UNIVERSITY OF CALIFORNIA,
IRVINE

Towards Analytics-Optimized Document Stores

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Wail Y. Alkowaileet

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Sharad Mehrotra

2022

DEDICATION

To my parents Norah M. Almalag and Yousef A. Alkowaileet, and to all my ten siblings: Abdullah, A'Adel (Rest in Peace), Waleed, Nabiel, Yasser, Nesreen, Walaa, Suhail, Faris, and Sarah.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of many people.

First and foremost, I am deeply thankful for the mentorship, support, and guidance that my advisor Professor Michael J. Carey provided over the past several years. Throughout the years, and even before being a Ph.D. student, Professor Carey has inspired me to pick and solve practical problems that can have positive impacts on day-to-day user experience. His simple yet effective model of Build, Measure, and Write (BMW) has shaped me into an independent researcher, who aspires to follow in his footsteps in the field of Data Management Systems. Despite his busy schedule, Professor Carey's door has always been open and available whenever I need his advice and direction. I could not have asked for a better advisor, a role model, and a leader to sail with during this journey. I am forever grateful.

My supervisors at King Abdulaziz City for Science and Technology (KACST), Anas Alfaris, Sattam Alsubaiee, and Hotham Altwaijry have my deepest gratitude for their valuable guidance and direction. Also, I would like to extend my gratitude to my old friend and colleague Mohanned Qunaibit for all the moral and technical support. Also, I would like to thank Ian Maxon for all the support and guidance during my work on Apache AsterixDB.

I would like to thank all my colleagues and my all current and previous AsterixDB teammates for their valuable suggestions, discussions, and code reviews: Abdullah Alamoudi, Murtadha Alhubail, Ali Alsuliman, Michael Blow, Yingyi Bu, Glenn Galvizo, Thomas Hütter, Shiva Jahangiri, Taewoo Kim, Chen Luo, Dmitry Lychagin, Mikhail Lychagin, Phanwadee (Gift) Sinthong, Hussain Towaileb, Xikui Wang, and Till Westmann.

I would also thank Professor Chen Li and Professor Sharad Mehrotra for joining my thesis committee. I thank them for their constructive feedback and comments during my topic defense. I also thank Professor Michael Franz and Professor Vladimir N. Minin for agreeing to be on my committee for the candidacy exam.

Outside the academic sphere, I owe my deepest gratitude to my friends: Rakan Alseghayer, Mohammed Alrished, and Yasser Alismail for all the fun and enjoyable times we have spent together during the last five years.

I would also extend my gratitude to KACST for providing me with a generous Graduate Study Fellowship during my PhD work. The work reported in this dissertation has been supported in part 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).

Last but not least, this work would not have been possible without the care, the love, and the patience of my family. Your daily voice and picture, although it was only through a screen, have given me all the moral support I needed. I owe this dissertation to my parents Norah and Yousef, and to all my ten siblings.

Chapter 4 of this dissertation contains material from "An LSM-based Tuple Compaction Framework for Apache AsterixDB" by Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey, which appeared in Proceedings of the VLDB Endowment 2020 (PVLDB 2020). The dissertation author was the primary investigator of this paper.

Chapter 5 of this dissertation contains material from "Columnar Formats for Schemaless LSM-based Document Stores" by Wail Y. Alkowaileet and Michael J. Carey, which appeared in Proceedings of the VLDB Endowment 2022 (PVLDB 2022). The dissertation author was the primary investigator of this paper.

VITA

Wail Y. Alkowaileet

EDUCATION

Doctor of Philosophy in Computer Science **2022**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Science **2013**
University of California, Irvine *Irvine, CA*

Bachelor of Science in Computer Science **2008**
King Saud Unieversity *Riyadh, Saudi Arabia*

PUBLICATIONS

Columnar Formats for Schemaless LSM-based Document Stores **2022**
Proceedings of the VLDB Endowment (PVLDB)

An LSM-based Tuple Compaction Framework for Apache AsterixDB **2020**
Proceedings of the VLDB Endowment (PVLDB)

End-to-End Machine Learning with Apache AsterixDB **2018**
DEEM: Workshop on Data Management for End-to-End Machine Learning @ SIGMOD

Enhancing Big Data with Semantics: The AsterixDB Approach (Poster) **2018**
The 12th IEEE International Conference on Semantic Computing (ICSC)

ABSTRACT OF THE DISSERTATION

Towards Analytics-Optimized Document Stores

By

Wail Y. Alkowaileet

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Michael J. Carey, Chair

In the last two decades, relational databases for analytics have been specialized to address the needs of analytical workloads. For instance, analytical workloads often focus on a few attributes (or columns) instead of the whole tuple. Thus, many analytical databases have opted to store the data in a columnar layout to reduce the I/O cost. Additionally, relational databases for analytics have shifted from interpreter structures for query processing to compiling queries into executable programs that can achieve the performance of hand-written specialized programs.

Document store database systems have gained traction for storing and querying large volumes of semi-structured data without requiring the users to pre-define a schema. This flexibility allows users to change the structure of incoming records without worrying about taking the system offline or hindering the performance of currently running queries. Despite their popularity, document stores have lacked the advances made for analytics-specialized relational databases for handling analytical workloads. In fact, the redundant information in the records ("embedded schemas") stored in document stores can introduce an unnecessary storage overhead that can render document stores even less performant than non-analytics-specialized relational databases.

Despite the performance gap between relational and document databases, many users have

no choice but to use the slower yet flexible document stores. In this dissertation, we aim to optimize document stores and show that users can enjoy a similar performance of the analytics-specialized relational databases without sacrificing the flexibility of the document model. Specifically, this dissertation makes the following main contributions:

We first address the lack of schemas and the storage overhead in document stores by proposing the tuple compactor, a framework for inferring and extracting the schemas of self-describing semi-structured records. Our tuple compactor exploits the events of Log-structured Merge tree (LSM) storage, namely the flush operation, to infer and extract the schema during data ingestion. We experimentally demonstrate the efficiency of our tuple compactor on real, large datasets.

Second, we use the inferred schema and introduce two layouts for storing nested semi-structured data in columnar formats. We also propose several extensions to the Dremel format, a popular columnar format for nested data, to comply with document stores' flexible data model. Our experiments show significant performance gains, improving the query execution time by orders of magnitude while minimally impacting ingestion performance.

Third, we present our code generation framework, which can handle the document data model's polymorphic nature and improve query execution performance in Apache AsterixDB. We analytically and empirically show that the CPU overhead of an interpreter-style execution engine can be a bottleneck even in disk-based databases like AsterixDB. However, our code generation framework can remedy the CPU overhead and significantly improve the performance of analytical queries. We also propose several improvements to AsterixDB's aggregation framework to optimize group-by queries, a crucial class of queries in analytical workloads.

Chapter 1

Introduction

Specialized Database Management Systems (DBMSs) for analytics have been around since the 1970s; Teradata [27] is an example of an analytical DBMS founded in 1979 [**?**]. However, the market for analytical DBMSs flourished in the 2000s due to the internet boom. Nowadays, analytical DBMSs are highly specialized to address the challenges of analyzing the ever-growing size of data. For example, systems such as Vertica [29, 89] and MonetDB [21, 94] store relational tables as columns instead of rows to accelerate analytical workloads, which tend to focus on querying a few attributes instead of the whole tuples. Another architectural shift in analytical DBMSs is the query execution model. For instance, in MonetDB\X100 [94], the authors proposed a query execution model (moving from the iterator model [69]) that is cache-friendly, which consequently reduces the overhead of accessing memory when executing CPU-intensive analytical queries. The advancement in query processing took another step to further reduce the CPU overhead by translating queries into code that can be compiled and executed to obtain the desired results [12, 65, 77, 83]. This code generation and query compilation approach can improve query execution performance and bring it closer to the performance of hand-written specialized programs.

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. As a result, document store systems have emerged as popular solutions for storing, indexing, and querying self-describing semi-structured data. In document stores, users do not need to define a schema or transform the data before ingesting it – making document stores more attractive for schema-flexible (or even schemaless) applications. However, document stores relinquished the advancements made for improving the performance of analytical workloads in the relational model by forgoing the rigidity of schemas. The lack of schema, for instance, makes it impossible to store nested semi-structured in a columnar-oriented fashion as the attributes (or fields) are unknown. Additionally, adopting the code generation and compilation model mentioned earlier would be challenging as values' types are only known at runtime and not compile time. For those reasons, major document stores such as MongoDB [22] and Couchbase Server [17] (at the time of writing this dissertation) do not support storing the data in a columnar format nor employ the code generation and compilation execution model – rendering them less performant for analytical workloads compared to relational DBMSs.

In this dissertation, we aim to bridge the performance gap between document stores and relational databases for handling analytical workloads without affecting the flexibility provided by the former. In particular, we study three problems and provide an efficient solution for each one in the context of Apache AsterixDB [3] based on the following topics:

Schema Inference and Tuple Compaction. Due to the lack of a centralized schema, each record stored in a document store embeds additional (schema) information that describes its structure and the stored values' types. By storing the "schema" within each record, records can have different structures and even types for a single field (e.g., storing birth dates as integers or strings). The cost of such an approach can be high since records often share the same schema; hence, storing such information in each record is redundant and can

unnecessarily inflate the storage footprint. Extracting the schema information from each record and storing it in a "centralized" manner can alleviate this storage overhead. In the first part of this dissertation, we introduce the tuple compactor, a framework that infers and compacts the schema for document stores' semi-structured records during data ingestion. The tuple compactor appeared in VLDB'20 [36].

Columnar Formats for Document Stores. Since the scheme can be inferred efficiently, one possible extension to the previous topic is to store documents as columns – a step further to bring document stores closer to matching the performance of analytical relational databases. In this topic, we explore different approaches for storing data in columnar formats without compromising the flexibility of the document model. We also study the viability of secondary indexes for querying data stored in a columnar format, where secondary indexes have at times been deemed unnecessary for columnar databases. The proposed techniques for storing semi-structured data in a columnar format will appear in VLDB'22 [37]

Code Generation and Compilation. Storing the data in a columnar format can reduce the I/O cost significantly for analytical workloads. With the current advancement in storage technologies, disks have become faster – shifting the cost to the CPU, which became apparent, especially for complex analytical workloads. Relational DBMS developers have addressed this issue by translating queries into compilable and executable code. However, document stores still employ interpreter-like execution engines, where values (or records) are passed from one operator to another for processing. As in dynamically-typed programming languages, a major challenge for shifting to a code generation and compilation model is the polymorphic nature of document stores, where values' types are only known at runtime. In the last part of this dissertation, we propose a solution for translating AsterixDB's SQL++ queries into code written using an internal language that can be compiled and executed efficiently. The generated code, which runs using Oracle's Truffle Framework [93], can adapt dynamically to type changes at runtime using Truffle's Just-in-Time (JIT) compiler. We also

propose several improvements to AsterixDB's aggregation framework to further improve the performance of analytical workloads.

The rest of this dissertation is organized as follows. Chapter 2 discusses the differences between the various physical storage formats and query execution models, and gives background information on Apache AsterixDB's internals. Chapter 3 discusses related work on inferring schemas, storing data in columnar formats, and translating queries into compilable and executable code. Chapter 4 presents a novel approach for inferring the schema and using the inferred schema to compact ingested records. Chapter 5 proposes one approach for storing schemaless semi-structured data in columnar formats. Chapter 6 describes the design and implementation of our code generation framework in Apache AsterixDB. Finally, Chapter 7 concludes this dissertation and discusses future research directions.

Chapter 2

Background

2.1 Storage Formats

In any data management system, the storage subsystem determines how the data is stored and retrieved. Different data management systems store the data differently to accelerate the data ingestion and retrieval process for specific workloads. For instance, database systems targeted for analytical workloads focus on processing a large volume of data while minimizing query execution time. Thus, a suitable physical storage format that reduces the overall I/O and CPU costs is crucial for analytical databases. In this section, we summarize the traits of the two major physical storage formats: row and columnar formats.

2.1.1 Row Format

In row-store data management systems, the records are stored contiguously on disk (or in memory for in-memory databases) as rows; hence the name "row-store". Figure 2.1 shows the `Employee` table whose records are stored as contiguous rows on disk. In document stores

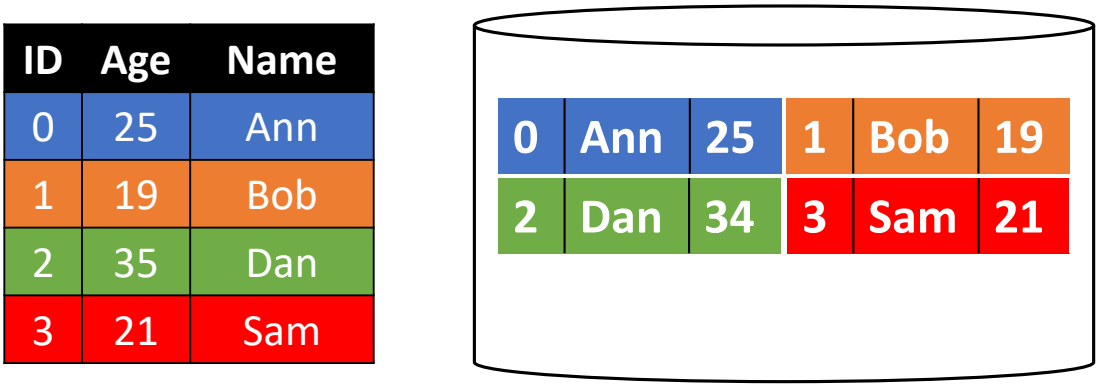


Figure 2.1: The `Employee` table stored in a row format

such as AsterixDB, records could also be stored in a row-like format, where each row is a document (e.g., a JSON document). Handling inserts, deletes, and updates efficiently are crucial for operational workloads. Since the records (or tuples) are the operational units (for insert, delete, and update operations) in database systems, storing the data as rows makes it naturally easier to manipulate. For instance, inserting a new record can be achieved by simply appending the record (as-is) to a file. Thus, operational databases often store data as rows to maximize the performance of processing a large number of record-oriented transactions.

2.1.2 Columnar Format

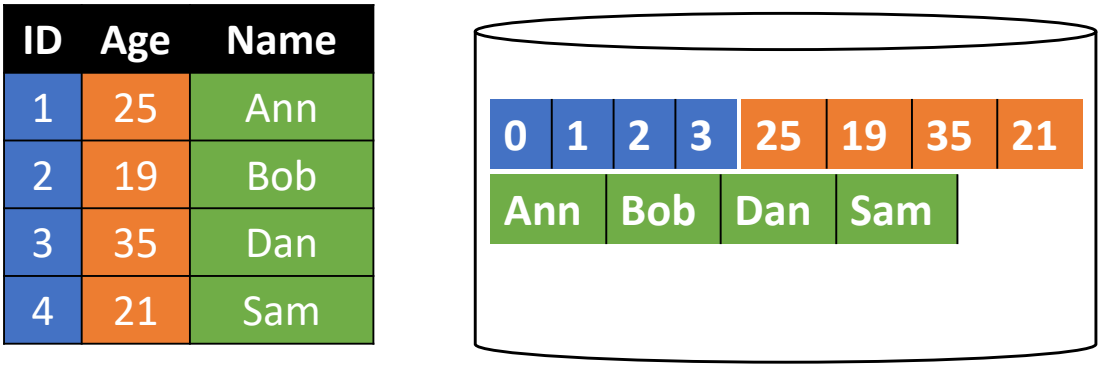


Figure 2.2: The `Employee` table stored in a columnar format

Handling operational workloads was initially the main objective for database systems. Thus,

storing data as rows is still the default format in legacy systems. However, using database systems has gone beyond handling operational workloads to analyze the stored data to draw valuable insights. In contrast to operational workloads, analytical workloads are heavy on running ad-hoc queries against a large volume of data. Furthermore, analytical queries tend to focus only on a few attributes (or columns) relevant to specific questions or insights. Therefore, column-store data management systems have emerged to address those needs. As opposed to rows, column-store systems store the records as contiguous columns on disk (or in memory). Figure 2.2 shows the columnar storage layout for the same table in Figure 2.1.

To illustrate the benefits of storing the data as columns for analytical workloads, let us take the following simple analytical query:

```
SELECT AVG(age) FROM Employee
```

The query computes the average age of all employees in the `Employee` table. When stored as rows, the system reads each record and extracts the age value to compute the required aggregate. However, when the data is stored as columns, the system reads only the `age`'s column and skips all irrelevant columns. Consequently, the I/O cost for such a query becomes significantly lower in column-store systems compared to row-store systems.

Additionally, storing the values of each column separately and contiguously allows for encoding the data to reduce the I/O cost further. Let us take an example where we want to encode the `ID`'s values (Figure 2.2): 1, 2, 3, and 4. The values are first encoded using the delta-encoding, which produces the numerical sequence [1, 1, 1, 1], where the first value of the sequence is the first unencoded value (i.e., $ID = 1$), and the rest of the values represent the differences (*deltas*) between the original values. The resulting sequence can then be passed to a Run-length encoder, which encodes the sequence's values as a pair of two integers $< 1, 4 >$ – indicating that the value "1" is repeated four times. As a result, using the two encoding schemes, it takes two integers to store the `ID`'s four integer values – a 50%

reduction.

However, storing the data as columns comes with a cost. In contrast to row-stores, column-stores systems must shred newly inserted records into columns before persisting them to disk (an additional cost). Consequently, the system must also ensure that the columns can be reassembled (or restitched) back to their original form. Some mechanisms are used to track which columns' values belong to which record. For instance, a naïve approach would be attaching a record identifier (RID) to each column's value – requiring the system to store the same RIDs for as many columns as a table has (e.g., three times for the table shown in Figure 2.2). Another approach could exploit the position of each value (i.e., the order in which the values are stored) to associate each value to its record. However, handling updates using the latter approach would be more challenging, as a new column value may need more space, which requires moving the new column's value to a different position. With encoding, handling updates would be even more challenging for both approaches.

2.2 Query Execution Models

Similar to the storage formats, different data management systems employ different models for processing queries, and each model has its pros and cons. When a user submits a query, the system translates the submitted query into a "physical" algebraic expression (also known as the query plan), which the system evaluates to produce the query result. In this section, we explain the workflow of different query execution models – namely, the iterator model, the vectorized model, and the code generation and query compilation model. We also give an overview of the advantages and disadvantages of each execution model.

```
SELECT name, age
FROM Employee
WHERE age > 20
```

Figure 2.3: A query against the `Employee` table (Section 2.1)

2.2.1 Iterator Model

In the iterator model [69] (sometimes referred to as the Volcano model [60]), the evaluation of a query plan starts by calling the `next()` function of the root algebraic operator of the query plan. That, in turn, calls the `next()` function for the root's child operator. The callings of `next()` cascade to every operator in the query plan until it reaches the leaf operators. The result of each calls to `next()` is a processed tuple and the resulting tuples from calling `next()` on the root operator are the query result.

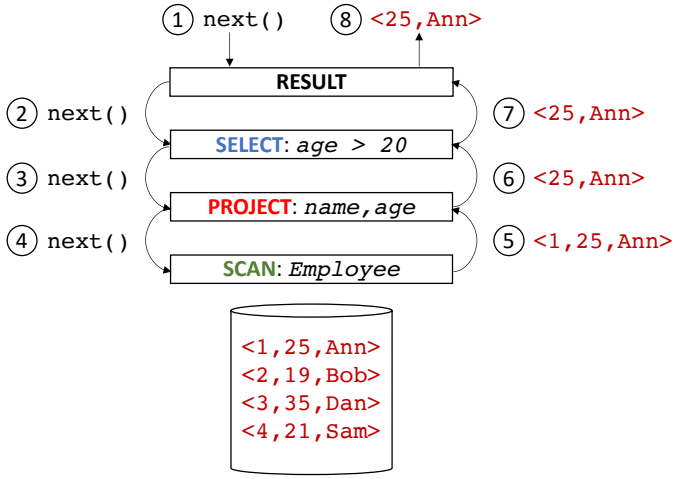


Figure 2.4: The execution plan for the query in Figure 2.3

To illustrate, Figure 2.4 shows the query plan for the query in Figure 2.3 as well as the sequence in which the function `next()` is called. Each call to `next()` cascades until the `next()` of the leaf `SCAN` operator is called, which feteches the first tuple from disk and passes it as an output of calling its `next()` function. Then, each operator processes the received tuple and produces a new one as a result of calling its `next()` function as shown in Figure 2.4.

The workflow of the iterator model resembles the interpreter model in programming languages. As opposed to compilers, interpreters are easy to implement; however, interpreted programming languages tend to be less performant than compiled ones. Similarly, the iterator model is less performant than other query execution models for the reasons explained in [77] and summarized in the following points:

- The many calls to the function `next()` in the iterator model can hinder the performance (e.g., one million tuples and three operators equates to three million calls).
- Each call to `next()` is probably a virtual call that the compiler cannot inline and can degrade the ability of the CPU's branch predictor.

2.2.2 Vectorized Model

To address the issues of the iterator model, the vectorized model [81, 94] (also known as the batch-at-a-time model) produces a batch of tuples (instead of a single tuple) for each call to the function `next()`. This batching mechanism reduces the number of calling `next()`, as the cost of each call is "distributed" among the produced tuples in the batch. However, in this model, each operator materializes the processed tuples into some temporary memory buffer to form a batch for the next operator. Some operators, such as the `SELECT` operator, do not alter the form of the received tuples. One advantage of the iterator model over the vectorized model is that the results of such operators are pipelined – avoiding the unnecessary cost of the materialization.

2.2.3 Code Generation and Query Compilation Model

Akin to the performance of a compiled language vs. an interpreted language, a compiled hand-written code outperforms both the iterator and the vectorized models [84, 94]. For

instance, in [84], the authors compared the performance of the iterator model against "*a hand-written code written by a college freshman*" when evaluating a simple aggregate query. They found that the "college freshman" hand-written code was an order of magnitude faster than the iterator model for the following reasons:

- The hand-written code eliminated the virtual function calls in the iterator model.
- The hand-written code placed the intermediate results in CPU registers (vs. in memory in the iterator model).
- The hand-written code benefited from the compiler's optimizations, such as unrolling loops and exploiting SIMD instructions to process multiple values in a single CPU instruction.

Several data systems [12, 65, 77, 83] translate algebraic operators into code, which can then be compiled and executed efficiently. The translated code "fuses" the work of multiple operators into a single function call. Thus, the generated code processes the data in situ instead of passing the data from one operator to another. In Chapter 6, we describe our approach (inspired by [77]) to translating the algebraic operators of a query into an executable code.

Despite the performance gains that systems can get from this model, one should consider the compilation time, which can be significantly high – especially for latency-sensitive applications (i.e., operational workloads). Thus, the code generation model is often used to reduce the execution times of complex analytical queries.

2.3 Apache AsterixDB

Apache AsterixDB is our system of choice for implementing the proposed techniques in this dissertation. Apache 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 [38, 51] and its query execution engine Hyracks [49].

2.3.1 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).

```
CREATE TYPE DependentType
AS CLOSED {
   name: string,
   age: int
};

CREATE TYPE EmployeeType
AS OPEN {
   id: int,
   name: string,
   dependents:{{DependentType}}?
};

CREATE DATASET Employee(EmployeeType) PRIMARY KEY id;
```

Figure 2.5: Defining Employee type and dataset in ADM

There are two options when defining a datatype in AsterixDB: *open* and *closed*. Figure 2.5 shows an example of defining a dataset of employee information. In this example, we first define *DependentType*, 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 *DependentType*, 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 *DependentType* 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.

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

Figure 2.6: An example of a SQL++ query

To query the data stored in AsterixDB, users can submit their queries written in SQL++ [52, 80], a SQL-inspired declarative query language for semi-structured data. Figure 2.6 shows an example of a SQL++ aggregate query posed against the dataset declared in Figure 2.5.

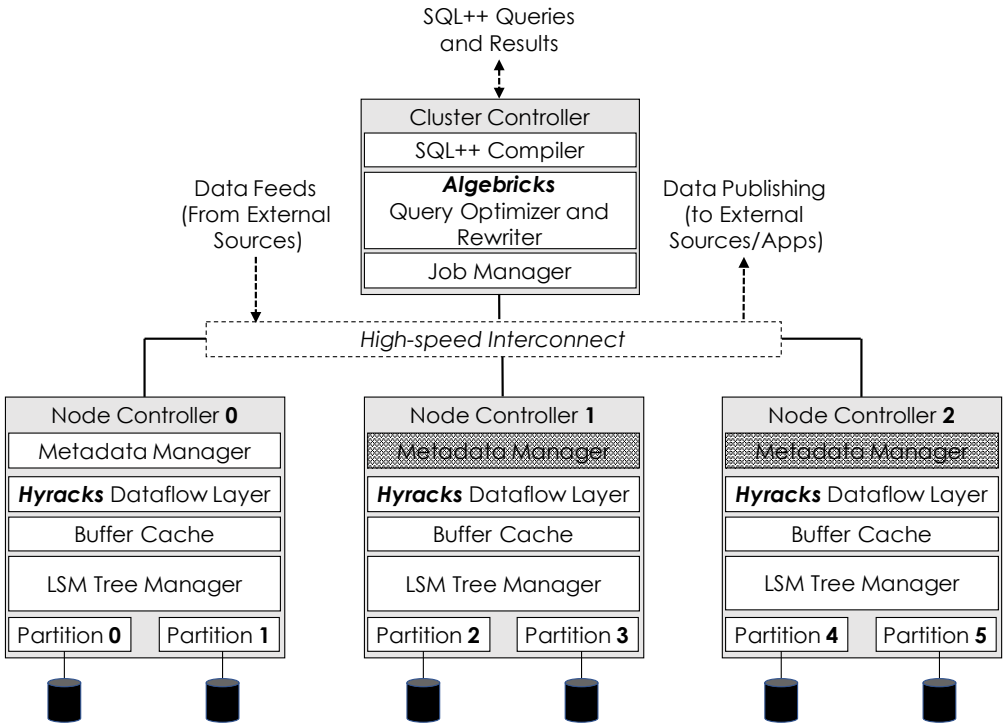


Figure 2.7: An AsterixDB cluster configured with two partitions in each of the three NCs

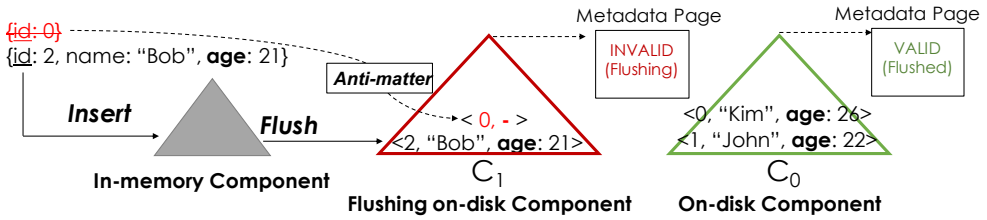
2.3.2 Storage and Data Ingestion

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 2.7 shows an AsterixDB cluster of three NCs, each of which has two data

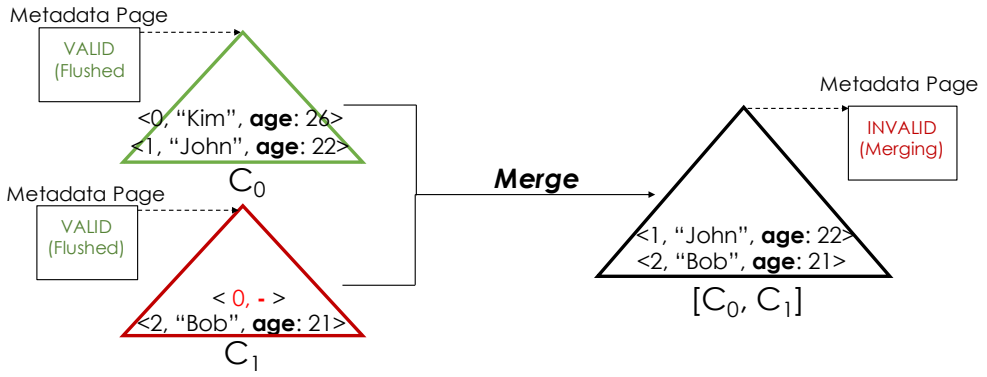
partitions that hold data on two separate storage devices. Data partitions in the same NC (e.g., Partition 0 and Partition 1 in NC0) share the same buffer cache and memory budget for LSM in-memory components; however, each partition manages the data stored in its storage device independently. In this example, NC0 also acts as a metadata node, which stores and provides access to AsterixDB metadata such as the defined datatypes and datasets.

AsterixDB stores the records of its datasets, spread across the data partitions in all NCs, as rows 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 2.7) 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 2.8a. 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.

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 [39] to indicate that a record with a specified key has been deleted. An upsert is an insert of a new record with the same key, which replaces the older record. For example, in Figure 2.8a, we delete the record with $id = 0$. Since the target record is stored in C_0, we insert an "anti-



(a) Flushing component C_1



(b) Merging the two components C_0 and C_1 into a new component $[C_0, C_1]$

Figure 2.8: An example of LSM flush and merge operations

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 [39, 70] that determines when and what to merge. Deleted and updated records are garbage-collected during the merge operation. In Figure 2.8b, after merging C_0 and C_1 into $[C_0, C_1]$, we do not write the record with $id = 0$ since 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 [70]. In this work, we explain how

to use this property later in Section 4.3.2.

Datasets' records (of both *open* and *closed* types) in the LSM primary index are stored in a binary-encoded physical ADM format [5]. 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 2.8, 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).

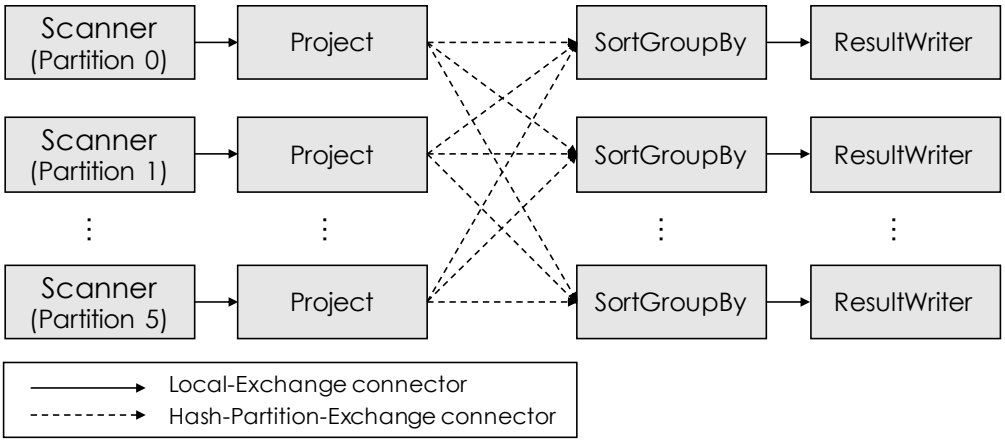


Figure 2.9: A compiled Hyracks job for the query in Figure 2.6

2.3.3 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 (similar to the vectorized execution model explained in Section 2.2.2). Figure 2.9 depicts the compiled Hyracks job for the query in Figure 2.6. As

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

shown in Figure 2.9, batches (or frames) of 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* [50] may rewrite the translated function when necessary. As an example, the field access expression $e.name$ in the query shown in Figure 2.6 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.

