in Table 2. The execution times for Q1 (for the datasets *cell*, *sensors*, *tweet_1*, and *wos*) against the different layouts correlated with their storage sizes shown in Figure 8a except for the *AMAX* layout. As Q1 only counts the number of records, we only need to count the number of primary keys on Page 0 of the *AMAX* layout — thereby minimizing the I/O cost. For all datasets except for *cell*, Q1 took a subsecond to finish in the *AMAX* layout.

The trends of the execution times for other queries using the three layouts also varied, as shown in Figure 9a. In the *cell* dataset, in contrast to Q1, Q2 requires grouping, aggregating, and sorting to compute the query's results, and hence, it takes more time to execute. For the *AMAX* layout, in addition to the primary keys on Page 0, Q2 accesses two more columns (the caller ID and the call duration columns), which means that more pages were accessed to execute Q2. Despite the additional costs, *AMAX* was the fastest to execute Q2. Q3 also shows a similar trend as Q1, where the I/O costs of the *AMAX* layout were the smallest.

Table 2: Summary of the queries used in the evaluation

*	Q1	The number of records
cell	Q2	The top 10 callers with the longest call durations
	Q3	The number of calls with durations ≥ 600 seconds
sensors	Q2	The maximum reading ever recorded
	Q3	The IDs of the top 10 sensors with maximum readings
	Q4	Similar to Q3, but for readings in a given a day
tweet_1	Q2	The top 10 users who posted the longest tweets
	Q3	The top 10 users with the highest number of tweets that contain a popular hashtag
wos	Q2	The top 10 scientific fields with the highest number of publications
	Q3	The top ten countries that co-published the most with US-based institutes
	Q4	The top ten pairs of countries with the largest number of co-published articles

For the *sensors* dataset, the execution times of Q1 – Q4, shown in Figure 9b, for the *AMAX* records were faster. One reason is that *AMAX* records took 6.5GB to store the data (Figure 8a), which is less than the 10GB of memory allocated for the system's buffer cache. Thus, AsterixDB was able to cache the *AMAX* records in memory and eliminate the I/O cost.

For *tweet_1*'s queries (Table 2), we observed an order of magnitude improvement in the query performance using the *AMAX* layout vs. the other layouts. *VB* and *AMAX* used comparable space to store the *tweet_1* data; however, reading only the columns involved in the queries in the *AMAX* layout improved their execution times significantly. For example, Q2 took 3.1 seconds to execute in *AMAX* vs. 48.5 and 39.9 seconds for *Open* and *VB*, respectively.

The *wos* dataset is the last one used to evaluate scan-based queries. As mentioned earlier in Section 6.1, the *wos* dataset contains several values with heterogeneous types. We used this dataset to evaluate the impact of querying over heterogeneous types for the columnar layouts. Specifically, Q3 and Q4 (Table 2) access the authors' affiliated countries, which is stored as either an array, for articles with multiple co-authors, or as an object, for single-authored articles. For Q2 – Q4, *AMAX* improved their execution times by at least 64% compared to the other layouts. Thus, the *AMAX* layouts can efficiently handle values with heterogeneous types, and the impact on query performance was negligible.

Code Generation: To illustrate the benefit of our proposed code generation technique, Figure 9e execution times of running Q1 and Q2 of the *sensors* dataset in the three different formats. Q1 was faster to execute using the AMAX format, as discussed earlier. However, for Q2-I (**I** stand for **I**nterpreted), querying the

data in the AMAX format – using AsterixDB's original execution engine – was even slower than querying data in the VB format due to the reassembly cost (Section 5). In contrast, when executing the same query using the **C**ode **G**eneration approach (Q2-CG in Figure 9e), the execution times were improved significantly in the three formats as a result of reducing the CPU costs discussed in Section 5.

Index-based Queries: We used the *tweet_2* dataset to evaluate the impact of secondary indexes on query performance for the *AMAX* layout. We used a *timestamp* secondary index to run range-queries with different selectivities that count the number of non-NULL values of different columns. Each query counts the appearances of different columns' values (i.e., non-NULL values) and varies the number of columns accessed to be 1, 5, and 10. The columns were picked at random and varied in terms of their types and sizes. Table 3 shows the execution times for index-based and scan-based queries that access the different number of columns. As expected, accessing more columns in the *AMAX* format negatively impacts the scan-based query performance. For example, reading ten different columns was 9.5X slower than reading a single column for the *AMAX* layout. Compared to the scan-based queries, the index-based queries took less time to execute when accessing more than one column and were less sensitive w.r.t the number of columns. Thus, secondary indexes can accelerate queries against records in a columnar layout and can help to minimize the impact of reading multiple columns for *AMAX*-like layouts.

