The Evolution of Storage in Apache AsterixDB

Murtadha Al Hubail, Ali Alsuliman, Wail Alkowaileet[†], Michael Blow, Michael Carey[*‡], Peeyush Gupta, Ian Maxon[‡], Daniel Nagy, Ritik Raj, Shahrzad Shirazi, Hussain Towaileb
Couchbase, Inc.
San Jose, CA, USA

ABSTRACT

This paper describes the evolution and current state of the storage subsystem of Apache AsterixDB. An initial shared-nothing version of AsterixDB, with an LSM-based storage system, was open-sourced in 2013 after four years of R&D; its storage system was described in a 2014 VLDB paper. In the dozen years since then, much has changed in the computing environment, including the move from magnetic storage to SSDs, the migration of database services to the cloud, and the availability and popularity of object storage. In this paper, we describe how AsterixDB storage has evolved in response, including the addition of column-oriented JSON storage, compute-storage separation with object store backing, and resource management policies targeting multi-user workloads over databases with many collections and indexes. Apache AsterixDB's storage system is also the storage system for the Capella Analytics and Enterprise Analytics product offerings from Couchbase, Inc.

PVLDB Reference Format:
Murtadha Al Hubail, Ali Alsuliman, Wail Alkowaileet, Michael Blow, Michael Carey, Peeyush Gupta, Ian Maxon, Daniel Nagy, Ritik Raj, Shahrzad Shirazi, Hussain Towaileb. The Evolution of Storage in Apache AsterixDB. PVLDB, 19(12): 4290 - 4302, 2026.
doi:10.14778/3827998.3828033

1 INTRODUCTION

AsterixDB was initially built, back in the MapReduce era, to serve as a feature-rich Big Data Management System (BDMS) for handling large volumes of semistructured data[12]. It aims to reduce the degree of database system sprawl in enterprises (with a motto of "one size fits a bunch"), to minimize the mismatch between an application's view of data and the persisted database state, to support the declarative management and querying of schemaless flexible data, to scale horizontally, and to accomplish all of those goals both as an on-prem database platform and now as a data management service in the cloud. AsterixDB is based on a declarative SQL-like query language, namely SQL++ [17, 20, 34]. It supports complex analytical queries (ad hoc joins, aggregations, windowing, and OLAP) over large collections of JSON documents that can originate from a variety of data sources, and its data can either be stored in and managed by the system or it can reside in external object storage and be accessed on demand at query time.

The Apache AsterixDB storage system has changed significantly over the last decade. The goal of this paper is to describe how it has evolved in response to changes in the computing environment, as well as through innovations contributed by both university and industrial project participants, and what it looks like today in terms of features and capabilities. More attention will be paid to those aspects of the system that have changed the most. To that end, the paper will proceed as follows: Section 2 provides an overview of AsterixDB and its JSON-based data model. Section 3 provides an overview of the AsterixDB storage layer and how it has evolved since its inception. Section 4 provides a more detailed description of the system's LSM based storage approach, including the current policies used to manage LSM components and how the system's memory is utilized. Section 5 double-clicks on the system's row- and column-oriented storage formats. Section 6 describes the approach taken to separate compute from cloud storage. Section 7 presents a set of performance results to highlight row- versus column-based storage format performance, as well as showing the system's ability to scale effectively. Section 8 discusses some of the lessons learned as the system has evolved. Finally, Section 9 concludes the paper.

2 APACHE ASTERIXDB

Apache AsterixDB [12, 16] was initially designed to have a set of features suitable for use cases such as web data warehousing and social media data analytics. AsterixDB's overall set of features includes the following:

- A NoSQL-style data model (ADM) based on extending JSON with a few concepts borrowed from object databases;
- A declarative query language (now SQL++) supporting a wide range of queries against multiple semi-structured datasets;
- A partition-aware query optimizer for choosing execution plans for parallel (MPP style) queries;
- An efficient data flow execution engine (Hyracks) for running partitioned-parallel query plans;
- Partitioned and LSM-based native storage and indexing for very large datasets;
- Support for querying external data (e.g., data in object storage, data lakes, or HDFS) like natively managed data;
- Rich field type support, including numeric, textual, temporal, and simple spatial types;
- Secondary indexing through B+ trees, R-trees, and inverted keyword indexes;
- Basic NoSQL-like transactional support similar to that of other document databases.

[*]Contact author. Email: mike.carey@couchbase.com
[†]Current affiliation: King Abdulaziz City for Science and Technology. Work done while at Couchbase, Inc.
[‡]Academic affiliation: UC Irvine.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828033

2.1 JSON-Based Data Model: ADM

The user-level data model for AsterixDB is based on JSON, a lightweight text-based data format with a simple specification [5]. By design, JSON is self-describing (no schema required) and readable and writable by both humans and machines. JSON objects are composed of primitive types, flexible records, and arrays. JSON is in wide use for data exchange, for building and invoking web applications and services, and it has become the defacto data format of choice for AI-infused applications. Most document databases have embraced JSON as their data model, including (alphabetically): Apache AsterixDB [3, 16], CosmosDB [7], Couchbase [2, 4, 15, 26, 27], DynamoDB [1, 25], and of course MongoDB [6]. Document databases are popular for the schema flexibility that they offer through their support for collections of semistructured objects that can be nested, heterogeneous, and schemaless [35]. AsterixDB's data model, ADM, extends JSON with multisets in addition to arrays, a richer set of field types, and an option to declare data types for datasets in order to specify some or all of their objects' fields if desired; schemalessness is thus accepted but not not mandated [12].

```
CREATE TYPE OrderLineType AS {
   lineNumber: Integer,
   amount: Double
};

CREATE TYPE OrderType AS {
   id: String,
   orderDate: DateTime,
   total: Double,
   orderStatus: String?,
   paymentMethod: String,
   paymentStatus: String,
   currency: String?,
   country: String?,
   createdAt: DateTime?,
   updatedAt: DateTime?,
   orderLines: [OrderLineType]
};

CREATE COLLECTION TypedOrders(OrderType)
   PRIMARY KEY id
   WITH {"storage-format": {"format":"row"}};
```

Figure 1: `TypedOrders` Collection Definition

To illustrate, Figure 1 shows an example where AsterixDB's data model is used to define a typed collection called *TypedOrders* for which many of the fields are known and declared a priori. The first CREATE TYPE statement defines a type *OrderLineType* for line items in an order, and the second CREATE TYPE defines a type *OrderType* to model order objects. The CREATE COLLECTION statement defines the typed collection itself, specifying its primary key (*id*) and the desired storage format (*row* format in this example). Notice that *orderLines* is a nested array of *OrderLineType* objects. In the example, the fields *orderStatus*, *currency*, *country*, *createdAt*, and *updatedAt* are optional, so they can either have a value of the indicated field type or they can be NULL or altogether MISSING (absent). The remaining fields are mandatory; they must be present and of the indicated type for an object to conform to the type definition. The AsterixDB data model is open, so additional undeclared fields are also acceptable (unless the type definition had said AS CLOSED instead of simply saying AS [12]).

```
INSERT INTO TypedOrders ([
 { "id": "101",
   "orderDate": datetime("2026-01-15T00:00:00"),
   "total": 299.99,
   "discount": 10.00,
   "orderStatus": "shipped",
   "paymentMethod": "credit_card",
   "paymentStatus": "captured",
   "currency": "USD",
   "country": "US",
   "createdAt": datetime("2026-01-15T10:10:00"),
   "updatedAt": datetime("2026-01-15T11:00:00"),
   "paymentDetails": {
     "card_network": "visa",
     "card_last4": "4242" },
   "orderLines": [
     { "lineNumber": 1, "amount": 190 },
     { "lineNumber": 2, "amount": 109.99 } ]
 },
 { "id": "102",
   "orderDate": datetime("2026-02-01T00:00:00"),
   "total": 100,
   "discount": "12%",
   "orderStatus": "pending",
   "paymentMethod": "ach",
   "paymentStatus": "pending",
   "currency": "USD",
   "country": "US",
   "createdAt": datetime("2026-02-01T08:25:00"),
   "updatedAt": datetime("2026-02-01T08:30:00"),
   "paymentDetails": {
     "bank_name": "Chase",
     "ach_trace_id": "ACH99817263"
   },
   "orderLines": [
     { "lineNumber": 1, "amount": 90.00 },
     { "lineNumber": 2, "amount": 10.00 } ]
 }
]);
```

Figure 2: `TypedOrders` Collection Insertion

Figure 2 shows a SQL++ statement that adds two objects to the *TypedOrders* collection and illustrates several additional features of the data model. First, note that the two objects each have a field *discount* that was not specified a priori. This field can be of any type, and in fact the two objects happen to have different value types for the field (a double and a string, respectively). Second, note that the inserted objects also have an undeclared nested *paymentDetails* field that differs between the objects. Again, this is perfectly fine, so the example INSERT statement will succeed since both of the objects being inserted conform to the definition of *OrderType*.