Chapter 3

Related Work

In this section, we overview current work relevant to inferring schemas in schemaless DBMSs, storing records in a columnar-oriented fashion, and using code generation and compilation techniques to accelerate query execution for analytical workloads.

3.1 Schema Inferrence

Schema inference for self-describing, semi-structured data has appeared in early work for the Object Exchange Model (OEM) and later for XML and JSON documents. For OEM (and later for XML), [57] presented the concept of a dataguide, which is a summary structure for schema-less semi-structured documents. A dataguide could be accompanied by values' summaries and samples (annotations) about the data. In [91], Wang et al. presented an efficient framework for extracting, managing, and querying a schema-view of JSON datasets. Their work targeted data exploration, where showing a frequently appearing structure can be good enough. In another work [54], the authors detailed an approach for automatically inferring and generating a normalized (flat) schema for JSON-like datasets, which can be

utilized in an RDBMS to store the data.

The schemas inferred in [57] and [91] were targeted for users to explore and build a general understanding of the data. As we describe later in Chapter 4, we share some of the mechanisms proposed in both works to infer the schema for schemaless semi-structured data. However, our objective differs in that we use the inferred schemas to reduce the storage overhead in document stores. Thus, inferring the exact schema efficiently under heavy inserts and updates is crucial. Also, our work in Chapter 4 is orthogonal to [54]; we target document stores without the need for changing the underlying data model of such systems.

3.2 Column Stores

Open-source and commercial column-store systems [1, 8, 16, 21, 29, 43] have gained more popularity as data warehouse solutions due to their superior performance in handling analytical workloads. Furthermore, legacy systems such as Microsoft SQL Server have added the capability to store tables as columns [67, 66], either as a primary or secondary index, to improve the performance of analytical workloads. For nested data, Parquet [9], which is a columnar file format, has become the de-facto format for big data systems such as Apache Spark [12] and Apache [7] due to Parquet's compactness. Those systems, however, still require the user to declare a schema a priori. Hence, the process of columnizing the data in such systems cannot be adopted in document stores, where declaring a schema is either not required or optional.

In [55], the authors have proposed *Json Tiles*, a columnar format for semi-structured records integrated into Umbra [78], 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 [90] 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 in Chapter 5 aims to support columnar formats for existing document stores.

In Chapter 5, we detail our approach to storing schemaless semi-structured data in document stores using the inferred schema from Chapter 4. Additionally, in the same chapter, we compare our approach to other recent work on storing data in columnar formats in schemaless document stores. We defer those discussions to the end of Chapter 5 as some of the discussion points require background knowledge of previous work that Chapter 5 details.

3.3 Code Generation and Compilation for DBMSs

Instead of using an interpreter structure (either the iterator or the vectorized models), several big data analytical systems [12, 65, 75, 77] have adopted the code generation and compilation model due to its superior performance for processing complex queries. Those systems utilize strongly-typed languages (e.g., Java, C++, or LLVM bitcode) to translate declarative queries into imperative code – relying on pre-declared schemas to provide the values' types.

For schemaless document stores, little work has focused on addressing the challenge of handling their polymorphic nature for code generation and compilation. For instance, [76] extends JSINQ [64], a JavaScript implementation of Language-integrated Query (LINQ), to query documents stored in MongoDB. JSINQ is utilized to process objects resulting from querying MongoDB, with the ability to push filter predicates down to MongoDB to reduce the number of retrieved objects. However, the proposed approach does not target replacing MongoDB's execution mode with a code generation and compilation model as in the aforementioned schema-ful systems. Instead, it targets bringing the popular LINQ model to

application developers who use MongoDB as their database.

In Chapter 6, we propose a solution for adopting the code generation and compilation model into schemaless document stores in a way that can adapt to type changes at runtime. There we also review the recent work on querying in-memory objects and arrays in dynamically typed languages, where value types are known only at runtime.

Chapter 4

An LSM-based Tuple Compaction Framework for Apache AsterixDB

4.1 Introduction

As described in Chapter 1, 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 [22] and Couchbase Server [17], 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 [38]. From a user perspective, declaring a schema requires a thorough a priori understanding of the dataset's fields and their types.

In this chapter, 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 [79] based storage engines, which are used in many prominent document store systems [17, 22] including Apache AsterixDB [39]. In LSM-backed engines, records are first accumulated in memory (*LSM in-memory component*) and 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 first introduce our implementation of page-level compression in AsterixDB. This is a similar solution to those adopted by other NoSQL DBMSs to reduce the storage overhead of self-describing records.
- 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 evaluate the feasibility of our design, prototyped using AsterixDB, to ingest and query a variety of large semi-structured datasets.

The remainder of this chapter is structured as follows: Section 4.2 provides the details of our implementation for page-level compression. Section 4.3 details the design and implementation of our tuple compaction framework in AsterixDB. Section 4.4 presents an experimental evaluation of the proposed framework. Section 4.5 discusses related work on utilizing the LSM lifecycle and on schema inference for semi-structured data. Finally, Section 4.6 presents our conclusions for this work.

4.2 Page-level Compression

As shown in Figure 2.8 in Chapter 2.3, the information of the undeclared field *age* is stored within each record. This could incur an unnecessary higher storage overhead if all or most records of the Employee dataset have the same undeclared field, as records store redundant information. MongoDB and the Data Service in Coucbase Server have introduced compression to reduce the impact of storing redundant information in self-describing records. In AsterixDB, we can take a similar step by introducing page-level compression to compress the leaf pages of the B^+-tree of the primary index.

To minimize its software engineering impact, AsterixDB's new page-level compression is designed to operate at the buffer-cache level. On write, pages can be compressed and then persisted to disk. On read, pages can be decompressed to their original configured fixed-size

and stored in memory in AsterixDB's buffer cache. Each such compressed page can be of any arbitrary size. However, the AsterixDB storage engine was initially designed to work with fixed-size data pages where the size is a configurable parameter. Larger data pages can be stored as multiple fixed-size pages, but there is no mechanism to store smaller compressed pages. Any proposed solution to support variable-size pages must not change the current storage physical layout of AsterixDB.

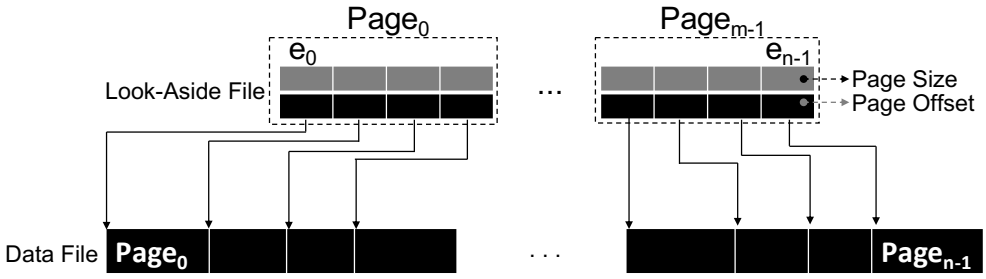
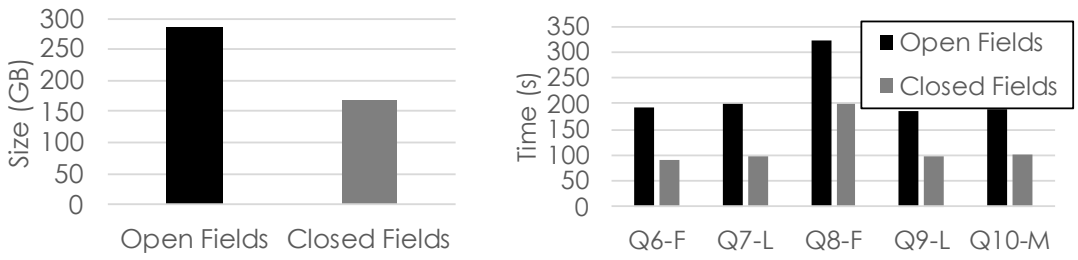


Figure 4.1: Compressed file with its Look-Aside File (LAF)

To address this issue, we use Look-aside Files (LAFs) to store offset-length entry pairs for the stored compressed data pages. When a page is compressed, we store both the page's offset and its length in the LAF before writing it to disk. Figure 4.1 shows a data file consisting of n compressed pages and its corresponding LAF. The number of entries in the LAF equals the number of pages in the data file, where each entry (e.g., e_0) stores the size and the offset of its corresponding compressed data page (e.g., $Page_0$). LAF entries can occupy more than one page, depending on the number of pages in the data file. Therefore, to access a data page, we need first to read the LAF page that contains the required data page's size and offset and then use them to access the compressed data page. This may require AsterixDB to perform an extra IO operation to read a data page. However, the number of LAF pages is usually small due to the fact that the entry size is small (12-bytes in our implementation). For instance, a 128KB LAF page can store up to 10,922 entries. Thus, LAF pages can be easily cached and read multiple times. Our proposed approach to support compression has helped multiple current AsterixDB users reduce the storage cost and improve the query execution time while not impacting their previously loaded data.



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

Figure 4.2: Summary of the findings in [82]

In addition to this "syntactic" approach based on compression, the next section introduces a "semantic" approach to reducing the storage overhead by inferring and stripping the schema out of self-describing records in AsterixDB. In Section 4.4, we evaluate both approaches (syntactic and semantic) when they are applied separately and when they are combined and show their impact on storage size, data ingestion rate, and query performance.

4.3 LSM-based Schema Inference and Tuple Compaction

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. [82] 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 4.2, 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.

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

Figure 4.3: Enabling the tuple compactor for a dataset

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 4.3. 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 4.3.1. In Section 4.3.2, we show the structure of an inferred schema and a way of maintaining it on update and delete operations. Then, in Section 4.3.3, 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 4.3.4, we address the challenges of querying compacted records stored in distributed partitions of an AsterixDB cluster.

4.3.1 Tuple Compactor Workflow

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

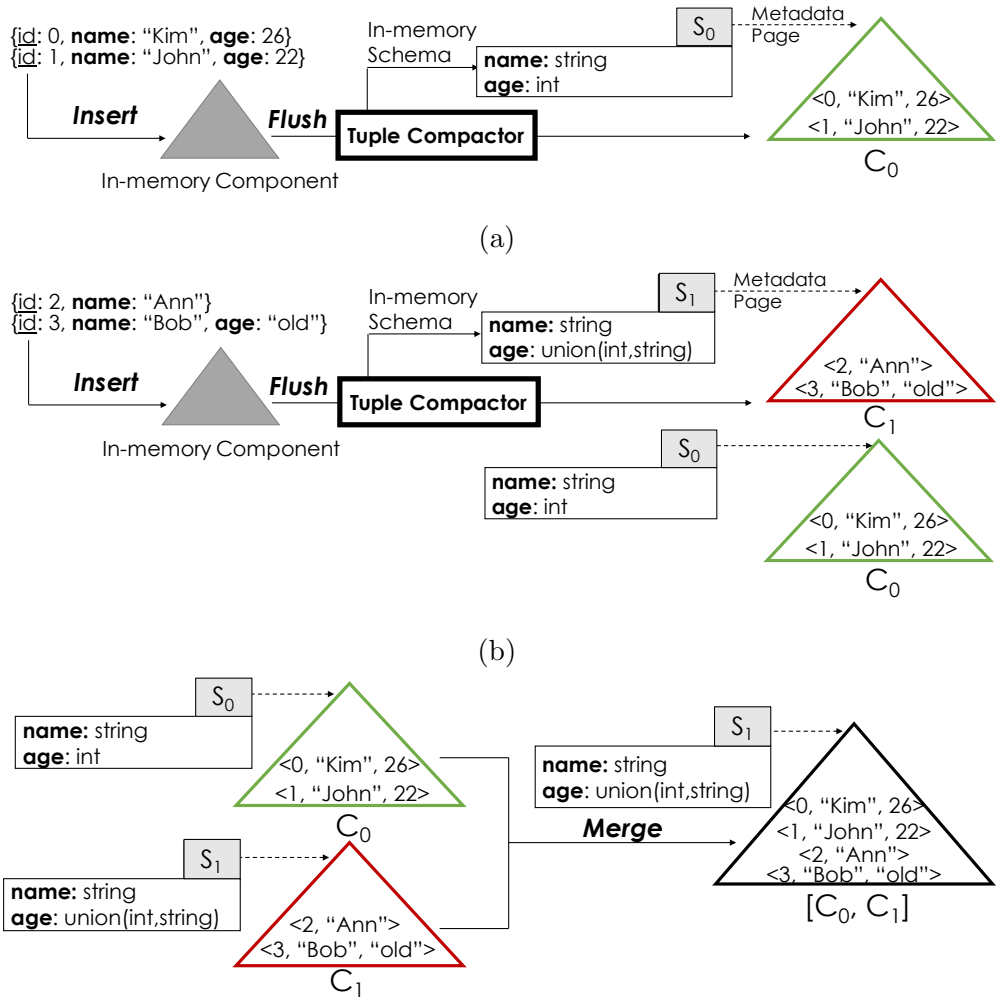


Figure 4.4: **(a)** Flushing the first component C_0 **(b)** Flushing the second component C_1 **(c)** Merging the two components C_0 and C_1 into the new component $[C_0, C_1]$

Data Ingestion. When creating the *Employee* dataset (shown in Figure 4.3) in the AsterixDB cluster illustrated in Figure 2.7, 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 4.4a, factors the schema information out of each record and builds a traversable in-memory structure that holds the schema (described in Section 4.3.2). 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 2.3.2). 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.

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 4.4b, 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 4.4c 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 2.3.2). To illustrate, let us consider the case where a system crash occurs during the second flush as shown in Figure 4.4b. 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.

4.3.2 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.

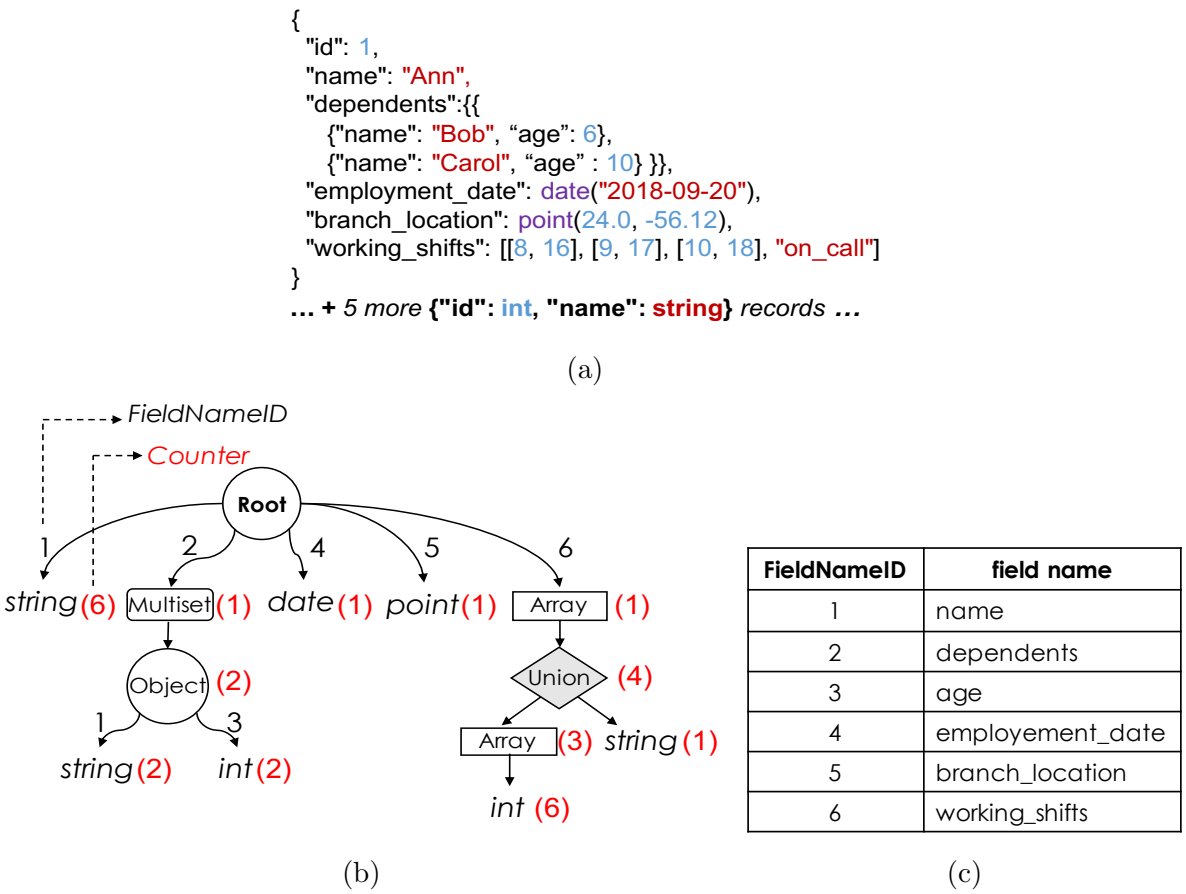


Figure 4.5: **(a)** An ADM record **(b)** Inferred schema tree structure **(c)** Dictionary-encoded field names

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 4.5a 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 4.5b, and (ii) a dictionary that encodes the inferred field names strings into *FieldNameID*s, as shown in Figure 4.5c. The *Counter* in the schema tree-structure represents the number of

occurrences of a value, which we further explain in Section 4.3.2.

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 4.3.1. 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 *FieldNameID*s (shown as integers on the edges of object nodes in Figure 4.5b) that reference the stored field names in the dictionary shown in Figure 4.5c. 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 4.5a. 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 [4]. Hence, a union node could have up to 27 children.

Schema Structure Maintenance

In Section 4.3.1 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 4.4 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 4.5b, 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 4.5a. Also, we can infer from the schema structure that all fields other than *name* belong to the record shown in Figure 4.5a. Therefore, after deleting this record, the schema structure should only have the field *name* as shown in Figure 4.6.

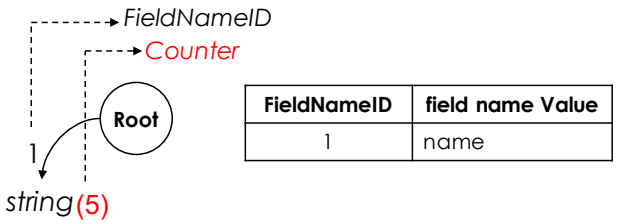


Figure 4.6: After deleting the record shown in Figure 4.5a

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 4.6, after deleting the record in Figure 4.5a, 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 4.5a) 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 [39] and LSM filters [41]. Luo et al. [70, 71] 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 4.4, we evaluate the data ingestion performance of our tuple compactor under heavy updates using the suggested primary key index.

4.3.3 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 4.5b 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.

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 2.3.2). Figure 4.7 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.

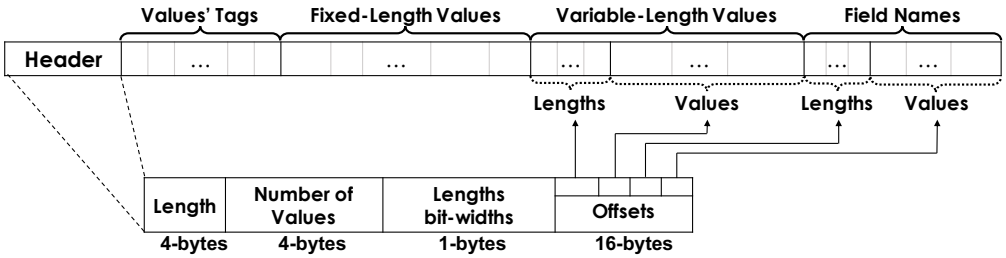


Figure 4.7: The structure of the vector-based format.

Figure 4.8 shows an example of a record in the vector-based format (we refer interested readers to Appendix A.1 for an additional example). 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.

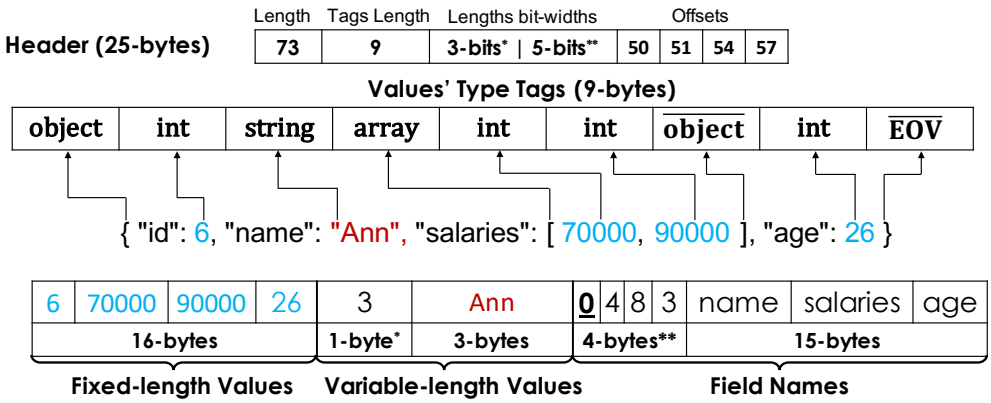


Figure 4.8: An example record in the vector-based format

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 [5, 13]. We address this issue in more detail in Section 4.3.4.

Schema Inference and Tuple Compaction

Records in vector-based format separate values from metadata. The example shown in Figure 4.8 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 4.9 shows the compacted structure of the record in Figure 4.8 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 4.7) to zero to indicate that field names were removed and stored in the schema structure. As shown in the example in Figure 4.9, 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 4.8.

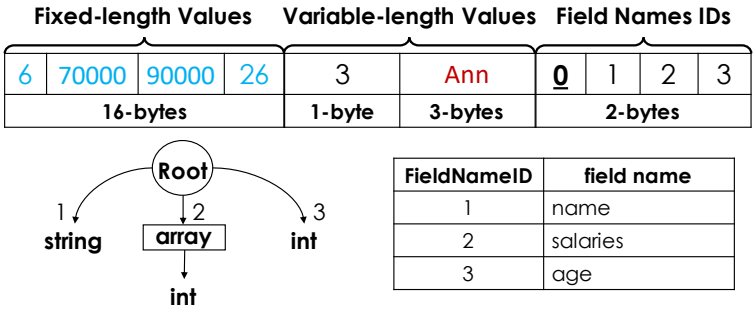


Figure 4.9: The record in Figure 4.8 after compaction

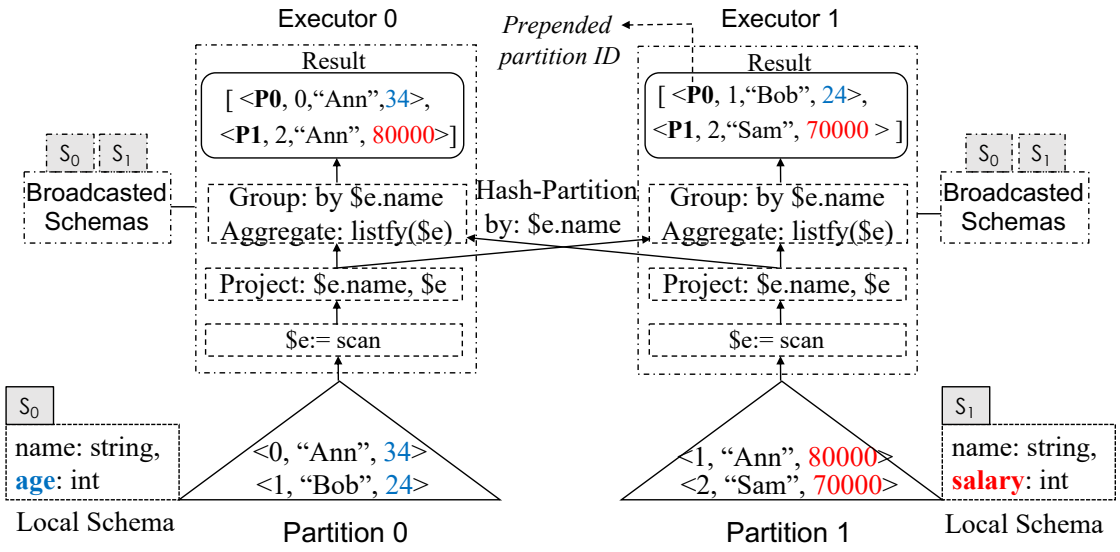
4.3.4 Query Processing

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.

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 4.10. 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*.

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,




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

Figure 4.10: Two partitions with two different schemas

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 4.10. 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.

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 4.3.3). The AsterixDB query optimizer can move field access expressions within the plan when doing so is advanta-

geous. 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*.

4.4 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.

We also conduct additional experiments to evaluate:

1. The performance accessing values in records in vector-based format (Section 4.3.3) with and without the optimization techniques explained in Section 4.3.4.
2. The impact of our approach on query performance using secondary indexes.
3. The scalability of our framework using computing clusters with different number Amazon EC2 instances.

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. Section 4.4.5 details the setup for our additional scale-out experiments.

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.

In Section 4.2, we introduced our implementation of the page-level compression in AsterixDB.

Throughout our experiments, we also evaluate the impact of compression (using Snappy [26] compression scheme) 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.

4.4.1 Datasets

In our experiments, we used three datasets (summarized in Table 5.1) which have different characteristics in terms of their record's structure, size, and value types.

Using the first dataset, we want to evaluate ingesting and querying social network data. We obtained a sample of tweets using the Twitter API [28]. 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.

The second dataset we used is the Web of Science (WoS) [14] publication dataset[1]. The WoS dataset encompasses meta information about scientific publications (such as authors, fundings and abstracts) from 1980 to 2016 with a total dataset size of 253GB. We transformed the dataset's record format from its XML original structure to a JSON one using an

[1]We obtained the dataset from Thomson Reuters. Currently, Clarivate Analytics maintains it [15].

existing XML-to-JSON converter [30]. The resulting JSON documents contain some fields with heterogeneous types, specifically a union of object and array of objects. The reason behind using such a converter is to mimic the challenges a data scientist can experience when resorting to existing solutions. (Due to a lack of support for declared union types in AsterixDB, we could only pre-declare the fields with homogeneous types in the closed schema case.)

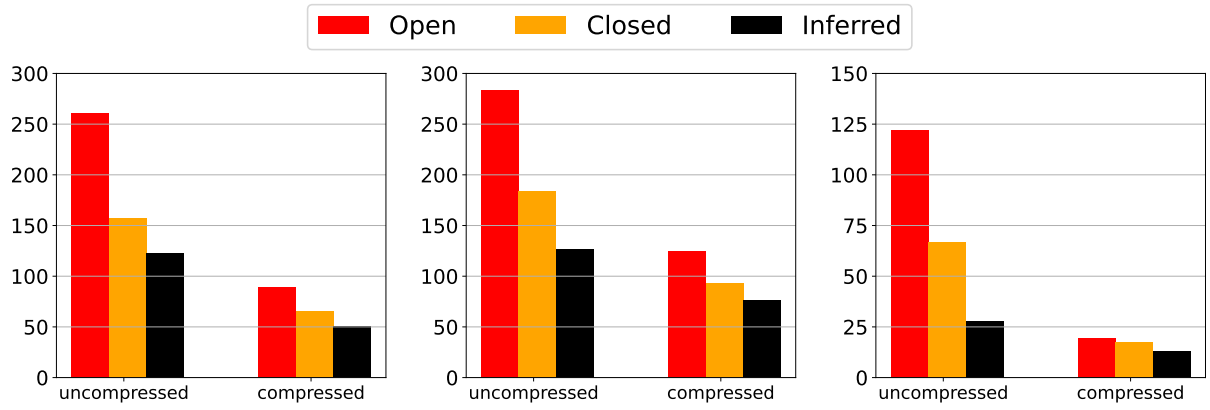
To evaluate more numeric Internet of Things (IoT)-like workloads, we generated a third synthetic dataset that mimics data generated by sensors. Each record in the sensors' dataset contains captured readings and their timestamps along with other information that monitors the health status of the sensor. 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.

	Twitter	WoS	Sensors
Source	Scaled	Real-world	Synthetic
Total Size	200GB	253GB	122GB
# of Records	77.6M	39.4M	25M
Record Size	∼2.7KB	∼6.2KB	5.1KB
# of Scalar val. (min, max, avg)	53, 208, 88	71, 193, 1430	248, 248, 248
Max. Depth	8	7	3
Dominant Type	String	String	Double
Union Type?	No	**Yes**	No

Table 4.1: Datasets summary

4.4.2 Storage Size

In this experiment, we evaluate the on-disk storage size after ingesting the Twitter, WoS and 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).



(a) Twitter dataset (b) WoS dataset (c) Sensors dataset

Figure 4.11: On-disk sizes (GB)

We first evaluate the total on-disk sizes after ingesting the data into the open, closed and inferred datasets. We begin with the Twitter dataset. Figure 4.11a 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).

For the WoS dataset, Figure 4.11b shows that the inferred dataset again has the lowest storage overhead. Even after compression, the open dataset (and the compressed MongoDB collection) had about the same size as the uncompressed inferred dataset. The reason is that WoS dataset structure has more nested values compared with the Twitter dataset. The vector-based format has less overhead for such data, as it does not store the 4-byte offsets for each nested value.

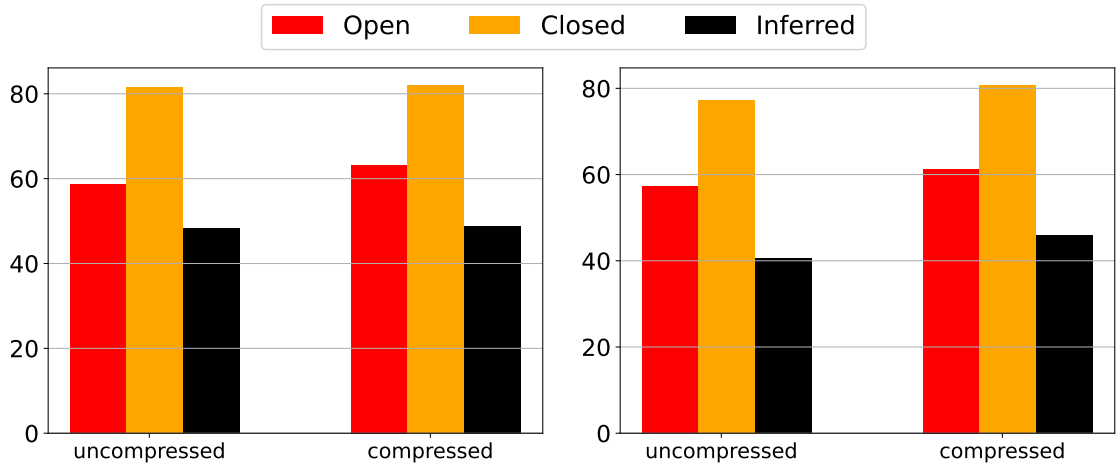
The Sensors dataset contains only numerical values that describe the sensors' status along with their captured readings, so this dataset's field name size to value size ratio is higher compared to the previous datasets. Figure 4.11c shows that, in the uncompressed dataset, the

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 — {"temp": **double**, "ts": **bigint**}. Compression reduced the sizes of the open and closed datasets 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 and WoS datasets, 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 size findings, both the syntactic (page-level compression) and semantic (tuple compactor) approaches alleviated the storage overhead as shown in Figure 4.11. The syntactic approach was more effective than the semantic approach for the Twitter dataset and the two were comparable for the WoS 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, 3.7x and 9.8x for the Twitter, WoS and Sensors datasets, respectively, compared to the open schema case in AsterixDB.

