An LSM-based Tuple Compaction Framework for Apache AsterixDB

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

Sattam Alsubaiee
King Abdulaziz City for Science and Technology
Riyadh, Saudi Arabia
ssubaiee@kacst.edu.sa

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

ABSTRACT

Document database systems store self-describing semi-structured records, such as JSON, "as-is" without requiring the users to pre-define a schema. This provides users with the flexibility to change the structure of incoming records without worrying about taking the system offline or hindering the performance of currently running queries. However, the flexibility of such systems does not free. The large amount of redundancy in the records can introduce an unnecessary storage overhead and impact query performance.

Our focus in this paper is to address the storage overhead issue by introducing a tuple compactor framework that infers and extracts the schema from self-describing semi-structured records during the data ingestion. As many prominent document stores, such as MongoDB and Couchbase, adopt Log Structured Merge (LSM) trees in their storage engines, our framework exploits LSM lifecycle events to piggyback the schema inference and extraction operations. We have implemented and empirically evaluated our approach to measure its impact on storage, data ingestion, and query performance in the context of Apache AsterixDB.

PVLDB Reference Format:
Wail Y. Alkowaileet, Sattam Alsubaiee and Michael J. Carey. An LSM-based Tuple Compaction Framework for Apache AsterixDB. *PVLDB*, 13(9): 1388 - 1400, 2020.
DOI: https://doi.org/10.14778/3397230.3397236

1. INTRODUCTION

Self-describing semi-structured data formats like JSON have become the de facto format for storing and sharing information as developers are moving away from the rigidity of schemas in the relational model. Consequently, NoSQL Database Management Systems (DBMSs) have emerged as popular solutions for storing, indexing, and querying self-describing semi-structured data. In document store systems such as MongoDB [11] and Couchbase [10], users are not required to define a schema before loading or ingesting their data since each data instance is self-describing (i.e., each record embeds metadata that describes its structure and values). The flexibility of the self-describing data model provided by NoSQL systems attracts applications where the schema can change in the future by adding, removing, or even changing the type of one or more values without taking the system offline or slowing down the running queries.

The flexibility provided in document store systems over the rigidity of the schemas in Relational Database Management Systems (RDBMSs) does not come without a cost. For instance, storing a boolean value for a field named *hasChildren*, which takes roughly one byte to store in an RDBMS, can take a NoSQL DBMS an order of magnitude more bytes to store. Defining a schema prior to ingesting the data can alleviate the storage overhead, as the schema is then stored in the system's catalog and not in each record. However, defining a schema defies the purpose of schema-less DBMSs, which allow adding, removing or changing the types of the fields without manually altering the schema [18]. From a user perspective, declaring a schema requires a thorough a priori understanding of the dataset's fields and their types.

Let us consider a scenario where a data scientist wants to ingest and analyze a large volume of semi-structured data from a new external data source without prior knowledge of its structure. Our data scientist starts by acquiring a few instances from the data source and tries to analyze their structures; she then builds a schema according to the acquired sample. After ingesting a few data instances, our data scientist discovers that some fields can have more than one type, which was not captured in her initial sample. As a result, she stops the ingestion process, alters the schema to accommodate the irregularities in the types of those fields, and then reinitiates the data ingestion process. In this case, our data scientist has to continuously monitor the system and alter the schema if necessary, which may result in taking the system offline or stopping the ingestion of new records. Having an automated mechanism to infer and consolidate the schema information for the ingested records without losing the flexibility and the experience of schema-less stores would clearly be desirable.

In this work, we address the problem of the storage overhead in document stores by introducing a framework that infers and compacts the schema information for semi-structured data during the ingestion process. Our design utilizes the lifecycle events of Log Structured Merge (LSM) tree [32] based storage engines, which are used in many prominent document store systems [10, 11] including Apache AsterixDB [19]. In LSM-backed engines, records are first accumulated in memory (*LSM in-memory component*) and

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397236

then subsequently written sequentially to disk (*flush operation*) in a single batch (*LSM on-disk component*). Our framework takes the opportunity provided by LSM *flush* operations to extract and strip the metadata from each record and construct a schema for each flushed LSM component. We have implemented and empirically evaluated our framework to measure its impact on the storage overhead, data ingestion rate and query performance in the context of Apache AsterixDB. Our main contributions can be summarized as follows:

- We propose a mechanism that utilizes the LSM workflow to infer and compact the schema for NoSQL systems semi-structured records during flush operations. Moreover, we detail the steps required for distributed query processing using the inferred schema.
- We introduce a non-recursive physical data layout that allows us to infer and compact the schema efficiently for nested semi-structured data.
- We introduce page-level compression in AsterixDB. This is a similar solution to these adopted by other NoSQL DBMSs to reduce the storage overhead of self-describing records. We refer interested readers to the extended version [17] of our paper for more details.
- We evaluate the feasibility of our design, prototyped using AsterixDB, to ingest and query a variety of large semi-structured datasets.

The remainder of this paper is structured as follows: Section 2 provides a preliminary review of the AsterixDB architecture. Section 3 details the design and implementation of our tuple compaction framework in AsterixDB. Section 4 presents an experimental evaluation of the proposed framework. Section 5 discusses related work on utilizing the LSM lifecycle and on schema inference for semi-structured data. Finally, Section 6 presents our conclusions and discusses potential future directions for our work.

2. APACHE ASTERIXDB OVERVIEW

In this paper, we use Apache AsterixDB to prototype our tuple compactor framework. AsterixDB is a parallel semi-structured Big Data Management System (BDMS) which runs on large, shared-nothing, commodity computing clusters. To prepare the reader, we give a brief overview of AsterixDB [18, 24] and its query execution engine Hyracks [22].

2.1 User Model

The AsterixDB Data Model (ADM) extends the JSON data model to include types such as temporal and spatial types as well as data modeling constructs (e.g., bag or multiset). Defining an ADM *datatype* (akin to a schema in an RDBMS) that describes at least the primary key(s) is required to create a *dataset* (akin to a table in an RDBMS).

There are two options when defining a datatype in AsterixDB: *open* and *closed*. Figure 1 shows an example of defining a dataset of employee information. In this example, we first define *Dependent*, which declares two fields *name* and *age* of types *string* and *int*, respectively. Then, we define *EmployeeType*, which declares *id*, *name* and *dependents* of types *int*, *string* and a multiset of *Dependent*, respectively. The symbol "?" indicates that a field is optional. Note that we defined the type *EmployeeType* as *open*, where data instances of this type can have additional undeclared fields. On the other hand, we define the *Dependent* as *closed*, where data instances can only have declared fields. In both the open and closed datatypes, AsterixDB does not permit data instances that do not have values for the specified non-optional fields. Finally, in this example, we create a dataset *Employee* of the type *EmployeeType* and specify its *id* field as the primary key.

```
CREATE TYPE Dependent            CREATE TYPE EmployeeType
AS CLOSED {                      AS OPEN {
   name: string,                  id: int,
   age: int                       name: string,
};                                dependents:{{Dependent}}?
                                 };
CREATE DATASET Employee(EmployeeType) PRIMARY KEY id;
```

Figure 1: Defining Employee type and dataset in ADM

To query the data stored in AsterixDB, users can submit their queries written in SQL++ [25, 33], a SQL-inspired declarative query language for semi-structured data. Figure 2 shows an example of a SQL++ aggregate query posed against the dataset declared in Figure 1.

```
SELECT VALUE nameGroup FROM Employee AS emp
GROUP BY emp.name GROUP AS nameGroup
```

Figure 2: An example of a SQL++ query

2.2 Storage and Data Ingestion

In an AsterixDB cluster, each worker node (Node Controller, or NC for short) is controlled by a Cluster Controller (CC) that manages the cluster's topology and performs routine checks on the NCs. Figure 3 shows an AsterixDB cluster of three NCs, each of which has two data partitions that hold data on two separate storage devices. Data partitions in the same NC (e.g., Partition 0 and Partition 1 in NC0) share the same buffer cache and memory budget for LSM in-memory components; however, each partition manages the data stored in its storage device independently. In this example, NC0 also acts as a metadata node, which stores and provides access to AsterixDB metadata such as the defined datatypes and datasets.