Figure 3 shows an example of an alternative type and collection definition that would also work for Figure 2's order data; in this case, only the primary key field is specified a priori, and the remaining fields of an object can be present or absent at the per object level. In this case, "anything goes", so objects inserted into the *LooseOrders* collection are only required to have a string-valued field named *id*. This approach is maximally flexible, making application changes simpler, but comes with the disadvantage that objects in the collection can vary widely from one another, so any desired type constraints must be handled by the application.

```
CREATE TYPE LooseOrderType AS { id: String };

CREATE COLLECTION LooseOrders(LooseOrderType)
   PRIMARY KEY id
   WITH {"storage-format": {"format":"column"}};
```

Figure 3: `LooseOrders` Collection Definition

```
CREATE INDEX TypedOrderDateIndex
    ON TypedOrders(orderDate);

CREATE INDEX TypedOrderDiscountIndex
    ON TypedOrders(discount);

CREATE INDEX OrderLineAmountIndex
    ON TypedOrders(UNNEST orderLines SELECT amount)
    EXCLUDE UNKNOWN KEY;
```

Figure 4: Secondary Index Examples

AsterixDB includes support for defining secondary indexes to speed queries involving selective predicates. Figure 4 shows three examples of creating secondary indexes for the *TypedOrders* collection. The first example indexes the orders on a declared, homogeneously typed field, namely *orderDate*. The second example indexes the orders on a field that was not specified in the collection definition, one that, in fact, has heterogeneous values.* The final example shows the creation of an array index that indexes the orders on a field of their nested line items. Note that while these examples show just a single secondary key field, composite keys are also fine, so one could opt to index the orders on (currency, total).

2.2 SQL-Based Query Language: SQL++

AsterixDB's query language is a generalization of SQL, extended to support nesting and schema optionality, known as SQL++ [17, 20, 34]. Figure 5 shows an example of a SQL++ analytical query over the *TypedOrders* collection. The example query searches the orders collection to find orders made in U.S. Dollars since the beginning of 2025 that contained at least line item priced at under fifty dollars. For such orders, it groups the orders based on how they were paid for and their current status, computing both the average number of items per order and the average total cost per order for each group. It delivers its results sorted by payment method and (within each method) payment status.

```
SELECT o.paymentMethod,
          o.paymentStatus,
          AVG(array_count(o.orderLines)) AS avgOrderSize,
          AVG(o.total) AS avgOrderTotal
FROM TypedOrders o
WHERE o.currency = "USD"
   AND o.orderDate >= datetime("2025-01-01T00:00:00")
   AND (SOME l IN o.orderLines SATISFIES l.amount < 50.00)
GROUP BY o.paymentMethod, o.paymentStatus
ORDER BY o.paymentMethod, o.paymentStatus;
```

Figure 5: An Example SQL++ Query

In addition to queries, of course, SQL++ in AsterixDB supports data modification statements. We already saw an INSERT in our example. SQL++ also supports UPSERT, UPDATE, DELETE, and TRUNCATE statements for modifying a collection's contents.

2.3 Shared-Nothing Architecture

Figure 6 shows a high-level overview of a traditional "on-prem" AsterixDB cluster. In this configuration, the system has a traditional shared-nothing architecture. Incoming SQL++ queries are handled by the Cluster Controller, which is the node that hosts the SQL++ query compiler and optimizer and orchestrates their execution. Optimized queries are translated into dataflow jobs that are distributed to the cluster's Node Controllers and are executed in parallel by the Hyracks dataflow layer. Each Node Controller manages one or more storage partitions for each AsterixDB collection based on LSM (Log-Structured Merge) tree technology. LSM data is paged and accessed through a fairly traditional buffer cache.

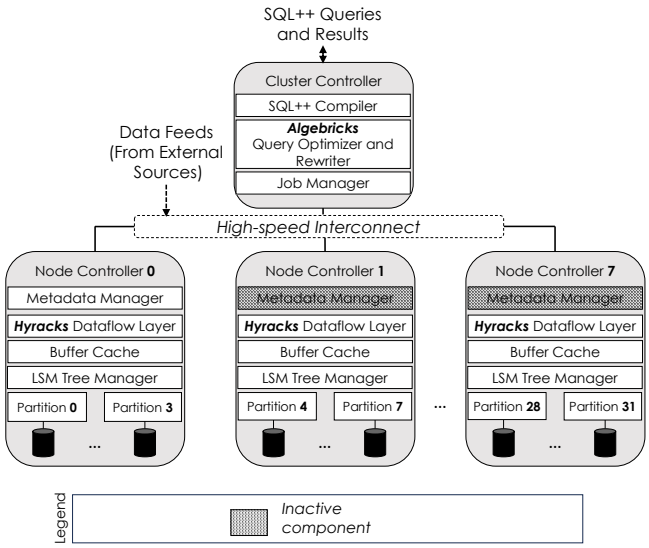

Figure 6: Apache AsterixDB Cluster (Shared Nothing).

Specifically targeting truly "Big Data" of the sort expected in the initial web and social media analytics use cases, a fundamental assumption from the very start of the AsterixDB project has been that the portion of data stored on a given node can significantly exceed the size of its main memory, and likewise (potentially) for intermediate query results [28]. The initial storage technology for the system was magnetic disk, but that was later replaced in most settings by SSD technology. Later in the paper we will explain how the emergence of cloud object storage has led AsterixDB to evolve further into a compute-storage separated architecture (informally called "cloud mode").

3 LSM-BASED STORAGE

AsterixDB's storage system is entirely LSM-based, though it is somewhat non-traditional (or unusually traditional, depending on your viewpoint) in its approach. Much of the community today thinks of LSM storage components in terms of layered SSTables, introduced in Google's early BigTable storage system [21] and adopted by the popular RocksDB [24] key-value store. As detailed in the original AsterixDB storage paper, the components of an LSM B+ tree in AsterixDB are themselves B+ trees – as in the original LSM index paper by O'Neill et al [33] – and as in HBase [14]. The decision to use LSM-based indexing with a fractal-like approach where each LSM component is an index instance, rather than instead going with in-place indexing or LSM with SSTables, was taken for three reasons:

(1) LSM-based indexing enabled the storage system to achieve higher ingestion throughput for append-mostly big data, which was our expected target workload.

*Until recently, AsterixDB required that the indexed fields be predeclared in the schema or have their field types specified in the index definition. These limitations have now been relaxed [36].

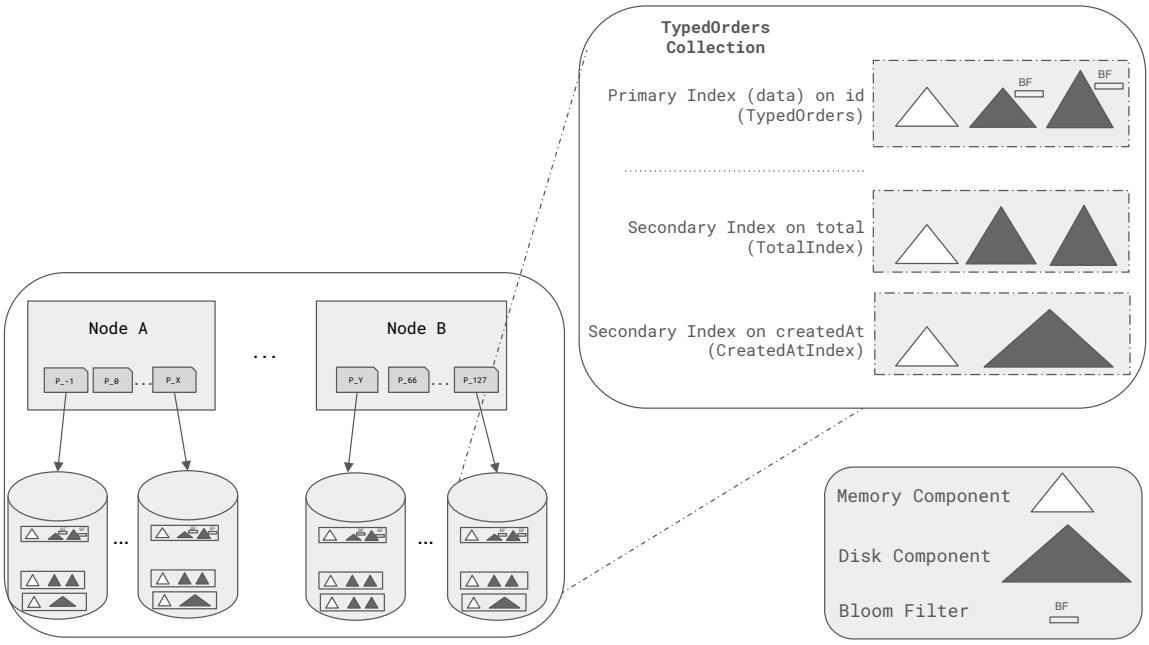

Figure 7: Primary and Secondary LSM Indexes.