Table 3: Query execution times (Scan vs. Index)

# of Columns	Scan	Index (Selectivity %)			
		0.001%	0.01%	0.1%	1.0%
1	2.080	0.021	0.066	0.405	3.487
5	14.133	0.033	0.079	0.456	3.917
10	25.710	0.047	0.097	0.553	4.636

7 RELATED WORK

Columnar layouts with dynamic schema: Storing schemaless semi-structured data in a columnar layout has gained more interest lately, and several approaches have been proposed to address the issues imposed by schema changes. Delta Lake [27], a storage layer for cloud object stores, addresses the challenges of updating and deleting records stored in Parquet files. Delta Lake recently added support for schema evolution; however, it still lacks support for storing heterogeneous values, as per Parquet's limitation. Alsubaiee et al. proposed a patented technique [26] that exploits Parquet's file organization to store datasets with heterogeneous values. The main idea of their approach is congregating records with the same value types within a group. In this work, we proposed an extension to Dremel to natively support union types, storing values with different types as different columns.

In [33], the authors have proposed *Json Tiles*, a columnar format for semi-structured records integrated into Umbra [45], a disk-based column-store RDBMS. The proposed approach infers the structure of the ingested records and materializes the common parts of the records' values, including heterogeneous values, as *JSON Tiles*. Similarly, Sinew [52] utilizes an RDBMS (potentially a columnar one) to store the JSON data, where JSON scalar values are either stored physically as columns (i.e., declared in the RDBMS schema) or virtually as key-value pairs in a separate table. However, our work's objective is to natively support columnar formats for existing LSM-based document stores.

For LSM-based document stores, Rockset [14] supports storing values of semi-structured records in a columnar format, with the values of a column being stored in RocksDB [13] (Rockset's storage engine) using a shared key prefix. Thus, a column's values from different records are stored contiguously on disk. When accessing a column, Rockset will only read the required values from disk, which minimizes the I/O cost. However, this approach does not support encoding the column's values (e.g., via run-length encoding).

LSM-based column stores: Most column-store databases employ a similar mechanism to LSM-based storage engines, where newly inserted records are batched in memory and then flushed to disk, during which time the flushed records are encoded and compressed. For example, Vertica [51] and Microsoft SQL Server's column store [35, 36] employ an LSM-like mechanism, while column-store systems such as Apache Kudu [5] and ClickHouse [8] are LSM-based. This work is no exception, as we share similar objectives. Again, however, our focus is on nested and schemaless data.

Code generation and query compilation: Data processing engines like Spark and DBMSs like Vector use code generation techniques to improve performance. Most such systems utilize strongly-typed languages for code generation, which is sufficient for schema-ful systems. However, for schemaless systems like MongoDB and Apache AsterixDB, utilizing a strongly-typed language would require adding additional checks to ensure the types of each processed value, which means more branches in the generated code. The Truffle framework addresses this issue for dynamically-typed languages such as Python and JavaScript. Recent work [50] has proposed using the Truffle framework for code generation and query compilation to run Language-integrated Query (LINQ) over a dynamically-typed collection in JavaScript or R and showed that the performance of their approach was comparable to hand-written code. In this work, we have also used the Truffle framework, where we generate code for parts of a query plan in Apache AsterixDB.

8 CONCLUSION

In this paper, we presented several techniques to store and query data in a columnar format for schemaless, LSM-based document stores. We first proposed several extensions to the Dremel format to make storing arrays' values more concise and to accommodate heterogeneous data values. Next, we introduced AMAX, a columnar layout for organizing and storing records in LSM-based document stores. Experiments showed that the AMAX layout significantly reduced the overall storage overhead compared to the row-major formats. The impact of transforming records into columns during data ingestion varied according to the structure of the ingested records, and it was seen that the AMAX layout's ingestion rate was relatively faster as compared to AsterixDB's current schemaless format. With the proposed code generation technique, the AMAX layout improved query performance by orders of magnitude.

ACKNOWLEDGMENTS

This work was supported by a fellowship from KACST. It was also supported by NSF awards IIS-1838248 and CNS-1925610, industrial support from Amazon, Google, Microsoft and Couchbase, and the Donald Bren Foundation (via a Bren Chair).