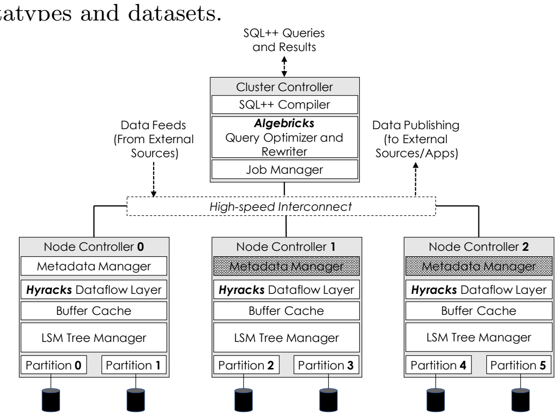


Figure 3: Apache AsterixDB cluster with three NCs

AsterixDB stores the records of its datasets, spread across the data partitions in all NCs, in primary LSM B^+-tree indexes. During data ingestion, each new record is hash-partitioned using the primary key(s) into one of the configured partitions (Partition 0 to Partition 5 in Figure 3) and inserted into the dataset's LSM in-memory component. AsterixDB implements a no-steal/no-force buffer management policy with write-ahead-logging (WAL) to ensure the durability and atomicity of ingested data. When the in-memory component is full and cannot accommodate new records, the *LSM Tree Manager* (called the "tree manager" hereafter) schedules a *flush operation*. Once the flush operation is triggered, the tree manager writes the in-memory component's records into a new LSM on-disk component on the partition's storage device, Figure 4a. On-disk components during their flush operation are considered *INVALID* components. Once it is completed, the tree manager marks the flushed component as *VALID* by setting a validity bit in the component's *metadata page*. After this point, the tree manager can safely delete the logs for the flushed component. During crash recovery, any disk component with an unset validity bit is considered invalid and removed. The recovery manager can then replay the logs to restore the state of the in-memory component before the crash.

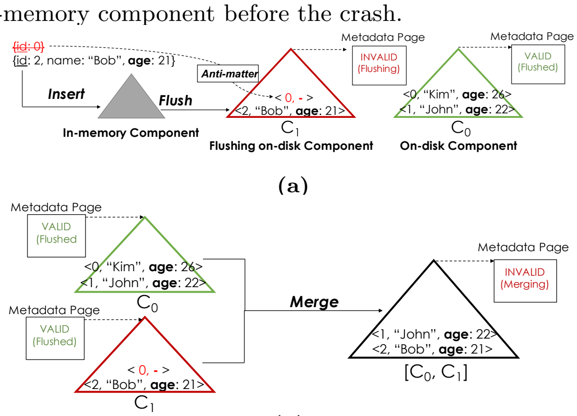


Figure 4: (a) Flushing component C_1 **(b)** Merging the two components C_0 and C_1 into a new component $[C_0, C_1]$

Once flushed, 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 [19] to indicate that a record with a specified key has been deleted. An upsert is simply a delete followed by an insert with the same key. For example, in Figure 4a, we delete the record with $id = 0$. Since the target record is stored in C_0, we insert an "anti-matter" entry to indicate that the record with $id = 0$ is deleted. As on-disk components accumulate, the tree manager periodically merges them into larger components according to a merge policy [19, 30] that determines when and what to merge. Deleted and updated records are garbage-collected during the merge operation. In Figure 4b, after merging C_0 and C_1 into $[C_0, C_1]$, we do not write the record with $id = 0$ as the record and the anti-matter entry annihilate each other. As in the flush operation, on-disk components created by a merge operation are considered *INVALID* until their operation is completed. After completing the merge, older on-disk components (C_0 and C_1) can be safely deleted.

On-disk components in AsterixDB are identified by their *component IDs*, where flushed components have monotonically increasing component IDs (e.g., C_0 and C_1) and merged components have components IDs that represent the range of component IDs that were merged (e.g., $[C_0, C_1]$). AsterixDB infers the recency ordering of components by inspecting the component ID, which can be useful for maintenance [30]. In this work, we explain how to use this property later in Section 3.2.

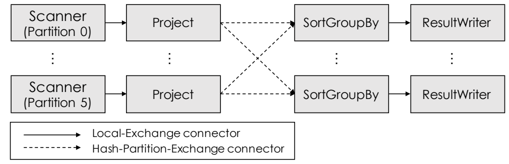


Figure 5: A compiled Hyracks job for the query in Figure 2

Datasets' records (of both *open* and *closed* types) in the LSM primary index are stored in a binary-encoded physical ADM format [3]. Records of *open* types that have undeclared fields are self-describing, i.e., the records contain additional information about the undeclared fields such as their types and their names. For our example in Figure 4, AsterixDB stores the information about the field **age** as it is not declared. For declared fields (*id* and *name* in this example), their type and name information are stored separately in the metadata node (NC0).

2.3 Runtime Engine and Query Execution

To run a query, the user submits an SQL++ query to the CC, which optimizes and compiles it into a Hyracks job. Next, the CC distributes the compiled Hyracks job to the query executors in all partitions where each executor runs the submitted job in parallel [1].

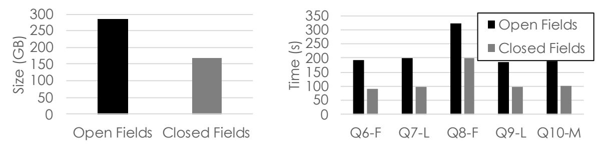
Hyracks jobs consist of *operators* and *connectors*, where data flows between operators over connectors as a batch of records (or a frame of records in Hyracks terminology). Figure 5 depicts the compiled Hyracks job for the query in Figure 2. As shown in Figure 5, records can flow within an executor's operators through *Local-Exchange* connectors or they can be repartitioned or broadcast to other executors' operators through non-local exchange connectors such as the *Hash-Partition-Exchange* connector in this example.

Operators in a Hyracks job process the ADM records in a received frame using AsterixDB-provided functions. For instance, a field access expression in a SQL++ query is translated into AsterixDB's internal function $getField()$. AsterixDB's compiler *Algebricks* [23] may rewrite the translated function when necessary. As an example, the field access expression $e.name$ in the query shown in Figure 2 is first translated into a function call $getField(emp, \text{"}name\text{"})$ where the argument emp is a record and *"name"* is the name of the requested field. Since *name* is a declared field, Algebricks can rewrite the field access function to $getField(emp, 1)$ where the second argument **1** corresponds to the field's index in the schema provided by the Metadata Node.

3. SCHEMA INFERENCE AND TUPLE COMPACTION FRAMEWORK

The flexibility of schema-less NoSQL systems attracts applications where the schema can change without declaring those changes. However, this flexibility is not free. In the context of AsterixDB, Pirzadeh et al. [34] explored query execution performance when all the fields are declared (*closed type*) and when they are left undeclared (*open type*). One conclusion from their findings, summarized in Figure 6, is that queries with non-selective predicates (using secondary indexes) and scan queries took twice as much time to execute against *open* type records compared to *closed* type records due to their storage overhead.

[1]The default number of query executors is equal to the number of data partitions in AsterixDB.



(a) On-disk storage size **(b)** Query execution time

Figure 6: Summary of the findings in [34]

In this section, we present a tuple compactor framework (called the "tuple compactor" hereafter) that addresses the storage overhead of storing self-describing semi-structured records in the context of AsterixDB. The tuple compactor automatically infers the schema of such records and stores them in a compacted form without sacrificing the user experience of schema-less document stores. Throughout this section, we run an example of ingesting and querying data in the *Employee* dataset declared as shown in Figure 7. The *Employee* dataset here is declared with a configuration parameter — {"tuple-compactor-enabled": **true**} — which enables the tuple compactor.

We present our implementation of the tuple compactor by first showing the workflow of inferring schema and compacting records during data ingestion and the implications of crash recovery in Section 3.1. In Section 3.2, we show the structure of an inferred schema and a way of maintaining it on update and delete operations. Then, in Section 3.3, we introduce a physical format for self-describing records that is optimized for the tuple compactor operations (schema inference and record compaction). Finally, in Section 3.4, we address the challenges of querying compacted records stored in distributed partitions of an AsterixDB cluster.

```
CREATE TYPE EmployeeType AS OPEN { id: int };
CREATE DATASET Employee(EmployeeType)
PRIMARY KEY id WITH {"tuple-compactor-enabled": true};
```

Figure 7: Enabling the tuple compactor for a dataset

3.1 Tuple Compactor Workflow

We first discuss the tuple compactor workflow during normal operation of data ingestion and during crash recovery.