4.4.3 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 ex-



(a) SATA SSD (b) NVMe SSD

Figure 4.12: Data ingestion time for the Twitter dataset — feed (Minutes).

periment measures the overhead caused by performing point lookups to get the anti-schemas of previously ingested records. The Sensor dataset was also ingested through a data feed and showed similar behavior to the Twitter dataset; we omit these results here.

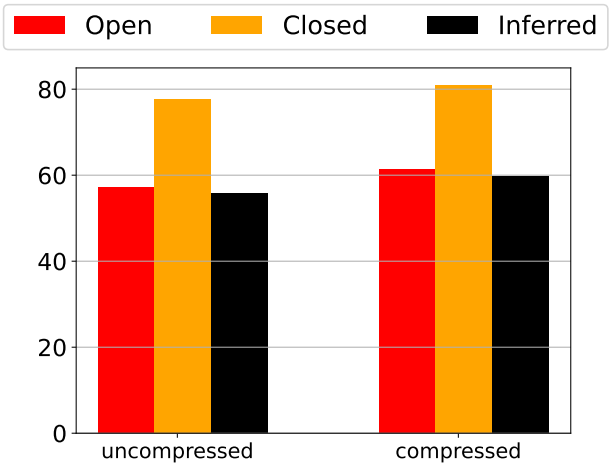
Continuous data ingestion from a data feed is sensitive to LSM configurations such as the merge policy and the memory budget. For instance, when cutting the memory budget by half, the size of flushed components would become 50% smaller. AsterixDB's default "prefix-merge policy" [39] could then suffer 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 bulk-loading, which builds a single on-disk component for the loaded dataset. (We evaluated the performance of bulk-loading into open, closed and inferred datasets using the WoS dataset.)

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 4.12 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 4.11a). 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.

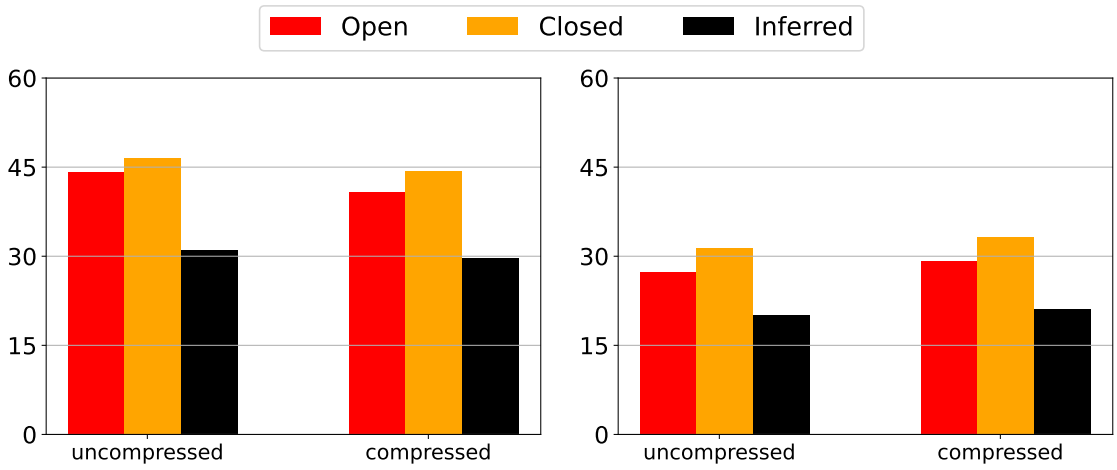
Data Feed (50% Updates) As explained in Section 4.3.2, updates require point lookups to maintain the schema, which can negatively impact the data ingestion rate. We evaluated the ingestion performance for update-intensive workload when the tuple compactor is enabled. In this experiment, we randomly updated 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 were updated equally. We created a primary key index, as suggested in [70, 71], to reduce the cost of point lookups of non-existent (new) keys. Figure 4.13 shows the ingestion time of the Twitter dataset, using the NVMe SSD



(a) NVMe SSD

Figure 4.13: Ingestion time for the Twitter dataset with 50% updates (Minutes)

drive, for the open, closed and inferred dataset with updates. The ingestion times for both open and closed datasets were about the same as with no updates (Figure 4.12b). For the inferred dataset, the ingestion time with updates took ∼27% and ∼23% more time for the uncompressed and compressed datasets, respectively, compared to one with no updates (Figure 4.12b). The ingestion times of the inferred and open datasets were comparable and both took less time than the closed dataset.



(a) SATA SSD

(b) NVMe SSD

Figure 4.14: Loading time for the WoS dataset — bulkload (Minutes)

Bulk-load. As mentioned earlier, continuous data ingestion is sensitive to LSM configurations such as the allocated memory budget for in-memory components. Additionally, the sizes of flushed on-disk components are smaller for the inferred and closed datasets as they have a smaller storage overhead than the open dataset (as we saw while ingesting the Twitter dataset). Smaller on-disk components may trigger more merge operations to reach the maximum mergeable component size. To eliminate those factors, we also evaluated the time it takes AsterixDB to bulk-load the WoS dataset. When loading a dataset, AsterixDB sorts the records and then builds a single on-disk component of the B^+-tree in a bottom-up fashion. The tuple compactor infers the schema and compacts the records during this process. When loading finishes, the single on-disk component will have a single inferred schema for the entire set of records.

Figure 4.14 shows the time needed to load the WoS dataset into open, closed, and inferred datasets. As for continuous data ingestion, the lower per-record construction cost of the vector-based format was the main contributor to the performance gain for the inferred dataset. We observed that the cost of the sort was relatively the same for the three schema datasets. However, the cost of building the B^+-tree was higher for both the open and closed schema datasets due to their higher storage overheads (Figure 4.11b).

As loading a dataset in AsterixDB does not involve maintaining transaction logs, the higher throughput of the NVMe SSD was noticeable here compared to continuous data ingestion. When compression is enabled, the SATA SSD slightly benefited from the lower IO cost; however, the faster NVMe SSD was negatively impacted by the compression due to its CPU cost.

4.4.4 Query Performance

We next evaluated the impact of our work on query performance by running analytical queries against the ingested Twitter, WoS, 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 six times and we report the average execution time of the last five.

We ran four queries (listed in Appendix A.2.1) against the Twitter dataset which retrieve:

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`.

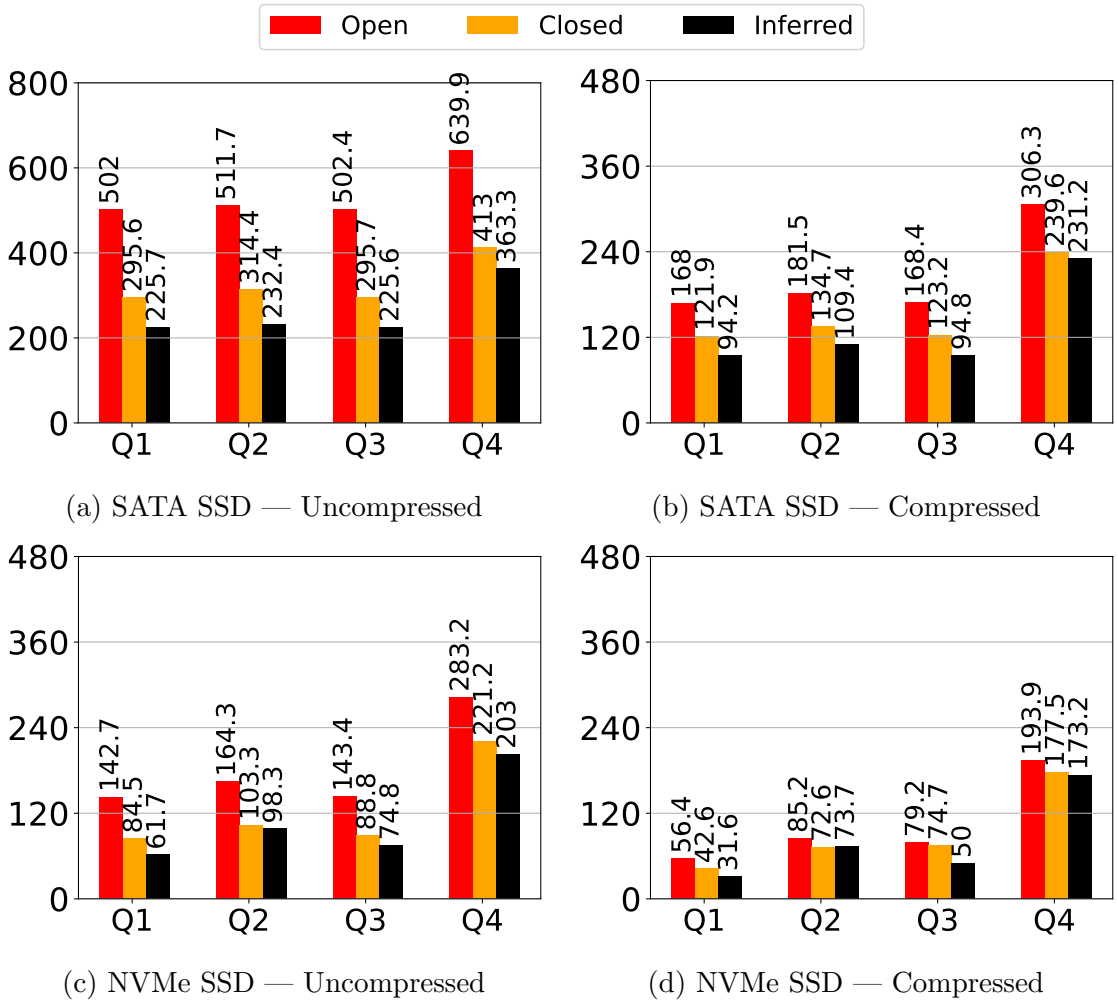
Q3. The top ten users who have the largest number of tweets that contain a popular hashtag — `EXISTS/GROUP BY ORDER BY`.

Q4. All records of the dataset ordered by the tweets' posting timestamps — `SELECT */ORDER BY`[2].

Twitter Dataset

Figure 4.15 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 4.11a. 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

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



(a) SATA SSD — Uncompressed

(b) SATA SSD — Compressed

(c) NVMe SSD — Uncompressed

(d) NVMe SSD — Compressed

Figure 4.15: Query execution time for the Twitter dataset (Seconds)

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 (as discussed in Section 4.3.4), 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. This experiment shows that our schema inference and tuple compaction approach can match (or even improve in some cases) the performance of querying datasets with fully declared schemas — without a need for pre-declaration.

WoS Dataset

We also ran four queries (listed in Appendix A.2.2) against the WoS dataset:

Q1. The number of records in the dataset — `COUNT(*)`.

Q2. The top ten scientific fields with the highest number of publications
— `GROUP BY/ORDER BY`.

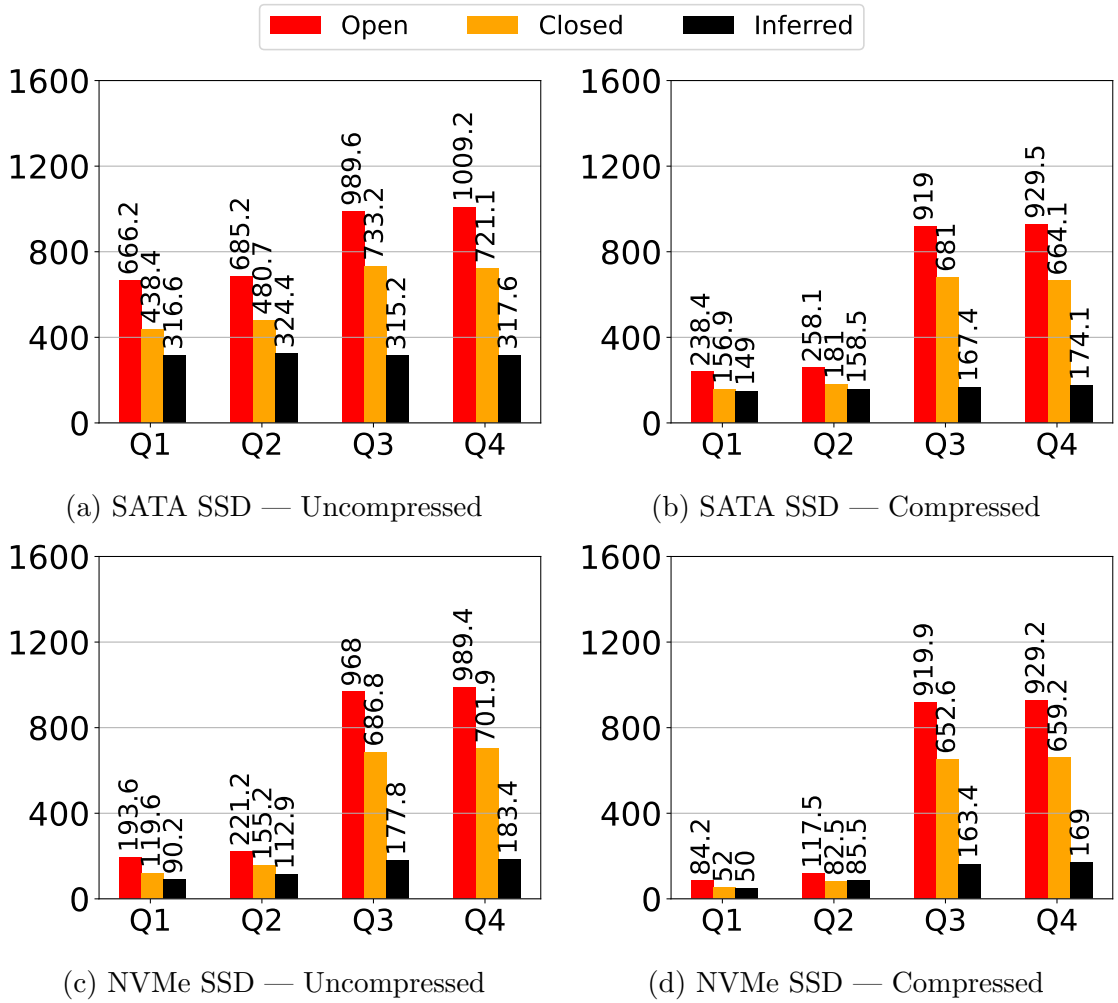
Q3. The top ten countries that co-published the most with US-based institutes
— `UNNEST/EXISTS/GROUP BY/ORDER BY`

Q4. The top ten pairs of countries with the largest number of co-published articles
— `UNNEST/GROUP BY/ORDER BY`

As Figure 4.16 illustrates, the execution times for Q1 and Q2 are correlated with the storage sizes of the three datasets (Figure 4.11b). For Q3 and Q4, the execution times were substantially higher in the open and closed datasets as compared to the inferred dataset. Similar to Q3 in the Twitter dataset, field access expression consolidation and pushdown were beneficial. Even after enabling compression, the open and closed schema execution times for Q3 and Q4 remained about the same despite the storage savings. (We will evaluate that behavior in more detail in Section 4.4.4).

Sensors Dataset

We again ran four queries (listed in Appendix A.2.3):



(a) SATA SSD — Uncompressed

(b) SATA SSD — Compressed

(c) NVMe SSD — Uncompressed

(d) NVMe SSD — Compressed

Figure 4.16: Query execution time for the WoS dataset (Seconds)

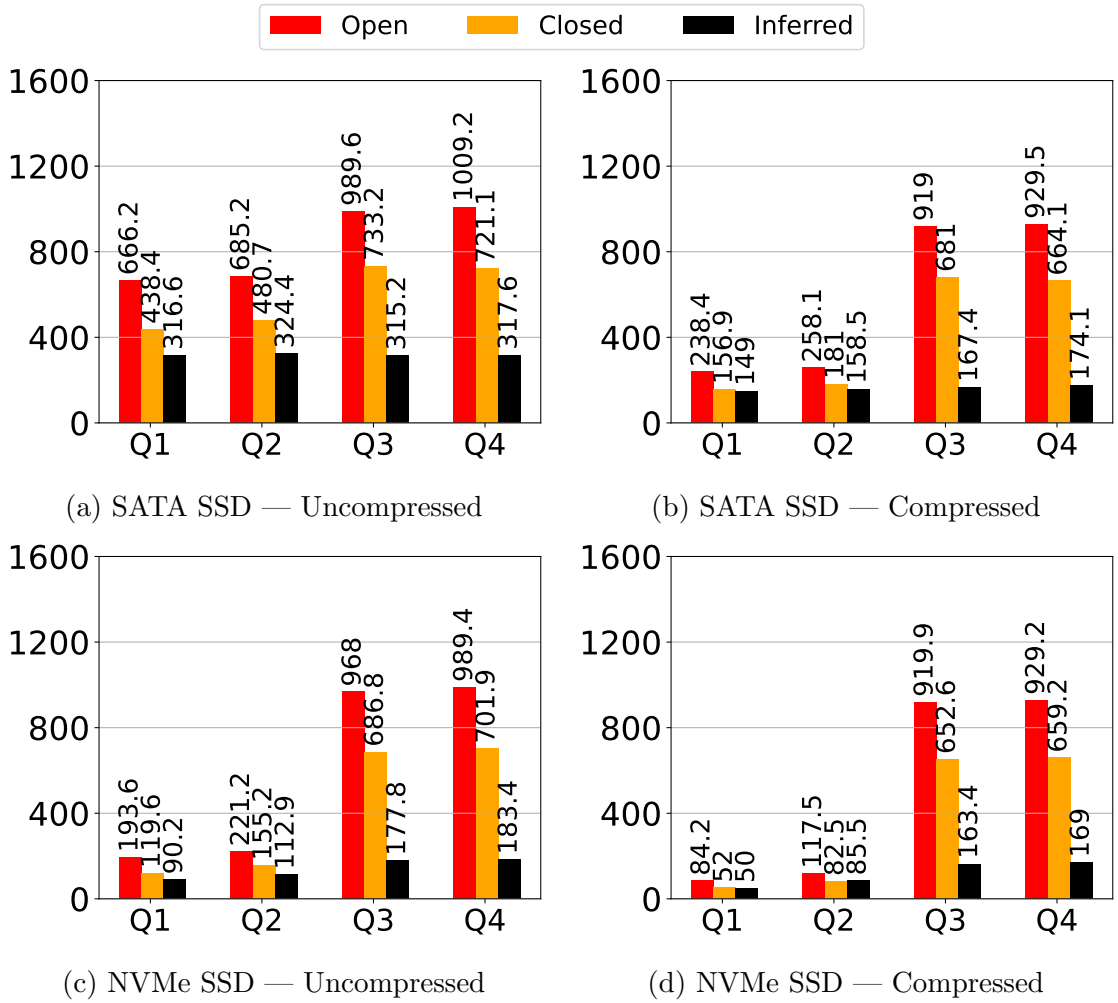
Q1. The number of records in the dataset — `COUNT(*)`.

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`

The execution times are shown in Figure 4.17. The execution times for Q1 on both uncompressed and compressed datasets correlate with the storage sizes of the datasets from



(a) SATA SSD — Uncompressed

(b) SATA SSD — Compressed

(c) NVMe SSD — Uncompressed

(d) NVMe SSD — Compressed

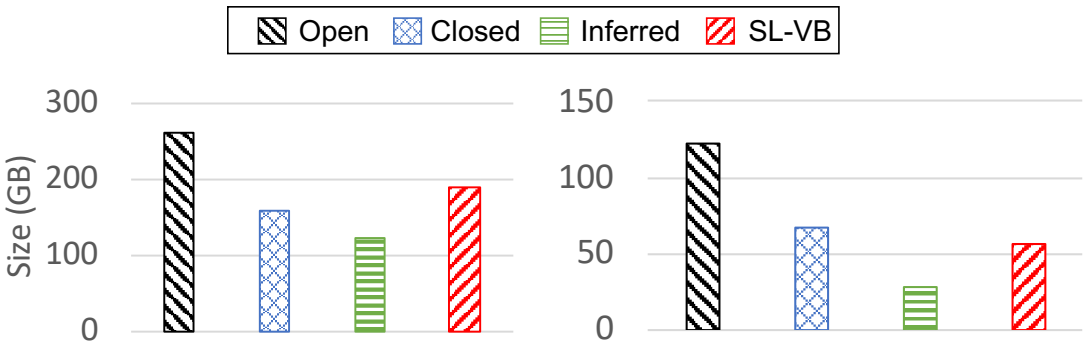
Figure 4.17: Query execution time for the Sensors dataset (Seconds)

Figure 4.11c. 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.

Impact of the Vector-based Optimizations

Breakdown of the storage savings. 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 4.3.3). To measure the impact of the newly proposed format, we reevaluate the storage size of the vector-based without inferring the schema or compacting the records (i.e., a schema-less version using the vector-based format), which we refer to as *SL-VB*.

In Figure 4.18a, 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 4.18b shows a similar pattern; however, the SL-VB Sensors dataset is smaller than the closed dataset for the reasons explained in Section 4.4.2.



(a) Twitter (b) Sensors

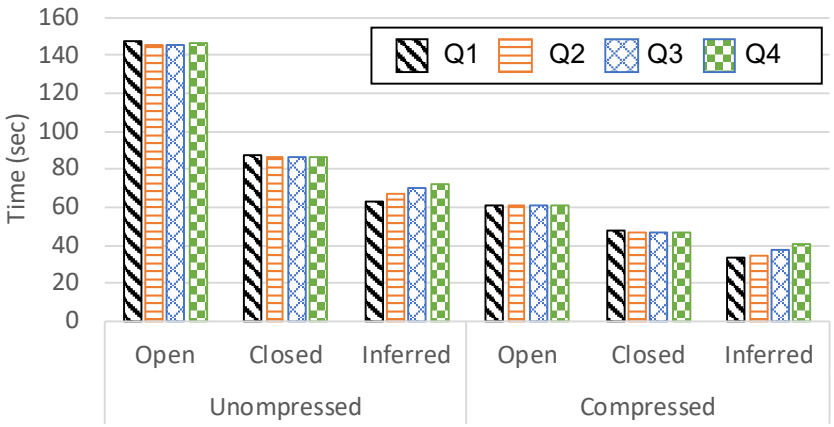
Figure 4.18: Impact of the vector-based format on storage

Linear-time field access. 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

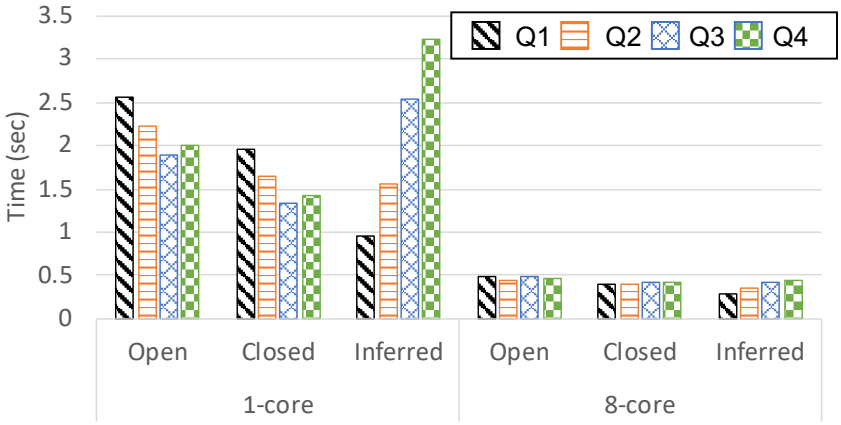
is faster than accessing a value that resides at the end. To measure the impact of linear access in the vector-based format, we ran four queries against the Twitter dataset (using the NVMe SSD drive) where each counts the number of appearances of a value. The positions (or indexes) of those values in the vector-based format are 1, 34, 68, and 136 for Q1, Q2, Q3 and Q4, respectively, where position 1 means the first value in the record and the position 136 is the last. Figure 4.19a shows the time needed to execute the queries. In the inferred datasets, the position of the requested value affected the query times, where Q1 was the fastest and Q4 was the slowest. For the open and closed datasets, the execution times for all queries were about the same. However, all four queries took less time to execute in the inferred cases, due to the storage savings. When all the data fits in-memory, the CPU cost becomes more apparent as shown in Figure 4.19b. In the case of a single core, the vector-based format was the slowest to execute Q3 and Q4. When using all 8-cores, the execution time for all queries were about the same for the three datasets.

Field-access consolidation and pushdown. Also in our experiments, 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.

The execution times of the queries are shown in Figure 4.20. We refer to *Inferred (un-op)* as querying the inferred dataset without our optimization of consolidating and pushing down field access expressions. When we disable our optimizations, 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 to get the (i) sensor ID, (ii) readings array, and (iii) reporting timestamp. 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



(a) Large: 200GB



(b) Small: 5GB

Figure 4.19: Impact of the vector-based format on storage

Q3 took twice as much time to finish for Inferred (un-op). Q2 is still faster to execute in the Inferred (un-op) case than in the closed case whereas Q3 took slightly more time to execute. Finally, disabling our optimizations improved the execution time for Q4 on the NVMe SSD, as delaying the evaluation of field accesses can be beneficial for queries with highly selective predicates.

Vector-based format vs. others. Other formats, such as Apache Avro [6], Apache Thrift [31], and Google Protocol Buffers [32], also exploit schemas to store semi-structured data more efficiently. In fact, providing a schema is not optional for writing records in such

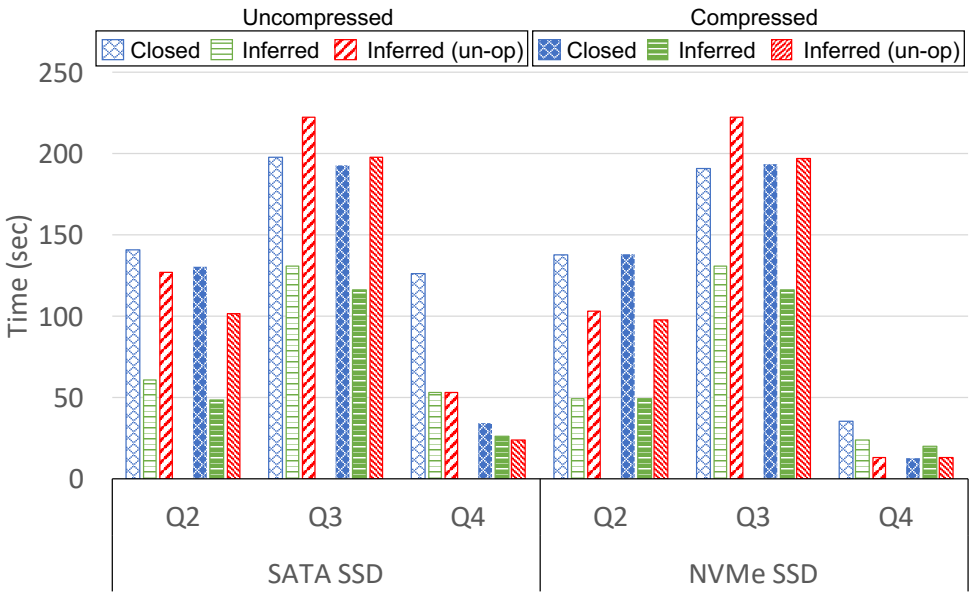


Figure 4.20: Impact of consolidating and pushing down field access expressions

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 4.2 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.

	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

Table 4.2: Writing 52MB of Tweets in different formats

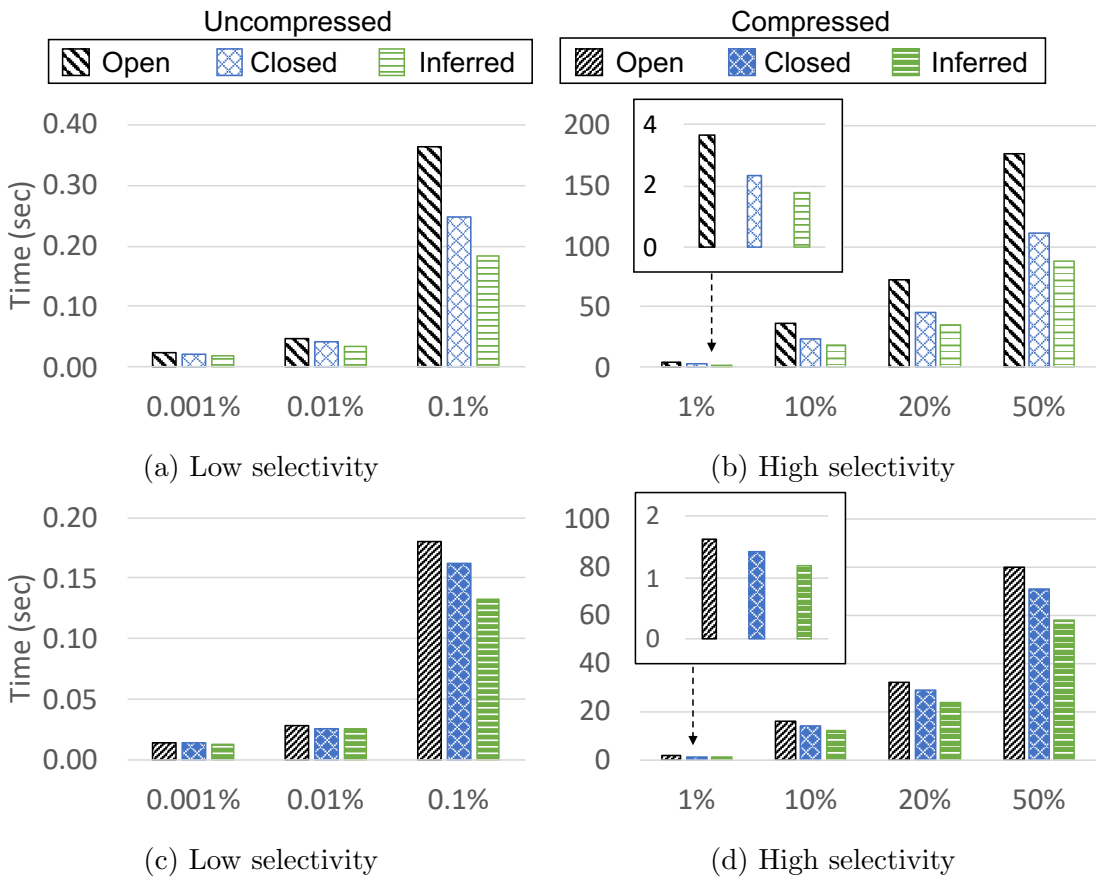


Figure 4.21: Query with secondary index (NVMe)

Secondary Index Query Performance

Pirzadeh et al. [82] 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. In this experiment, we evaluated the impact of having a secondary index using the Twitter dataset.