(2) LSM-based indexing simplified the implementation of concurrency control and recovery due to LSM's component immutability via flushes and merges and management of an active components list [13].
(3) The fractal-like approach enables the storage system to support the "LSM-ification" of any existing index data structure as long as an efficient component merge approach could be found. As a result, AsterixDB currently provides LSM B+ tree, R-tree, and inverted text indexes as its available index options, and an LSM friendly vector index is currently in the works.

We will see several additional advantages of AsterixDB's LSM-based storage architecture later in the paper.

3.1 Data and Index Partitioning

Figure 7 shows how the LSM-based storage for a given collection is partitioned across a cluster. The organization of AsterixDB data is based on a combination of hash partitioning, a primary index to hold the collection's data objects, and zero or more local secondary indexes. The disk icons in the figure represent node-attached storage partitions for the collection, and the arrow associations of these icons with the cluster's nodes indicate which nodes are responsible for reading and writing the data locally for which storage partition.

Data objects in a collection are assigned to partitions by hashing them based on the collection's primary key. The data and access paths for each partition of a collection reside in a set of interrelated LSM tree indexes, as indicated in the figure. The primary index contains the actual data objects, ordered by primary key, with zero or more secondary indexes serving to map values from an object's other attributes (secondary keys) to their associated primary key to speed the execution of selective queries. Secondary index partitions refer only to objects in their corresponding primary index partition, so a secondary key-based lookup will run in parallel on all nodes. Several optimizations are employed to speed searches, including Bloom filters for primary index components, sorting of the primary keys from a secondary lookup prior to accessing the primary index to fetch the objects, executions of primary index lookups in batches rather than individual probes, remembrence of the most recently visited page in a search to streamline cases where the next (sorted) lookup would lead back to the same page, and so on [29].

4 LSM RESOURCE MANAGEMENT

To operate on very large volumes of data reliably and efficiently, AsterixDB carefully controls how the main memory of its nodes is used. Figure 8 shows how the main memory resources of a node are allocated to different storage- and query execution-related needs. Each node's memory is divided into three regions: a buffer cache, for holding pages of on-disk components; in-memory component memory, for holding pages of in-memory components as new data is ingested or existing data is modified; and working memory, which provides scratch pages for memory-intensive operators like joins, sorts, and grouped aggregation.

4.1 Regional Memory Management

The AsterixDB buffer cache is a fairly traditional database-style pool of in-memory pages. It has evolved over time from initially supporting traditional in place indexes, to supporting memory management for LSM-based indexes, and now to supporting the needs of LSM trees for AsterixDB's columnar format. Buffer cache pages that follow a traditional lifecycle of pinning and unpinning, which is the case for all reads, are paged out with a Clock-based replacement policy. For objects that exceed the system's normal page size, "large pages" are allocated as contiguous multiples of single pages. (This is necessary because the code base expects the bytes for a given row format object to be contiguous in memory; large object

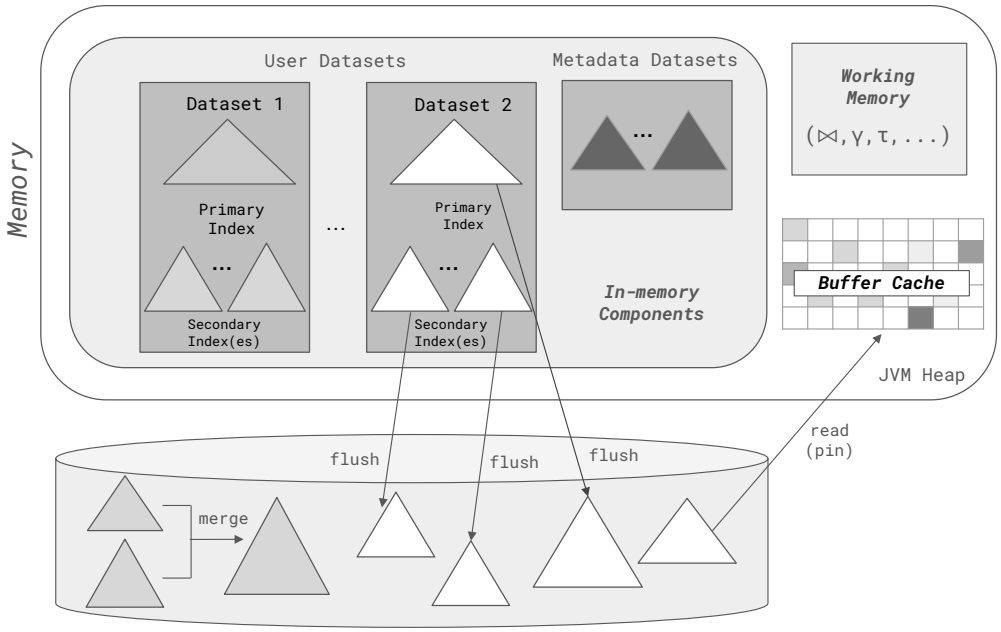

Figure 8: Main Memory Regions.

support came later in the system's history.) AsterixDB's initial default page size was 32KB; today it is 128KB to better utilize the bandwidth potential of today's storage devices. Additionally, to support LSM operations (flush or merge) where all disk writes must happen in-order and only once, pages can also be "confiscated" from the buffer cache. Page confiscation is a combination of pinning but delaying the assignment of a file offset to a newly allocated page. This enables true append-only construction of tree structures, as is required by the write semantics of many object stores. As an example, when constructing a LSM B+ tree disk component, it is created via a bulkload-like process. This process can be made append-only by using page confiscation for interior nodes, delaying the assignment of a page offset until they are full and thus ready to be written. (Note that this results in a file format where the root page of the tree is at the end of the file rather than at the beginning.)

Similar to the buffer cache, the in-memory component memory is a shared pool of pages. It contains the in-memory components of the primary and secondary indexes of the collections that are currently undergoing modifications. When an in-memory LSM component needs more space for its incoming data, it requests a page from this shared pool. Once a certain percentage of the pool is used, one or more collections' in-memory components are selected to be flushed in a round-robin fashion. When a flush is triggered for a collection, the in-memory components of all its indexes are flushed simultaneously; this aids in identifying a shared rollback point across the collection's indexes.

Lastly, to support efficient algorithms for hash and sort-based query operators, each operator is given a budget that it uses while reading and writing intermediate files (e.g., run files and hash buckets) and queries are queued if their aggregate memory requirements cannot be met with currently unclaimed pages in working memory [28]. By default, AsterixDB devotes 25% of its overall memory (i.e., its JVM heap) to the buffer cache, 25% to the in-memory component page pool, and the other 50% to operators' working memory.

4.2 LSM Component Management

As discussed above, in-memory components are flushed to disk based on memory pressure. When the number of LSM components for an index starts to add up, a sequence of smaller components will be merged into a larger component. Concurrent merges are supported, even for the same index. The concurrent merge policy has a set of properties that can be left to their default values or tuned if necessary. These properties include:

- min-merge-component-count (default: 3), to provide a lower bound on the number of components in a merge;
- max-merge-component-count (default: 10), to provide an upper bound on the number of components in a merge;
- size-ratio (default: 1.2), to avoid merging a big component with too many small components;
- max-component-count (default: 30), to limit the overall number of components allowed to co-exist;
- maxComponentSize (default: 1TB), to avoid merging components that are already large enough, and to account for cloud object storage services with a maximum file size.

The defaults were chosen based on experimentation with a variety of different workload characteristics [29].