Data Ingestion. When creating the *Employee* dataset (shown in Figure 7) in the AsterixDB cluster illustrated in Figure 3, each partition in every NC starts with an empty dataset and an empty schema. During data ingestion, newly incoming records are hash-partitioned on the primary keys (*id* in our example) across all the configured partitions (Partition 0 to Partition 5 in our example). Each partition inserts the received records into the dataset's in-memory component until it cannot hold any new record. Then, the tree manager schedules a flush operation on the full in-memory component. During the flush operation, the tuple compactor, as shown in the example in Figure 8a, factors the schema information out of each record and builds a traversable in-memory structure that holds the schema (described in Section 3.2). At the same time, the flushed records are written into the on-disk component C_0 in a compacted form where their schema information (such as field names) are stripped out and stored in the schema structure. After inserting the last record into the on-disk component C_0, the inferred schema S_0 in our example describes two fields *name* and *age* with their associated types denoted as $FieldName : Type$ pairs. Note that we do not store the schema information of any explicitly declared fields (field *id* in this example) as they are stored in the Metadata Node (Section 2.2). At the end of the flush operation, the component's inferred in-memory schema is persisted in the component's Metadata Page before setting the component as $VALID$. Once persisted, on-disk schemas are immutable.

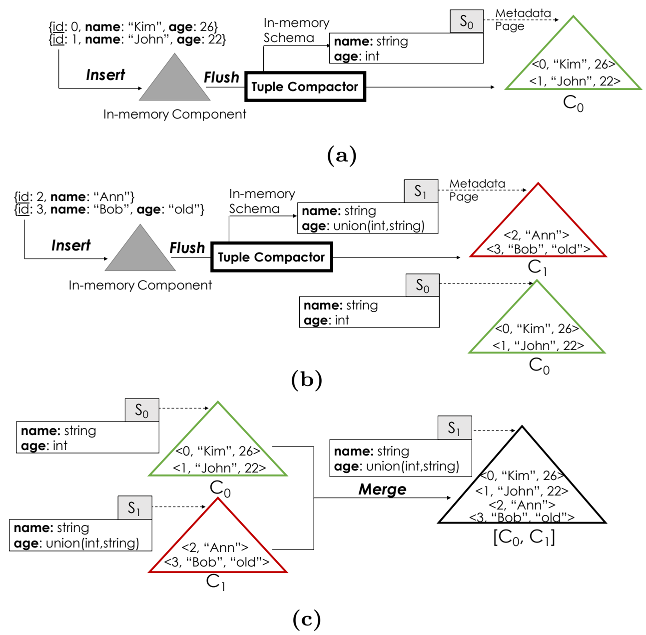


Figure 8: (a) Flushing the first component C_0 **(b)** Flushing the second component C_1 **(c)** Merging the two components C_0 and C_1 into the new component $[C_0, C_1]$

As more records are ingested by the system, new fields may appear or fields may change, and the newly inferred schema has to incorporate the new changes. The newly inferred schema will be a super-set (or union) of all the previously inferred schemas. To illustrate, during the second flush of the in-memory component to the on-disk component C_1 in Figure 8b, the records of the new in-memory component, with *id* 2 and 3, have their *age* values as *missing* and *string*, respectively. As a result, the tuple compactor changes the type of the inferred age field in the in-memory schema from *int* to $union(int, string)$, which describes the records' fields for both components C_0 and C_1. Finally, C_1 persists the latest in-memory schema S_1 into its metadata page.

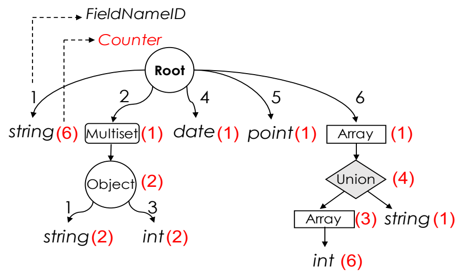
Given that the newest schema is always a super-set of the previous schemas, during a merge operation, we only need to store the most recent schema of all the mergeable components as it covers the fields of all the previously flushed components. For instance, Figure 8c shows that the resulting on-disk component $[C_0, C_1]$ of the merged components C_0 and C_1 needs only to store the schema S_1 as it is the most recent schema of $\{S_0, S_1\}$.

We chose to ignore compacting records of the in-memory component because (i) the in-memory component size is relatively small compared to the total size of the on-disk components, so any storage savings will be negligible, and (ii) maintaining the schema for in-memory component, which permits concurrent modifications (inserts, deletes and updates), would complicate the tuple compactor's workflow and slow down the ingestion rate.

Crash Recovery. The tuple compactor inherits the LSM guarantees for crash recovery (see Section 2.2). To illustrate, let us consider the case where a system crash occurs during

```
{
  "id": 1,
  "name": "Ann",
  "dependents":{{
    {"name": "Bob", "age": 6},
    {"name": "Carol", "age" : 10} }},
  "employment_date": date("2018-09-20"),
  "branch_location": point(24.0, -56.12),
  "working_shifts": [[8, 16], [9, 17], [10, 18], "on_call"]
}
... + 5 more {"id": int, "name": string} records ...
```

(a)



(b)

FieldNameID	field name
1	name
2	dependents
3	age
4	employement_date
5	branch_location
6	working_shifts

(c)

Figure 9: (a) An ADM record **(b)** Inferred schema tree structure **(c)** Dictionary-encoded field names

the second flush as shown in Figure 8b. When the system restarts, the recovery manager will start by activating the dataset and then inspecting the validity of the on-disk components by checking their validity bits. The recovery manager will discover that C_1 is not valid and remove it. As C_0 is the "newest" valid flushed component, the recovery manager will read and load its schema S_0 into memory. Then, the recovery manager will replay the log records to restore the state of the in-memory component before the crash. Finally, the recovery manager will flush the restored in-memory component to disk as C_1, during which time the tuple compactor operates normally.

3.2 Schema Structure

Previously, we showed the flow of inferring the schema and compacting the tuples during data ingestion. In this section, we focus on the inferred schema and present its structure. We also address the issue of maintaining the schema in case of delete and update operations, which may result in removing inferred fields or changing their types.

3.2.1 Schema Structure Components

Semi-structured records in document store systems are represented as a tree where the inner nodes of the tree represent nested values (e.g., JSON objects or arrays) and the leaf nodes represent scalar values (e.g., strings). ADM records in AsterixDB also are represented similarly. Let us consider the example where the tuple compactor first receives the ADM record shown in Figure 9a during a flush operation followed by five other records that have the structure `{"id": int, "name": string}`. The tuple compactor traverses the six records and constructs: (i) a tree-structure that summarizes the records structure, shown in Figure 9b, and (ii) a dictionary that encodes the inferred field names strings into *FieldNameIDs*, as shown in Figure 9c. The *Counter* in the schema tree-structure represents the number of occurrences of a value, which we further explain in Section 3.2.2.

The schema tree structure starts with the root object node which has the fields at the first level of the record (*name*, *dependents*, *employment_date*, *branch_location*, and *working_shifts*). We do not store any information here about the dataset's declared field *id* as explained previously in Section 3.1. Each inner node (e.g., *dependents*) represents a nested value (object, array, or multiset) and the leaf nodes (e.g., *name*) represent the scalar (or primitive) values. Union nodes are for object fields or collection (array and multiset) items if their values can be of different types. In this example, the tuple compactor infers the array item type of the field *working_shifts* as a union type of an array and a string.

The edges between the nodes in the schema tree structure represent the nested structure of an ADM record. Each inner node of a nested value in the schema tree structure can have one or more children depending on the type of the inner node. Children of object nodes (e.g., fields of the *Root* object) are accessed by *FieldNameIDs* (shown as integers on the edges of object nodes in Figure 9b) that reference the stored field names in the dictionary shown in Figure 9c. Each field name (or $FieldNameID$ in the schema tree structure) of an object is unique, i.e., no two children of an object node share the same field name. However, children of different object nodes can share the same field name. Therefore, storing field names in a dictionary allows us to canonicalize repeated field names such as the field name *name*, which has appeared twice in the ADM record shown in Figure 9a. A collection node (e.g., *dependents*) have only one child, which represents the items' type. An object field or a collection item can be of heterogeneous value types, so, their types may be inferred as a union of different value types. In a schema tree structure, the number of children a union node can have depends on the number of supported value types in the system. For instance, AsterixDB has 27 different value types [2]. Hence, a union node could have up to 27 children.

3.2.2 Schema Structure Maintenance

In Section 3.1 we described the flow involved in inferring the schema of newly ingested records, where we "add" more information to the schema structure. However, when deleting or updating records, the schema structure might need to be changed by "removing" information. For example, the record with *id* 3 shown in Figure 8 is the only record that has an *age* field of type *string*. Therefore, deleting this record should result in changing the type of the field *age* from $union(int, string)$ to int as the dataset no longer has the field *age* as a string. From this example, we see that on delete operations, we need to (i) know the number of appearances of each value, and (ii) acquire the old schema of a deleted or updated record.