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. We created a secondary index on this generated timestamp attribute and ran multiple range-queries with different selectivities. For each query selectivity, we executed queries with different range predicates to warm up the system's cache and report the average stable execution

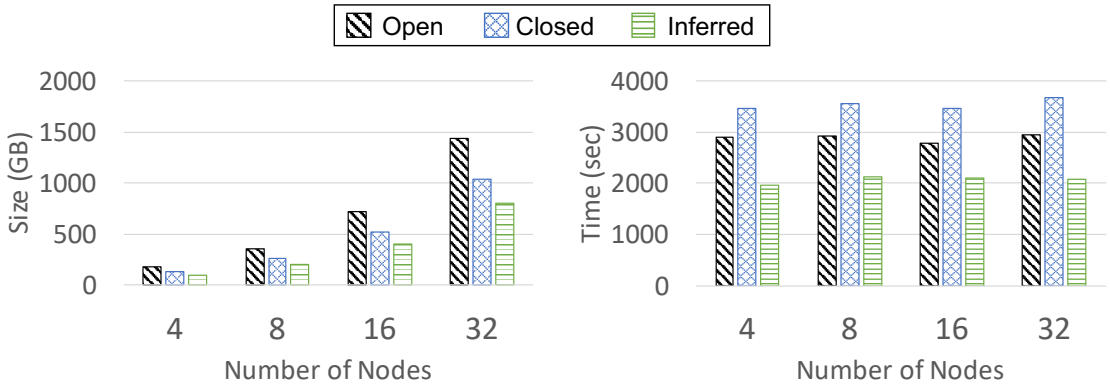
time. Figures 4.21a and 4.21b show the execution times for queries, with both low and high selectivity predicates, using the NVMe SSD, for uncompressed datasets. The execution times for all queries correlated with the storage sizes (Figure 4.11a), where the closed and inferred datasets have lower storage overhead compared to the open dataset. The execution times for compressed datasets (Figures 4.21c and 4.21d) showed similar relative behavior.

4.4.5 Scale-out Experiment

Finally, to evaluate the scalability our approach, 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 evaluate the ingestion and query performance of the Twitter dataset using clusters with 4, 8, 16 and 32 nodes. We configure each node with 10GB of total memory, with 6GB for the buffer cache and 1GB for the in-memory component budget. The remaining 3GB is allocated for working buffers. We used the instance *ephemeral* storage to store the ingested data. Due to the lack of storage space in a `c5d.2xlarge` instance (200GB), we only evaluate the performance on compressed datasets.

Figure 4.22a shows the total on-disk size after ingesting the Twitter data into the open, closed and inferred datasets. The raw sizes of the ingested data were 400, 800, 1600 and 3200 GB for the 4, 8, 16 and 32 node clusters, respectively. Figure 4.22b shows the time taken to ingest the Twitter data into the three datasets. As expected, we observe the same trends seen for the single node cluster (see Figure 4.11a and Figure 4.12), where the inferred dataset has the lowest storage overhead with the highest data ingestion rate.

To evaluate query performance, we ran the same four Twitter queries as in Section 4.4.4. Figure 4.23 shows the execution times for the queries against the open, closed and inferred datasets. 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,



(a) On-disk size (b) Ingestion time

Figure 4.22: Storage and ingestion performance (scale-out)

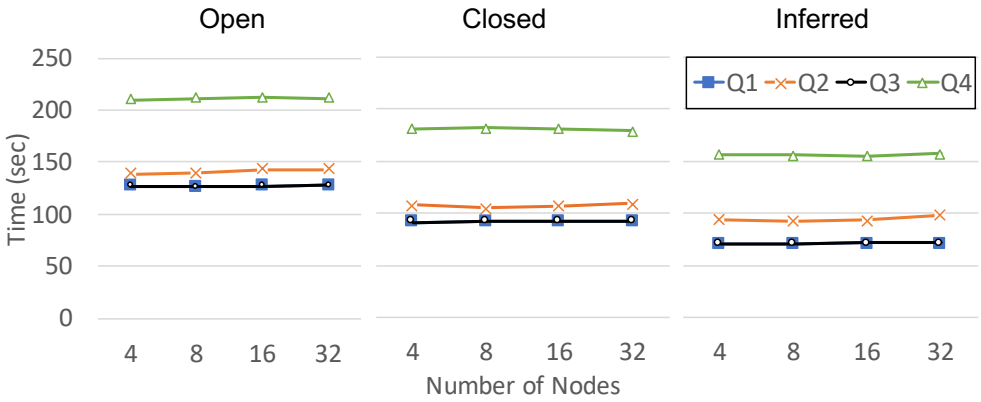


Figure 4.23: Query performance (scale-out)

each partition broadcasts its schema to the other nodes in the cluster (Section 4.3.4) at the start of a query. However, the performance of the queries was essentially unaffected and was still faster to execute in the inferred dataset.

4.5 Related Work

In Chapter 3, we have comprehensively discussed the recent research on reducing the storage overhead in document stores. Here we mainly focus on recent work related to our proposed tuple compactor framework.

Creating **secondary indexes** is related to declaring attributes in schemaless document stores. Azure DocumentDB [87] 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 in the primary index.

Semantically compacting self-describing, semi-structured records using schemas appears in popular big data systems such as Apache Spark [12] and Apache Drill [7]. 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 [2]). 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 those formats, whereas it is optional for the vector-based format. Apache Parquet [9] (or Google Dremel [74]) uses 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 [42, 68]. 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 [33, 41, 89]. 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 [41], 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. [33] 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 [89], 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.

4.6 Conclusion

In this chapter, after first introducing a syntactic compression scheme, we introduced a tuple compaction framework that addresses the overhead of storing self-describing records in LSM-based document stores. The semantic 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, we saw that the tuple compactor and vector-based record format can improve the 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.

Chapter 5

Columnar Formats for Schemaless LSM-based Document Stores

5.1 Introduction

In recent years, as described in Chapter 1, 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 [89, 74]. As a result, open source and commercial relational column-store systems such as MonetDB [21, 94] (and the commercial version Actian Vector [1]), and C-Store [89] (commercialized as Vertica [29]) have gained more popularity as data warehouse solutions.

For nested data, Dremel [74] and its open source implementation Apache Parquet [9] 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 [22], Couchbase Server [17] or, Apache AsterixDB [3, 38, 51] 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 chapter, we show that users with analytical 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 [79] 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 Chapter 4 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 two new models in our work here for storing columns in an LSM B^+-tree index. In the first model, we store columns using a Partitioned Attributes Across (PAX)-like [35] format, where each column occupies a contiguous region (called a minipage) within a B^+-

tree's leaf page. We refer to this model as the AsterixDB PAX model or APAX for short. In the second model, we stretch the PAX minipages to become megapages, where a column could occupy multiple pages. We refer to this model as the AsterixDB Mega-Attributes Across (AMAX). Despite their names, Despite their names, these layouts are agnostic of the columns' structure and see each column as a series of bytes; hence they should only require a few modifications to be adopted by other LSM-based document stores.

To show their benefits, we have implemented both the APAX and AMAX layouts to store document data in a columnar format in Apache AsterixDB. This has enabled us to conduct an extensive evaluation of the APAX and AMAX formats and present their tradeoffs in terms of (1) ingestion performance, (2) query performance, and (3) memory and CPU consumption for different datasets.

The remainder of this chapter is structured as follows: Section 5.2 details our extensions to the Dremel format. Section 5.3 presents the structure of both the APAX and AMAX layouts and discusses the challenges of reading and writing records in both. Section 5.4 presents our evaluation of the propsed columnar formats. Section 5.5 surveys related work. Finally, Section 5.6 concludes the chapter.

5.2 A Flexible Columnar Format for Nested Semi-structured Data

Inferring the schema and compacting schemaless semi-structured records, using Chapter 4's tuple compactor framework, 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 [74] 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.

5.2.1 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 5.2.2, we detail how our extensions to the Dremel format allow for schema changes. For better illustration, Figure 5.1a 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 5.1b for the Dremel format and in Figure 5.1c 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

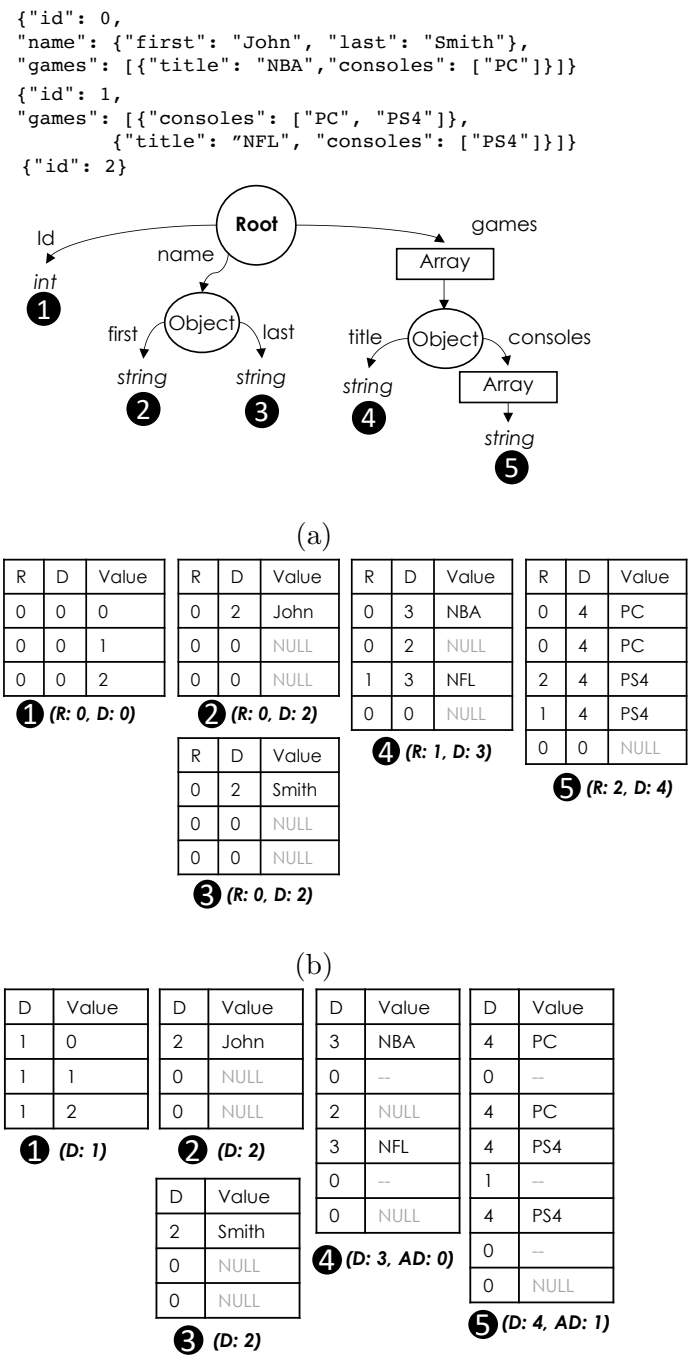


Figure 5.1: (a) Raw JSON records and their schema (b) Dremel columnar representation (c) Extended Dremel representation

store model, which is the scope of this paper. We encourage interested readers to refer to [74, 53] for more details on the representation of non-optional values.

The tables in Figure 5.1b and Figure 5.1c depict the columnar-striped representation of the records' atomic values from Figure 5.1a 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 [74]. 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 5.1b, 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 5.1a. 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) $\rightarrow$ $name$ (1) $\rightarrow$ $first$ (2)). In the first record in Figure 5.1b, 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 5.1 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 5.1c. 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 5.1b – as it is a non-optional field. However, in our extension, the maximum definition level (for

the same column ❶ shown in Figure 5.1c) is 1 even though the field *id* is also not optional. As discussed later in Section 5.2.3, 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 5.1b, the values of column ❺ belong to the two nested arrays *games* and *consoles* in Figure 5.1a. 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 5.1c 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.

5.2.2 Schema Changes

For LSM-based document stores, one could use the approach proposed in Chapter 4 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. Similarly, document stores allow storing an array that consists of heterogeneous values, such as the array [0, `"1"`, {`"seq"`: 2}]. Limiting the support for storing data in a columnar format to datasets with homogeneous values is maybe enough for many cases, as evidently shown by the popularity of Parquet. However,

including support for datasets with heterogeneous values is a desired feature for certain use cases, especially when the users have no control over how the data is structured, like when ingesting data from web APIs [56, 88]. In this section, we address the two main challenges: (i) handling schema changes and (ii) handling data with heterogeneous types.

In Chapter 4, we introduced union types in our inferred schemas to represent values with heterogeneous types. Figure 5.2 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. The first is the field *name*, which could be a string or an object. Thus, we infer *name*'s type as a union of string and object. The second union type corresponds to the *games* array's elements, where each element could be of type string or array of strings. 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 5.2, has two children, where the key "string" corresponds to the left child, and the key "object" corresponds to the right child.

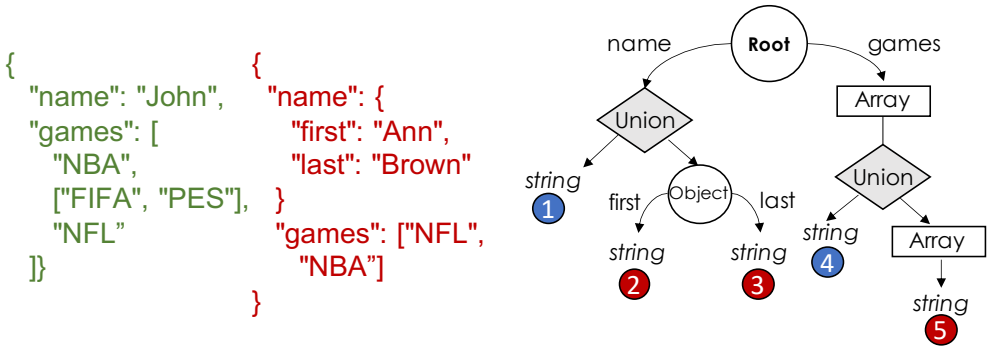


Figure 5.2: Example of heterogeneous values and their schema

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

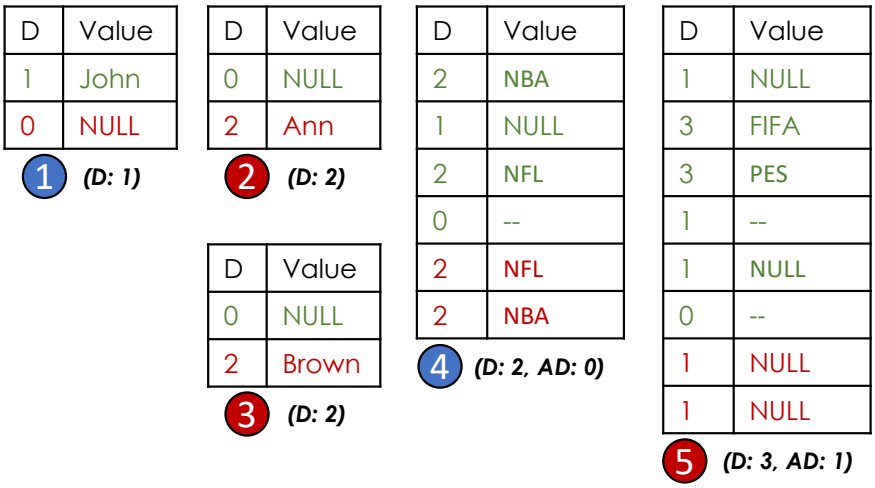


Figure 5.3: Columnar representation of the records in Figure 5.2

the records' values can be performed while inferring the schema in a single pass, as in the compaction process in Chapter 4. After inserting the first record of Figure 5.2, 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 1 in Figure 5.3. 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 5.2. Since the second record is the first to introduce the field *name* as an object, we can write NULLs in the newly inferred columns 2 and 3 for all previous records. Then, we write the values "Ann" and "Brown", with definition levels 2 in 2 and 3, respectively. Recall that only a single value can be present in a union type; therefore, we write a NULL in column 1. After injecting the union node in the path root → union → string, we do not change the definition level of column 1 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 2 and 3. 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 5.2 and Figure 5.3, accessing *name* goes as follows: First, we inspect column ①, which corresponds to the string child of the union. If we get a NULL from ①, we need to proceed to the following type, an object with two fields, $first$ ② and *last* ③. In this case, we need to inspect one of the values' definition levels, say column ②. 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 ③ is sufficient to determine whether the value is present or not. Thus, the results of accessing the value *name.last* are NULL in the first record and "Brown" in the second record.

The types of repeated values (array elements) can alternate between two or more types, as in the array *games* in Figure 5.2. In the first record, the elements' types of the array *games* are either a string or an array of strings. Similar to the value *name*, when accessing the value *games*, we need to inspect both columns ④ and ⑤ to determine which element of the two types is present. When accessing the value *games*, we see that the first value's definition level in column ④ is 2, which is the maximum definition level of the column for the string value "NBA". In column ⑤, however, the definition level is 1, which indicates that the inner array of the union type is NULL. Thus, we know that the first element of the array *games* corresponds to the string alternative of the union type. The following definition level 1 in column ④ indicates that the second element is NULL, whereas it is 3 in column ⑤, which is the maximum definition level of the column. Hence, the second element of the array is of type array of strings. The two values with definition levels 3 and the following

delimiter with definition level 1 correspond to the two elements ["FIFA", "PES"] of the first record. Following the delimiter, the definition level 1 in column 5 indicates that the third element of the outer array is NULL. However, the definition level 2 in column 4 for the value "NFL" indicates that the third value of the outer array is a string. The delimiter 0 in both columns 4 and 5 indicate the *games*'s end of values for the first record. The final result of accessing the value *games* in the first record, therefore, is ["NBA", ["FIFA", "PES"], "NFL"], which preserves the original value of the record shown in Figure 5.2. In the second record, we can see that the array consists of two elements, both of type string. Hence, the two NULL values in column 5 indicate that neither of the two elements is of the array of strings alternative of the *games*'s union type.

5.2.3 LSM Anti-matter

In Section 2.3, 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 5.4 shows the columnar representation of the component C_1 from Figure 2.8a in Section 2.3. 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 2.3). When the definition level is 1, we know that it is a non-anti-matter record. Figure 5.4 also shows that the anti-matter has an entry for the column *id* but none of the others.

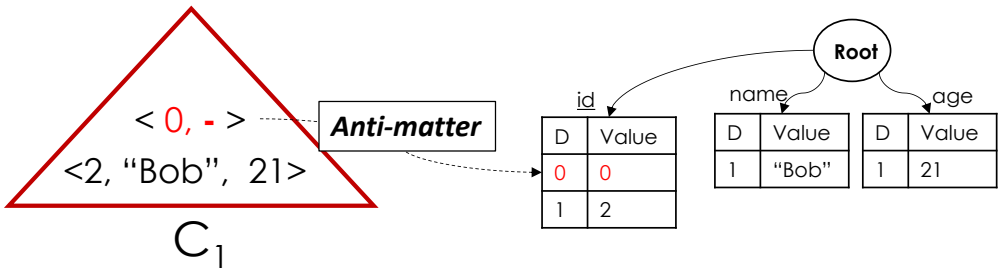


Figure 5.4: Representing anti-matter tuples

5.2.4 Record Assembly

When accessing a nested value such as the nested value *name* in Figure 5.1b 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 [74]. 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.

5.3 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 two layouts for storing the columns in LSM-based document stores: (i) AsterixDB Partitioned Attributes Across *(APAX)* and (ii) AsterixDB Mega Attributes Across *(AMAX)*. We have implemented and evaluated both layouts in Apaches AsterixDB, hence the names. In the following sections, we first briefly explain the supported techniques used to encode the column values. Then, we detail the

structures of both the APAX and AMAX layouts. Next, we describe the lifecycle of reading and writing the columns, and finally, we cover challenges related to answering queries with secondary indexes.

5.3.1 Encoding

Apache Parquet offers a rich set of encoding algorithms [11] 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.)

5.3.2 *APAX* Layout

Ailamaki et al. proposed Partition Attributes Across (PAX) [35], a cache-friendly page layout as compared to the commonly used row-major layout (or N-ary Storage Model, a.k.a., slotted pages). PAX pages store each attribute's values contiguously in minipages. APAX minipages can be reached by relative pointers stored in the pages' header. Within a PAX page, fixed-length and variable-length values are stored in F-minipages and V-minipages, respectively. Along with the values, F-minipages contain a bit vector to indicate whether a value is present or NULL. The V-minipage stores values similar to the F-minipage values; however, instead of the presence bits, the V-minipage uses values' offsets to determine the lengths of each variable-length value. NULL offsets (e.g., offset zero) indicate NULL values on the V-minipage.

Our APAX layout is a modified version of the PAX layout, as shown in Figure 5.5, where fixed-length and variable-length values are encoded and stored in homogeneous mini-pages

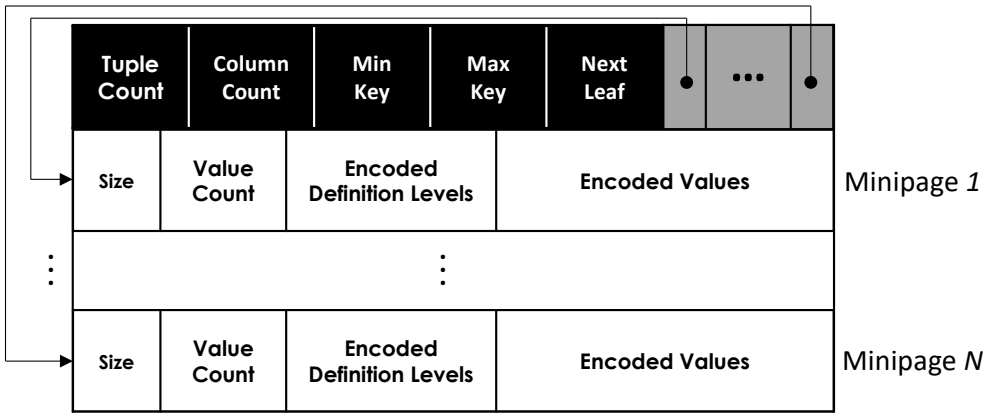


Figure 5.5: APAX page layout

(i.e., no F-minipages and V-minipages). Thus, APAX is agnostic of its minipages' contents, and it is up to the minipages' readers and decoders to interpret the minipages' content, where the inferred schema determines the minipages' appropriate readers and decoders. Figure 5.5 shows the organization of an APAX page. The reader will read the first four bytes to determine the size of the encoded definition level. Then, it will pass both the encoded definition levels and the encoded values to the appropriate decoders. The resulting decoded definition levels and values are then processed, as explained earlier in Section 5.2. As in PAX, we can reach each minipage via pointers stored in the APAX page header. Since APAX pages reside as leaf pages in a B^+-Tree, we store the minimum and the maximum keys (primary keys) within the APAX page header. By doing so, we can access their minimum and maximum keys directly when performing B^+-tree operations (e.g., search) without the need to decode the primary keys. The header also stores the number of minipages (or columns) and the number of records stored in the APAX page.

5.3.3 *AMAX* Layout

The PAX and APAX layouts store 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 5.6 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 with the same structure as an APAX column as shown in Figure 5.6. 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 5.6 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. As in an APAX's minipage, the content of an AMAX's megapage is encoded, and specific readers and decoders (determined by the schema) are used for interpreting their content.

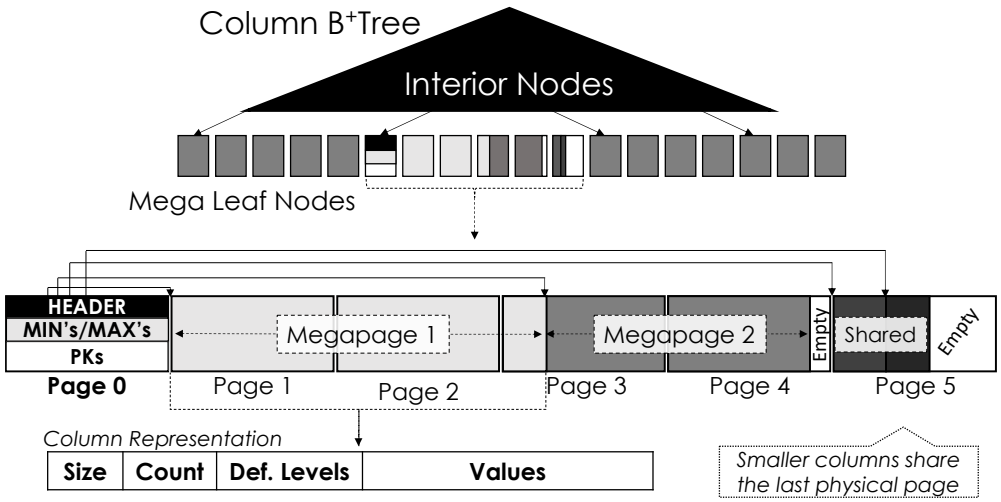


Figure 5.6: AMAX multi-page layout in a B-$^+$tree

5.3.4 Writing

As in the Tuple Compactor Framework (Chapter 4), 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 – storing the columns as APAX or AMAX pages. Finally, the inferred schema (e.g., Figure 5.1a) is persisted into the flushed component's metadata page as in the tuple compactor (Chapter 4).

Storing columns in APAX or AMAX layouts has different implications in terms of CPU and memory usage. In the following, we show and discuss our approach for writing the columns' values as APAX and AMAX pages.

Writing APAX Pages

Determining the sizes of our APAX minipages is even more challenging than determining PAX minipages' sizes, as we incrementally encode each column's values. To address this issue, we first write the columns' values into temporary buffers, where each temporary buffer is dedicated to a single column. Once the temporary buffers have a page's-worth of values, we copy and align their contents as APAX minipages and write the resulting APAX page into the disk. We reuse the same temporary buffers to construct the following APAX pages for the remaining records of the flushed in-memory component.

Writing AMAX Pages:

As opposed to APAX minipages, 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 5.3.7. 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 [10] – for example, a smaller data page size is more suitable for single row lookups.

5.3.5 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., APAX minipages or AMAX mega-pages) for executing the query. Moreover, in AMAX, 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 in AMAX 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 APAX or 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, whether APAX or AMAX. 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.

5.3.6 Impact of LSM Merge Operations

From time to time, an LSM merge operation is scheduled to compact the on-disk components. In both the AMAX and APAX layouts, 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 5.3.5, 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 [34], 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 [72], 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 [72], should be conducted to measure those tradeoffs; however, it is beyond the scope of this work, so we leave it for future work.

5.3.7 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 APAX or AMAX layouts is more expensive than in their row-major counterparts, as we need to decode primary keys and search for the requested value in both layouts. To alleviate the cost of point lookups for datasets in both the APAX and AMAX layouts, 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 [70, 71]. 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 [70] 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.

5.4 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 Chapter 4, (iii) APAX, and (iv) AMAX. Again, Open and VB are both row-major formats, whereas APAX and AMAX are columnar formats.

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. Unless otherwise noted, we configured AsterixDB with a single node and eight partitions (Section 2.3). 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 *APAX* or *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 [26] compression scheme to reduce the storage footprint for all formats.

5.4.1 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 5.1 lists and summarizes the characteristics of the five datasets. In Table 5.1, *# of Columns* refers to the number of inferred columns for records in *APAX* or *AMAX* layouts.

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 [14], 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 [30]. 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 columnar layouts. Lastly, we obtained the *tweet_1* and *tweet_2* datasets using the Twitter API [28], where we collected the tweets in *tweet_1* from September 2020 to January 2021. The *tweet_2* dataset is a sample of tweets ($\sim$ 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 5.1.

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 5.3.7. 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 5.3.7. 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 5.4.3.

	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. Record Size	141B	3.8KB	5.3KB	6.2KB	2.7KB
# of Columns	7	16	933	296	275
Dominant Type	Mix	Integer	String	String	String

Table 5.1: Datasets summary

5.4.2 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 5.7 shows the total on-disk size after ingesting the five datasets using the four layouts: *Open*, *VB*, *APAX*, 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 5.7 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 ($\sim$ 17% smaller) due to compaction [36]. Similarly, the records in both of the columnar layouts, *APAX* and *AMAX*, took about the same space; however, compared to the records in the *Open* format, the sizes are 45% and 50% smaller for the *APAX* and *AMAX* layouts, respectively. The storage overhead reductions in the *APAX* and *AMAX* layouts are due to (i) storing no additional information (e.g., field names) with the values, compared to *Open*, and (ii) the values being encoded, which is not possible in the row-major layouts.

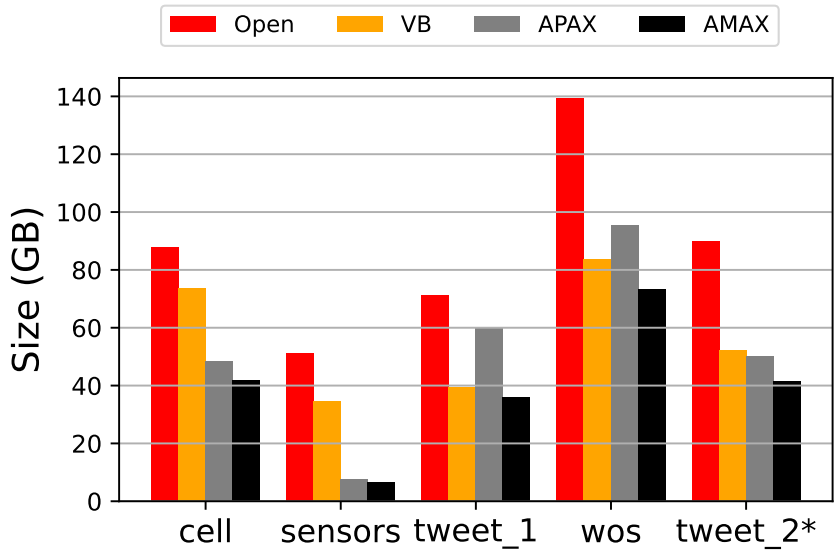


Figure 5.7: Storage size

The impact of encoding in both the $APAX$ and $AMAX$ layouts becomes apparent for storing the *sensors* dataset, where the values' are primarily numeric. Figure 5.7 shows that the *sensors* records in the *Open* and VB layouts took 7.2X and 4.8X more space compared to the records in the $APAX$ layout, respectively, and 8.5X and 5.6X more space compared to the records in 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 and the encoding becomes less effective. Storing this data in a columnar layout did not show a significant improvement, as shown in Figure 5.7, compared to the records in a row-major layout. In fact, the records in the $APAX$ layout took 35% more space than the records in the VB layout. The reason behind the high storage overhead in $APAX$ is the excessive number of columns of the *tweet_1* dataset, as shown in Table 5.1, so each minipage stores a small number of values compared to the *cell* and *sensors* datasets. Thus, in some instances, the encoding imposes a negative impact, as the encoded values store additional information for decoding, and this information occupies more space than the encoding saves. However, the $AMAX$ layout is not as sensitive to the number of columns, as a column can span multiple pages, and hence, the number of values is sufficient for the encoding to be effective. The storage saving from the $AMAX$ layout is negligible compared to the VB layout, as encoding large textual values is relatively less effective compared to numerical values.

Storing the *wos* dataset using the four layouts shows a similar trend, shown in Figure 5.7, as in the *tweet_1* dataset, even though the number of columns in the *wos* dataset is not as excessive, as shown in Table 5.1. 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 could consist of multiple paragraphs — exceeding the number of characters of a tweet. Hence, the larger values of the *wos* dataset limited the number of values we could store in an *APAX* page, which again reduced the effectiveness of encoding. The *wos* records took more space in the *Open* layout than other layouts due to the *Open* layout's recursive structure (as detailed in [36]), where deeply nested values require 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.