5 ALTERNATIVE STORAGE FORMATS

As described in Section 2.1, AsterixDB offers users a choice between a row-based format (its first offered format) and a newer column-based format (now the default format) for the primary index of each collection. Also, as shown there, users can choose to specify

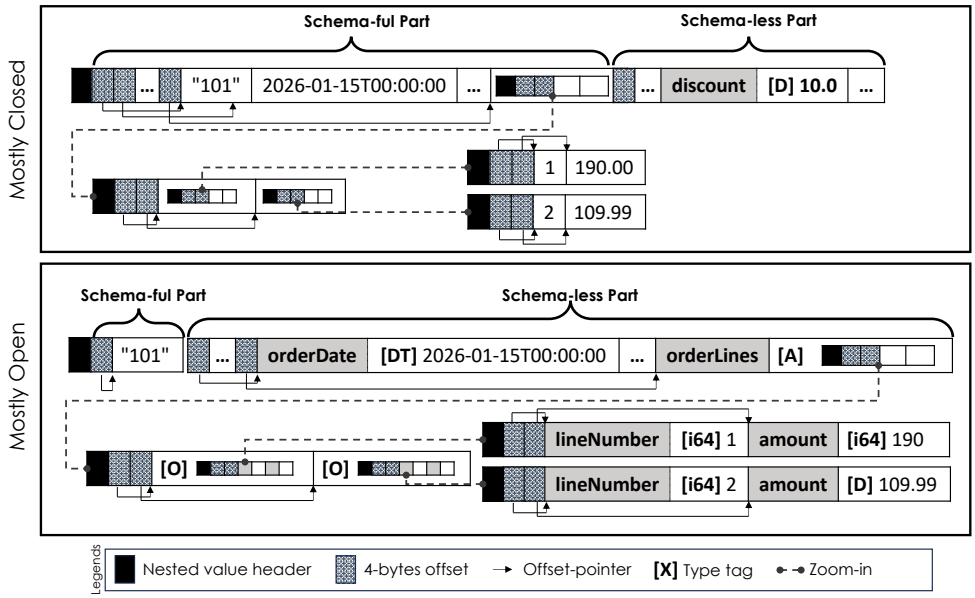

Figure 9: Row Format(s).

as much about the schema of a given collection as they wish to reveal (and have the system enforce).

5.1 Row Format

Figure 9 illustrates the internal data representation for the row format by showing how the first example order in Figure 2 from Section 2.1 would look if its schema were mostly predefined (as in OrderType in Figure 1) or left mostly open (as in LooseOrderType in Figure 3). One of the underlying motivations for providing users with the ability to provide schema information was to have an easy path to achieving relational-style storage efficiency in situations where the schema is known a priori, yet fully supporting schemaless data in settings where the user prefers to leave things open. As shown, a row object has a "schema-ful" part that contains the data values for the object's known fields; their field names and data types are stored separately in the system metadata catalog that describes the collection and are thus not stored on a per-object basis. In the upper half of Figure 9, the schema-ful part accounts for the majority of the object's fields. In the lower half of the figure, everything but the order's primary key resides in the "schema-less" part, where the name, data type, and data value of each field are stored per object. Last but not least, the figure also shows how the nested data (line items) appears in each case.

5.2 Columnar Format

Due to AsterixDB's focus on analytical use cases, a columnar format was added as its new default format. The goal was to achieve the well-known benefits of columnarization for analytics, but to do so for JSON and without requiring the user to provide a schema. The benefits include enabling encoding as well as the ability to avoid reading columns that are not needed for a given analytical query [8]. Figure 10 and Figure 11 provide an overview of the inner workings of AsterixDB's unique binary columnar data representation, called AMAX [11], for efficiently storing and supporting analytics over collections of wide JSON documents.

Figure 10 shows how AMAX infers and separates the schema information for a collection's objects from the objects' values in AsterixDB's LSM-based storage world. When new data is added to the primary index, it first goes into the index's in-memory LSM component. Then, when that component fills, it will be flushed and become the index's newest immutable disk component. During this phase of the component's LSM lifecycle, when its data is flushed, the schema of its contents are observed and recorded just once for the newly created component. Also, at the same time, the component's data content will be transformed into AMAX's generalized columnar JSON format, as indicated in the bottom left part of the figure. Note that the "columns" are essentially paths through the data object and that, similar to the schema-ful part of an AsterixDB row object, the field name and type information are not repeated per object; rather, they appear just once per component, and only the data values themselves are stored per object.

AMAX differs from similar formats like Dremel [31] (which it extends), its open source clone Parquet, and its BigQuery descendent Capacitor [32]. In addition to supporting nesting, AMAX supports heterogeneity (via union types, needed for the discount field in the example) and does not require a schema. Observing every object by camping out on the LSM lifecyle enables the system to infer an exact schema (as opposed to an approximate one, as might be obtained by sampling). Moreover, as was shown experimentally in [10, 11], this is done with very little overhead since the objects are observed at the point when the system is transforming them and writing them to a newly created on-disk LSM component.

Figure 11 provides a more detailed look at the columnar LSM B+ tree structure of an AMAX-based primary index. The tree's interior nodes are essentially the same as for the row format, but the tree's leaves differ – the tree has mega leaf nodes, made up of Page 0 – the first page of a mega leaf node – followed by a series of megapages,

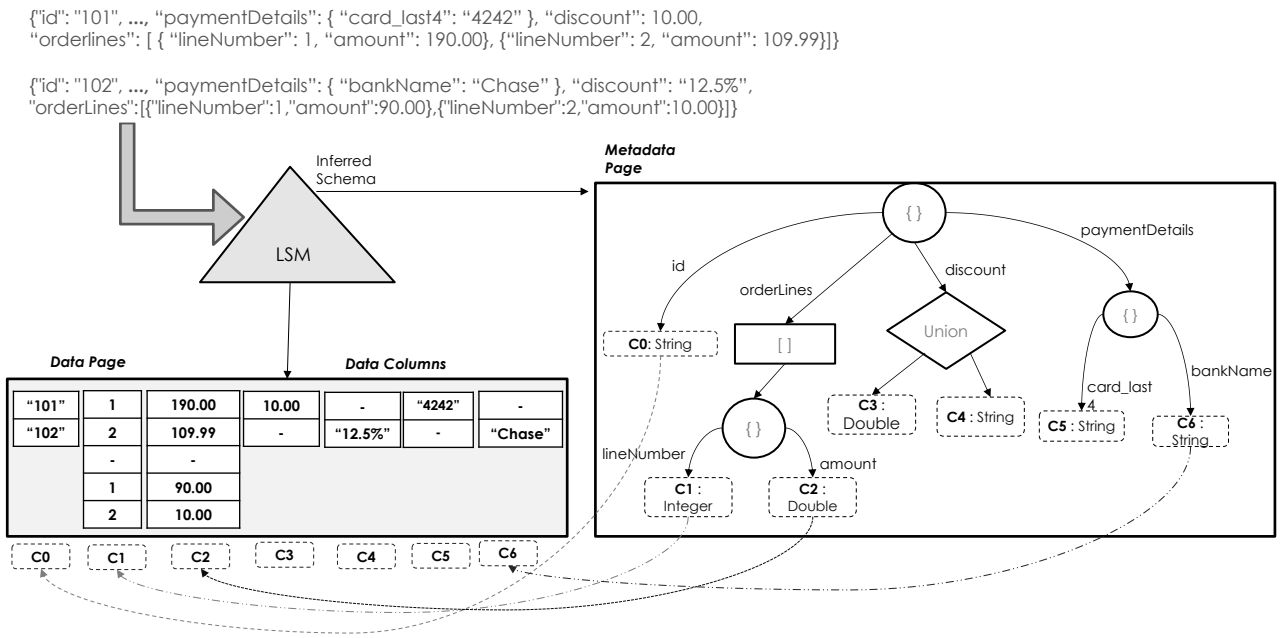

Figure 10: Columnar Format and LSM Schema Inference.

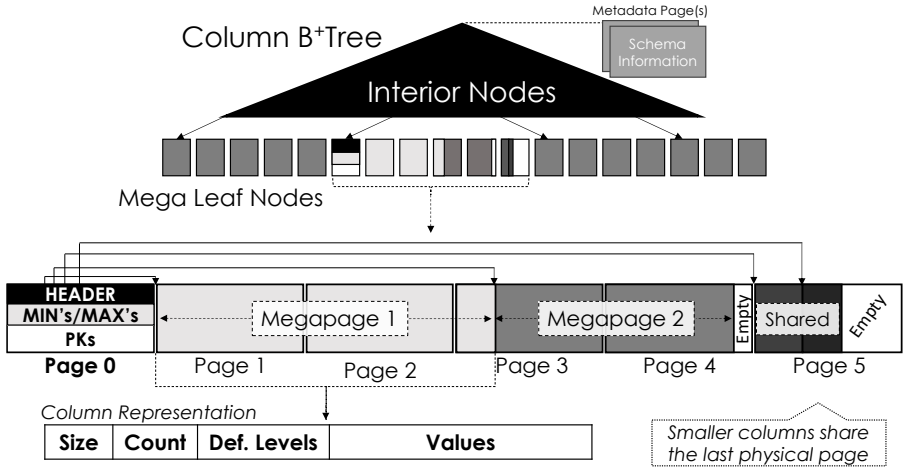

Figure 11: Columnar LSM B+ Tree Structure.

which are where the columnar representation for batches of objects reside. Page 0 stores the primary keys for the batch's objects, serves as a directory for the megapages, and also has min/max information that is used to avoid unnecessary reads. Each megapage contains the batch's data values for a given column, with smaller columns sharing the last physical page to avoid potential fragmentation. Figure 11 shows the structure of each megapage, which stores the **size** of the megapage (in bytes), the **count** of values, the values' **definition levels**, and the data **values** themselves. It is worth noting that AMAX uses only the **definition levels** (no **repetition levels** as in Dremel [31]) to determine a value is `present`, `NULL` or `MISSING`, as well as the start and the end of each repeating value when storing columnar arrays' values [11].