During the schema inference process, the tuple compactor counts the number of appearances of each value and stores it in the schema tree structure's nodes. In Figure 9b, each node has a *Counter* value that represents the number of times the tuple compactor has seen this node during the schema inference process. In the same figure, we can see that there are six records that have the field *name*, including the record shown in Figure 9a. Also, we can infer from the schema structure that all fields other than *name* belong to the record shown in Figure 9a. Therefore, after deleting this record, the schema structure should only have the field *name* as shown in Figure 10.

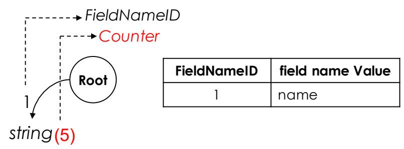


Figure 10: After deleting the record shown in Figure 9a

On delete, AsterixDB performs a point lookup to get the old record from which the tuple compactor extracts its schema (we call the schema of a deleted record the "*anti-schema*"). Then, it constructs an anti-matter entry that includes the primary key of the deleted record and its *anti-schema* and then inserts it into the in-memory component. During the flush operation, the tuple compactor processes the anti-schema by traversing it and decrementing the Counters in the schema tree structure. When the counter's value of a node in the schema tree structure reaches zero, we know that there is no record that still has this value. Then, the tuple compactor can safely delete the node from the schema structure. As shown in Figure 10, after deleting the record in Figure 9a, the counter value corresponding to the field *name* is decremented from 6 to 5 whereas the other nodes of the schema structure (shown in Figure 9a) have been deleted as they were unique to the deleted record. After processing the anti-schema, the tuple compactor discards it before writing the anti-matter entry to disk. Upserts can be performed as deletes followed by inserts.

It is important to note that performing point lookups for maintenance purposes is not unique to the schema structure. For instance, AsterixDB performs point lookups to maintain secondary indexes [19] and LSM filters [20]. Luo et al. [29, 30] showed that performing point lookups for every upsert operation can degrade data ingestion performance. More specifically, checking for key-existence for every upserted record is expensive, especially in cases where the keys are mostly new. As a solution, a primary key index, which stores primary keys only, can be used to check for key-existence instead of using the larger primary index. In the context of retrieving the anti-schema on upsert, one can first check if a key exists by performing a point lookup using the primary key index. Only if the key exists, an additional point lookup is performed on the primary index to get the anti-schema of the upserted record. If the key does not yet exist (new key), the record can be inserted as a new record. In Section 4, we evaluate the data ingestion performance of our tuple compactor under heavy updates using the suggested primary key index.

3.3 Compacted Record Format

Since the tuple compactor operates during data ingestion, the process of inferring the schema and compacting the records needs to be efficient and should not degrade the ingestion rate. As the schema can change significantly over time, previously ingested records must not be affected or require updates. Additionally, sparse records should not need to store additional information about missing values such as null bitmaps in RDBMSs' records. For example, storing the record `{"id": 5, "name": "Will"}` with the schema shown in Figure 9b should not include any information about other fields (e.g., dependents). Moreover, uncompacted records (in-memory components) and compacted records (on-disk components) should be evaluated and processed using the same evaluation functions to avoid any complexities when generating a query plan. To address those issues, we introduce a compaction-friendly physical record data format into AsterixDB, called the vector-based format.

3.3.1 *Vector-based Physical Data Format*

The main idea of the vector-based format is that it separates the metadata and values of a self-describing record into vectors that allow us to manipulate the record's metadata efficiently during the schema inference and record compaction processes. To not be confused with a columnar format, the vectors are stored within each record and the records are stored contiguously in the primary index (Section 2.2). Figure 11 depicts the structure of a record in the vector-based format. First comes the record's header, which contains information about the record such as its length. Next comes the values' tags vector, which enumerates the types of the stored primitive and nested values. Fixed-length primitive (or scalar) values such as integers are stored in the fixed-length values vector. The next vector is split into two sub-vectors, where the first stores lengths and the second stores the actual values of variable-length values. Lastly, the field names sub-vectors (lengths and values) store field name information for all objects' fields in the record.

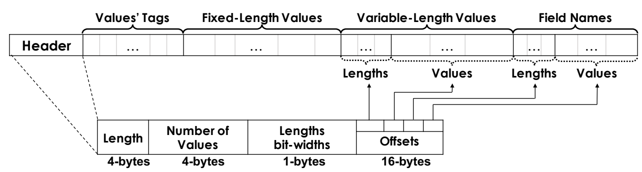


Figure 11: The structure of the vector-based format.

Figure 12 shows an example of a record in the vector-based format (we refere intersted readers to [17] for more examples). The record has four fields: *id*, *name*, *salaries*, and *age* with the types integer, string, array of integers and integer, respectively. Starting with the header, we see that the record's total size is 73-bytes and there are nine tags in the values' type tags vector. Lengths for variable-length values and field names are stored using the minimum amount of bytes. In our example, the maximum lengths of the variable-length values and field names are 3 (Ann) and 8 (salaries), respectively. Thus, we need at most 3-bits and 5-bits to store the length of each variable-length value or field name, respectively. We only actually need 4-bits for name lengths; however, the extra bit is used to distinguish inferred fields (e.g., name) from declared ones (e.g., id) as we explain next.

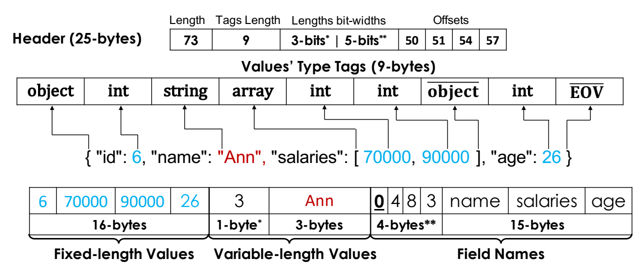


Figure 12: An example record in the vector-based format

After the header, we store the values' type tags. The values' type tags encode the tree structure of the record in a *Depth-First-Search* order. In this example, the record starts with an object type to indicate the root's type. The first value of the root object is of type integer, and it is stored in the first four bytes of the fixed-length values. Since an object tag precedes the integer tag, this value is a child of that object (root) and, hence, the first field name corresponds to it. Since the field *id* is a declared field, we only store its index (as provided by the metadata node) in the lengths sub-vector. We distinguish index values from length values by inspecting the first bit. If set, we know the length value is an index value of a declared field. The next value in the example record is of type string, which is the first variable-length value in the record. The string value is stored in the variable-length values' vector with its length. Similar to the previous integer value, this string value is also a child of the root and its field name (*name*) is next in the field names' vector. As the field *name* is not declared, the record stores both the name of the field and its length. After the string value, we have the array tag of the field *salaries*. As the array is a nested value, the subsequent tags (integers in this example) indicate the array items' types. The array items do not correspond to any field name, and their integer values are stored in the fixed-length values' vector. After the last item of the array, we store a control tag $\overline{object}$ to indicate the end of the array as the current nesting type and a return to the parent nesting type (object type in this example). Hence, the subsequent integer value (age) is again a child of the root object type. At the end of the value's tags, we store a control tag $\overline{EOV}$ to mark the end of the record.

As can be inferred from the previous example, the complexity of accessing a value in the vector-based format is linear in the number of tags, which is inferior to the logarithmic time provided by some traditional formats [3, 9]. We address this issue in more detail in Section 3.4.2.

3.3.2 Schema Inference and Tuple Compaction

Records in vector-based format separate values from metadata. The example shown in Figure 12 illustrates how the fixed-length and variable-length values are separated from the record's nested structure (values' types tags) and field names. When inferring the schema, the tuple compactor needs only to scan the values' type tags and the field names' vectors to build the schema structure.

Compacting vector-based records is a straightforward process. Figure 13 shows the compacted structure of the record in Figure 12 along with its schema structure after the compaction process. The compaction process simply replaces the field names string values with their corresponding FieldNameIDs after inferring the schema. It then sets the fourth offset to the field names' values sub-vector in the header (Figure 11) to zero to indicate that field names were removed and stored in the schema structure. As shown in the example in Figure 13, the record after the compaction needs just two bytes to store the field names' information, where each FieldNameID takes three bits (one bit for distinguishing declared fields and two for the IDs), as compared to the 19 (4+15) bytes in the uncompacted form in Figure 12.