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 four layouts. Hence, the differences between the sizes for the different layouts, shown in Figure 5.7, correspond to the layouts' characteristics. For instance, the sizes of the records in the *VB*, *APAX*, and *AMAX* layouts are comparable, with *AMAX* being slightly smaller. However, the *Open* layout took more space due to the reasons explained earlier.

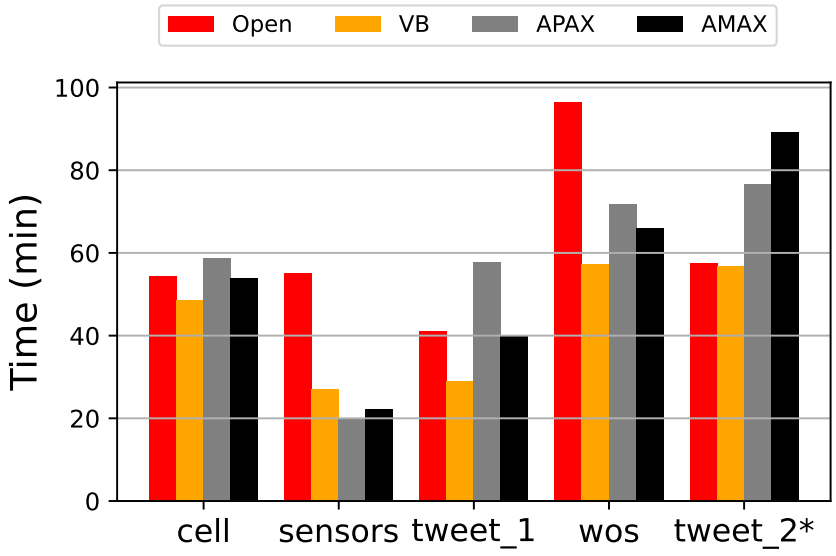


Figure 5.8: Ingestion time

5.4.3 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 $APAX$ and $AMAX$ formats 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 [72], 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 5.3.6) while merging $APAX$ and $AMAX$ components. Additionally, we limited the number of primary keys in AMAX's Page 0 to 15K (default value) for the reasons discussed in Section 5.3.4.

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 5.1. However, it also has the most records. In AsterixDB's original configuration (i.e., a single NC with eight partitions), the ingestion rate of the *cell* dataset was the slowest among the datasets — it took more than 8000 seconds to ingest the dataset

under the four layouts. The main reason is that the eight partitions share the same resources, including the transaction log buffer where each partition writes entries to commit their transactions. With the *cell* dataset's high record cardinality, writing to the transaction log buffer became a bottleneck. To alleviate the contention on the transaction log buffer, we reconfigured AsterixDB to have four virtual nodes [1], each with two partitions. We divided the memory budget equally among the four nodes. Using this configuration, Figure 5.8 shows the time it took to ingest the *cell* dataset using the four layouts. We see that the ingestion rate is about the same for the four layouts, as writing to the transaction log buffer is still a major bottleneck. However, the ingestion rate using the new configuration improved significantly — from more than 8000 seconds to the vicinity of 3000 seconds.

In the *sensors* dataset, in contrast to the *cell* dataset, the ingestion rate varied among the different layouts as shown in Figure 5.8. Ingesting *Open* records took more time than records in the other layouts due to the record construction cost of the *Open* layout [36]. The recursive nature of the *Open* layout requires copying the child's values to the parent from the leaf to the root of the record, which means multiple memory copy operations for the same value. In contrast, constructing records in the VB layout is more efficient, as the values are written only once [36]. Thus, ingesting records in the VB layout took 50% less time in comparison. Recall that the records of the in-memory components are in the VB format for $APAX$ and $AMAX$ (as discussed in Section 5.3.4), and during the flush operation, the records are transformed into a columnar layout. Thus, the lower construction cost of the VB records contributed to the higher ingestion rate of both the $APAX$ and $AMAX$ layouts. 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

[1] Usually, only a single node is configured per computing node.

more apparent due to the higher number of columns in those two datasets. Figure 5.8 shows that the ingestion time for *tweet_1* using the *APAX* layout was the longest. As explained earlier in Section 5.4.2, a higher number of columns can negatively affect the number of values that we can store in *APAX* pages, and thus, more pages are required to store the ingested records. Also, recall that the columns' values are first written into temporary buffers and then copied to form an *APAX* page (Section 5.3.4). Consequently, we need to iterate over the temporary column buffers (933 in total as shown in Table 5.1) to construct each *APAX* page. The cost of constructing a large number of *APAX* pages took most of the time to ingest the *tweet_1* dataset. We did not observe a similar behavior when constructing the *AMAX* pages; most of the time here was spent performing LSM merges, as we need to fetch all 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 5.8) for the *APAX* and *AMAX* layouts. 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 *APAX* and *AMAX* layouts was comparable and slightly slower than the *VB* layout, as the cost of transforming the ingested records into a column-major layout during the flush operation and decoding and encoding the values during the merge operation was higher for the *wos* dataset compared to the *sensors* dataset.

Update-intensive

We evaluated the ingestion performance for insert-only workloads using different datasets, and we saw that the ingestion rate using columnar layouts, in general, was faster or compara-

ble 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 are updated equally. Prior to starting the data ingestion, we created two indexes: one on the primary keys, which we call a primary key index, to minimize the cost of point lookups of non-existent (new) keys. The second index is on the *timestamp* values. Figure 5.8 shows *tweet_2* the ingestion time for the four different layouts. The ingestion times for records in the $APAX$ and $AMAX$ were $\sim$ 24% and $\sim$ 35% slower than the *Open* layout, respectively. 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. Recall that the cost of searching for a value in a columnar layout is linear (v.s. logarithmic in a row-major layout), and 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 layouts became higher than for the row-major layouts. Even though we only need to read the pages corresponding to the *timestamp* in the $AMAX$ layout (i.e., less I/O cost), its update cost was higher than the $APAX$ layout. This was due to decoding large numbers of *timestamp* values stored in $AMAX$ megapages (a CPU cost) for each update. We will soon (Section 5.4.4) discuss the benefit of secondary indexes when answering queries; however, the cost of maintaining the correctness of secondary indexes is high for columnar layouts. Thus, one should consider how often the index would be utilized.

5.4.4 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. Appendix B lists the queries and Table 5.2 summarizes the queries used for each dataset. Q1, which

*	Q1	The number of records
cell	Q2	The top 10 callers with the longest call durations
	Q3	The number of calls with durations $\geq$ 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

Table 5.2: A summary of the queries used in the evaluation

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. 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.

For the scan-based experiment, we used an early version of our code generation framework, which we describe in detail in Chapter 6. Additionally, in Chapter 6, we present the benefit of the code generation framework over AsterixDB's original query execution model.

Scan-based Queries:

Figure 5.9 shows the execution time for the three queries (summarized in Table 5.2) executed against the *cell* dataset using the four different layouts. The execution times for Q1 against different layouts correlated with their storage sizes shown in Figure 5.7, except for the *AMAX* layout, where Q1 took the least time to execute — about 88% faster than the

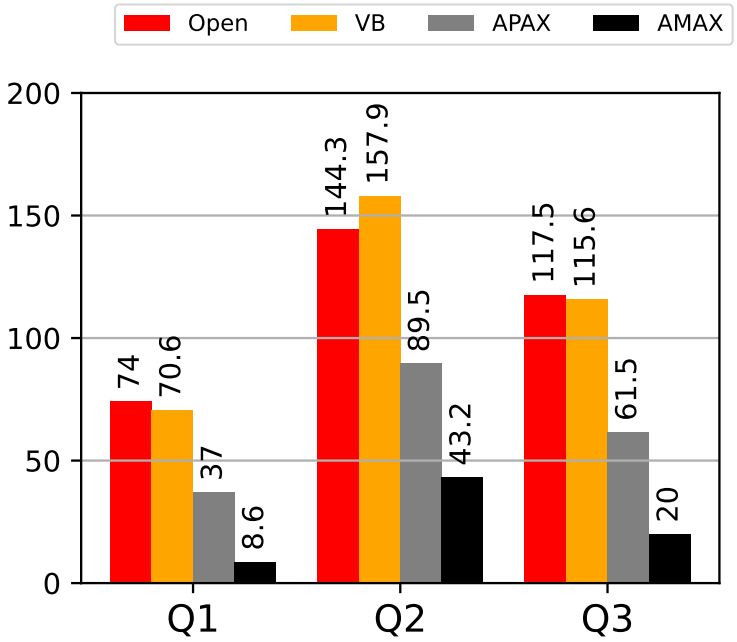


Figure 5.9: Query execution times of the *cell* dataset (Seconds)

Open and VB layouts. 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. This is also true for Q1 against the other datasets, as discussed in the following sections. 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, the execution times for Q2 showed a similar trend as in Q1. Querying the $APAX$ and $AMAX$ formats were 38% and 70% faster than the *Open* layout, and 40% and 72% than the VB layout, respectively. The slowdown of querying VB is due to how the fields' values are linearly accessed, as detailed in [36]. Q3 also shows a similar trend as Q1, where the I/O costs of the two columnar layouts were the smallest.

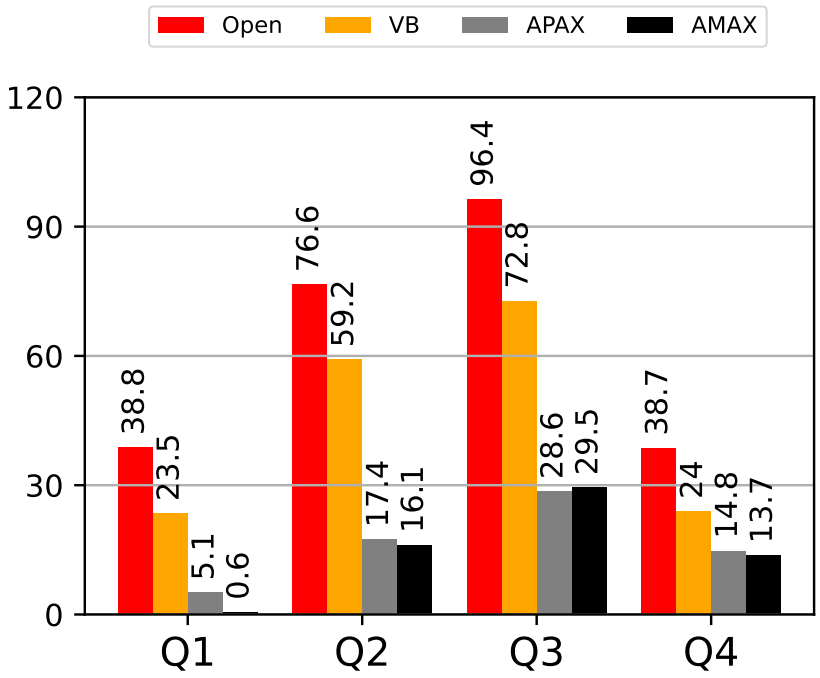


Figure 5.10: Query execution times of the *sensors* dataset (Seconds)

Querying the *sensors* dataset shows similar trends as in the *cell* dataset, where the queries' execution times (Figure 5.10) correlated with the formats' storage sizes (Figure 5.7). Executing Q1 against the $AMAX$ layout took 0.65 seconds, and it took 5.1 seconds for $APAX$. As in the *cell* dataset, Q1 only read Page 0 of $AMAX$, and hence, it was the fastest. For Q2 – Q4 (summarized in Table 5.2), the execution times for the $APAX$ and $AMAX$ records were comparable, as they took 7.7GB and 6.5GB to store the data, respectively, which is less than the 10GB of memory allocated for the system's buffer cache. Thus, AsterixDB was able to cache the $APAX$ and $AMAX$ records in memory and eliminate the I/O cost. For the two row-major layouts, it was faster to execute the queries against VB than *Open*, as VB took less space.

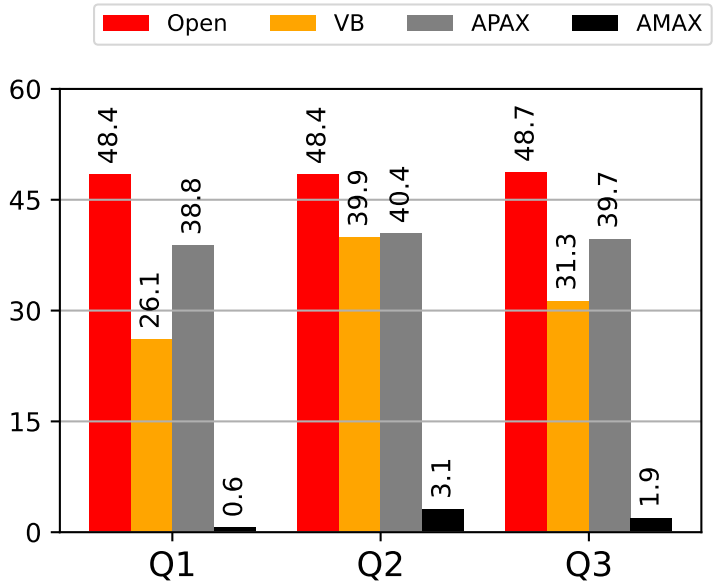


Figure 5.11: Query execution times of the *tweets_1* dataset (Seconds)

For *tweet_1*'s queries (Table 5.2), 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 for the $AMAX$ layout improved their execution times significantly. For example, Q1 took only 0.6 seconds to execute against the $AMAX$ format compared to 48.4, 26.1, and 38.8 seconds for *Open*, VB, and $APAX$, respectively. For Q2, $AMAX$ took 3.1 seconds to execute vs. 48.5, 39.9, and 40.3 seconds for *Open*, VB, and $APAX$, respectively. Storing and querying the *tweet_1* dataset using $APAX$ showed less improvement than for the *sensors* dataset. Excluding the $AMAX$ layout, the VB layout, in comparison, was more suitable for storing and querying a text-heavy Twitter dataset, as its records took less space to store and less time to query.

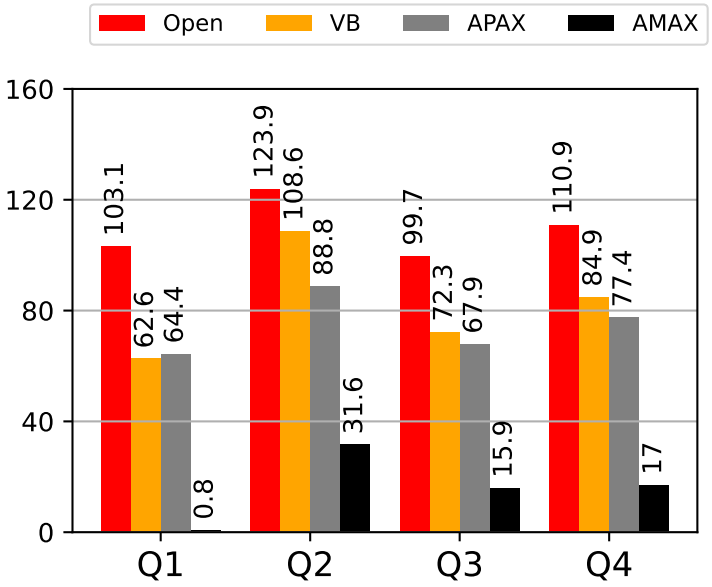
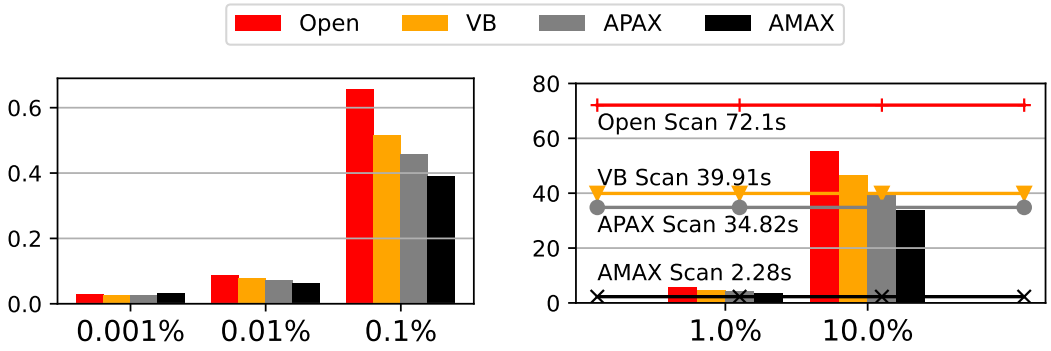


Figure 5.12: Query execution times of the *wos* dataset

The *wos* dataset is the last one used to evaluate scan-based queries. As mentioned earlier in Section 5.4.1, 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 5.2) 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. Figure 5.12 shows the execution times for Q1 - Q4, where $AMAX$ was the fastest to query. Q1 took only 0.83 seconds to execute, compared to 103.1, 62.5, and 64.4 seconds for *Open*, VB, and $APAX$, respectively. For Q2 - Q4, $AMAX$ improved their execution times by at least 64% compared to the other layouts. The queries' execution times against $APAX$ were slightly shorter than the VB layout. Thus, both the $APAX$ and $AMAX$ layouts can efficiently handle values with heterogeneous types, and the impact of mixed types on query performance was negligible.



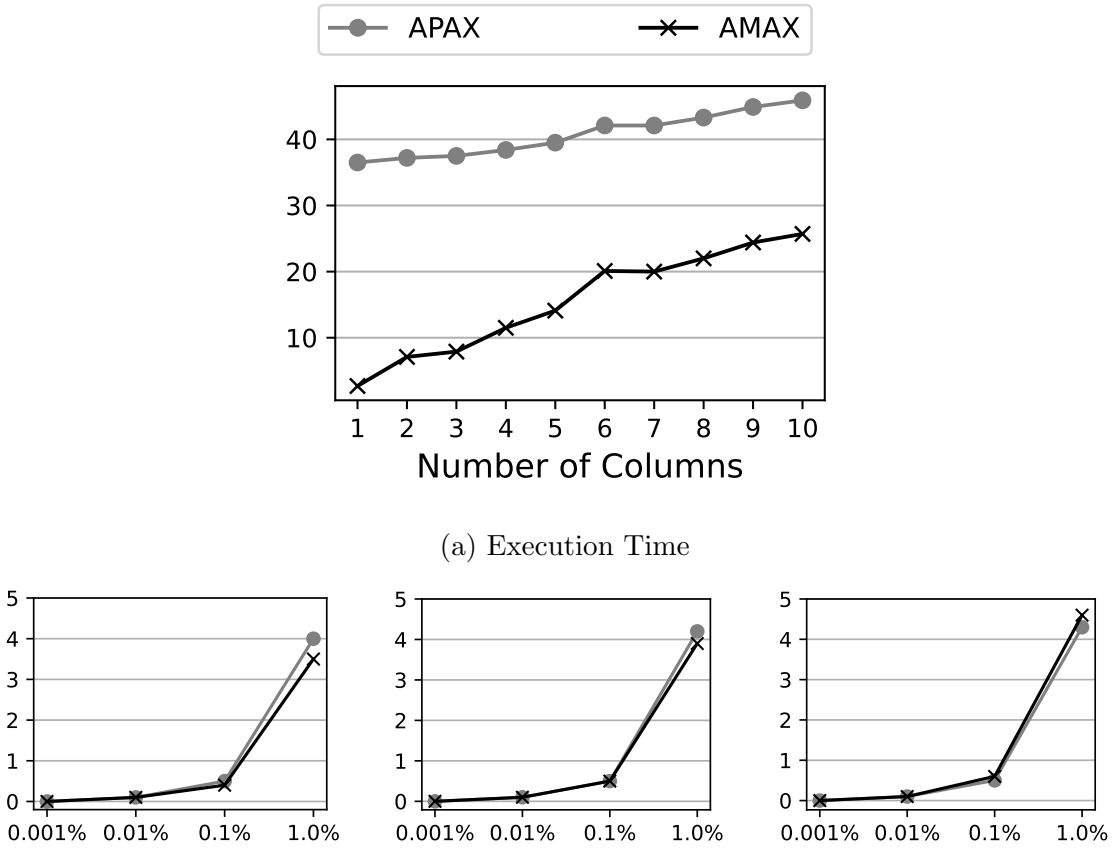
(a) Low selectivity

(b) High selectivity

Figure 5.13: Query with secondary index

Index-based Queries

We used the *tweet_2* dataset to evaluate the impact of secondary indexes on query performance for the four different layouts. We used a created *timestamp* secondary index to run range-queries with different selectivities that count the number of records. For each query selectivity, we executed queries with different range predicates to measure their actual I/O cost and report the average execution time. Figure 5.13 shows the execution times for both low and high selectivity predicates. For queries with low selectivity predicates, their execution times using the four different layouts were comparable, as shown in Figure 5.13a), and all queries took less than a second to finish. However, the execution times for queries that are 0.1% selective were correlated with storage sizes (Figure 5.7). Figure 5.13b shows the execution times for queries with high selectivity predicates both with and without utilizing the *timestamp* index. The secondary index accelerated the execution of queries with high selectivity predicates, except for the $AMAX$ layout. We observed that the scan-based query in the $AMAX$ layout (AMAX Scan) was faster to execute than its index-based queries.



(a) Execution Time

(b) 1 Column

(c) 2 Columns

(d) 10 Columns

Figure 5.14: Impact of accessing different number of columns: (a) for scan-based, and (b) - (d) for index-based queries

# of Columns	Scan	Index (Selectivity %)			
		0.001%	**0.01%**	**0.1%**	**1.0%**
1	2.080	0.021	0.066	0.405	3.487
2	7.083	0.026	0.067	0.424	3.711
3	7.940	0.028	0.068	0.436	3.487
4	11.477	0.029	0.074	0.442	3.487
5	14.133	0.033	0.079	0.456	3.917
6	20.005	0.035	0.085	0.503	4.120
7	20.815	0.038	0.089	0.508	4.301
8	21.987	0.041	0.092	0.538	4.420
9	24.422	0.047	0.093	0.551	4.600
10	25.710	0.047	0.097	0.553	4.636

Table 5.3: Scan-based vs. Index-based queries' execution times

The previous experiment does not depict a full picture, as counting the number of records only accesses Page 0 of $AMAX$ and skips the rest of the pages. The benefit of using secondary indexes for such queries becomes more apparent when a query accesses more columns. Figure 5.14 shows the impact of running queries that read different number of columns in the $APAX$ and $AMAX$ formats. Each query counts the appearances of different columns' values (i.e., non-NULL values) and varies the number of columns accessed from 1 to 10. The columns were picked at random and varied in terms of their types and sizes. Figure 5.14a shows the execution times for scan-based queries that access different number of columns. As expected, accessing more columns in the $AMAX$ format negatively impacts query performance, while the performance was relatively stable in the $APAX$. For example, reading ten different columns was 9.5X slower than reading a single column for the $AMAX$ layout, whereas the impact was less noticeable in $APAX$. Despite the slowdown, querying records in the $AMAX$ layout was still faster than $APAX$. The time variance in accessing different columns shown in Figure 5.14a is due to the time needed for reading and decoding values with different sizes. For example, the rate of change for reading from five to six columns was higher than for six to seven columns, as the sixth column contained required values, while the seventh column's values were mostly NULLs. That was for the scan-based queries. Figures 5.14b – 5.14d show the execution times of index-based queries with different selectivities (0.001% – 1.0%). The execution times for all queries were comparable for both layouts, despite the number of columns each query reads. Compared to the scan-based queries, the index-based queries took less time to execute and were less sensitive w.r.t the number of columns – Table 5.3 shows the execution times for the scan-based vs. index-based queries in the $AMAX$ format. Thus, as for row-major layout, 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.

5.5 Related Work

In Chapter 3, we have generally discussed the recent research on storing data in columnar formats. Here we mainly focus on recent work more directly related to columnar formats with dynamic schemas and LSM-based column stores.

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 [43], 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 [40] 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.

For LSM-based document stores, Rockset [25] supports storing values of semi-structured records in a columnar format, with the values of a column being stored in RocksDB [24] (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 only reads 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 [89] and Microsoft SQL Server's column store [67, 66] employ an LSM-like mechanism, while column-store systems such as Apache Kudu [8] and ClickHouse [16] are

LSM-based. This work is no exception, as we share similar objectives. Again, however, our focus is on nested and schemaless data.

5.6 Conclusion

In this chapter, 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 APAX and AMAX, two columnar layouts for organizing and storing records in LSM-based document stores. Furthermore, we highlighted the challenges involved reading and writing records in the APAX and AMAX layouts and proposed solutions to overcome those challenges. Experiments showed that the AMAX layout significantly reduced the overall storage overhead compared to the row-major formats, while the APAX format had mixed results depending on the number of columns in the dataset. The impact of transforming records into columns during data ingestion varied according to the structure of the ingested records. It was seen that the AMAX layout's ingestion rate was relatively stable compared to APAX and was faster compared to AsterixDB's current schemaless format.

To the best of our knowledge, most column store databases, except Vector [1], do not support secondary indexes, as scan-based queries are often considered good enough for data warehouse workloads. In this work, we also evaluated the impact of secondary indexes on the data ingestion and query performance of columnar formats and showed that the ingestion rate might be negatively impacted; however, the impact of reading multiple columns in AMAX was reduced when answering queries with secondary indexes.

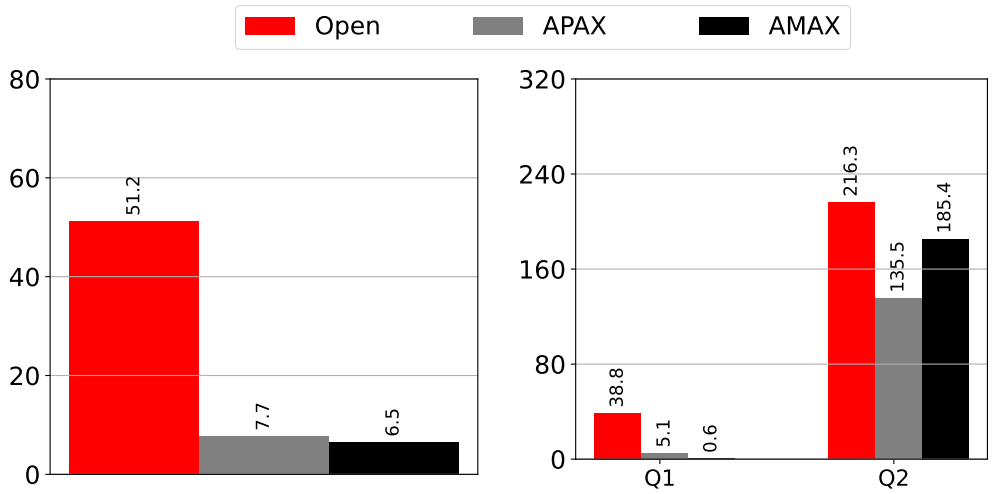
Chapter 6

A Code Generation Framework for Apache AsterixDB

6.1 Introduction

The goal of continuing to reduce the I/O cost in disk-based databases is objectively justified (and is a focus of this dissertation). However, with the ever-growing advancements in storage technologies, the CPU cost for query processing becomes even more apparent, especially for analytical workloads. The dominant factor determining the CPU cost is the query execution model. As described earlier in Chapter 1, modern DBMSs have moved away from using the traditional iterator model [69, 60] (also known as Volcano-style processing) to use other execution models (such as the batch model [81] and the materialization model [73]) to minimize the CPU overhead.

To reduce CPU and networking costs, Apache AsterixDB employs the batch execution model – explained in Chapter 2 – instead of the iterator model. However, in our early evaluation in Chapter 5, we observed an interesting phenomenon in certain types of datasets and analytical



(a) Storage Size (GB) (b) Query Execiton Time (Seconds)

Figure 6.1: Storage size and query execution times for the *sensors* dataset

workloads. We used an IoT-like dataset (called the *sensors* dataset) to evaluate the benefit of storing such data in columnar formats – namely, in the APAX and AMAX formats from Chapter 5. We first observed that both the APAX and AMAX formats reduced the overall storage size by at least 7x compared to AsterixDB's original format [5] (referred to as Open), as shown in Figure 6.1a. Despite the storage savings of the APAX and AMAX formats, however, the query execution times for Q1 and Q2, shown in Figure 6.1b, varied drastically. Q1, which counts the number of tuples, took 5 seconds in the APAX format and 0.6 seconds in the AMAX format. However, Q2, which computes the maximum recorded temperature, took at least 27X more time to execute than Q1 in the APAX and AMAX formats. After profiling, we found that the CPU cost of AsterixDB's query execution engine eclipsed the I/O savings when querying the *sensors* dataset in the APAX and AMAX formats.

Several factors contributed to the high CPU cost, which we address in detail in this chapter. Some of those factors are well-known in the relational model [77, 94], such as the materialization cost in the batch model. In [77], Neumann discussed those CPU costs 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 oper-

ators. For instance, operators such as $SCAN$, $SELECT$, and $PROJECT$ are fused and replaced with generated code, eliminating the fused operators' materialization costs. Thus, code generation and query compilation have become major contributors to the performance gains of many data systems [7, 75, 84].

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 only known at runtime, and they may change from one tuple to another – making the code generation process trickier. Luckily, the issue of handling dynamically-typed values at runtime has been addressed in work on popular dynamically-typed languages such as Python. For example, PyPy [23] and ZipPy [92] both provide a Just-in-Time (JIT) compiler for Python, which outperforms Python's original interpreter by orders of magnitude in some cases [85]. Inspired by those results, we have selected the Oracle Truffle framework (called Truffle hereafter) [93], a framework for implementing dynamically typed languages (e.g., Python), to implement an internal language that we will use for our proposed code generation framework.

In this chapter, we first discuss the CPU costs imposed by the batch model. Then, we shed light on the possibility of using query compilation techniques for document stores – where the value types are unknown until runtime – as a solution to address the batch model's CPU cost. Next, we revisit the current aggregation framework in Apache AsterixDB and propose several optimizations to further improve the performance of aggregate queries. Finally, we evaluate our proposed techniques in the context of Apache AsterixDB.

6.2 Background

In this section, we first review the batch query execution model and show the CPU costs associated with this model. Next, to prepare the reader, we give an overview of the Truffle framework, which we utilize to implement an internal dynamically-typed language for our proposed code generation framework.