6 CLOUD-BASED STORAGE MANAGEMENT

The most significant changes in the database storage world over the last decade, arguably, have been the migration of database management services to the cloud and the rise of cloud object stores (such as AWS S3) that offer virtually unlimited capacity and highly available data storage. AsterixDB has responded to these changes by (1) separating its (query) compute from its storage, (2) using object storage as the underlying source of truth for data, and (3) incorporating cache management techniques that allow AsterixDB to also offer virtually unlimited capacity.

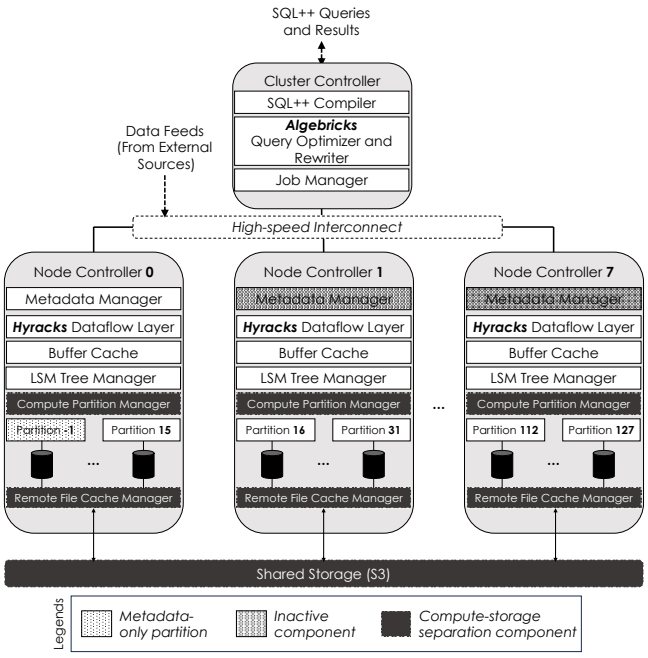

Figure 12: Apache AsterixDB Cluster (Shared Storage).

6.1 Compute-Storage Separation

Figure 12 shows AsterixDB's storage subsystem when deployed in its newer "cloud mode". Comparing Figure 12 with Figure 6 reveals that architecturally, AsterixDB is essentially a shared-nothing/shared-storage hybrid system. Data still lives in AsterixDB's internal storage format(s), but now it is backed by cloud object storage, and it is physically accessible from all of the compute nodes in a shared-storage-like sense. However, at any given moment, the system actually runs in shared-nothing mode, with reading and writing from and to a given storage partition being the responsibility of exactly one compute node. This responsibility can be reassigned in the event of failures or to scale the system's query processing power up or down without having to move data. As shown, data storage is now over-partitioned – each collection is statically divided into 128 partitions by default – with each compute node being responsible for managing its assigned fair share of the partitions.

The decision to rely on object storage for the data at rest with compute-storage separation was driven by a mixture of cost and availability considerations. Cost-wise, the non-ephemeral storage options offered by cloud providers (e.g., AWS EBS) are expensive. Ephemeral (per compute instance) storage provides better performance than EBS, and essentially for free, but its data disappears when instances are turned off. The use of object storage (e.g., AWS S3) to persist the data, while using ephemeral storage as a cache, provides better performance and lower cost. Availability-wise, cloud object storage has the benefit of providing very high availability. AsterixDB's approach to compute-storage separation was inspired by Snowflake [23]; space precludes a more detailed comparison.

AWS S3, Google Cloud Storage (GCS), and Azure Blob Storage are the current flavors of object storage that can be used, depending on the cloud platform on which the service is deployed. To minimize the latency between the compute nodes and remote storage, each compute node has a local NVMe cache with limited capacity, as discussed above. When it nears capacity, a novel eviction policy will be triggered (more on this soon). The policy selectively evicts infrequently accessed data, prioritizing the retention of data that is frequently accessed by the workload. If there is no space available in the local cache for hot data, reads and writes will occur directly against the remote object storage. When space becomes available in the local cache (e.g., if a hot data collection is dropped), frequently accessed data will be re-cached when it is next read from the remote object storage.

6.2 Scaling in the Cloud

Figure 13 and Figure 14 provide a high-level view of how AsterixDB's compute-storage separated architecture enables a cluster to be scaled up (or down) to adjust the query performance delivered by the cluster. The upper portion of Figure 13 shows a two-node cluster with two compute partitions per node. The lower portion shows the 128 partitions[†] in object storage, and the arrows show how they have been assigned, in a balanced manner, to the cluster's four compute partitions. Let us suppose that users are dissatisfied with the cluster's current query performance. If they would like queries to be twice as fast, as we will see in Section 7, they can double the size of the cluster to achieve their goal.

Figure 14 shows the resulting four-node cluster, again with two compute partitions per node. The upper portion of the figure shows the new cluster, and the middle of the figure shows how the 128 storage partitions of the system's data have been re-assigned, again in a balanced manner, to the cluster's eight compute partitions. Thanks to compute-storage separation, the scaling operation need only notify the compute nodes of their new responsibilities. No storage-level data movement is involved; the compute nodes will simply re-cache any data in NVMe that they subsequently access after the scale-up operation completes.

6.3 Remote Cache Management

The remote file caching mechanism uses a four-tiered approach to prioritize storage artifacts when the data being operated on exceeds the NVMe cache capacity. Figure 15 shows the tiers: Tiers 0, 1, 2, and 3. Each on-disk storage artifact is placed into one of the four tiers. The tiers are logical classifications and do not affect the cache's underlying storage structure (which is the same as the on-disk structure in AsterixDB's shared-nothing mode). Note that the figure only shows two storage partitions (out of 129 in total), namely Partition -1 and Partition 0. Partition -1 is a dedicated partition for cluster metadata (including the catalog) and it is classified as Tier 0. Partition 0 is a data partition (as are the other 127 partitions) that stores the partitions of collections and secondary indexes. In Figure 15, we see a columnar collection called MyCollection – with four columns stored in four megapages M1, M2, M3, and M4 – and a secondary index on the column values stored in M1 called MyIndex. MyCollection and MyIndex each have two on-disk LSM components labeled as their C1 and C2 components in the figure.

For the collection MyCollection, we can see that the storage artifacts of both of its components (C1 and C2) are spread across

[†]Partition -1 is a special metadata-related storage partition that is used to hold the AsterixDB system catalog.

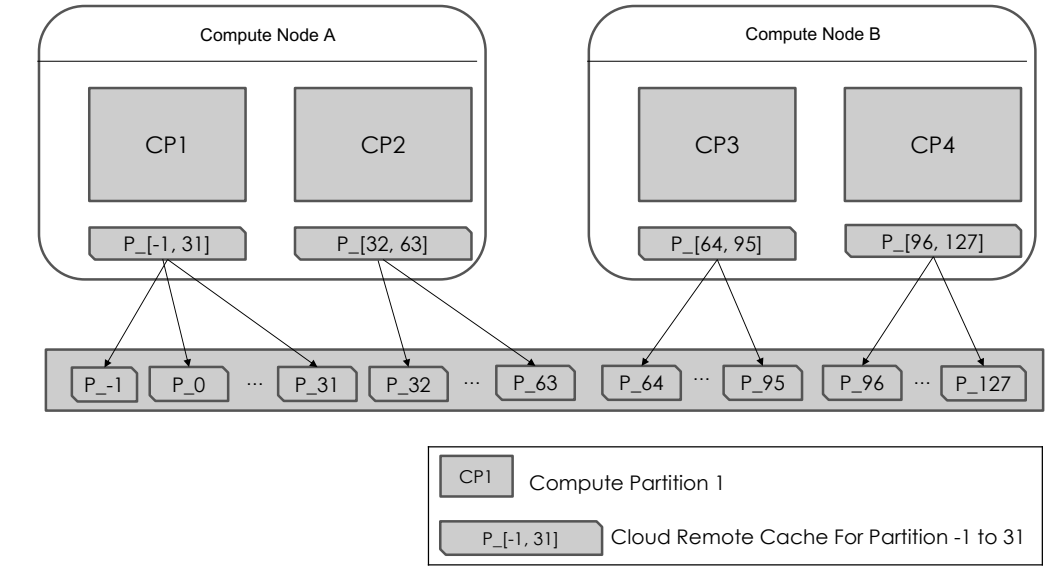

Figure 13: Pre-Scaling Partition Assignments.