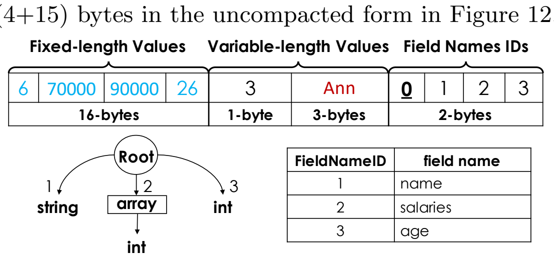


Figure 13: The record in Figure 12 after compaction

3.4 Query Processing

In this section, we explain our approach of querying compacted records in the vector-based format. We, first, show the challenges of having distributed schemas in different partitions and propose a solution that addresses this issue. Next, we zoom in into each query executor and show the optimizations needed to process compacted records.

3.4.1 Handling Heterogeneous Schemas

As a scalability requirement, the tuple compaction framework operates in each partition without any coordination with other partitions. Therefore, the schema in each partition can be different from other schemas in other partitions. When a query is submitted, each distributed partition executes the same job. Having different schemas becomes an issue when the requested query needs to repartition the data to perform a join or group-by. To illustrate, suppose we have two partitions for the same dataset but with two different inferred schemas, as shown in Figure 14. We see that the schemas in both partitions have the field *name* of type string. However, the second field is *age* in partition 0 and *salary* in partition 1. After hash-partitioning the records by the *name* value, the resulting records are shuffled between the two query executors and the last field can be either *age* or *salary*. Recall that partitions can be in different machines within the AsterixDB cluster and have no access to the schema information of other partitions. Consequently, query executors cannot readily determine whether the last field corresponds to *age* or *salary*.

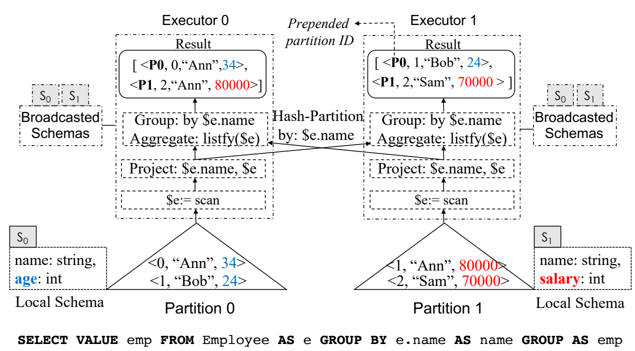


Figure 14: Two partitions with two different schemas

To solve the schema heterogeneity issue, we added functionality to broadcast the schema information of each partition to all nodes in the cluster at the beginning of a query's execution. Each node receives each partition's schema information along with its partition ID and serves the schemas to each executor in the same node. Then, we prepend each record resulting from the scan operator with the source partition ID. When an operator accesses a field, the operator uses both the prepended partition ID of the record and the distributed schema to perform the field access. Broadcasting the partitions' schemas can be expensive, especially in clusters with a large number of nodes. Therefore, we only broadcast the schemas when the query plan contains a non-local exchange operator such as the hash-partition-exchange in our example in Figure 14. When comparing the schema broadcasting mechanism to handling self-describing records, a broadcasted schema represents a *batch of records*, whereas the redundant schemas embedded in self-describing records are carried through the operators on a *record-by-record basis*.

Thus, transmitting the schema once per partition instead of once per record is more efficient.

3.4.2 Processing Compacted Records

One notable difference between the vector-based format and the ADM physical format is the time complexity of accessing a value (as discussed in Section 3.3.1). The AsterixDB query optimizer can move field access expressions within the plan when doing so is advantageous. For instance, the query optimizer inlines field access expressions with WHERE clause conjunct expressions as in the example:

$$emp.age > 25\ AND\ emp.name = \text{“}Ann\text{”}$$

The inlined field access expression *emp.name* is evaluated only if the expression *emp.age* > 25 is true. However, in the vector-based format, each field access requires a linear scan on the record's vectors, which could be expensive. To minimize the cost of scanning the record's vectors, we added one rewrite rule to the AsterixDB query optimizer to consolidate field access expressions into a single function expression. Therefore, the two field access expressions in our example will be written as follows:

$$[\$age, \$name] \leftarrow getValues(emp, \text{“}age\text{”}, \text{“}name\text{”})$$

The function $getValues()$ takes a record and path expressions as inputs and outputs the requested values of the provided path expressions. The two output values are assigned to two variables $\$age$ and $\$name$ and the final conjunct expression of our WHERE clause example is transformed as:

$$\$age > 25\ AND\ \$name = \text{“}Ann\text{”}$$

The function $getValues()$ is also used for accessing array items by providing the item's index. For example, the expression $emp.dependents[0].name$ is translated as follows:

$$[\$d\_name] \leftarrow getValues(emp, \text{“}dependents\text{”}, 0, \text{“}name\text{”})$$

Additionally, we allow "wildcard" index to access nested values of all items of an array. For instance, the output of the expression $emp.dependents[*].name$ is an array of all *name*s' values in the array of objects *dependents*.

4. EXPERIMENTS

In this section, we experimentally evaluate the implementation of our tuple compactor in AsterixDB. In our experiments, we compare our compacted record approach with AsterixDB's current *closed* and *open* records in terms of (i) on-disk storage size after data ingestion, (ii) data ingestion rate, and (iii) the performance of analytical queries. Additionally, we evaluated the performance of accessing values in records in the vector-based format (Section 3.3.1) with and without the optimizations explained in Section 3.4.2.

We also conducted additional experiments, summarized in Section 4.5 and detailed in our extended paper [17], to further evaluate other aspects of the proposed framework.

Experiment Setup We conducted our initial experiments using a single machine with an 8-core (Intel i9-9900K) processor and 32GB of main memory. The machine is equipped with two storage drive technologies SATA SSD and NVMe SSD, both of which have 1TB of capacity. The SATA SSD drive can deliver up to 550 MB/s for sequential read and 520 MB/s for sequential write, and the NVMe SSD drive can deliver up to 3400 MB/s for sequential read and 2500 MB/s for sequential write.

We used AsterixDB v9.5.0 after extending it with our tuple compaction framework. We configured AsterixDB with 15GB of total memory, where we allocated 10GB for the buffer cache and 2GB for the in-memory component budget. The remaining 3GB is allocated as temporary buffers for operations such as sort and join. Throughout our experiments, we also evaluate the impact of our page-level compression (detailed in [17]) using Snappy [13] on the storage size, data ingestion rate, and query performance.

Schema Configuration. In our experiments, we evaluated the storage size, data ingestion rate, and query performance when defining a dataset as **(i) open, (ii) closed, and (iii) inferred** using our tuple compactor. For the open and inferred datasets, we only declare the primary key field, whereas in closed datasets, we pre-declare all the fields. The records of open and closed datasets are stored using the ADM physical format, whereas the inferred datasets are using the new vector-based format. Note that the AsterixDB open case is similar to what schema-less NoSQL systems, like MongoDB and Couchbase, do for storage.

4.1 Datasets

In our experiments, we used three datasets (two of which are listed in Table 1) that differ in terms of their records' structure, size, and value types. The third one, with more field heterogeneity, is included in [17].

Table 1: Datasets summary

	Twitter	Sensors
Source	Scaled	Synthetic
Total Size	200GB	122GB
# of Records	77.6M	25M
Record Size	∼2.7KB	5.1KB
Max. Depth	8	3
# of values (min, max, avg)	53, 208, 88	248, 248, 248
Dominant Type	String	Double

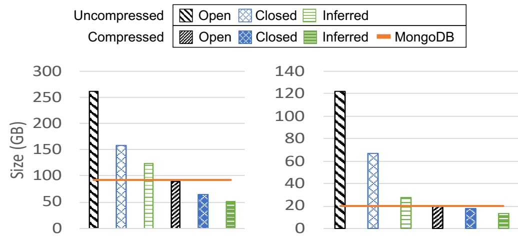
Using the first dataset, we want to evaluate ingesting and querying social network data. We obtained a sample of tweets using the Twitter API [14]. Due to the daily limit of the number of tweets that one can collect from the Twitter API, we replicated the collected tweets ten times to have 200GB worth of tweets in total. Replicating the data would not affect the experiment results as (i) the tuple compactor's scope is the records' metadata (not the values) and (ii) the original data is larger than the compressible page size.

To evaluate more numeric Internet of Things (IoT)-like workloads, we generated a second synthetic dataset that mimics data generated by sensors. Each record in the sensors' dataset contains captured readings and their timestamps along with other information that monitors the health status of the sensor. The sensor data contains mostly numerical values and has a larger field-name-size to value-size ratio. The total size of the raw Sensors data is 122GB.