6.2.1 Apache AsterixDB's Query Execution Model

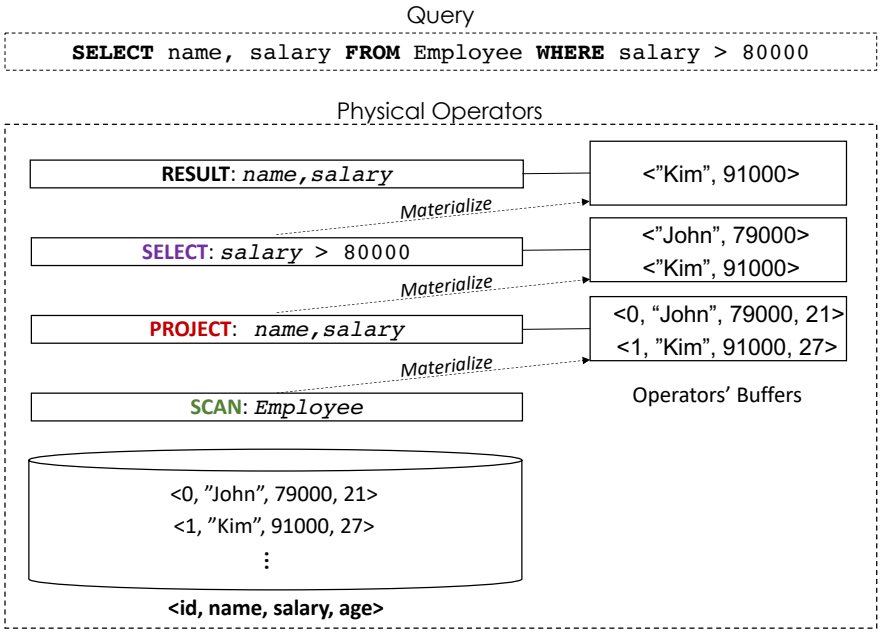


Figure 6.2: Running a query using the batch execution model

Section 2.2 gave an overview of the vectorized query execution model. Here, we provide more details of AsterixDB's vectorized execution model.

In AsterixDB, each operator receives a batch of records, then processes it and materializes the results from the processed batch to the next operator. Figure 6.2 illustrates an example of a batch of records being processed by different operators when executing a query. The chain of operators starts with the `SCAN` operator, which fetches the records stored in the

Employee collection. The SCAN operator produces a batch of records (two records in this case) and materializes it to the PROJECT's operator buffer. The PROJECT operator then produces another batch that contains only the name and salary fields and projects out the other fields. The resulting batch of the PROJECT operator is then materialized to the SELECT's operator buffer, which filters out records that do not satisfy the condition $salary > 80,000$. The resulting batch from the SELECT operator is then delivered to the user as a part of the final query result.

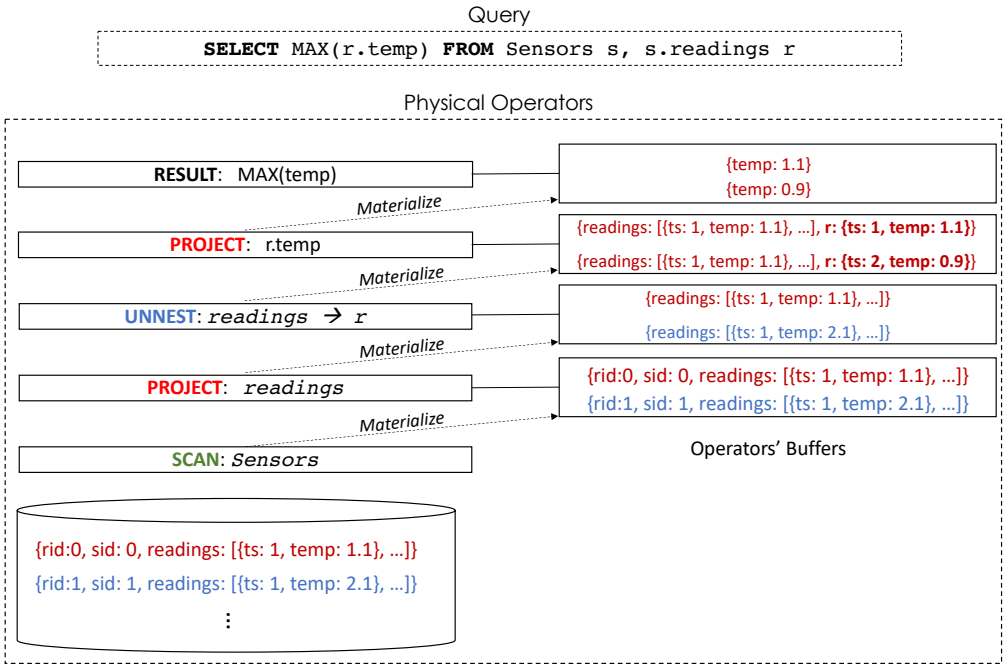


Figure 6.3: Running Q2 from Figure 6.1b using the batch execution model

The materialization cost between the operators in the batch execution model grows as the number of operators grows. In the document model, additional CPU-intensive operators are needed to navigate the nested structure of the stored documents. To illustrate, let us consider the earlier *sensors* example. Figure 6.3 shows both Q2 and its physical operators. When executing the query, the SCAN operator first fetches the records of the *sensors* collection and produces a batch for the PROJECT operator. The PROJECT operator then produces a batch that contains only the *readings* array. Then, the UNNEST operator flattens and "joins" every element of the array *readings* in the tuple with the tuple itself. In Figure 6.3, the

first resulting batch from the `UNNEST` operator contains only the values from the first record (colored as red) – assuming for illustration that each operator's buffer fits only two records. The resulting batch from the `UNNEST` is then materialized to another `PROJECT` operator, which only projects the *temp* values. Finally, the resulting values are aggregated to compute the final maximum recorded temperature.

Looking at the execution times shown in Figure 6.1b, it turns out that the `UNNEST` operator is one of the significant contributors to the high CPU in Q2. More specifically, in the *sensors* dataset, each *readings* array consists of 144 {`ts:` .., `temp:` ..} pairs. As a result, the total number of tuples resulting from the `UNNEST` operator is 144X larger than its input cardinality.

In Figure 6.1b, executing Q2 was significantly slower than executing Q1, whether the data was stored as rows or columns. However, querying records stored in a row format has an advantage over querying records stored in columnar formats in Apache AsterixDB, as its query execution engine was originally designed to operate on records in a row format. When the data is stored in a columnar format, then, the columns have to be reassembled back as rows, which incurs an additional CPU overhead — making querying records in a columnar format sometimes slower than records in a row format. One could consider changing AsterixDB's query execution engine to operate on columnar values natively as a solution, where the final result is reassembled (and usually consists of a few values) at the last operator instead of doing it eagerly at the `SCAN` operator. However, changing AsterixDB's query execution engine to support columnar formats natively would be a laborious task and might not yield better performance. Additionally, such a change would not address the cost of the `UNNEST` operator nor address the materialization cost of the batch execution model.

```
1  distribute result [$$45]
2  -- DISTRIBUTE_RESULT
3    exchange
4    -- ONE_TO_ONE_EXCHANGE
5      aggregate [$$45] <- [agg-global-sql-max($$48)]
6      -- AGGREGATE
7        exchange
8        -- RANDOM_MERGE_EXCHANGE
9          aggregate [$$48] <- [agg-local-sql-max($$41)]
10         -- AGGREGATE
11           project ([$$41])
12           -- STREAM_PROJECT
13             assign [$$41] <- [$$r.getField("temp")]
14             -- ASSIGN
15               project ([$$r])
16               -- STREAM_PROJECT
17                 unnest $$r <- scan-collection($$46)
18                 -- UNNEST
19                   project ([$$46])
20                   -- STREAM_PROJECT
21                     assign [$$46] <- [$$s.getField("readings")]
22                     -- ASSIGN
23                       project ([$$s])
24                       -- STREAM_PROJECT
25                         exchange
26                         -- ONE_TO_ONE_EXCHANGE
27                           data-scan []<-[$$44, $$s] <- IoT.Sensors: {readings:[{temp:any}]}
28                           -- DATASOURCE_SCAN
```

Figure 6.4: The optimized query plan of Q2 from Figure 6.3

6.2.2 Apache AsterixDB Query Optimizer

When a user submits a query for execution, AsterixDB first translates the submitted query into a logical query plan consisting of operators and connectors. The logical query plan is then optimized by applying several rewrite rules such as optimized field access and projection pushdown. Figure 6.4 shows the optimized query plan for Q2 shown in Figure 6.3. Clearly, the actual query plan consists of more operators than what Figure 6.3 showed. For instance, the *ASSIGN* operator (line 21) is used to access objects' fields as in the expression `$$s.getField("readings")`. However, the flows are similar in both versions – i.e., the simplified version in Figure 6.3 and the complete version in Figure 6.4.

Figure 6.4 also shows the use of two types of connectors (or exchange operators), namely `ONE_TO_ONE_EXCHANGE` (line 26) and a `RANDOM_MERGE_EXCHANGE` (line 8). Connectors are responsible for forwarding the records from one operator to another operator, either within a single partition (i.e., `ONE_TO_ONE_EXCHANGE`) or between partitions through the network

(e.g., `RANDOM_MERGE_EXCHANGE`) for parallel execution in AsterixDB. The query plan in Figure 6.4 shows that the two aggregate operators (in line 9 and line 5) are connected by a `RANDOM_MERGE_EXCHANGE` (line 8). The first aggregate in the plan computes the local maximum temperature (`temp` field from line 13) in each partition in a parallel fashion. Then the local maximums from all partitions are forwarded to a single partition (picked at random; hence the use of the connector `RANDOM_MERGE_EXCHANGE`) to compute the global maximum temperature in the second aggregate operator.

In addition to producing an optimized query plan, AsterixDB's query optimizer also computes "the requested schema". The query plan in Figure 6.4 shows the requested schema {`readings:`[{`temp:any`}]} (line 27) from the `Sensors`' collection. This computed schema is crucial when querying data stored in a columnar format for two reasons. First, the computed schema is used to determine which columns to read from the disk. Second, it provides the final expected structure of the records when reassembling the columns back into a row-oriented form.

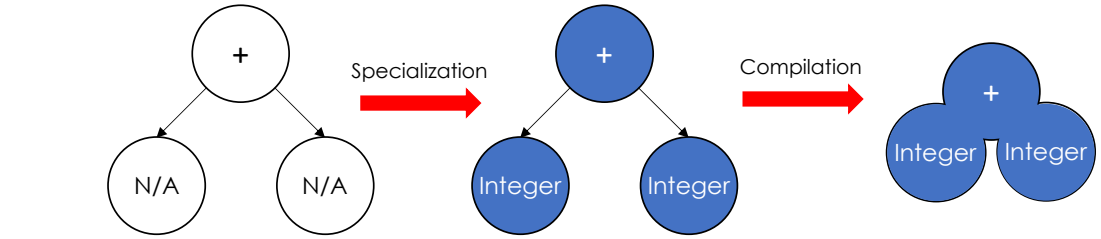
6.2.3 Truffle

Truffle is a framework for implementing dynamically typed languages, such as Python, JavaScript, and R, in order to run more efficiently (using a Java Virtual Machine or JVM) as compared to their original interpreters. To implement a new language using Truffle, one needs to write a language parser, which produces an Abstract Syntax Trees (AST). 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., an if statement). Additionally, the language implementer needs to specify the expected behavior for each expression given its inputs. For example, Figure 6.5 shows the implementation of `AddNode` of a language's AST nodes. The `AddNode` operation can either add two integers (`addInteger`) or two doubles (`addDoubles`).

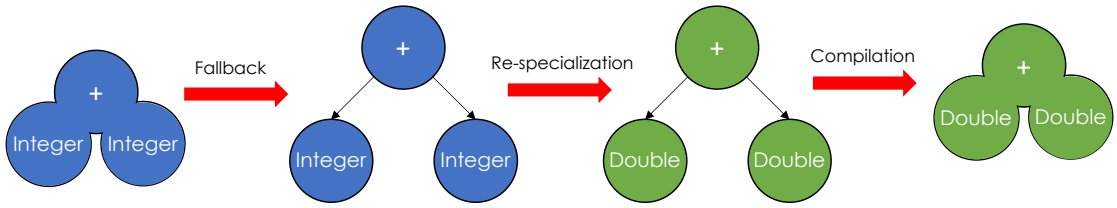
```
public abstract class AddNode extends BinaryNode {

    @Specialization
    protected int addInteger(int left, int right) {
        return left + right;
    }

    @Specialization
    protected double addDouble(double left, double right) {
        return left + right;
    }
}
```

Figure 6.5: Implementing the numerical add operation in Truffle

The two Java annotations `@Specialization` point Truffle to the two semantics of adding two values – whether integers or doubles.



(a) Initial specialization



(b) Re-specialization on type change

Figure 6.6: Truffle steps for specialization and re-specialization from [93]

On execution, Truffle accepts the constructed AST and starts evaluating the provided AST in an interpreted mode, during which Truffle rewrites the nodes by observing the input types of each node. For example, if the `AddNode` in Figure 6.5 executes `addInteger` for a few iterations, then, Truffle "specializes" the AST as if the two inputs are always integers. Next, Truffle compiles the generated specialized AST into a Java bytecode, where the generated bytecode is optimized further to machine code. In the case of type changes, Truffle falls back to the interpreter mode (i.e., going back to the first specialized AST). Truffle then

"re-specializes" the AST and recompiles it to a bytecode. Figure 6.6 illustrates the steps of the initial specialization and the re-specialization during an observed type change.

The Truffle optimizer does those specialization, re-specialization, and compilation steps automatically. Therefore, language implementers can focus on the semantics of their languages, while it is up to Truffle to optimize their executions. For this reason, we chose to use the Truffle framework in this work to implement an internal language for Apache AsterixDB's query code generation framework.

6.3 AsterixDB Internal Language (AIL)

```
function sumNum(arg, length) {
    total = 0;
    i = 0;
    while(i < length) {
        if(arg[i] > 100) {
            total = total + arg[i];
        }
        i = i + 1;
    }
    return total;
}
```

Figure 6.7: An example of a program written in AsterixDB Internal Language (AIL)

In this section, we first show some of the features and semantics of AsterixDB Internal Language (AIL)[1] – a Truffle language used in our code generation framework. As in any programming language, AIL provides control flow statements (such as if-statement and while-statement) and expressions (such as arithmetic addition and multiplication). Figure 6.7 shows a function that sums all the values of the array `arg` that are greater than `100`.

Since AIL is intended to replace SQL++ operators and expressions, AIL's semantics match AsterixDB's SQL++ semantics [4, 52], which are different in certain cases from usual pro-

[1] AIL is a fork from SL `https://github.com/graalvm/simplelanguage`

gramming languages. For example, in Figure 6.7, in the expression `arg[i] > 100`, the `arg[i]` value's type could be either an integer, a double, a string, object, array, or even NULL. For the two numerical types, integer and double, the expression would return `true` if the integer or the double values are greater than 100, and `false` otherwise – as in most programming languages. However, in AsterixDB SQL++ specification, comparing two incomparable types yields a `NULL` value and an issued warning – telling the user is comparing two incomparable types. Thus, in our example, if the type of `arg[i]` is a string, the result of the comparison `arg[i] > 100` should be `NULL` in AIL. However, since the expression `arg[i] > 100` is the condition of the if-statement, then the resulting `NULL` has to be evaluated as a boolean value, whether `true` or `false`. Since the NULL (resulting from comparing two incomparable types) is not a boolean value, in the WHERE-clause, it can be seen as non-true. In this setting, it can be seen physically as equivalent to false. Thus, in this case, AIL converts all non-boolean values of the if-statement's condition to `false`.

```
public abstract class AILGreaterThanNode extends AILBinaryNode {
    @Specialization
    protected boolean greaterThan(long left, long right) {
        return left > right;
    }

    //Comparing long and string (incompatible)
    @Specialization
    protected Object greaterThan(long left, AILStringRuntime right) {
        //Return NULL: incompatible
        return AILNullRuntime.INSTANCE;
    }
    //Other specializations...
}
```

Figure 6.8: A snippet from `AILGreaterThanNode` implementation

Figures 6.8, 6.9, and 6.10 show snippets from `AILGreaterThanNode`, `AILToBooleanNode`, and `AILIfNode` implementations in AIL, respectively. In `AILGreaterThanNode`, the incompatible comparison between an integer and a string returns `NULL`. `AILToBooleanNode` evaluates all boolean expressions as booleans and non-boolean expressions as `false`. Finally, in `AILIfNode`, the condition node is wrapped with the `AILToBooleanNode`, ensuring that the condition expression is always converted to a proper boolean value.

```
public abstract class AILToBooleanNode extends AILExpressionNode {
    @Specialization
    protected boolean fromBoolean(boolean value) {
        return value;
    }

    @Specialization(limit = "LIMIT")
    public static boolean fromGeneric(Object value,
        @CachedLibrary("value") InteropLibrary interop) {
        if (interop.isBoolean(value)) {
            //value is boolean --> return the boolean value
            return interop.asBoolean(value);
        }
        //Non-booleans return false
        return false;
    }
}
```

Figure 6.9: A snippet from `AILToBooleanNode` implementation

```
public final class AILIfNode extends AILStatementNode {
    //Members...

    //Constructor
    public AILIfNode(AILExpressionNode conditionNode,
        AILStatementNode thenPartNode, AILStatementNode elsePartNode) {
        //Wrap the condition expression with AILToBooleanNode
        this.conditionNode = AILToBooleanNodeGen.create(conditionNode);
        this.thenPartNode = thenPartNode;
        this.elsePartNode = elsePartNode;
    }
    //...
}
```

Figure 6.10: A snippet from `AILIfNode` implementation

As in the snippets above, other AIL expressions similarly follow AsterixDB's semantics (e.g., adding an integer and a string yields a `NULL` and a warning). However, in some cases, different implementations of an expression are needed. Let's take the following SQL++ query to illustrate:

```
SELECT SUM(salary) FROM Employee
```

In this query, some of the `salary` values could be `NULL`s or even non-numerics (e.g., strings). The aggregate function `SUM()` in AsterixDB ignores non-numeric values and computes the sum for all numeric values. For this case, AIL provides a different version of the '`+`' operator called the `AggregateAdd` (using the symbol '`++`'), which ignores non-numeric operands, including `NULL` values. For example, the result of `1 ++ NULL` is `1` instead of being `NULL` (per SQL's "interesting" semantics) as it would be in the scalar add operator in the expression

`1 + NULL` (also per SQL's semantics).

In addition to matching the SQL++ semantics, AIL provides helper data structures such as Lists and Maps. Those helper data structures can also re-specialize in case of type changes. We adopted the same approach used in Graal Python[2] for implementing such data structures. Those data structures can be helpful when translating operators such as the `GROUP-BY` operator, as we discuss later in Section 6.5.

6.4 Code Generation

In Section 6.2.1, we detailed the workflow of the batch query execution model as well as pointing out the CPU costs associated with such a model. In this section, we first show the workflow of our proposed code generation framework – inspired by [77] – in Apache AsterixDB. Next, we analyze the potential reduction of CPU costs by adopting the proposed framework as compared to using Apache AsterixDB's original execution engine.

6.4.1 Code Generation Workflow

Before showing how the proposed solution in [77] can be used to address the CPU costs of the batch execution model, let us first show Q2's (from Figure 6.4) query plan after enabling code generation. Figure 6.11 shows the generated code attached to the `DATASOURCE_SCAN` operator. In the new query plan, the generated code replaces all operators between `RANDOM_MERGE_EXCHANGE` and `DATASOURCE_SCAN` from the query plan shown in Figure 6.4.

First, let us go through the generated code and show what it actually does. The code begins with the function `Sensors0Func`'s (named after the collection's name), which takes three

[2]E.g., Graal Python List: `https://github.com/oracle/graalpython/tree/master/graalpython/com.oracle.graal.python/src/com/oracle/graal/python/builtins/objects/list`

```
distribute result [$$45]
-- DISTRIBUTE_RESULT
  exchange
  -- ONE_TO_ONE_EXCHANGE
    aggregate [$$45] <- [agg-global-sql-max($$48)]
    -- AGGREGATE
      exchange
      -- RANDOM_MERGE_EXCHANGE
        data-scan [$$48]<-[$$44, $$s] <- IoT.Sensors code >>
        01| //reader0: {readings:[{temp:any}]}
        02| function Sensors0Func (cursor, resultWriter, reader0) {
        03|     var0 = NULL;
        04|     while (cursor.next()) {
        05|        reader0.next();
        06|        while (!reader0.isEndOfArray()) {
        07|           var1 = reader0.getValue();
        08|           var0 = var0 _max_ var1;
        09|           reader0.next();
        10|        }
        11|     }
        12|     append(resultWriter, var0);
        13|     flush(resultWriter);
        14| }
        -- DATASOURCE_SCAN
```

Figure 6.11: The query plan of Q2 from Figure 6.4 after enabling code generation

parameters: `cursor`, `resultWriter`, and `reader0`. The first parameter `cursor` is a tuple cursor over the `Sensors`' collection tuples. The second parameter `resultWriter` is the final result writer, which pushes the processed tuples' values resulting from executing the generated code to the global `AGGREGATE` operator through the connector `RANDOM_MERGE_EXCHANGE`. The last parameter, `reader0`, is a value accessor for the `Sensors` collection's tuples. Readers in the AIL language are agnostic of the storage format (i.e., row or columnar formats). For this query, `reader0` corresponds to the temperature values (the field `temp` in the `readings` array of objects) – the only requested values in Q2 as per the requested schema shown in Figure 6.4. As in line 01 in Figure 6.11, our code generator always adds comments that show the mapping of each reader and its corresponding schema, which describes the requested value. This mapping allows us to understand and debug the correctness of the generated code.

The function body begins by declaring the variable `var0`, initialized with the value `NULL`. At the end of the function's execution, `var0` will be holding the maximum recorded temperature, as expected from Q2. Next, the code loops through the `Sensors`' tuples by

calling `cursor.next()`. Then, `reader0` is advanced to prepare the first value by calling `reader0.next()`. Since the `readings` field is an array, the code iterates over `reader0`'s value by first calling `reader0.next()` (line 09) inside the while-loop body. To ensure that the loop (in line 06) stops, the loop condition checks if the end of the array is reached. At the beginning of the loop body (line 07), `var1` is declared with the initial value produced from calling `reader0.getValue()`, which retrieves the temperature value (or the value of field `temp`). As the loop iterates over the array items, the maximum recorded temperature is computed and stored in `var0` (line 08). Finally, in lines 12 and 13, the value of var0 is appended and flushed. Each partition in AsterixDB's cluster (Section 2.3) executes the generated code, which computes the maximum local value, in parallel. Computing the maximum temperature then goes as it did in the original plan – as was detailed in Section 6.2.2.

data-scan: IoT.Sensors: {readings:[{temp:any}]}
-- DATASOURCE_SCAN

```
01| //reader0: {readings:[{temp:any}]}
02| function Sensors0Func (cursor, resultWriter, reader0) {
03|   var0 = NULL;
04|   while (cursor.next()) {
05|     reader0.next();
06|     while (!reader0.isEndOfArray()) {
07|       var1 = reader0.getValue();
08|       var0 = var0 _max_ var1;
09|       reader0.next();
10|     }
11|   }
12|   append(resultWriter, var0);
13|   flush(resultWriter);
14| }
```

unnest $$r <- scan-collection($$46)
-- UNNEST

```
01| //reader0: {readings:[{temp:any}]}
02| function Sensors0Func (cursor, resultWriter, reader0) {
03|   var0 = NULL;
04|   while (cursor.next()) {
05|     reader0.next();
06|     while (!reader0.isEndOfArray()) {
07|       var1 = reader0.getValue();
08|       var0 = var0 _max_ var1;
09|       reader0.next();
10|     }
11|   }
12|   append(resultWriter, var0);
13|   flush(resultWriter);
14| }
```

aggregate [$$48] <- [agg-local-sql-max($$41)]
-- AGGREGATE

```
01| //reader0: {readings:[{temp:any}]}
02| function Sensors0Func (cursor, resultWriter, reader0) {
03|   var0 = NULL;
04|   while (cursor.next()) {
05|     reader0.next();
06|     while (!reader0.isEndOfArray()) {
07|       var1 = reader0.getValue();
08|       var0 = var0 _max_ var1;
09|       reader0.next();
10|     }
11|   }
12|   append(resultWriter, var0);
13|   flush(resultWriter);
14| }
```

Figure 6.12: Code generation steps for the three operators `DATASOURCE_SCAN`, `UNNEST`, and `AGGREGATE` from the query plan in Figure 6.4

Now, let us present the process of generating this code from the original query plan and see which operator contributes which part of the code. The generated code, shown in Figure 6.11, is produced by applying a rule that traverses the optimized plan, during which each operator contributes part of the generated code [77].

The code generation process starts when the code generation rewrite rule reaches the `DATASOURCE_SCAN` – the leaf of the query plan. Figure 6.12 shows contributed parts of the code by the three leading operators, `DATASOURCE_SCAN`, `UNNEST`, and `AGGREGATE`, from the original query plan in Figure 6.4. First, `DATASOURCE_SCAN` contributes the function's header

and the while-loop in line 04 – iterating over the collection's tuples. Next, the `UNNEST` operator contributes the calls `reader0.next()` in lines 05 and 09, as well as the while-loop in line 06. The logic here is to advance `reader0` first (line 05) and ensure that `reader0` corresponds to a non-empty array (line 06). Again, the cursor advances the reader at the end of the loop body (line 09) to iterate over the array's items. This iteration over the array resembles the logical operation performed by the `UNNEST` to get the array's items. Finally, the `AGGREGATE` operator contributed code first declares the variable var0 at the beginning of the function's body (line 03) and outside the cursor iteration loop (line 04) – ensuring that the final aggregate result is computed for every tuple and every element in the array `readings` of the `Sensors` collection. Since the `AGGREGATE` is the last operator, where the code generation stops, the value of the variable `var0` is appended and flushed as the local aggregate value (the local maximum temperature in this case) and the final result when executing this function.

The code generation framework is limited and currently does not support whole-stage code generation as in [77]. Thus, the framework currently stops when it encounters a non-local connector such as the `RANDOM_MERGE_EXCHANGE` connector in Figure 6.12. We plan to expand our code generation framework to overcome this limitation in the future in order to support code generation for later (upstream) operators as well.

6.4.2 Cost Analysis

In Section 6.2.1, we described the costs associated with the batch execution model, which could be categorized into three points: (i) the materialization cost between operators, (ii) the cost of operators such as the `UNNEST` operator in document stores, and (iii) the cost of eagerly reassembling columns back as records in columnar formats. The costs (i) and (ii) affect all query execution performance, whether the data is stored as rows or columns,

whereas (iii) is an additional cost that only affects columnar formats.

To illustrate how the proposed code generation technique addresses those costs, let us compare the two query plans in Figures 6.4 and 6.11. First, we see that the generated code in Figure 6.11 replaces the work of eight operators from the original plan shown in Figure 6.4 (line 9 – line 24). By eliminating those operators, the code generation technique eliminates the materialization costs of the replaced operators – addressing cost (i). Additionally, the generated code fuses the work of the `UNNEST` (line 17) and `AGGREGATE` (line 9) operators of the original query plan. By fusing the work of two operators, the generated code directly computes the aggregated value (the local maximum temperature) – eliminating the unnecessary "join" of the `UNNEST` operator (Section 6.2.1), which addresses cost (ii). Finally, the reader `reader0` in Figure 6.11 only accesses the values of the field `temp`, which are of the scalar type double (Figure 6.3). Consequently, the generated code avoids the cost of reassembling the array `readings` for columnar formats, which addresses (iii).

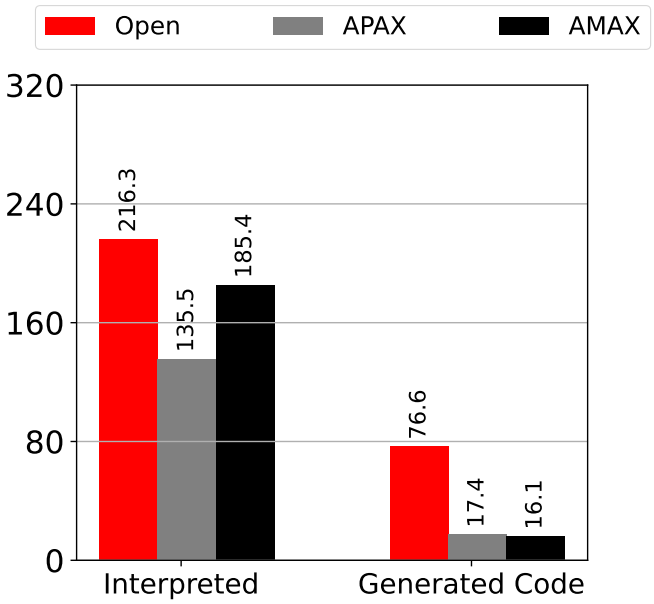


Figure 6.13: Interpreted vs. Generated Code: Q2 execution times (Seconds)

To empirically evaluate the practicality of reducing those costs, we re-executed query Q2 (Figure 6.3) after enabling the code generation. Figure 6.13 shows the execution times for

Q2 with and without the code generation for the three storage formats *Open*, *APAX*, and *AMAX*. The proposed code generation framework improved Q2's execution performance by ∼2.8X, ∼7.7X, and ∼11.5X for the *Open*, *APAX*, and *AMAX*, respectively. Those improvements in query execution performance confirm the impact of the three CPU costs (i) and (ii) for all three formats and (iii) for the columnar formats *APAX* and *AMAX*. The final results of our empirical evaluation of the proposed code generation framework will be presented later in Section 6.6.

6.5 Optimizing `GROUP-BY` Queries

```
SELECT sid, MAX(r.temp) max_temp
FROM Sensors s, s.readings r
GROUP BY s.sensor_id sid
ORDER BY max_temp DESC
LIMIT 10
```

Figure 6.14: Retrieving the top ten sensors that recorded the maximum temperature

Computing aggregates over a large volume of data is an integral part of analytical workloads. In previous sections, we discussed the approach used in AsterixDB for computing global aggregates (as in Q2 shown in Figure 6.1b). Besides global aggregates, computing aggregates per group (`GROUP-BY` queries) is also crucial for analytical workloads. Figure 6.14 shows an example of a query that finds the top ten sensors that recorded the ten maximum temperatures. Optimizing such `GROUP-BY` queries has been a focus in previous work [44, 46, 48, 58]. In this section, we propose several improvements (inspired by [46, 48, 58]) to Apache AsterixDB's aggregation framework in the context of our code generation framework.

6.5.1 Improving Parallel GROUP-BY I/O

In parallel databases, such as Apache AsterixDB, computing aggregates are done in two phases. In the first phase, each partition computes the aggregation locally. Then, in the second phase, the locally computed aggregate(s) are either merged by a single partition to compute a global aggregate (as explained in Section 6.2.2) or shuffled between the partitions to compute the final aggregate result for each group in GROUP-BY queries. To illustrate the process of processing GROUP-BY queries in AsterixDB, let us examine parts of the query plan (Figure 6.15) for the query shown in Figure 6.14. Notice that the two SORT_GROUP_BY operators (lines 17 and 8) correspond to the local and global aggregate operators, respectively. Additionally, AsterixDB's query optimizer picked the sort-based GROUP-BY algorithm (its default algorithm) instead of the alternative hash-based algorithm, which can be enabled by providing a hint[3].