REFERENCES

[1] 2021. Actian Vector. https://esd.actian.com/product/Vector
[2] 2021. Apache AsterixDB. https://asterixdb.apache.org
[3] 2021. Apache AsterixDB Object Serialization Reference. https://cwiki.apache.org/confluence/display/ASTERIXDB/AsterixDB+Object+Serialization+Reference
[4] 2021. Apache Drill. https://drill.apache.org
[5] 2021. Apache Kudu. https://kudu.apache.org
[6] 2021. Apache Parquet. https://parquet.apache.org
[7] 2021. CADRE: Collaborative Archive Data Research Environment. http://iuni.iu.edu/resources/cadre
[8] 2021. ClickHouse. https://clickhouse.tech
[9] 2021. Couchbase. https://couchbase.com
[10] 2021. MonetDB. https://www.monetdb.org
[11] 2021. MongoDB. https://www.mongodb.com
[12] 2021. Parquet encoding. https://github.com/apache/parquet-format/blob/master/Encodings.md
[13] 2021. RocksDB. https://rocksdb.org
[14] 2021. Rockset. https://rockset.com
[15] 2021. Snappy. http://google.github.io/snappy
[16] 2021. Twitter API Documentation. https://developer.twitter.com/en/docs.html
[17] 2021. Vertica. https://www.vertica.com
[18] 2021. xml-to-json: Library and command line tool for converting XML files to json. http://hackage.haskell.org/package/xml-to-json
[19] 2022. Parquet Documentation. https://parquet.apache.org/documentation/latest/
[20] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *PVLDB* 8, 8 (2015), 850–861.
[21] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal* 11, 3, 198–215.
[22] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2020. An LSM-based tuple compaction framework for Apache AsterixDB. *PVLDB* 13, 9 (2020), 1388–1400.
[23] Wail Y. Alkowaileet and Michael J. Carey. 2021. Columnar Formats for Schemaless LSM-based Document Stores. arXiv:2111.11517 [cs.DB]
[24] Sattam Alsubaiee et al. 2014. AsterixDB: A Scalable, Open Source BDMS. *PVLDB* 7, 14 (2014).
[25] Sattam Alsubaiee et al. 2014. Storage Management in AsterixDB. *PVLDB* 7, 10 (2014).
[26] Sattam Alsubaiee and Vinayak Borkar. 2017. Method, apparatus, and computer-readable medium for encoding repetition and definition level values for semi-structured data. US Patent App. 15/208,032.
[27] Michael Armbrust et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *PVLDB* 13, 12 (2020), 3411–3424.
[28] Vinayak Borkar et al. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *International Conference on Data Engineering (ICDE)*.
[29] Vinayak Borkar et al. 2015. Algebricks: a data model-agnostic compiler backend for big data languages. In *SoCC*.
[30] Michael J Carey. 2019. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. In *International Conference on Data Engineering (ICDE)*.
[31] Don Chamberlin. 2018. *SQL++ For SQL Users: A Tutorial.* Couchbase, Inc. (Available at Amazon.com).
[32] Julien Le Dem. 2013. Dremel made simple with Parquet. *Twitter Blog* (2013). https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html
[33] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *ACM International Conference on Management of Data (SIGMOD)*.
[34] Goetz Graefe and William J McKenna. 1993. The Volcano optimizer generator: Extensibility and efficient search. In *International Conference on Data Engineering (ICDE)*. IEEE, 209–218.
[35] Per-Åke Larson et al. 2011. SQL server column store indexes. In *ACM International Conference on Management of Data (SIGMOD)*. 1177–1184.
[36] Per-Ake Larson et al. 2013. Enhancements to SQL server column stores. In *ACM International Conference on Management of Data (SIGMOD)*. 1159–1168.
[37] Raymond A. Lorie. 1974. XRM - An extended (N-ary) relational memory. *IBM Research Report* G320-2096 (1974).
[38] Chen Luo and Michael J Carey. 2019. Efficient data ingestion and query processing for LSM-based storage systems. *PVLDB* 12, 5 (2019).
[39] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1.
[40] Chen Luo and Michael J Carey. 2020. On performance stability in LSM-based storage systems. *PVLDB* 13, 4 (2020), 449–462.
[41] Stefan Manegold, Martin L Kersten, and Peter Boncz. 2009. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB* 2, 2 (2009), 1648–1653.
[42] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *PVLDB* 3, 1-2 (2010), 330–339.
[43] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB* 11, 1 (2017), 1–13.
[44] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011), 539–550.
[45] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.
[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996).
[47] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631* (2014).
[48] Sriram Padmanabhan, Timothy Malkemus, Anant Jhingran, and Ramesh Agarwal. 2001. Block oriented processing of relational database operations in modern computer architectures. In *International Conference on Data Engineering (ICDE)*. IEEE, 567–574.
[49] Reynold Xin Sameer Agarwal, Davies Liu. 2013. Apache Spark as a compiler: Joining a billion rows per second on a laptop. *Databricks Blog* (2013). https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html
[50] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2021. Language-Agnostic Integrated Queries in a Managed Polyglot Runtime. *PVLDB* 14, 8 (2021), 1–13.
[51] Mike Stonebraker et al. 2005. C-store: A column-oriented DBMS. In *VLDB* (Trondheim, Norway). 553–564.
[52] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. 2014. Sinew: a SQL system for multi-structured data. In *ACM International Conference on Management of Data (SIGMOD)*. 815–826.
[53] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204.
[54] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin* 28, 2 (2005), 17–22.