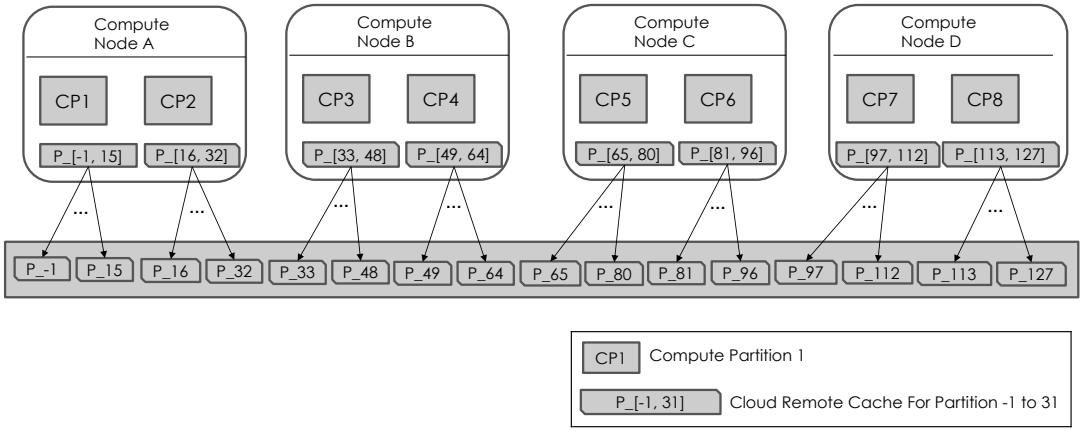

Figure 14: Post-Scaling Partition Assignments.

the four tiers. In Tier 0, we place its metadata file — a small file that describes the collection — and its checkpoint files. In Tier 1, we place the bloomfilter files (e.g., C1 Bloomfilter, see Section 4) as well as the compression files (e.g., C1 Compression) that store offset-length pairs for each compressed data page when a component's on-disk file has compression enabled (which it will by default). Finally, in Tiers 2 and 3, we can see that the B+ Tree component files (e.g., C1 B+ Tree) are placed into those two tiers. For the secondary index MyIndex, all of its files reside entirely in Tier 1.

With that background, we are now ready to describe each of the tiers shown in Figure 15 along with its characteristics:

Tier 0 (Metadata Class) is dedicated to handling all metadata files, which include the entirety of Partition -1, as well as each declared collection's metadata files and checkpoint files. Tier 0 files are never evicted, as they are small and they are essential to operating on collections.

Tier 1 (File Class) is dedicated to handling all other files except columnar format B+ tree indexes. Storage artifacts for row-format collections (none shown in the figure) reside in Tier 0 (for their metadata and checkpoint files) and Tier 1 (for their component data files). Secondary index-related files — as shown for MyIndex in Tier 1 in Figure 15 — also reside in Tier 1. Tier 1 files will be evicted whenever a collection (or secondary index) becomes inactive for an extended period of time (six hours by default).

Tiers 2 and 3 are together dedicated to handling the storage artifacts related to columnar B+ trees. As an example, in Figure 15 we see that the C1 B+ tree file is divided between these two tiers. That is, parts of a columnar B+ tree file are classified as Tier 2 and the rest as Tier 3.

Tier 2 (Guide Class) is dedicated to handling the interior nodes of columnar B+ trees, as well as their Page 0's (see Figure 11). Also, any/all secondary-indexed columns (such as M1 in Figure 15) are classified as Tier 2 (more on this shortly). Tier 2 content will only

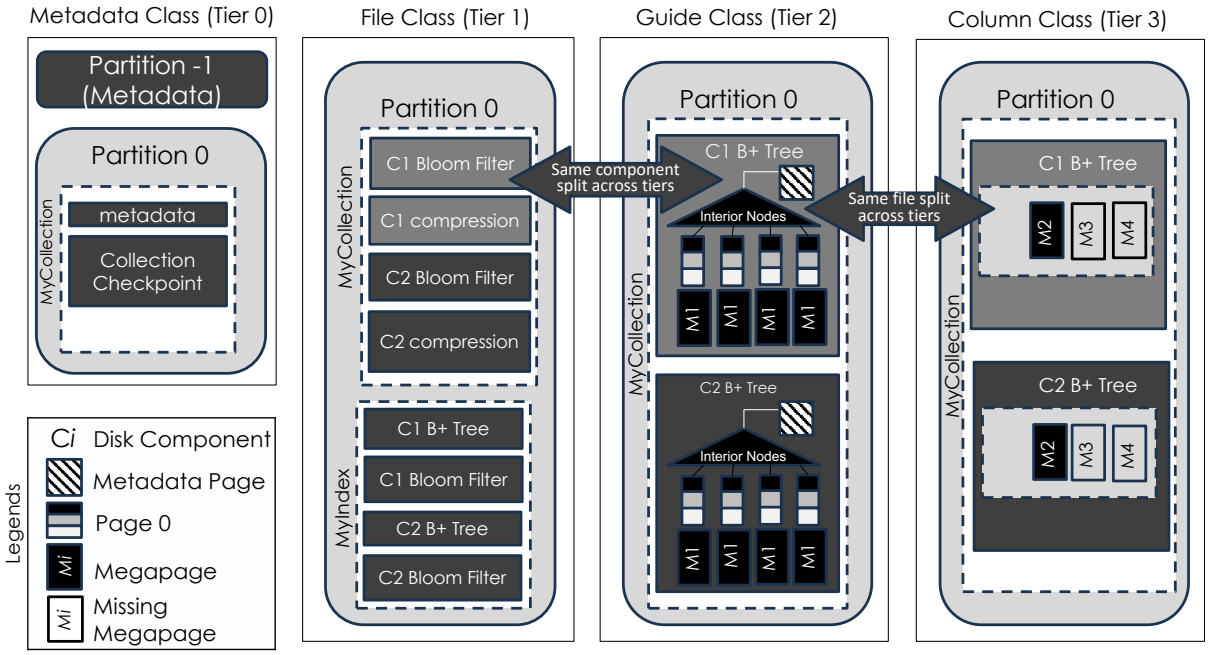

Figure 15: Tiered NVMe Caching.

be evicted if its Tier 1 content has been evicted, as the upper and Page 0 (guiding) portions of a columnar B+ tree are always needed when operating on it.

Tier 3 (Column Class) is dedicated to handling columnar B+ tree megapages (unless their column is indexed). Megapages for columns are usually the first to be considered for eviction when the cache's storage space becomes nearly full.

If a secondary index exists on a column (e.g., M1 in Figure 15), those columns will be "upgraded" to Tier 2, so they will not be evicted unless their collection is evicted — i.e., unless warranted by Tier 1 eviction. The reason for this is that indexed columns must be accessed frequently to maintain their secondary indexes. The other unindexed columns (M2-M4) will be evictable first if the cache's disk space becomes too tight.

When an AsterixDB cluster is resumed (after a pause), the local disks of the cluster nodes will initially be empty. Upon resuming, all files in Tier 0 and Tier 1 will be proactively cached from shared storage first. The content of those two tiers is usually small and does not occupy much space compared to the actual data. In contrast, the content for Tier 2 (columnar B+ trees' interior nodes, Page 0's, and indexed columns) will not be re-cached until they are once again activated. As for Tier 3's content, megapages are cached only when they are accessed explicitly, as illustrated in the case of M2 of the C1 B+ Tree in Figure 15.

When new data is ingested, all flushed and merged LSM components will be written both remotely and locally (write-through) if the local disk space permits. When space is pressured (based on a certain occupancy threshold), the remote file cache manager devises a plan for evicting data from the cache. It starts by looking for all inactive collections and indexes (Tier 1) that have not been accessed for a while, and it evicts them along with their Tier 2 contents. If disk space is still too tight, the remote file cache manager iterates over each active columnar collection and finds candidate megapages to evict (Tier 3) — based on factors such as their last access time and access frequency — and evicts them. If space pressure persists, the remote file manager will become more "aggressive" about evicting megapages. Note that when it considers a column for caching or eviction, it does so for all components of the column's collection. As an example, in Tier 3 in Figure 15, megapage M2 is being cached for both C1 and C2, while megapages M3 and M4 are not, which could be a case where both M3 and M4 were either not cached or were evicted.

7 PERFORMANCE AND SCALABILITY

This section provides some empirical performance results regarding the behavior of the AsterixDB storage system. Its scalability in several different dimensions is explored first. The impact of the choice of static over-partitioning is then briefly examined, as is the relative performance of its older row format and the new column format. Note that this is *not* a benchmark of different systems; the goal is to explore characteristics of the AsterixDB storage system owing to the design decisions that were made. (A competitive storage-level benchmark would be difficult, as benchmark workloads are driven at the query API level, which would end up comparing systems' whole stacks rather than just storage-level differences.)