When the query in Figure 6.14 is executed, each partition starts by scanning their corresponding Sensors records (line 38), unnesting the reading values (line 27), and projecting sensor_id and the temp value pairs (line 20). The projected sensor_id and temp value pairs are then pushed as inputs to the SORT_GROUP_BY operator, where sensor_id is the grouping key and the temp value is the input for the aggregate function MAX(). The received value pairs are then sorted by the grouping key (sensor_id in this query). Performing an external sort might be required if the total size of the produced value pairs (in bytes) exceeds the SORT_GROUP_BY memory budget. The sorted groups are then merged by computing their aggregate value (the maximum temperature in our query, as shown in line 12 in Figure 6.15). The final result of the local SORT_GROUP_BY operator (line 11) are the distinct groups with their aggregate values (i.e., the maximum temperature for each sensor_id in each partition) – which concluds the first phase. Since the data is partitioned by the records' primary keys when ingested, each partition may have a local maximum for the same sensor_id.

[3]Query Hints: https://nightlies.apache.org/asterixdb/sqlpp/manual.html#Query_hints

```
1  ...
2  group by ([$$sid := $$70]) decor ([]) {
3             aggregate [$$66] <- [agg_global_sql_max($$69)]
4             -- AGGREGATE
5             nested tuple source
6             -- NESTED_TUPLE_SOURCE
7          }
8  -- SORT_GROUP_BY[$$70] <-- GLOBAL
9      exchange
10     -- HASH_PARTITION_EXCHANGE [$$70]
11     group by ([$$70 := $$64]) decor ([]) {
12                aggregate [$$69] <- [agg_local_sql_max($$60)]
13                -- AGGREGATE
14                nested tuple source
15                -- NESTED_TUPLE_SOURCE
16            }
17     -- SORT_GROUP_BY[$$64] <-- LOCAL
18         exchange
19         -- ONE_TO_ONE_EXCHANGE
20         project ([$$60, $$64])
21         -- STREAM_PROJECT
22             assign [$$60] <- [$$r.getField("temp")]
23             -- ASSIGN
24             project ([$$64, $$r])
25             -- STREAM_PROJECT
26                 unnest $$r <- scan-collection($$67)
27                 -- UNNEST
28                 project ([$$64, $$67])
29                 -- STREAM_PROJECT
30                     assign [$$64, $$67] <- [$$s.getField("sensor_id"), $$s.getField("readings")]
31                     -- ASSIGN
32                     project ([$$s])
33                     -- STREAM_PROJECT
34                         exchange
35                         -- ONE_TO_ONE_EXCHANGE
36                         data-scan [$$65, $$s]<-[$$65, $$s] <- IoT.Sensors
37                             project-dataset ({sensor_id:any,readings:[{temp:any}]})
38                         -- DATASOURCE_SCAN
```

Figure 6.15: The optimized query plan for the query shown in Figure 6.14

Therefore, after computing the local maximums, the computed aggregates are then hash-partitioned (line 10) by the `sensor_id` to compute the final aggregate values with the same grouping key by a single partition – ensuring that the global maximum is computed in one place for each group. Similar to the first phase, the second `SORT_GROUP_BY` (line 8) computes each group's global aggregate value (the global maximum temperature in our query) – thus concluding the second and last phase of computing the `GROUP-BY` aggregate values.

The current approach to processing `GROUP-BY` queries in AsterixDB can be improved in several ways. First, we observed that using the sort-based algorithm may lead to unnecessary spilling to the disk. For example, when the number of distinct groups is small, using the hash-based algorithm with a sufficient memory budget would never spill to disk since the hash-

based algorithm only keeps the distinct groups in memory by performing early aggregation. However, the hash-based algorithm could suffer significantly if the number of distinct groups is too large to fit in memory – the sort-based algorithm would be superior in this case. As mentioned in [48], a mixed approach can be utilized to benefit from the best of the two algorithms. In the mixed approach, the hash-based algorithm is used until the hash-table can no longer fit more groups; then, the accumulated hash-table entries are sorted and written to disk as a sorted run. After the last spill, the sorted runs are merged, as in the sort-based algorithm, to compute the final aggregate result for each group. If the entire set of groups fits in the given memory budget, the early aggregation of the hash-based algorithm will not spill to the disk. If spilling to disk is needed, then the sorted runs at least will not contain any duplicate groups – making the spilled runs relatively smaller than the pure sort-based algorithm's runs would be.

Secondly, we observe that, in the local `GROUP-BY`, it is not actually necessary to compute the final local aggregate values for the entire set of records in each partition. Instead, the local `GROUP-BY` operator can ship partially computed aggregates to the final destination (the global aggregators) instead of spilling to disk, as the global aggregator would likely spill to disk as well. Doing this can alleviate pressure on the nodes' disks (a potentially pressured resource) since records are spilled once (at the global aggregator) instead of the two disk spillings in the original approach.

6.5.2 Optimizing `GROUP-BY` Queries with $UNNEST$

The query shown in Figure 6.14 computes the maximum temperature for each sensor. The grouping key `sensor_id` is a single scalar value, whereas the temperature values are repeated values produced by the `UNNEST` operator, as shown earlier in Figure 6.3. Hence, each record contains a single `sensor_id` value and 144 temperature values. One possible way to process

```
1  //reader0: {readings:[{temp:any}]}
2  //reader1: {sensor_id:any}
3  function Sensors0Func (cursor, resultWriter, reader0, reader1) {
4      var0 = newAggregator("MAX", 134217728, resultWriter);
5      while (cursor.next()) {
6          reader0.next();
7          reader1.next();
8          while (!reader0.isEndOfArray()) {
9              aggregate(var0, reader1.getValue(), reader0.getValue());
10             reader0.next();
11         }
12     }
13     append(resultWriter, var0);
14     flush(resultWriter);
15 }
```

Figure 6.16: Generated code for the query shown in Figure 6.14

this query in AIL is shown in Figure 6.16. The AIL code starts with two comments (lines 1 and 2) that describe the values produced by `reader0` and `reader1`, respectively, followed by the function header. In line 4, the AIL code allocates a new aggregator, which is a hash-based aggregator. The first argument of the function `newAggregator()` is the desired aggregate function, which is `MAX()`. The next parameter is the memory budget in bytes, which is 128MB. The last parameter is the `resultWriter`, which the aggregator uses to write the partial aggregate result to the global aggregator once the hash table's memory is full (as explained in the previous section). The rest of the code is similar to previously explained AIL-generated code, except for line 9. In line 9, we see the function `aggregate()`, which takes three parameters: (1) `var0`, which corresponds to the aggregator declared in line 4, (2) the value of `reader1`, which is the `sensor_id`, and (3) the value of `reader0`, which the temperature value. Here, the `aggregate()` function adds and computes the maximum temperature value for the grouping key (`sensor_id`) using the hash-based aggregator of `var0`. Finally, in lines 13 and 14, the stored aggregate values in `var0` are flushed to the global aggregator to compute the global aggregates.

The generated AIL code in Figure 6.16 could incur unnecessary overhead since the function `aggregate()` will compute the same hash code for `sensor_id` and may write to the hash

```
1  //reader0: {readings:[{temp:any}]}
2  //reader1: {sensor_id:any}
3  function Sensors0Func (cursor, resultWriter, reader0, reader1) {
4      var0 = newAggregator("MAX", 134217728, resultWriter);
5      while (cursor.next()) {
6          var1 = NULL;
7          reader0.next();
8          reader1.next();
9          while (!reader0.isEndOfArray()) {
10             var1 = var1 _max_ reader0.getValue();
11             reader0.next();
12         }
13         aggregate(var0, reader1.getValue(), var1);
14     }
15     append(resultWriter, var3);
16     flush(resultWriter);
17 }
```

Figure 6.17: An optimized version of the generated code in Figure 6.16

table repeatedly. More specifically, in this query, the same hash code will be computed 144 times (the size of the array `readings`) and may write a new maximum value in each call to `aggregate()`. Since the grouping key `sensor_id` is the same for the 144 temperature values in a record, an alternative approach (shown in Figure 6.17) eliminates these unnecessary repetitive operations. In this approach, in the loop body (lines 10 and 11), the maximum temperature is first computed and stored in `var1`. Then, in line 13, the key `sensor_id` and the computed maximum value are added once to the hash-based aggregator in `var0`. Thus, the aforementioned operations are performed once for each record instead of the 144 times in Figure 6.16.

```
SELECT ts, MAX(r.temp) max_temp
FROM Sensors s, s.readings r
GROUP BY r.ts ts
ORDER BY max_temp DESC
LIMIT 10
```

Figure 6.18: Retrieve the top ten timestamps with the maximum temperatures ever recorded

However, note that this optimization step is only possible if the grouping key did not originate from the `UNNEST` operator (e.g., as us true fo `sensor_id`). To illustrate, let us examine the query shown in Figure 6.18, which retrieves the top ten timestamps with the maximum

temperatures ever recorded. In this query, both the grouping key `ts` and the value `temp` originate from the unnested array `readings`. Therefore, each temperature value corresponds to a different timestamp. Consequently, each key-value pair must be added to the hash-based aggregator as shown in the generated AIL code in Figure 6.19.

```
1  //reader0: {readings:[{temp:any}]}
2  //reader1: {readings:[{ts:any}]}
3  function Sensors0Func (cursor, resultWriter, reader0, reader1) {
4      var0 = newAggregator("MAX", 134217728, resultWriter);
5      while (cursor.next()) {
6          reader0.next();
7          reader1.next();
8          while (!reader0.isEndOfArray()) {
9              aggregate(var0, reader1.getValue(), reader0.getValue());
10             reader0.next();
11             reader1.next();
12         }
13     }
14     append(resultWriter, var0);
15     flush(resultWriter);
16 }
```

Figure 6.19: Generated code for the query shown in Figure 6.18

6.5.3 Optimizing Top-K Queries with `MIN()` and `MAX()`

Most database systems process all `GROUP-BY` aggregate queries similarly even though different aggregate functions may have different traits. More specifically, top-K `GROUP-BY` queries, with the aggregate functions `MIN()` and `MAX()`, are unique and can be optimized differently as compared to other aggregate functions (e.g., `COUNT()`).

```
SELECT sid, COUNT(*) cnt
FROM Sensors s, s.readings r
GROUP BY s.sensor_id sid
ORDER BY cnt DESC
LIMIT 10
```

(a)

```
SELECT sid, MAX(r.temp) max_temp
FROM Sensors s, s.readings r
GROUP BY s.sensor_id sid
ORDER BY max_temp DESC
LIMIT 10
```

(b)

Figure 6.20: Comparing two queries: one with `COUNT()` and another with `MAX()`

To show their uniqueness, let us compare the two `GROUP-BY` queries shown in Figure 6.20. The query in Figure 6.20a counts the top ten sensors with the highest number of recorded readings, while the query in Figure 6.20b computes the top ten sensors with the maximum recorded temperatures. Both queries have the same grouping key `sensor_id`, and thus, both have the same number of distinct groups. In the first query, the system first computes the count for each group by incrementally "accumulating" the number of occurrences of readings for each sensor. In contrast, the maximum temperature is computed for a given grouping key in the second query by replacing the older maximum if the older maximum is a smaller number. Thus, the second query does not require any "accumulation" to compute the maximum value. Since the query only cares about the top ten sensors with the maximum temperature, it is enough to always keep only the top ten sensors with the highest temperature in memory without spilling to disk.

Algorithm 1 An algorithm to compute the maximum top-k aggregate values

```
1  procedure AddMaxTopK (map, key, value, k) {
2      //Scenario1
3      if(map.contains(key)) {
4          olderValue = map.get(key);
5          if(value > olderValue) {
6              map.put(key, value);
7              if(map.getMinimumKey() == key) {
8                  map.findAndSetNewMinimum();
9              }
10         }
11     }
12     //Scenario2
13     else if(map.size() < k) {
14         map.put(key, value);
15         if(map.size() == 1 || map.getMinimumValue() > value) {
16             map.setMinimum(key, value);
17         }
18     }
19     //Scenario3
20     else if(map.getMinimumValue() < value) {
21         map.removeMinimum();
22         map.put(key, value);
23         map.findAndSetNewMinimum();
24     }
25 }
```

To illustrate, Algorithm 1 shows a procedure that adds a new key-value pair to compute the

top-k maximum values. The procedure takes (1) `map`: a hash-map with additional procedures as we will see next, (2) `key`: the grouping key, (3) `value`: the value to be aggregated, and (4) `k`: an integer, which specifies the **K** of the top-**k** query. The procedure handles three scenarios: (i) `Scenario1` (lines 3-11), (ii) `Scenario2` (lines 13-18), and (iii) `Scenario3` (lines 20-24). In the first scenario, the grouping key already exists in `map`. In this case, the procedure replaces the older value (the value in `map`) if the older value is smaller than the new provided value (lines 4 and 5). If the older value is replaced, the procedure checks if the key corresponds to the minimum value stored in `map` (line 6). Since the older value was replaced, it may no longer be the minimum value stored in `map`, and hence, a search is performed to find the new minimum value (line 8). In the second scenario, `map` does not store the provided `key`; hence, the procedure checks whether the map contains the required number of values (i.e., `k`) in line 13. If not, then the new key-value pair is added to `map` (line 14). Lines 15-17 ensure that the key with the minimum value is recorded. In the last scenario, `map` already contains the required `k` values. However, the procedure checks whether the newly provided value is greater than the minimum value stored in `map` (line 20). If that is the case, the key and its corresponding previous minimum value are replaced by the newly provided pair, and a search is performed to find the new key with the minimum value (line 23). If none of the three scenarios' conditions are satisfied, that means the provided value is smaller than all `k` values stored in `map`, and the provided pair is ignored.

6.6 Experimental Evaluation

In this section, we evaluate and analyze the proposed code generation framework in Apache AsterixDB. At first, we conducted an experiment – similar to the one detailed in Section 5.4.4 from Chapter 5 – to compare the proposed code generation framework against AsterxiDB's original execution engine. The experiments in the previous chapter already only showed

execution times based on using the proposed code generation framework, which we detailed in this chapter. We used the code generation in the previous chapter as well because the CPU overhead of AsterixDB's original execution model obscured the impact of reducing the I/O costs when querying data in columnar formats, as we will show and explain later in this section. Furthermore, here we add additional queries with different aggregate functions to evaluate the techniques proposed in Section 6.5.

In a previous study [61], Apache AsterixDB was evaluated using the IRIS-HEP benchmark [18, 62], a benchmark that measures the suitability of systems for handling High-Energy Physics (HEP) data and analysis. That study showed that Apache AsterixDB was not practical for handling such workloads due to its inefficient interpreter-based execution engine. Therefore, we use the IRIS-HEP benchmark here to evaluate and measure the effectiveness of our proposed code generation framework for handling HEP workloads.

In a second experiment, we evaluate AsterixDB's current aggregation framework as well as the proposed improvements detailed in Section 6.5. In this experiment, we ran several `GROUP-BY` queries, where the number of distinct grouping keys varies in the different queries. Additionally, we varied the size of the dedicated memory budget allocated for the `GROUP-BY` operator. The objective here is to evaluate the behavior of the different approaches, namely (1) the sort-based approach, (2) the hash-based approach, and (3) the suggested mixed approach.

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 configured AsterixDB (v9.8.0) with a single node and eight partitions (Section 2.3). The eight partitions share 16GB of total allocated memory, and from this, we allocated 10GB for the system's buffer cache and 2GB for its in-memory component budget. The remaining 4GB is allocated for use as

temporary buffers for query operations such as sorting and grouping. The other 16GB of memory is kept for the operating system and for the automation tool we use to conduct the evaluation.

Since the main focus of this experiment is measuring the CPU cost/overhead, we used the columnar storage format *AMAX* and AsterixDB's page-level compression to minimize the overall I/O cost throughout our experiments.

Datasets In our evaluation, we used four different datasets that differ in terms of their records' structures, sizes, and value types. Table 6.1 lists and summarizes the traits of the four datasets. The first three datasets (namely *cell*, *sensors*, and *tweets*) are the same datasets used in evaluating the columnar formats *APAX* and *AMAX* proposed in Chapter 5. (The details of the three datasets were presented previously in Section 5.4.1.)

For the fourth dataset, the *iris_hep* dataset, each record (or tuple) represents an event recorded by the sensors of a particle collider. The structure of each record is a JSON document consisting of scalar values, object values, and several arrays of objects. The nested scalar values consist mainly of double values and some boolean and integer values.

	cell	*sensors*	*tweets*	*iris_hep* [62]
Type	Real	Synthetic	Real	Real
Size (GB)	172	212	210	158
# of Records	1.43B	40M	17M	53.4M
Dominant Type	Mix	Integer	String	Double

Table 6.1: Datasets summary

6.7 Evaluation Summary

We first compare AsterixDB's original query execution engine (referred to as `Interpreted`) against our proposed code generation framework (referred to as `Generated Code`) using

cell	Q1	The number of records
	Q2a	The top 10 callers with the longest call durations
	Q2b	The top 10 callers with the highest number of activities
	Q3	The number of calls with durations $\geq$ 600 seconds
sensors	Q1	The number of records
	Q2	The maximum reading ever recorded
	Q3a	The IDs of the top 10 sensors with maximum readings
	Q3b	The IDs of the top 10 sensors that have the maximum number of readings
	Q4a	Similar to Q3, but for readings in a given day
	Q4b	Similar to Q4, but for readings in a given day
tweets	Q1	The number of records
	Q2a	The top 10 users who posted the longest tweets
	Q2b	The top 10 users who posted the largest number of tweets
	Q3	The top 10 users with the highest number of tweets that contain a popular hashtag
iris_hep [62]	Q1	Plot the ${E_T}^{miss}$ of all events
	Q2	Plot the p_T of all jets
	Q3	Plot the p_T of jets with $\|\eta\| < 1$
	Q4	Plot the ${E_T}^{miss}$ of events that have at least two jets with $p_T > 40\ GeV$
	Q5	Plot the${E_T}^{miss}$ of events that have an opposite-charge *muon* pair with an invariant mass between 60 and 120 GeV

Table 6.2: Summary of the queries used in the evaluation

the four datasets *cell*, *sensors*, *tweets*, and *iris_hep*. We re-executed the same queries summarized in Section 5.4.4 and listed in Appendix B. We also included similar `GROUP-BY` queries (with the suffix `b`) but with different aggregate functions to evaluate the proposed techniques in Section 6.5. The SQL++ queries for the *iris_hep* dataset are listed in [18]. Table 6.2 summarizes the objectives of all queries for all four datasets. In our experiments, we ran each query five times and reported the average time for the last four.

6.7.1 *cell* Dataset

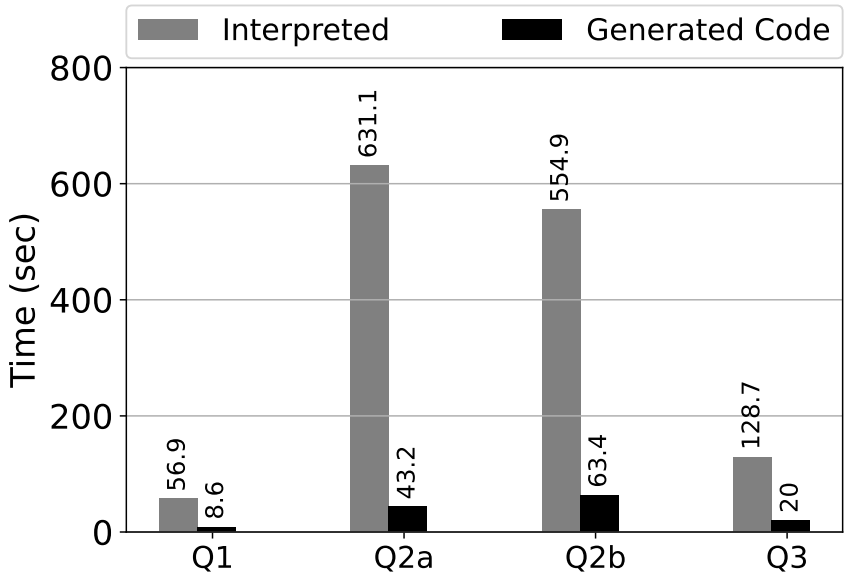


Figure 6.21: Query execution times for the *cell* dataset

In the *cell* dataset, all four queries took significantly less time to execute using our proposed code generation framework (`Generated Code`) than AsterixDB's original execution engine (`Interpreted`), as shown in Figure 6.21. For instance, with our proposed code generation framework (`Generated Code`), Q1 – a simple `SELECT COUNT(*)` query – took 8.6 seconds to execute compared to 56.9 seconds using AsterixDB's query execution engine (`Interpreted`). The 6.6X difference between the two approaches when executing Q1 is attributed to the number of records in the *cell* dataset, which is relatively high – 1.4 billion compared to the tens of millions in other datasets (Table 6.1). In the `Interpreted` mode, the number of memory copy operations used to construct frames that could be passed to the next operator was the main contributor to the higher execution time. In contrast, the `Generated Code` mode has eliminated the memory copy overhead by fusing the work of multiple operators into a single operator. The execution times for the other queries exhibited a similar trend. The `Generated Code` mode was ∼14.61X, ∼8.75X, and ∼6.4X faster to execute Q2a, Q2b, and Q3 compared to the `Interpreted` mode, respectively.

Note that Q2a and Q2b are similar (both are Top-K queries), and the only difference is the aggregate function – `MAX(duration)` in Q2a and `COUNT(*)` in Q2b. In the `Interpreted` mode, Q2b took less time to execute than Q2a, as Q2a incurred an additional I/O cost by reading the *duration* column to compute the maximum call's duration, whereas Q2b only reads the *caller_id*'s column. In contrast, in the `Generated Code` mode, Q2a took less time to execute since computing Top-K queries with the aggregate function `Max()` does not need to spill to disk, as opposed to computing Top-K queries with the `COUNT()` aggregate function (Section 6.5.3). Hence, the I/O cost of reading the *duration* column was negligible compared to the spilling I/O cost for the `GROUP-BY` operator.

6.7.2 *sensors* Dataset

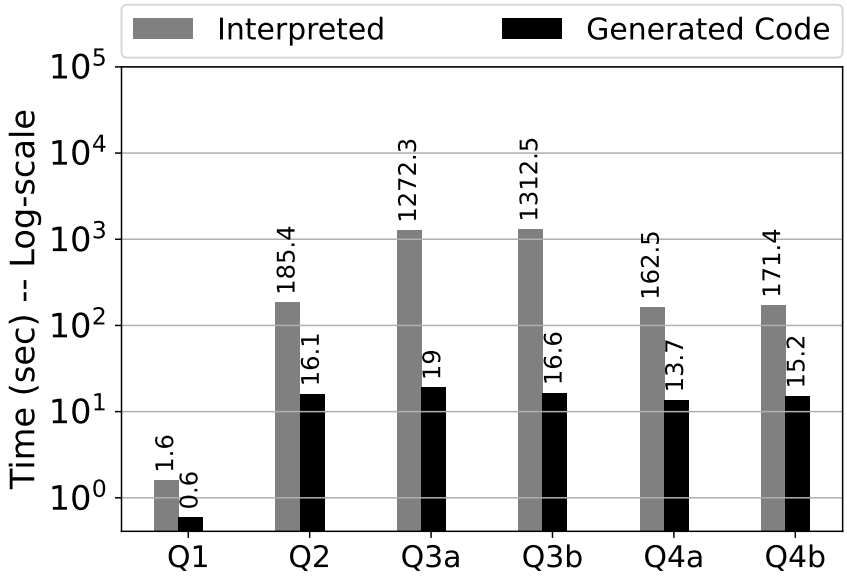


Figure 6.22: Query execution times for the *sensors* dataset

For the *sensors* dataset, Figure 6.22 (log-scale) shows that the `Generated Code` mode was at least an order of magnitude faster than the `Interpreted` mode except for Q1, where the `Generated Code` mode was only 2.66X faster. The main reasons for this gap between the two execution modes in query execution performance are the cost of the `UNNEST` operator and the

cost of assembling the columns back as records. In Section 6.4.2, we analyzed the associated costs in `Interpreted` mode for executing Q2 in details. The execution of the other queries in the `Interpreted` mode also incurred the same costs. For Q3a and Q3b, the difference between the two modes is even more significant (∼67X in Q2a and ∼79X in Q2b) because AsterixDB uses the sort-based algorithm by default. In contrast, the `Generated Code` mode employs a mixed approach (as detailed in Section 6.5.1), which enjoys the benefits of both the sort-based and hash-based algorithms. In AsterixDB, the user can provide a hint to use the hash-based algorithm. Later in Section 6.7.5, we will show the result of comparing the mixed approach against AsterixDB's different algorithms with different memory budgets.

6.7.3 *tweets* Dataset

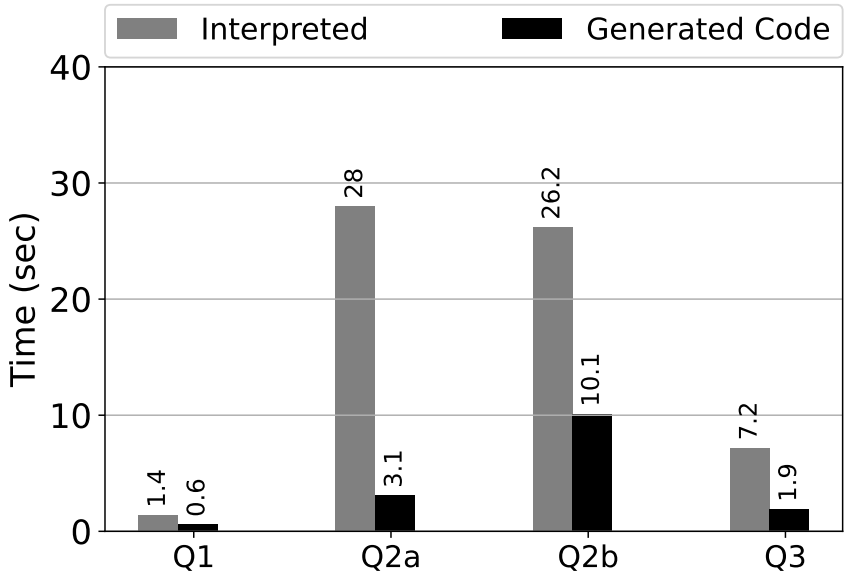


Figure 6.23: Query execution times for the *tweets* dataset

Figure 6.23 shows the execution times in both execution modes: `Interpreted` and `Generated Code`. For all queries, the `Generated Code` mode was faster; however, the differences were less significant than for the queries of the *sensors* dataset. For example, Q2a, which exhibited the highest contrast between the two modes, took 28 seconds to execute

in the `Interpreted` mode and 3.1 seconds in the `Generated Code` mode – the latter mode was ∼8.9X faster. When profiled, we found that disk spilling in AsterixDB's default sort-based algorithm for computing `GROUP-BY` queries was a major overhead in Q2a and Q2b in the `Interpreted` mode. Q3 needs to unnest the *hashtags* field; however, posted tweets that contain hashtags are relatively rare (i.e., the *hashtags* field in most records are `NULL`s). Thus, the CPU cost of unnesting the *hashtags* field was negligible as compared to unnesting the *readings* field in the *sensors* dataset.

6.7.4 *iris_hep* Dataset

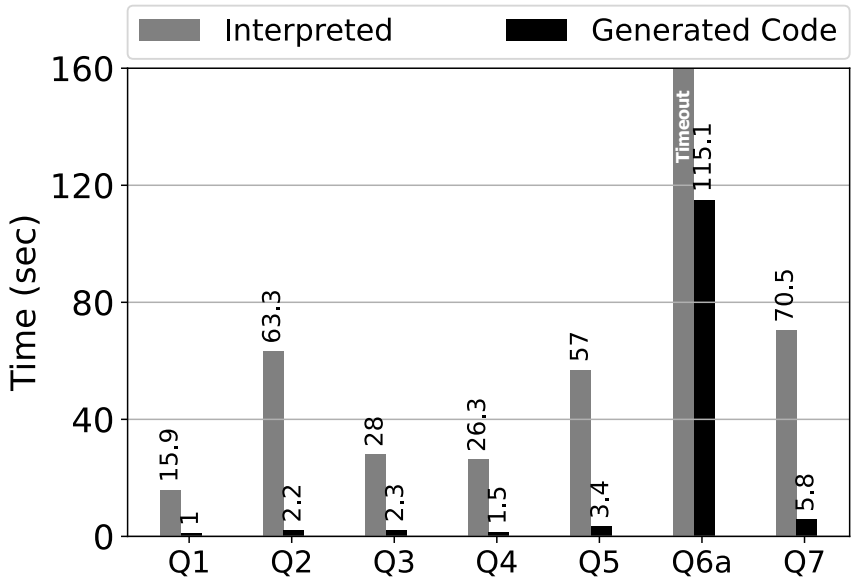


Figure 6.24: Query execution times for the *iris_hep* dataset

The *iris_hep* dataset is the last one that we used in our evaluation. Figure 6.24 shows the execution times for Q1-Q7. All queries except for Q6a took less than six seconds to execute using the `Generated Code` mode. For Q6a, the query took more than 100 seconds to finish in the `Generated Code` mode. The `Interpreted` mode failed to execute Q6a in a reasonable time, and we canceled the execution after waiting more than 30 minutes. In [61], Q6a was

the slowest query to execute in all the systems that the authors evaluated. Similarly, Q6a was also the slowest to execute in our evaluation of the *iris_hep* queries.

The *iris_hep* query performance differences between the two execution modes range between ∼16X and ∼29X. Similar to the *sensors* dataset's queries, *iris_hep*'s queries are heavy on unnesting arrays – potentially an inefficient operation as discussed earlier in Section 6.4.2. In contrast to the *sensors* queries, the *iris_hep* queries are more complex and contain nested calls to SQL++ UDFs[4]. In multiple cases, AsterixDB's compiler failed to fold constant values or to eliminate common expressions – thus increasing the number of function calls that evaluate those expressions and consequently hindering query performance. The code generation framework does not resolve those issues at the translation phase, as it simply maps the operation and expressions of the generated plans by AsterixDB's compiler. However, when Truffle compiles the generated code, it performs all of those optimizations in addition to other (and more complex) optimizations such as inlining function calls and unrolling loops. Therefore, AIL (a Truffle-based language) benefits from Truffle's optimizations. As a result, the optimizations missed by AsterixDB's compiler did not impact the performance of the `Generated Code` mode.