4.2 Storage Size

In this experiment, we evaluate the on-disk storage size after ingesting the Twitter and the Sensors datasets into AsterixDB using the three formats (open, closed and inferred) and we compare it with MongoDB's storage size. Our goal of comparing with MongoDB's size is simply to show that the compressed open case is comparable to what other NoSQL systems take for storage using the same compression scheme (Snappy). (It is not the focus of this paper to compare both systems' data ingestion and query performance.)

We first evaluate the total on-disk sizes after ingesting the data into the open, closed and inferred datasets. We



(a) Twitter dataset **(b)** Sensors dataset

Figure 15: On-disk sizes

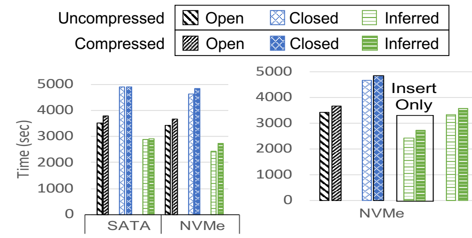
begin with the Twitter dataset. Figure 15a shows its total on-disk sizes. We see that the inferred and closed schema datasets have lower storage footprints compared to the open schema dataset, as both avoid storing field names in each record. When compression is enabled, both formats still have smaller size compared to the open format and to MongoDB's compressed collection size. The size of the inferred dataset is slightly smaller than the closed schema dataset since the vector-based format does not store offsets for every nested value (as opposed to the ADM physical format in the closed schema dataset).

The Sensors dataset contains only numerical values that describe the sensors' status along with their captured readings, so this dataset's field name size to value size ratio is higher compared to the previous datasets. Figure 15b shows that, in the uncompressed dataset, the closed and inferred datasets have about 2x and 4.3x less storage overhead, respectively, than the open dataset. The additional savings for the inferred dataset results from eliminating the offsets for readings objects, which contain reading values along with their timestamps — `{"value": double, "timestamp": bigint}`. Compression reduced the open and closed dataset sizes by a factor of 6.2 and 3.8, respectively, as compared to their uncompressed counterparts. For the inferred dataset, compression reduced its size only by a factor of 2.1. This indicates that both the open and closed dataset records incurred higher storage overhead from storing redundant offsets for nested fixed-length values (readings objects). As in the Twitter, the sizes of both the compressed open dataset in AsterixDB and the compressed collection in MongoDB were comparable in the Sensors dataset.

To summarize our findings, both the syntactic (page-level compression) and semantic (tuple compactor) approaches alleviated the storage overhead as shown in Figure 15. The syntactic approach was more effective than the semantic approach for the Twitter dataset. For the Sensors dataset, the semantic approach (with our vector-based format) was more effective for the reasons explained earlier. When combined, the approaches were able to reduce the overall storage sizes by 5x and 9.8x for the Twitter and Sensors datasets, respectively, compared to the open schema case in AsterixDB.

4.3 Ingestion Performance

We evaluated the performance of continuous data ingestion for the different formats using AsterixDB's data feeds for the Twitter dataset. We first evaluate the insert-only ingestion performance, without updates. In the second experiment, we evaluate the ingestion performance for an update-intensive workload, where previously ingested records are updated by either adding or removing fields or changing the types of existing data values. The latter experiment measures the overhead caused by performing point lookups to get the anti-schemas of previously ingested records. The



(a) Insert-only **(b)** 50% Updates

Figure 16: Data ingestion performance (Twitter Dataset)

Sensor dataset was also ingested through a data feed and showed similar behavior to the Twitter dataset; we omit these results here to conserve space. In [17], we evaluated

Data Feed (Insert-only). To evaluate the performance of continuous data ingestion, we measured the time to ingest the Twitter dataset using a data-feed to emulate Twitter's firehose. We set the maximum mergeable component size to 1GB and the maximum tolerable number of components to 5, after which the tree manager triggers a merge operation.

Figure 16a shows the time needed to complete the data ingestion for the 200GB Twitter dataset. Ingesting records into the inferred dataset took less time than ingesting into the open and closed datasets. Two factors played a role in the data ingestion rate. First, we observed that the record construction cost of the system's current ADM physical format was higher than the vector-based format by ∼40%. Due to its recursive nature, the ADM physical format requires copying the values of the child to the parent from the leaf to the root of the record, which means multiple memory copy operations for the same value. Closed records took even more time to enforce the integrity constraints such as the presence and types of none-nullable fields. The second factor was the IO cost of the flush operation. We noticed that the inferred dataset's flushed on-disk components are ∼50% and ∼25% smaller than the open and closed datasets, respectively. This is due to the fact that compacted records in the vector-based format were smaller in size than the closed and open records in ADM format (see Figure 15a). Thus, the cost of writing larger LSM components of both open and closed datasets was higher.

The ingestion rate for the SATA SSD and the NVMe SSD were comparable, as both were actually bottlenecked by flushing transaction log records to the disk. Enabling compression had a slight negative impact on the ingestion rate for each format due to the additional CPU cost.

Data Feed (50% Updates). As explained in Section 3.2.2, updates require point lookups to maintain the schema, which can negatively impact the data ingestion rate. We evaluated the ingestion performance for update-intensive workload when the tuple compactor is enabled. In this experiment, we randomly updated 50% of the previously ingested records by either adding or removing fields or changing existing value types. The updates followed a uniform distribution, where all records are updated equally. We created a primary key index, as suggested in [29, 30], to reduce the cost of point lookups of non-existent (new) keys. Figure 16b shows the ingestion time of Twitter dataset, using the NVMe SSD drive, for the open, closed and inferred datasets with updates. The ingestion times for both open and closed datasets were about the same as with no updates (Figure 16a). For the inferred dataset, the ingestion time with updates took ∼27% and ∼23% more time for the uncompressed and compressed datasets, respectively, as compared to no updates. The ingestion times of the inferred and open datasets were comparable and took less time than the closed dataset.

LSM Write-amplification. Continuous data ingestion from a data feed is sensitive to LSM configurations such as the merge-policy and the memory budget. AsterixDB's default "prefix-merge" policy [19] could then suffer from higher write-amplification by repeatedly merging smaller on-disk components until their combined size reaches a certain threshold. To eliminate those factors, we also evaluated the performance of bulkloading (detailed in [17]), which builds a single on-disk component for the loaded dataset. We observed that the cost of building the primary index was higher for both the open and closed schema datasets for the same reasons explained earlier and the cost of the LSM write amplification did not change the trends in Figure 16a.

4.4 Query Performance

We next evaluated the impact of our work on query performance by running analytical queries against the ingested Twitter and Sensor datasets. The objective of our experiments is to evaluate the IO cost of querying against open, closed, and inferred datasets. Each executed query was repeated 6 times and we report the average execution time of the last 5. All queries are listed in [17].

4.4.1 *Twitter Dataset*

We ran four queries against the Twitter dataset:

Q1. The number of records in the dataset — `COUNT(*)`.
Q2. The top ten users whose tweets' average length are the largest — `GROUP BY/ORDER BY`.
Q3. The top ten users who have the largest number of tweets that contain a popular hashtag — `EXISTS/GROUP BY/ORDER BY`.
Q4. All records of the dataset ordered by the tweets' posting timestamps — `SELECT */ORDER BY`[2].

Figure 17 shows the execution time for the four queries in the three datasets (open, closed, and inferred) when the data is on the SATA SSD drive and the NVMe SSD drive. On the SATA SSD, the execution times of the four queries, with and without compression, correlated with their on-disk sizes from Figure 15a. This correlation indicates that the IO cost dominates the execution time. However, on the NVMe SSD drive, the CPU cost becomes more evident, especially when page-level compression is enabled. For Q2 and Q4, the ∼2x reduction in storage after compression reduced their execution times in the SATA case in all three datasets. However, the execution times for Q2 and Q4 in the closed and inferred datasets did not improve as much after compression in the NVMe case, as the CPU became the bottleneck here. Q3, which filters out all records that do not contain the required hashtag, took less time to execute in the inferred dataset. This is due to the way that nested values of records in the vector-based format are accessed. In the Twitter dataset, hashtags are modeled as an array of objects; each object contains the hashtag text and its position in the tweet's text. We consolidate field access expressions for records in the vector-based format (Section 3.4.2), and the query optimizer was able to push the consolidated field access through the unnest operation and extract only the hashtag text instead of the hashtag objects. Consequently, Q3's intermediate result size was smaller in the inferred dataset compared to the other two datasets, and executing Q3 against the inferred dataset was faster.