The experiments here are based on borrowing the analytical query workload from a recent benchmark – namely CH2++ [18]. CH2++ is designed to evaluate JSON data platforms with respect to support for modern operational and analytical workloads. It extends the CH benchmark for relational HTAP systems [22] by asking "what might the CH benchmark schema have looked like if its commerce application and its analytical queries had been designed for today's document database world?" Differences from

CH's schema (itself being based on merging the schemas of TPC-C and TPC-H) and its 22-query workload include the natural application of nesting and arrays as well as the addition of 64 additional columns, untouched by the queries, that represent today's often wider schemas for several of the benchmark's main collections [18]. We are not aware of other available benchmarks that would be suitable for evaluating system performance for JSON analytics.

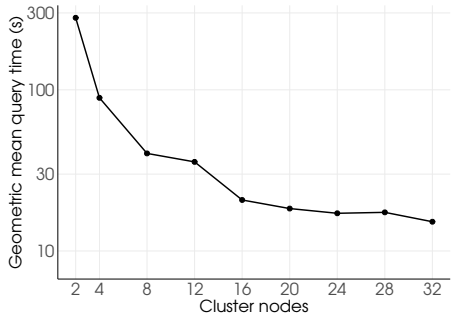

Figure 16: CH2++ Query Speedup (GeoMean).

We ran the CH2++ queries on an EC2 cluster in the AWS cloud. Since Apache AsterixDB is the engine under the hood of Couchbase's Enterprise Analytics (EA) product, and we had a readily available CH2++ based performance suite for it, we ran the benchmark against an EA cluster. The cluster consisted of 2 to 32 nodes with 8 vCPUs and 32 GB of memory; each node had a 474 GB NVMe local disk. Each node provides 8 compute partitions. We used an AWS c5.24xlarge instance with 96 vCPUs, 192 GB of memory, 100 GB gp3 storage and up to 25 Gbps of network bandwidth to run the client driver. We ran CH2++ at a scale factor of 4000 warehouses using the columnar format (except where otherwise specified).

Due to the wide variations in the complexity and thus performance of the 22 analytical queries, CH2++'s query performance is reported in terms of the geometric mean ("power") of the response times of the benchmark's queries when they are looped and run in a sequence. The order of execution for the 22 queries of CH2++ follows the list of permutations specified in the original TPC-H specification [37] (from which the queries in CH2++ are derived).

We begin by experimentally examining the *speedup* of the system. Figure 16 shows how the query performance for the benchmark relates to the cluster size when the overall data size is fixed, i.e., when more nodes are being enlisted to handle the same data size (hopefully more quickly). The trend that we observe here is that the system provides essentially linear speedup, particularly at cluster sizes that are powers of two. Thus, the system's query performance can be doubled by doubling the size of its cluster. For the non-power-of-two cluster sizes we see slight performance variations due to the fact that those sizes necessarily lead to slight imbalances in the number of partitions per node. Finally, on the far right in the figure we see that the additional benefit of adding nodes eventually wanes for the volume of data at this CH2++ scale factor.

Next, we examine the system's *batch scaleup*. Figure 17 plots query performance versus cluster size when the data per node is fixed so that the overall database size increases as nodes are added to the cluster. Each point's annotation shows its ratio of compute partitions to storage partitions (CP:SP). When the number of nodes

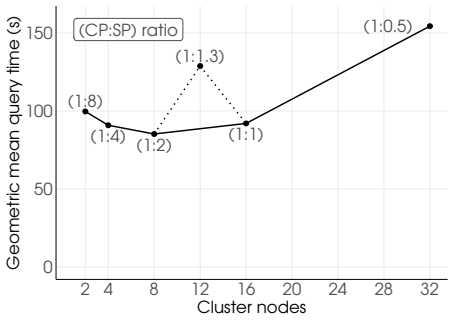

Figure 17: CH2++ Batch Scaleup (GeoMean).

is a power of two, the system is able to deliver linear scaleup up until the point (at 16 nodes) where each compute partition has just one associated storage partition. In that range, the system can query twice the data volume with the same performance if the cluster size is doubled. Performance then worsens at 32 nodes because the data scan parallelism is still 128-way (since only half the compute partitions have data) whereas the rest of the query runs with 256-way parallelism. The extra coordination and data-passing is all pain and no gain in this case. Note that workloads that aren't dominated by data scans would continue to scale beyond 16 nodes. The performance observed for the 12-node cluster underscores the importance of avoiding non-integral CP:SP ratios when sizing a cluster's compute partitions. (For example, if the number of storage partitions for the 12-node cluster had been 192 instead of 128, the geometric mean query time would have been 88s, which is comparable to the neighboring times in the graph.)

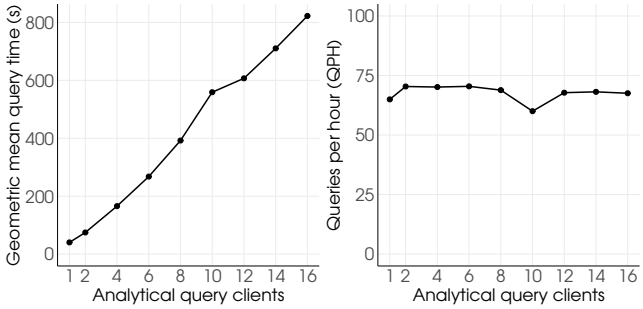

Figure 18: CH2++ Concurrency Scaleup.

In a different dimension of scalability, Figure 18 shows how the system's performance is affected when the number of concurrent users (modeled by running concurrent CH2++ query loops) is increased. The things to notice here are: (1) Since the system's CPU and I/O resources are shared across users, increasing the number of concurrent users naturally leads to a proportional slowdown in terms of the query response times per user, as shown by the graph on the left. (2) The query throughput delivered (in terms of queries per hour, or QpH) is not negatively affected by the increase in the user load, as the graph on the right shows.

Figure 19 shows how the CH2++ query performance on our 4-node cluster – with its 32 compute partitions – is affected by setting the number of storage partitions to a value other than the default of 128. Doubling the number of storage partitions provides a 2x performance gain as long as there is at least a 1:1 ratio of compute

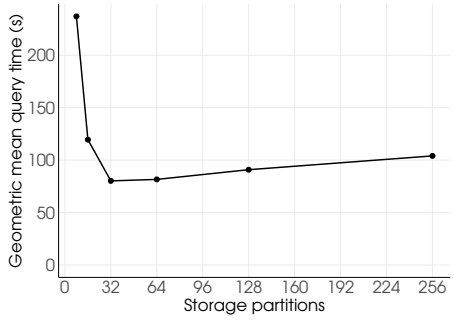

Figure 19: CH2++ Storage Partition Impact.

partitions to storage partitions. Beyond that point (> 32 storage partitions) the additional storage partitioning doesn't help. The scan parallelism is capped due to having only 32 compute partitions to do the scanning.

Storage Format	Storage Size	Geo. Mean Query Time	Data Ingest Time
Column (native)	789 GB	79 secs	53 mins
Row (native)	1000 GB	252 secs	44 mins
Column (external Parquet)	412 GB	167 secs	-

Table 1: CH2++ Object Format Performance (4 nodes).

Lastly, turning to the choice of storage formats, Table 1 shows the storage sizes and query and ingestion performance of AsterixDB's row and columnar formats. As expected, the columnar format performs significantly better for queries than the row format for the CH2++ benchmark, as its data size is smaller and (much more importantly) the reading of unnecessary columns is avoided. As another point of comparison, the table also shows the query performance obtained if the same data is resident in S3 but resides there *externally* in Parquet files (as might be the case for a data lake). This scenario was tested by storing the CH2++ benchmark data in files in S3 and defining the benchmark's collections to be external AsterixDB collections (which are queried on demand at runtime). This comparison may interest readers familiar with Parquet, as they may have been wondering how effective AsterixDB's columnar format is relative to Parquet. From this test we see that the external Parquet performance beats AsterixDB's natively managed row format. For this test the collections' schemas were left open (i.e., no predeclared schemas) in order to evaluate the negative impact of the schema-less row representation, so the inferiority of the row format is not a big surprise. Note that ingesting data is faster with the row format, as the columnar format does more restructuring when components are flushed to disk. We also see in the table that AsterixDB's natively managed columnar format outperforms the use of externally stored Parquet files.[‡] Additional format-related experiments, both with additional object formats and a variety of other data sets, can be found in [10].

[‡]Table 1's relative size results stem from the facts that (1) the 64 extra columns of CH2++ have UUIDs as values, taking up significant hard-to-compress space, and (2) Parquet's use of dictionary compression, which the AsterixDB columnar format compression trick bag does not yet include, seems to be effective for CH2++ data.