The IRIS-HEP benchmark consists of eight queries in all. The last query (Q8) contains a subquery that `UNION`s two nested subqueries, a plan structure that the proposed code generation framework does not support currently and consequently it bailed out. In our work to date, the code generation framework is a proof-of-concept and does not yet cover all of AsterixDB's operators (e.g., its `UNION`, `DISTINCT`, and `WINDOW` operators). We plan to expand our proof-of-concept to include such operators in the future.

```
SELECT gkey, COUNT(*) cnt
FROM Tweets
GROUP BY <"GROUP-KEY"> AS gkey
ORDER BY cnt DESC
LIMIT 10
```

Figure 6.25: The query template used to evaluate `GROUP-BY` queries

6.7.5 `GROUP-BY` Query Evaluation

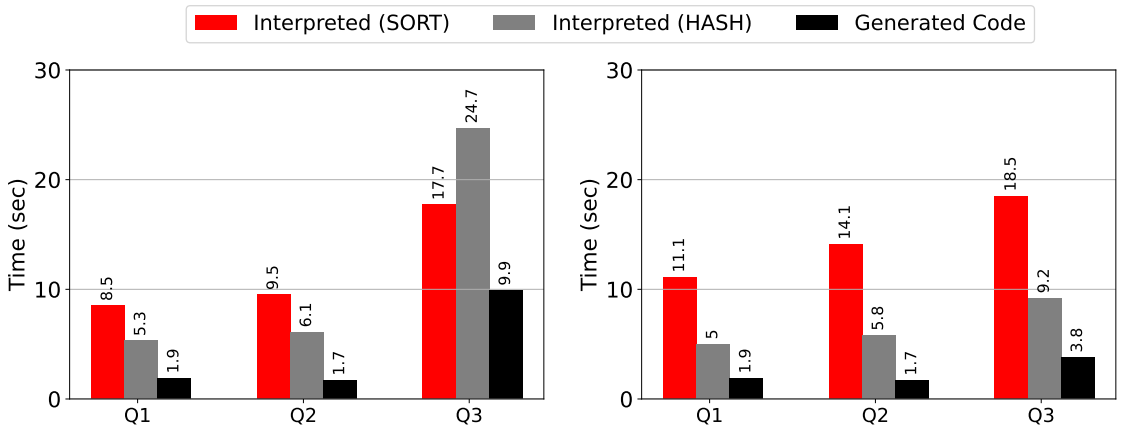
In Section 6.5, we proposed several improvements that could be adopted in AsterixDB's aggregation framework. In this section, we detail evaluation results of for proposed improvements (using our proposed code generation framework) and compare them to the current implementation of AsterixDB's aggregation framework. In this experiment, we ran three queries with the basic structure shown in Figure 6.25 against the *tweets* dataset. We varied the grouping key (`<"GROUP-KEY">`) to use either of three fields: `lang`, `user.followers_count`, and `user.name`. Table 6.3 summarizes the salient characteristics of the three fields.

Query	Grouping Key	# of Distinct Groups	Type
Q1	`lang`	66	String
Q2	`user.followers_count`	192,120	Integer
Q3	`user.name`	4,553,812	String

Table 6.3: Grouping keys characteristics

As was mentioned earlier, AsterixDB uses the sort-based approach by default to process `GROUP-BY` queries. However, a user can provide a hint to direct it to use the hash-based approach instead. Since both approaches can arguably be sensitive to the size of the memory allocated for the `GROUP-BY` operator, we set the memory size to `8MB` for the first run, and then we changed it to `256MB` for a second run.

[4]IRIS-HEP's SQL++ UDF functions: `https://github.com/RumbleDB/iris-hep-benchmark-sqlpp/blob/1fd70b10b72691ac7a9d53ccc90fb0f8f37e5e08/queries/common/functions.sqlpp`



(a) 8MB (b) 256MB

Figure 6.26: Query execution times for the GROUP-BY queries using different memory budgets

Figures 6.26a and 6.26b show the time that it took to run the three queries (see Figure 6.25 and Table 6.26) using the Interpreted mode for both the sort-based (Interpreted (SORT)) and the hash-based (Interpreted (HASH)) approaches. For the Generated Code mode, we used the proposed improvements from Section 6.5.

We first compare the performance of both the Interpreted (SORT) and Interpreted (HASH) approaches. Notice that allocating more memory for the Interpreted (SORT) approach slightly (negatively) impacted the execution performance for the three queries. For instance, Q1 took 8.5 seconds to execute with 8MB of memory and 11.1 seconds with 256MB of memory. On closer inspection, we first noticed that the generated sorted runs for the three queries were merged in a single pass. I.e., the sorted runs were written once and read once from the disk, even when using only 8MB of memory. Asymptotically, the CPU cost of performing external sorting is the same despite using 8MB or 256MB of memory, as the I/O costs were the same for both budgets in our case (i.e., writing and reading once). However, when profiled, we saw that, with the larger memory budget of 256MB, the CPU cost for the in-memory sorting was higher. The first reason for higher CPU cost is that more memory means a larger number of tuples; hence, the CPU needs to perform more work to prepare those tuples for sorting (e.g., constructing an array of

pointers to the tuples). However, the main reason was that more memory means more compare – where each requires two indirect memory accesses (pointer → tuple) – and swap operations. Adding normalized keys [47, 59, 63] to the pointers' array can minimize or even eliminate the cost of indirect memory accesses when comparing two tuples. However, AsterixDB's in-memory sorter uses normalized keys only for values with declared types, and in our experiments, we only declare the types of primary keys.

For the `Interpreted (HASH)` approach, Q1 and Q2 took about the same time to execute. However, Q3, which has more than 4 million distinct groups, showed a significant performance gap. With 8MB of memory, Q3 took 24.7 seconds vs. 9.2 seconds with 256MB of memory. In comparison to the `Interpreted (SORT)` approach, the `Interpreted (HASH)` approach is more sensitive to the memory size and the number of distinct groups in the grouping key – a well-understood behavior of the hash-based approach.

For the `Generated Code` mode, all three queries took less time to execute when compared to the `Interpreted` mode using the two approaches. Since the three queries are reasonably simple, most of the performance gains are attributed to the mixed approach, which benefits from both the hash-based approach's early aggregation and the sort-based approach's robustness. With only 8MB of memory, the `Generated Code` mode (with the proposed improvements) outperformed the two approaches used in the `Interpreted` mode even with 256MB of memory – except for Q3, where the `Interpreted (HASH)` with the larger memory budget approach was slightly faster than the lower budget `Generated Code` approach.

6.8 Related Work

In Chapter 3, we discussed recent research on code generation and related query compilation. Here we mainly focus on recent work on code generation and query compilation for

polymorphic data processing systems.

For RDBMSs, a plethora of work (summarized in Section 3) focuses on code generation and query compilation. Some of that work went beyond generating efficient code for query processing. For instance, in [77], Neumann proposed a technique that utilizes LLVM to ensure that values stay in CPU registers as much as possible and that, in turn, minimizes trips to memory and even to the CPU cache.

To the best of our knowledge, none of the prominent schemaless databases use code generation and compilation for query processing, as work on code generation and compilation for schemaless databases are still in its infancy. Data processing engines like Spark use code generation techniques when querying structured nested data (e.g., Parquet). However, such systems require the schema to be known a priori and should not contain heterogeneous fields. Thus, most such systems utilize strongly-typed languages for code generation, which is sufficient for schema-ful systems.

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 would mean more branches in the generated code. The Truffle framework addresses this issue for dynamically-typed languages such as Python and JavaScript. Recent work [86] has proposed using the Truffle framework for code generation and query compilation to run Language-integrated Query (LINQ) queries over a dynamically-typed collection in JavaScript or R, and that work showed that the performance of their approach was comparable to hand-written code. In this work, we have also used the Truffle framework to generate code for parts of a query plan, in this case for SQL++ in Apache AsterixDB.

The code generation and compilation model imposes additional challenges when measuring the performance of different operations or debugging their correctness. Thus, in [45], the authors proposed a new vectorized execution engine called Photon, which replaces parts of

Spark's older code generation execution engine. The authors showed that Photon outperforms the older engine as Photon's operators were implemented using the native language C++. As a result of using a native language, Photon developers skipped the JVM limitations and gained explicit control over memory and access to SIMD intrinsics. However, the limitations imposed by the JVM are orthogonal to the execution model, as the code generation model could also benefit from gaining control over memory and access to SIMD intrinsics. Moreover, the JVM developers' community is aware of those limitations and proposed several additions to Java 17 to give Java developers more control over memory [19] and access to vectorized computation [20]. Those additions could allow us to improve our code generation framework further in the future without compromising the portability of Java.

6.9 Conclusion

In this chapter, we have presented an approach to code generation and compilation for schemaless document stores using the Truffle framework's JIT compilation capability to process values with heterogeneous types. Additionally, we proposed several enhancements that can be adopted in Apache AsterixDB to improve the performance of `GROUP-BY` queries. In our evaluation, those enhancements, along with the proposed code generation framework, showed very significant improvements over the current AsterixDB execution engine – improving its aggregate query execution performance by orders of magnitude for some workloads.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this dissertation, we have presented several techniques to optimize document stores for handling analytical workloads without sacrificing the flexibility of the document data model.

In Chapter 4, we presented a novel approach for inferring and extracting documents' embedded schemas during data ingestion to minimize the storage overhead of storing JSON-like documents. We first described the workflow of the proposed tuple compactor, which exploits the LSM-lifecycle events to infer the schema and compact the records efficiently. Then, we presented the vector-based format, a compaction-friendly physical record format, to minimize the impact of inferring schemas and compacting records on ingestion performance. Our experiments have shown that the proposed tuple compactor can reduce the storage overhead and improve both the query and ingestion performance of Apache AsterixDB. Combined with our page-level compression, we further reduced the overall storage size and improved the system's query performance.

In Chapter 5, we proposed an additional application for the inferred schema mechanism, where we used it to columnize nested semi-structured data in columnar formats. We first presented our extensions to Dremel to (1) handle schema changes and (2) allow heterogeneous values. Then, we introduced APAX and AMAX, two columnar layouts for storing columns in LSM B^+-trees, and analyzed both layouts' advantages and disadvantages. We also proposed an efficient approach for deleting and upserting previously inserted records. Furthermore, we showed the challenges involved in reading and writing records in the APAX and AMAX layouts and proposed solutions to overcome those challenges. We evaluated the two proposed columnar layouts and showed that querying data stored in the AMAX format was orders of magnitude faster than in AsterixDB's original and the vector-based row formats. The APAX format was faster than both row formats only for datasets that contain a moderate number of columns. In another aspect, our experiments have shown that secondary indexes can accelerate querying data stored in columnar formats, especially when accessing more than a few columns; however, the cost of maintaining the entries of secondary indexes is high.

In Chapter 6, we showed that reducing the I/O costs alone (e.g., by using columnar formats) is only partially sufficient for improving the performance of analytical workloads. Additionally, we described and evaluated (analytically and empirically) the overheads associated with AsterixDB's batch-at-a-time query execution model for analytical workloads. As a result, we introduced our code generation framework to remedy those overheads by translating queries into executable programs to process queries more efficiently. By utilizing Truffle, a code generation framework can re-specialize its code at runtime in case of type changes – addressing the challenge of handling polymorphic types in document stores. We also described and evaluated several techniques to optimize `GROUP-BY` queries, a crucial class of queries in analytical workloads. Our experiments have shown that the combined techniques proposed in this chapter can improve query execution performance for document analysis by orders of magnitude – showing that the CPU cost can be a bottleneck, even for disk-based databases such as Apache AsterixDB.

7.2 Future Work

In Chapter 5, we introduced a new parameter into AsterixDB. The new parameter determines the number of records that can be stored in a single mega leaf node in the AMAX format. We saw that increasing the number of records, for instance, can be beneficial for scan-heavy workloads. However, the resulting pressure on the buffer cache can be high as more temporary buffers are needed to construct the mega leaf nodes, especially if the number of columns of ingested records is high. Having such a static limit that can be tuned manually is less than ideal, however, as it becomes the user's responsibility to configure the limit parameter. A more adaptive approach that considers the number of columns, their sizes, and the current state of the buffer cache would be more user-friendly and may improve both the query and ingestion performance.

Cloud-based object storage solutions are becoming cheaper and more reliable. As a result, many cloud-based analytical databases are adopting the shared-disk architecture instead of the shared-nothing architecture, which is the most common architecture for on-premises parallel databases. With the immutability and the batching nature of LSM, combined with the efficiency of the AMAX format, a shared-disk version of AsterixDB would make a viable candidate for a cloud-based document store. The immutability and the batching of LSM could amortize the monetary cost of updates (or upserts) by bundling them into a single write. At the same time, the columnar AMAX format could reduce the monetary cost of data transfer in cloud-based storage solutions by reading only the relevant columns of a query.

In Chapter 6, we described a mechanism for translating queries into executable programs. However, the proposed code generation framework is a prototype and still lacks the support for translating some of AsterixDB's operators, such as its `UNION`, `DISTINCT` and `WINDOW` operators. We plan to extend our prototype in the future to include such operators. Another

extension for the code generation framework could exploit the interoperability of the Truffle languages to allow users to write and more efficiently run User-defined Functions (UDFs) in their favorite languages. For example, today an AsterixDB user can write a UDF using Python, one of the languages supported to run in Truffle, and call it in SQL++. Since one of the languages that Truffle supports is Python, its generated AIL language could then call the declared UDF and produce the desired result. The interoperability between Truffle languages could avoid today's way of marshaling and unmarshalling the values between two different languages (as both would run in the same JVM) – reducing the cost of UDF calls. Another interesting aspect of the interoperability between Truffle's programming languages and AIL is that AsterixDB's code generation framework could handle users' UDFs in a "white-box" manner. For example, any field access in those UDFs could be pushed down to the `SCAN` operator. Such optimization cannot be performed in AsterixDB's today's Python UDFs.

Bibliography

[1] Actian Vector. https://esd.actian.com/product/Vector.

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

[3] Apache AsterixDB. https://asterixdb.apache.org.

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

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

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

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

[8] Apache Kudu. https://kudu.apache.org.

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

[10] Apache Parquet Documentation. https://parquet.apache.org/documentation/latest.

[11] Apache Parquet Encodings. https://github.com/apache/parquet-format/blob/master/Encodings.md.

[12] Apache spark. https://spark.apache.org.

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

[14] CADRE: Collaborative Archive Data Research Environment. http://iuni.iu.edu/resources/cadre.

[15] Calrivate: WoS. https://clarivate.com/products/web-of-science/.

[16] ClickHouse. https://clickhouse.tech.

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

[18] High-energy Physics Analysis Queries using SQL++ (AsterixDB). https://github.com/RumbleDB/iris-hep-benchmark-sqlpp.

[19] Java Foreign Function & Memory API. https://openjdk.org/jeps/412.

[20] Java Vector API. https://openjdk.org/jeps/414.

[21] MonetDB. https://www.monetdb.org.

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

[23] PyPy. https://www.pypy.org.

[24] Rocksdb. https://rocksdb.org.

[25] Rockset. https://rockset.com.

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

[27] Teradata. https://www.teradata.com.

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

[29] Vertica. https://www.vertica.com.

[30] xml-to-json: Library and command line tool for converting XML files to JSON. http://hackage.haskell.org/package/xml-to-json.

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

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

[33] I. Absalyamov, M. J. Carey, and V. J. Tsotras. Lightweight cardinality estimation in LSM-based systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 841–855, 2018.

[34] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *PVLDB*, 8(8):850–861, 2015.

[35] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. volume 11, pages 198–215. Springer-Verlag New York, Inc., 2002.

[36] W. Y. Alkowaileet, S. Alsubaiee, and M. J. Carey. An LSM-based tuple compaction framework for Apache AsterixDB. *PVLDB*, 13(9):1388–1400, 2020.

[37] W. Y. Alkowaileet and M. J. Carey. Columnar formats for schemaless LSM-based document stores. *To appear in PVLDB*, 2022.

[38] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14), 2014.

[39] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *PVLDB*, 7(10), 2014.

[40] S. Alsubaiee and V. Borkar. Method, apparatus, and computer-readable medium for encoding repetition and definition level values for semi-structured data, July 6 2017. US Patent App. 15/208,032.

[41] S. Alsubaiee, M. J. Carey, and C. Li. LSM-based storage and indexing: An old idea with timely benefits. In *Second international ACM workshop on managing and mining enriched geo-spatial data*, 2015.

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

[43] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. Delta lake: high-performance acid table storage over cloud object stores. *PVLDB*, 13(12):3411–3424, 2020.

[44] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System r: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[45] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan, et al. Photon: A fast query engine for lakehouse systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2326–2339, 2022.

[46] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems (TODS)*, 8(2):255–265, jun 1983.

[47] M. W. Blasgen, R. G. Casey, and K. P. Eswaran. An encoding method for multifield sorting and indexing. *Communications of the ACM*, 20(11):874–878, 1977.

[48] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[49] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *International Conference on Data Engineering (ICDE)*, 2011.

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

[51] M. J. Carey. AsterixDB mid-flight: A case study in building systems in academia. In *International Conference on Data Engineering (ICDE)*, 2019.

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

[53] J. L. Dem. Dremel made simple with Parquet. *Twitter Blog*, 2013. `https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html`.

[54] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *ACM International Conference on Management of Data (SIGMOD)*, pages 295–310, 2016.

[55] D. Durner, V. Leis, and T. Neumann. Json tiles: Fast analytics on semi-structured data. In *ACM International Conference on Management of Data (SIGMOD)*, 2021.

[56] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau. An empirical study on web service evolution. In *2011 IEEE International Conference on Web Services*, pages 49–56. IEEE, 2011.

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

[58] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[59] G. Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3):10–es, 2006.

[60] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *International Conference on Data Engineering (ICDE)*, pages 209–218. IEEE, 1993.

[61] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso. Evaluating query languages and systems for high-energy physics data. *Proc. VLDB Endow.*, 15(2):154–168, oct 2021.

[62] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso. Benchmark Scripts for "Evaluating Query Languages and Systems for High-Energy Physics Data", Apr. 2022. This repository hosts the experiment scripts used for the following paper. Please cite both, the software and the paper, when citing in academic contexts. Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, Gustavo Alonso. "Evaluating Query Languages and Systems for High- Energy Physics Data." In: PVLDB 15(2), 2022. DOI: 10.14778/3489496.3489498.

[63] T. Härder. A scan-driven sort facility for a relational database system. In *VLDB*, pages 236–244, 1977.

[64] K. Jäger. JSINQ–A JavaScript implementation of LINQ to objects. 2009.

[65] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *International Conference on Data Engineering (ICDE)*, pages 613–624. IEEE, 2010.

[66] P.-Å. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL Server column stores. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1159–1168, 2013.

[67] P.-Å. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server column store indexes. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1177–1184, 2011.

[68] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *ACM International Conference on Management of Data (SIGMOD)*, pages 153–164, 2000.

[69] R. A. Lorie. Xrm - an extended (n-ary) relational memory. *IBM Research Report*, G320-2096, 1974.

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

[71] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. volume 29, 2020.

[72] C. Luo and M. J. Carey. On performance stability in lsm-based storage systems. *PVLDB*, 13(4):449–462, 2020.

[73] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[74] 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.

[75] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[76] K. Nakabasami, T. Amagasa, and H. Kitagawa. Querying MongoDB with LINQ in a server-side JavaScript environment. In *International Conference on Network-Based Information Systems*, pages 344–349. IEEE, 2013.

[77] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[78] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

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

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

[81] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *International Conference on Data Engineering (ICDE)*, pages 567–574. IEEE, 2001.

[82] 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)*. IEEE, 2015.

[83] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using jvm. In *International Conference on Data Engineering (ICDE)*, pages 23–23, 2006.

[84] R. X. Sameer Agarwal, Davies Liu. 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`.

[85] G. Savrun-Yeniçeri, M. L. Van de Vanter, P. Larsen, S. Brunthaler, and M. Franz. An Efficient and Generic Event-Based Profiler Framework for Dynamic Languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, page 102–112, New York, NY, USA, 2015.

[86] F. Schiavio, D. Bonetta, and W. Binder. Language-agnostic integrated queries in a managed polyglot runtime. *PVLDB*, 14(8):1–13, 2021.

[87] 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. volume 8, pages 1668–1679. VLDB Endowment, 2015.

[88] S. Sohan, C. Anslow, and F. Maurer. A case study of web api evolution. In *2015 IEEE World Congress on Services*, pages 245–252. IEEE, 2015.

[89] M. Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[90] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: a sql system for multi-structured data. In *ACM International Conference on Management of Data (SIGMOD)*, pages 815–826, 2014.

[91] 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.

[92] C. Wimmer and S. Brunthaler. ZipPy on Truffle: A Fast and Simple Implementation of Python. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity*. Association for Computing Machinery, 2013.

[93] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM international symposium on New ideas, New Paradigms, and Reflections on Programming & Software*, pages 187–204, 2013.

[94] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.

Appendix A

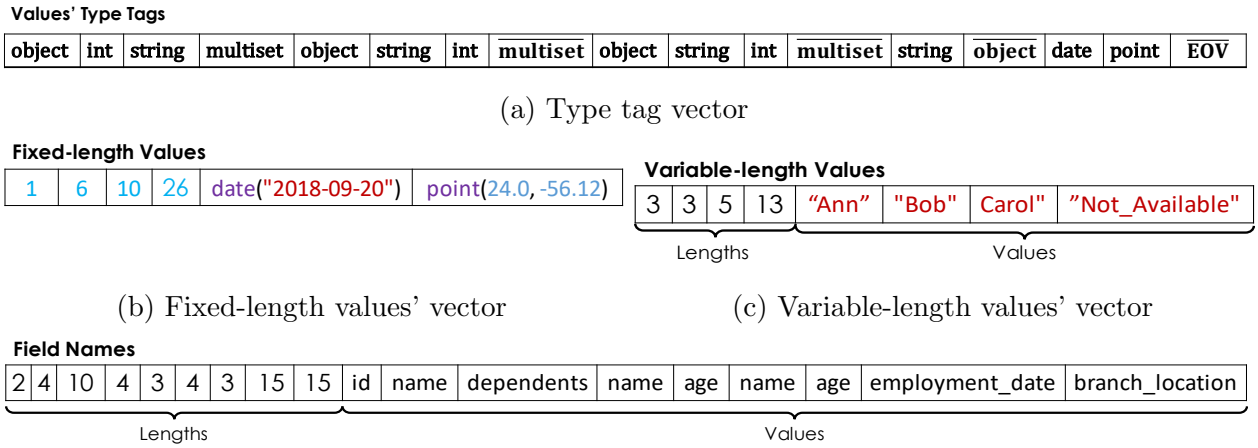
Chapter 4 Supplementary Material

A.1 Vector-based Format: Additional Example

In Section 4.3.3, we showed an example of a record in the vector-based format. However, the example may not clearly illustrate the structure of a record with complex nested values. In this section, we walk through another example of how to interpret a record in a vector-based format with more nested values.

```
{
  "id": 1,
  "name": "Ann",
  "dependents":{{
     {"name": "Bob", “age”: 6},
     {"name": "Carol", “age” : 10} ,
      ”Not_Available" }},
  "employment_date": date("2018-09-20"),
  "branch_location": point(24.0, -56.12),
}
```

Figure A.1: JSON document with more nested values



(a) Type tag vector

(b) Fixed-length values' vector

(c) Variable-length values' vector

(d) Field names' vector

Figure A.2: A record in a vector-based format (another example)

Figure A.2 shows the structure of the JSON document, shown in Figure A.1, in the vector-based format. Starting with the header (not show), we determine the four vectors, namely **(i)** the type tag vector (Figure A.2a), **(ii)** the fixed-lengths values' vector (Figure A.2b), **(iii)** the variable-length values' vector (Figure A.2c), and finally **(v)** the field names' vector (Figure A.2d).

After processing the header, we start by reading the first tag (*object*), which determine the root type. As explained in Section 4.3.3, the tags of nested values (i.e., *object, array*, and *multi*) and control tags (i.e., $\overline{object}$, $\overline{array}$, $\overline{multiset}$, and $\overline{EOV}$) are neither fixed or variable length values. We continue to the next tag which is (*int*). Since *int* is a fixed-length value, we know it is stored in the first four bytes (assuming *int* is a 4-byte int). Also, because it is preceded by the nested tag (*object*), we know it is a field of this object, and thus its field name corresponds to the first field name ***id*** in the field names' vector. Next, we see the tag (*string*). As it is the first variable-length value, the length (i.e., **3**) and the value `"Ann"` in the variable-length vector belong to this string value.

Followed by the string, we get the tag (*multiset*), which is a nested value and a child of the root object type. Therefore, the third field name (length: 10, value: dependents) corresponds

to the multiset value. We see in Figure A.1 that the the field *dependents* is a multiset of three elements of types: (*object*), (*object*) and (*string*). Thus, the following tag (*object*) corresponds to the first element of the multiset. As it is a child of a multiset, our object value does not have a field name. The next two tags are of type (*string*) and (*int*), which correspond to the field names ***name*** and ***age***, respectively, as they are children of the preceded (*object*) tag. The following tag ($\overline{multiset}$) marks the end of the current nesting type (i.e., object) and we are back to the nesting type (i.e., mutliset). The following tag (*object*) marks the beginning of the second element of the mutliset (See Figure A.1) and we process it as we did for the first element. After the second control tag ($\overline{multiset}$), we get the tag (*string*), which is the type of the third and the last element of our multiset. The following control tag ($\overline{object}$) tells it is the end of the multiset and we are going back to the root object type.

The next two tags (*date*) and (*point*) are the last two children of the root object type and they have the last two field names ***employment_date*** and ***branch_location***, respectively. Finally, the control tag ($\overline{EOV}$) marks the end of the record.

A.2 Queries

We provide the queries we ran in our experiments against the Twitter, WoS, and Sensors datasets.

A.2.1 Twitter Dataset's Queries

```
Q1:
SELECT VALUE count(*)
FROM Tweets
```

```
Q2:
SELECT VALUE uname,a
FROM Tweets t
GROUP BY t.users.name AS uname
WITH a AS avg(length(t.text))
ORDER BY a DESC
LIMIT 10

Q3:
SELECT uname, count(*) as c
FROM Tweets t
WHERE (
  SOME ht IN t.entities.hashtags
  SATISFIES lowercase(ht.text) = "jobs"
)
GROUP BY user.name as uname
ORDER BY c DESC
LIMIT 10

Q4:
SELECT *
FROM Tweets
ORDER BY timestamp_ms
```

A.2.2 WoS Dataset's Queries

```
Q1:
SELECT VALUE count(*)
FROM Publications as t

Q2:
SELECT v, count(*) as cnt
FROM Publications as t,
    t.static_data.fullrecord_metadata
    .category_info.subjects.subject
        AS subject
WHERE subject.ascatype = "extended"
GROUP BY subject.`value` as v
ORDER BY cnt DESC

Q3:
SELECT country, count(*) as cnt
FROM (
  SELECT value distinct_countries
  FROM Publications as t
  LET address = t.static_data
                .fullrecord_metadata
                .addresses.address_name,
  countries = (
      SELECT DISTINCT VALUE
          a.address_spec.country
      FROM address as a
  )
  WHERE is_array(address)
```

```
  AND array_count(countries) > 1
  AND array_contains(countries, "USA")
) as collaborators
UNNEST collaborators as country
WHERE country != "USA"
GROUP BY country
ORDER BY cnt DESC
LIMIT 10

Q4:
SELECT pair, count(*) as cnt
FROM (
    SELECT value country_pairs
    FROM Publications as t
    LET address = t.static_data
                  .fullrecord_metadata
                  .addresses.address_name,
    countries = (
        SELECT DISTINCT VALUE
            a.address_spec.country
        FROM address as a
        ORDER by a.address_spec.country
    ),
    country_pairs = (
        SELECT VALUE [countries[x], countries[y]]
        FROM range(0, array_count(countries) - 1) as x,
             range(x + 1, array_count(countries) - 1) as y
  )
  WHERE is_array(address)
```

```
  AND array_count(countries) > 1
) as country_pairs
UNNEST country_pairs as pair
GROUP BY pair
ORDER BY cnt DESC
LIMIT 10
```

A.2.3 Sensors Dataset's Queries

```
Q1:
SELECT count(*)
FROM Sensors s

Q2:
SELECT max(r.temp), min(r.temp)
FROM Sensors s, s.readings r

Q3:
SELECT sid, avg_temp
FROM Sensors s, s.readings as r
GROUP BY s.sensor_id as sid
WITH avg_temp as AVG(r.temp)
ORDER BY t DESC
LIMIT 10

Q4:
SELECT sid, avg_temp
FROM Sensors s, s.readings as r
WHERE s.report_time > 1556496000000
```

```
AND s.report_time < 1556496000000
                    + 24 * 60 * 60 * 1000
GROUP BY s.sensor_id as sid
WITH avg_temp as AVG(r.temp)
ORDER BY avg_temp DESC
LIMIT 10
```

Appendix B

Chapter 5 Queries

We provide the queries we ran in our experiments in Chapter 5 against the *cell*, *tweets_1*, *sensors*, and *wos* datasets.

B.1 *cell* Queries

```
Q1:
SELECT VALUE COUNT(*)
FROM Cell

Q2:
SELECT caller, MAX(c.duration) as m
FROM Cell c
GROUP BY c.caller AS caller
ORDER BY a DESC
LIMIT 10
```

Q3:

```
SELECT VALUE COUNT(*)
FROM Cell c
WHERE c.duration >= 600
```

B.2 *tweets_1* Queries

Q1:

```
SELECT VALUE COUNT(*)
FROM Tweets
```

Q2:

```
SELECT VALUE uname,a
FROM Tweets t
GROUP BY t.users.name AS uname
WITH a AS MAX(length(t.text))
ORDER BY a DESC
LIMIT 10
```

Q3

```
SELECT uname, COUNT(*) as c
FROM Tweets t
WHERE (
  SOME ht IN t.entities.hashtags
  SATISFIES LOWERCASE(ht.text) = "jobs"
)
GROUP BY user.name as uname
ORDER BY c DESC
LIMIT 10
```

B.3 *sensors* Queries

```
Q1:
SELECT VALUE COUNT(*)
FROM Sensors s, s.readings r

Q2:
SELECT MAX(r.temp), MIN(r.temp)
FROM Sensors s, s.readings r

Q3:
SELECT sid, max_temp
FROM Sensors s, s.readings as r
GROUP BY s.sensor_id as sid
WITH max_temp as MAX(r.temp)
ORDER BY t DESC
LIMIT 10

Q4:
SELECT sid, max_temp
FROM Sensors s, s.readings as r
WHERE s.report_time > 1556496000000
AND s.report_time < 1556496000000
                  + 24 * 60 * 60 * 1000
GROUP BY s.sensor_id as sid
WITH max_temp as MAX(r.temp)
ORDER BY max_temp DESC
LIMIT 10
```

B.4 *wos* Queries

```
Q1:
SELECT VALUE COUNT(*)
FROM Publications as t

Q2:
SELECT v, COUNT(*) as cnt
FROM Publications as t,
   t.static_data.fullrecord_metadata
   .category_info.subjects.subject
           AS subject
WHERE subject.ascatype = "extended"
GROUP BY subject.`value` as v
ORDER BY cnt DESC

Q3:
SELECT country, COUNT(*) as cnt
FROM (
  SELECT value countries
  FROM Publications as t
  LET  address = t.static_data
                 .fullrecord_metadata
                 .addresses.address_name,
       countries = ARRAY_DISTINCT(
              address[*].address_spec.country
              )
  WHERE IS_ARRAY(address)
  AND ARRAY_COUNT(countries) > 1
  AND ARRAY_CONTAINS(countries, "USA")
```

```
) as collaborators
UNNEST collaborators as country
WHERE country != "USA"
GROUP BY country
ORDER BY cnt DESC
LIMIT 10

Q4:
SELECT pair, COUNT(*) as cnt
FROM (
    SELECT value ARRAY_PAIRS(countries)
    FROM Publications as t
    LET  address = t.static_data
                 .fullrecord_metadata
                 .addresses.address_name,
         countries = ARRAY_DISTINCT(
            address[*].address_spec.country
            )
    WHERE IS_ARRAY(address)
    AND ARRAY_COUNT(countries) > 1
) as country_pairs
UNNEST country_pairs as pair
GROUP BY pair
ORDER BY cnt DESC
LIMIT 10
```