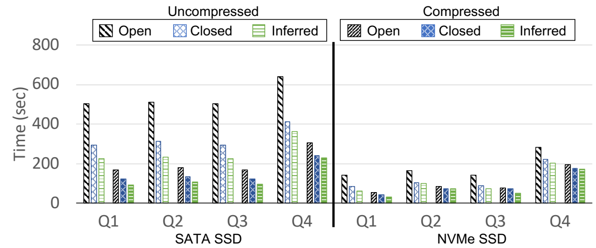


Figure 17: Query execution for the Twitter dataset

4.4.2 *Sensors Dataset*

We again ran four queries:

Q1. The number of records in the dataset — `COUNT(*)`.
Q2. The minimum and maximum reading values that were ever recorded across all sensors — `UNNEST/GROUP BY`.
Q3. The IDs of the top ten sensors that have recorded the highest average reading value — `UNNEST/GROUP BY/ORDER BY`.
Q4. Similar to Q4, but look for the recorded readings in a given day — `WHERE/UNNEST/GROUP BY/ORDER BY`

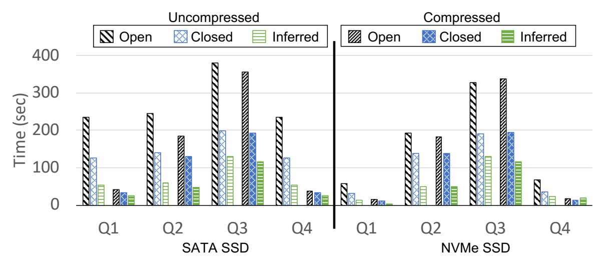


Figure 18: Query execution time for the Sensors dataset.

The execution times are shown in Figure 18. The execution times for Q1 on both uncompressed and compressed datasets correlate with the storage sizes of the datasets from Figure 15b. Q2 and Q3 exhibit the effect of consolidating and pushing down field value accesses of vector-based format, where both queries took significantly less time to execute in the inferred dataset. However, pushing the field access down is not always advantageous. When compression is enabled, the execution time of Q4 for the inferred dataset using NVMe SSD was the slowest. This is because the consolidated field accesses (of sensor ID, reading and reporting timestamp) are evaluated before filtering using a highly selective predicate (0.001%). In the open and closed datasets, delaying the evaluation of field accesses until after the filter for Q4 was beneficial. However, the execution times for the inferred dataset was comparable to the open case.

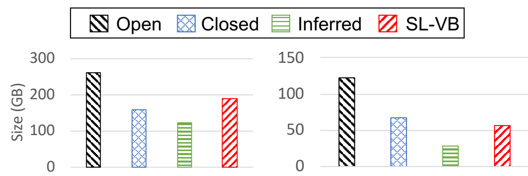
4.4.3 *Impact of the Vector-based Optimizations*

As we showed in our experiments, the time it takes for ingesting and querying records in the vector-based format (inferred) was smaller even when the schema is fully declared for the ADM format (closed). This is due to fact that the vector-baed format encodes nested values more efficiently using only the type tags (as in Section 3.3.1). To measure the impact of the newly proposed format, we reevaluate the storage size of the vector-based without inferring the schema

[2]In Q4, we report only the time for executing the query, excluding the time for actually retrieving the final formatted query result.

or compacting the records (i.e., a schema-less version using the vector-based format), which we refer to as *SL-VB*.

In Figure 19a, we see the total sizes of the four datasets *open, closed, inferred*, and *SL-VB* after ingesting the Twitter dataset. We see that the SL-VB dataset is smaller than the open dataset but slightly larger than the closed one. More specifically, about half of the storage savings in the inferred dataset (compared to the open dataset) is from the more efficient encoding of nested values in the vector-based format, and the other half is from compacting the record. For the Sensors dataset, Figure 19b shows a similar pattern; however, the SL-VB Sensors dataset is smaller than the closed dataset for the reasons explained in Section 4.2.



(a) Twitter **(b)** Sensors

Figure 19: Impact of the vector-based format on storage

Also, we showed that our optimizations of consolidating and pushing down field access expressions can tremendously improve query execution time. To isolate the factors that contributed to the performance gains, we reevaluated the execution times for Q2-Q4 of the Sensors dataset with and without these optimizations using the NVMe storage device.

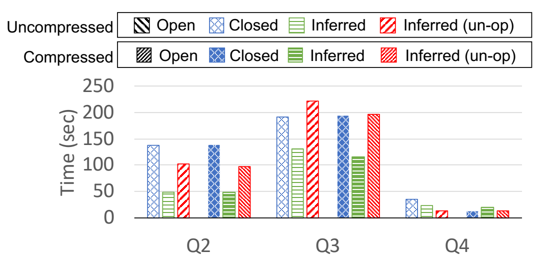


Figure 20: Impact of consolidating field access expressions

The execution times of the queries are shown in Figure 20. We refer to *Inferred (un-op)* as querying the inferred dataset without our optimization of consolidating and pushing down field access expressions. When not consolidated, the linear-time field accesses of the vector-based format are performed as many times as there are field access expressions in the query. For instance, Q3 has three field access expressions. Each field access requires scanning the record's vectors, which is expensive. Additionally, the size of the intermediate results of Q2 and Q3 were then larger (array of objects vs. array of doubles). As a result, Q2 and Q3 took twice as much time to finish for Inferred (un-op). Q2 was still faster to execute in the Inferred (un-op) case than in the closed case, whereas Q3 took slightly more time to execute. Finally, delaying the field accesses improved the execution time for queries with highly selective predicates, such as Q4.

4.5 Summary of Additional Experiments

We conducted additional experiments using the Twitter dataset to evaluate (i) the storage size and construction time of the vector-based format in comparison with schema-based formats; namely, Apache Avro, Apache Thrift and Google Protocol Buffers, (ii) the impact of the linear access in the vector-based format, (iii) the query performance with the presence of a secondary index, and (iv) the cluster scalability using Amazon EC2 instances.

Other formats, such as Apache Avro [4], Apache Thrift [8], and Google Protocol Buffers [12], also exploit schemas to store semi-structured data more efficiently. In fact, providing a schema is not optional for writing records in such formats — as opposed to the vector-based format, where the schema is optional. Nonetheless, we compared the vector-based format to Apache Avro, Apache Thrift using both Binary Protocol (BP) and Compact Protocol (CP), and Protocol Buffers to evaluate 1) the storage size and 2) the time needed to construct the records in each format using 52MB of the Twitter dataset. Table 2 summarizes the result of our experiment. We see that the storage sizes of the different formats were mostly comparable. In terms of the time needed to construct the records, Apache Thrift (for both protocols) took the least construction time followed by the vector-based format. Apache Avro and Protocol Buffers took 1.9x and 2.9x more time to construct the records compared to the vector-based format, respectively.

Table 2: Writing 52MB of Tweets in different formats

	Space (MB)	Time (msec)
Avro	27.49	954.90
Thrift (BP)	34.30	341.05
Thrift (CP)	25.87	370.93
ProtoBuf	27.16	1409.13
Vector-based	29.49	485.48

Accessing values in the vector-based format is sensitive to the position of the requested value. For instance, accessing a value that appears first in a record is faster than accessing a value that resides at the end. To evaluate the impact of the value's position, we ran four queries where each counts the number of appearances of a value. The positions (or indexes) of those values in the vector-based format were 1, 34, 68, and 136 for Q1, Q2, Q3 and Q4, respectively, where position 1 means the first value in the record and position 136 is the last. The impact of the value's position was negligible, and all queries took less time to execute in the inferred cases due to the storage savings. However, when all the data fits in-memory, the CPU cost becomes more apparent. When using all 8-cores, the execution time for all queries were about the same for the three datasets. In the case of a single core, the vector-based format was the slowest to execute Q3 and Q4.

Pirzadeh et al. [34] previously showed that predeclaring the schema in AsterixDB did not improve (notably) the performance of range-queries with highly selective predicates in the presence of a secondary index. We evaluated the impact of having a secondary index (on the *timestamp*) on query performance by running different queries with low and high selective predicates. We modified the scaled Twitter dataset by generating monotonically increasing values for the attribute *timestamp* to mimic the time at which users post their tweets. The storage savings had a negligible impact on the execution times for the queries with selectivities of 0.001% and 0.01% (as in [34]). However, the execution times for queries with higher selectivities (0.1%-0.5%) correlated with the storage sizes (Figure 15a), where the closed and inferred datasets have lower storage overhead.