8 REFLECTIONS

Before closing the paper, this section briefly highlights a few of the most significant lessons that we learned over the years as the AsterixDB storage system evolved.

First, the use of LSM-based storage and indexing yielded more advantages than we had anticipated. The immutability of LSM components was ideal for transitioning to cloud object stores, as they don't support in-place updates. Also, the LSM lifecycle provided excellent "air cover" for tasks such as incremental schema inference for AMAX or incremental statistics gathering [9].

Second, our columnar storage format had the expected performance and space benefits without needing user-provided schemas. However, AMAX turns out to rely on collections having sensible implicit schemas. As discussed in [19], there are certain JSON anti-patterns that can show up in collections in the wild, including "Anti-Pattern 2, Data as Metadata", where every object has a unique id as a *field name* rather than as a *field value*. When this unfortunate pattern is seen, e.g., if an application naively persists a JavaScript hash map as JSON, the collection's schema (in terms of its field names) can grow without bound. While this is easily handled by our row format, it can choke AMAX (which gracefully accommodates "only" a few thousand fields). Future work is needed to develop a hybrid "mostly columnar" approach to handle such cases.

Third, with the move to cloud storage, we had to add significantly more error-handling and retry logic to the I/O level of the system when handling NVMe cache misses or writing new LSM components to "disk". Despite our best efforts to stay below the cloud provider's object storage request rate limits, we still encountered unexpected rate limiting errors and sporadic connection failures.

Fourth, we chose a statically over-partitioned storage path to separate compute from storage. As we saw, this works well when the ratio of storage to compute is a power of two, yielding good load balancing. It works less well when the assignment of storage partitions to a cluster's compute nodes is uneven, and the (default) choice of 128 partitions can be limiting. In the future we expect to explore a more dynamic data partitioning scheme, such as [30].

9 CONCLUSION

In this paper, we have described how Apache AsterixDB's storage subsystem has evolved over the past dozen or so years. Much has changed in the computing environment since the early days of the project, including the move from magnetic storage to SSDs, the migration of database services to the cloud, and the availability and popularity of cloud object storage. We have described how AsterixDB storage has responded, including the addition of a column-oriented JSON storage format with incremental schema inference, compute-storage separation with object store backing, and memory and NVMe cache management policies designed to support multi-user workloads over databases with many LSM-based collections and secondary indexes. We also presented performance results to demonstrate the scalability of the system and the performance benefit of its columnar data format. Today, the system is in quite a few customers' hands, as Apache AsterixDB is the basis for the Capella Analytics and Enterprise Analytics products from Couchbase.

REFERENCES

[1] Amazon Web Services 2024. *Amazon DynamoDB.* Amazon Web Services. Retrieved November 26, 2024 from https://aws.amazon.com/dynamodb/

[2] Couchbase, Inc. 2025. *About Capella Columnar.* Couchbase, Inc. Retrieved February 2, 2025 from https://docs.couchbase.com/columnar/intro/intro.html

[3] Apache Software Foundation 2025. *Apache AsterixDB.* Apache Software Foundation. Retrieved January 25, 2025 from https://asterixdb.apache.org/index.html

[4] Couchbase, Inc. 2025. *Couchbase Server.* Couchbase, Inc. Retrieved January 25, 2025 from https://www.couchbase.com/products/server/

[5] 2025. *Introducing JSON.* Retrieved January 25, 2025 from https://www.json.org/json-en.html

[6] MongoDB, Inc. 2025. *Introduction to MongoDB.* MongoDB, Inc. Retrieved January 25, 2025 from https://www.mongodb.com/docs/manual/introduction/#introduction-to-mongodb

[7] Microsoft, Inc. 2025. *Queries in Azure Cosmos DB for NoSQL.* Microsoft, Inc. Retrieved January 25, 2025 from https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/query/

[8] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[9] Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 841–855. https://doi.org/10.1145/3183713.3183761

[10] Wail Y. Alkowaileet. 2022. *Towards Analytics-Optimized Document Stores.* Ph.D. Dissertation. University of California, Irvine, USA. https://www.escholarship.org/uc/item/0hq0s31w

[11] Wail Y. Alkowaileet and Michael J. Carey. 2022. Columnar Formats for Schemaless LSM-based Document Stores. *Proc. VLDB Endow.* 15, 10 (2022), 2085–2097.

[12] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *PVLDB* 7, 14 (2014), 1905–1916. http://www.vldb.org/pvldb/vol7/p1905-alsubaiee.pdf

[13] Sattam Alsubaiee, Alexander Behm, Vinayak R. Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *PVLDB* 7, 10 (June 2014), 12.

[14] Apache Software Foundation. 2026. Apache HBase. https://hbase.apache.org Accessed: 2026-02-27.

[15] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have Your Data and Query It Too: From Key-Value Caching to Big Data Management. In *Proc. 2016 Int'l. Conf. on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 239–251.

[16] Michael Carey. 2019. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. In *2019 IEEE 35th Int'l. Conf. on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12.

[17] Michael Carey, Don Chamberlin, Almann Goo, Kian Win Ong, Yannis Papakonstantinou, Chris Suver, Sitaram Vemulapalli, and Till Westmann. 2024. SQL++: We Can Finally Relax!. In *2024 IEEE 40th Int'l. Conf. on Data Engineering (ICDE)*. 5501–5510.

[18] Michael Carey, Vijay Sarathy, Daniel Nagy, Bo-Chun Wang, Keshav Murthy, M. Muralikrishna, Peeyush Gupta, and Till Westmann. 2026. CH2++: New HOAP for Benchmarking JSON Data Analytics. In *Performance Evaluation and Benchmarking - 17th TPC Technology Conference, TPCTC 2025, London, England, September 2025 (Lecture Notes in Computer Science)*, Raghunath Nambiar and Meikel Poess (Eds.). Springer.

[19] Michael J. Carey, Wail Y. Alkowaileet, Nick Digeronimo, Peeyush Gupta, Sachin Smotra, and Till Westmann. 2025. Towards Principled, Practical Document Database Design. *Proc. VLDB Endow.* 18, 12 (2025), 4804–4816. https://doi.org/10.14778/3750601.3750606

[20] Don Chamberlin. 2018. *SQL++ For SQL Users: A Tutorial.* Couchbase, Inc. https://www.couchbase.com/content/analytics/sql-book

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI '06)*. USENIX Association, 205–218.

[22] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems* (Athens, Greece) *(DBTest '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1988842.1988850

[23] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[24] Siying Dong, Andrew Kryczka, YanQin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–28.

[25] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *Proc. 2022 USENIX Annual Technical Conf., USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 1037–1048.

[26] Murtadha Al Hubail, Ali Alsuliman, Wail Y. Alkowaileet, Michael Blow, Michael J. Carey, Savyasach Enukonda, Peeyush Gupta, Santosh Hegde, Kamini Jagtiani, Abhishek Jindal, Nawazish Kahn, Mehnaz Tabassum Mahin, Ian Maxon, M. Muralikrishna, Keshav Murthy, Daniel Nagy, Preetham Poluparthi, Ankit Prabhu, Ritik Raj, Vijay Sarathy, Shahrzad Shirazi, Utsav Singh, Hussain Towaileb, Ayush Tripathi, Janhavi Tripurwar, Bo-Chun Wang, and Till Westmann. 2025. Cloudy With a Chance of JSON. *Proc. VLDB Endow.* 18, 12 (2025), 4938–4950. https://doi.org/10.14778/3750601.3750618

[27] Murtadha Al Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. 2019. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.* 12, 12 (Aug 2019), 2275–2286.

[28] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Murtadha Al Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, Chen Luo, Ian Maxon, and Pouria Pirzadeh. 2020. Robust and efficient memory management in Apache AsterixDB. *Softw. Pract. Exp.* 50, 7 (2020), 1114–1151.

[29] Chen Luo. 2020. *On Optimizing LSM-based Storage for Big Data Management Systems.* Ph.D. Dissertation. University of California, Irvine, USA. https://www.escholarship.org/uc/item/5x65p2dr

[30] Chen Luo and Michael J. Carey. 2022. DynaHash: Efficient Data Rebalancing in Apache AsterixDB. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 485–497. https://doi.org/10.1109/ICDE53745.2022.00041

[31] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[32] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[33] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[34] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL databases. *CoRR* abs/1405.3631 (2014).

[35] Pramod Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistnce.* Addison-Wesley, USA.

[36] S. Haji Amin Shirazi. 2025. *Query Optimization Techniques for Big Active Data.* PhD thesis. University of California, Riverside, USA.

[37] Transaction Processing Performance Council (TPC). 2022. *TPC Benchmark H (Decision Support) Standard Specification Revision 3.0.1.* Technical Report. Transaction Processing Performance Council. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf Available at https://www.tpc.org/tpch/.