Finally, to evaluate the scalability of our approach (including the handling of distributed schemas in different partitions), we conducted a scale-out experiment using a cluster of Amazon EC2 instances of type `c5d.2xlarge` (each with

16GB of memory and 8 virtual cores). We evaluated the ingestion and query performance of the Twitter dataset using clusters with 4, 8, 16 and 32 nodes. The raw sizes of the ingested data were 400, 800, 1600 and 3200 GB for the 4, 8, 16 and 32 node clusters, respectively. As expected, we observed the same trends as seen for the single node cluster (see Figures 15a and 16a), where the inferred dataset has the lowest storage overhead with the highest data ingestion rate. To evaluate query performance, we ran the same four queries as in Section 4.4.1. All four queries scaled linearly, as expected, and all four queries were faster in the inferred dataset. Since the data is shuffled in Q2 and Q3 to perform the parallel aggregation, each partition broadcasts its schema to the other nodes in the cluster (Section 3.4.1) at the start of a query. However, the performance of the queries was essentially unaffected and were still faster in the inferred dataset.

5. RELATED WORK

Schema inference for self-describing, semi-structured data has appeared in early work for Object Exchange Model (OEM) and later for XML and JSON documents. For OEM (and later for XML), [27] presented the concept of a dataguide, which is a summary structure for schema-less semi-structured documents. A dataguide could be accompanied with values' summaries and samples (annotations) about the data, which we also use in our schema structure to keep the number of occurrences in each value. In [37], Wang et al. present an efficient framework for extracting, managing and querying schema-view of JSON datasets. Their work targeted data exploration, where showing a frequently appearing structure can be good enough. However, in our work, the purpose of inferring the schema is to use it for compacting and querying the records, so, we infer the exact schema of the ingested dataset. In another work [26], the authors detail an approach for automatically inferring and generating a normalized (flat) schema for JSON-like datasets, which then can be utilized in an RDBMS to store the data. Our work here is orthogonal; we target document store systems with LSM-based storage engines.

Creating **secondary indexes** is related to declaring attributes in schema-less document stores. Azure DocumentDB [35] and MongoDB support indexing all fields at once without declaring the indexed fields explicitly. E.g., MongoDB allows users to create an index on all fields using a *wildcard index*. Doing so requires the system to "infer" the fields. Despite the similarities, our objective is different. In our work, we infer the schema to reduce storage overhead by compacting records residing in the primary index.

Semantically compacting self-describing, semi-structured records using schemas appears in popular big data systems such as Apache Spark [38] and Apache Drill [6]. For instance, Apache Drill uses schemas of JSON datasets (provided by the user or inferred by scanning the data) to transform records into a compacted in-memory columnar format (Apache Arrow [1]). File formats such as Apache Avro, Apache Thrift, and Google Protocol Buffers use the provided schema to store nested data in a compacted form. However, the schema is required for the those formats; whereas it is optional for the vector-based format. Apache Parquet [7] (or Google Dremel [31]) use the provided schema to store nested data in a columnar format to achieve higher compressibility. An earlier effort to semantically compact and compress XML data is presented in [21, 28]. Our work is different in targeting more "row"-oriented document stores with LSM-based storage engines. Also, we support data values with heterogeneous types, in contrast to Spark and Parquet.

Exploiting LSM lifecycle events to piggyback other operations to improve the query execution time is not new by itself and has been proposed in several contexts [15, 20, 36]. LSM-backed operations can be categorized as either non-transformative operations, such as computing information about the ingested data, or transformative operations, e.g., in which the records are transformed into a read-optimized format. An example of a non-transformative operation is [20], which shows how to utilize the LSM lifecycle operations to compute range-filters that can accelerate time-correlated queries by skipping on-disk components that do not satisfy the filter predicate. [15] proposes a lightweight statistics collection framework that utilizes LSM lifecycle events to compute statistical summaries of ingested data that the query optimizer can use for cardinality estimation. An example of a transformative operation is [36], which utilizes LSM-like operations to transform records in the writeable-store into a read-optimized format for the readable-store. Our work utilizes the LSM lifecycle operations to do both (i) non-transformative operations to infer the schema and (ii) transformative operations to compact the records.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduced a tuple compaction framework that addresses the overhead of storing self-describing records in LSM-based document store systems. Our framework utilizes the flush operations of LSM-based engines to infer the schema and compact the ingested records without sacrificing the flexibility of schema-less document store systems. We also addressed the complexities of adopting such a framework in a distributed setting, where multiple nodes run independently, without requiring synchronization. We further introduced the vector-based record format, a compaction-friendly format for semi-structured data. Experiments showed that our tuple compactor is able to reduce the storage overhead significantly and improve the query performance of AsterixDB. Moreover, it achieves this without impacting data ingestion performance. In fact, the tuple compactor and vector-based record format can actually improve data ingestion performance of insert-heavy workloads. When combined with our page-level compression, we were able to reduce the total storage size by up to 9.8x and improve query performance by the same factor.

The vector-based format store values (fixed and variable lengths) contiguously in the same region, which could be suitable for applying several encoding schemes. We plan to explore this opportunity for future work.. We also plan to extend this work to introduce a schema-adaptive columnar-oriented document store. First, we want to explore the viability of adopting the PAX [16] page format, which could potentially eliminate some of the vector-based format overheads. In a second direction, we want to explore ideas from popular static columnar file formats (such as Apache Parquet and Apache CarbonData [5]) to build an LSM-ified columnar indexes for self-describing, semi-structured data.

Acknowledgements This work was supported by a graduate 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).

7. REFERENCES

[1] Apache Arrow. https://arrow.apache.org.

[2] AsterixDB Documentation. https://ci.apache.org/projects/asterixdb/index.html.

[3] AsterixDB Object Serialization Reference. https://cwiki.apache.org/confluence/display/ASTERIXDB/AsterixDB+Object+Serialization+Reference.

[4] Apache Avro. https://avro.apache.org.

[5] Apache CarbonData. https://carbondata.apache.org.

[6] Apache Drill. https://drill.apache.org.

[7] Apache Parquet. https://parquet.apache.org.

[8] Apache Thrift. https://thrift.apache.org.

[9] Binary JSON: BSON specification. http://bsonspec.org/.

[10] Couchbase. https://couchbase.com.

[11] MongoDB. https://www.mongodb.com.

[12] Protocol Buffers. https://developers.google.com/protocol-buffers.

[13] Snappy. http://google.github.io/snappy/.

[14] Twitter API Documentation. https://developer.twitter.com/en/docs.html.

[15] I. Absalyamov, M. J. Carey, and V. J. Tsotras. Lightweight cardinality estimation in LSM-based systems. In *ACM SIGMOD*, pages 841–855, 2018.

[16] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[17] W. Y. Alkowaileet, S. Alsubaiee, and M. J. Carey. An LSM-based tuple compaction framework for Apache AsterixDB. *arXiv preprint arXiv:1910.08185*, 2018.

[18] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14), 2014.

[19] S. Alsubaiee et al. Storage management in AsterixDB. *PVLDB*, 7(10), 2014.

[20] S. Alsubaiee and Others. LSM-based storage and indexing: An old idea with timely benefits. In *Second international ACM workshop on managing and mining enriched geo-spatial data*, pages 1–6, 2015.

[21] A. Arion, A. Bonifati, G. Costa, S. dAguanno, I. Manolescu, and A. Pugliese. Efficient query evaluation over compressed XML data. In *EDBT*, pages 200–218, 2004.

[22] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[23] V. Borkar et al. Algebricks: a data model-agnostic compiler backend for big data languages. In *SoCC*, 2015.

[24] M. J. Carey. AsterixDB mid-flight: A case study in building systems in academia. In *ICDE*, pages 1–12, 2019.

[25] D. Chamberlin. *SQL++ For SQL Users: A Tutorial*. Couchbase, Inc., 2018. (Available at Amazon.com).

[26] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *ACM SIGMOD*, pages 295–310, 2016.

[27] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[28] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *ACM SIGMOD*, pages 153–164, 2000.

[29] C. Luo and M. J. Carey. LSM-based storage techniques: A survey. *arXiv preprint arXiv:1812.07527*, 2018.

[30] C. Luo and M. J. Carey. Efficient data ingestion and query processing for LSM-based storage systems. *PVLDB*, 12(5):531–543, 2019.

[31] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[33] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631*, 2014.

[34] P. Pirzadeh, M. J. Carey, and T. Westmann. Bigfun: A performance study of big data management system functionality. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 507–514. IEEE, 2015.

[35] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, et al. Schema-agnostic indexing with Azure DocumentDB. *PVLDB*, 8(12):1668–1679, 2015.

[36] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[37] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz. Schema management for document stores. *PVLDB*, 8(9):922–933, 2015.

[38] M. Zaharia et al. Spark: Cluster computing with working sets. In *Proc. HotCloud*, 2010.