

# M2: An Analytic System with Specialized Storage Engines for Multi-Model Workloads

Kyoseung Koo  
Seoul National University  
Seoul, Republic of Korea  
koo@db.snu.ac.kr

Bogyeong Kim\*  
Seoul National University  
Seoul, Republic of Korea  
bgkim@db.snu.ac.kr

Bongki Moon  
Seoul National University  
Seoul, Republic of Korea  
bkmoon@snu.ac.kr

## Abstract

Modern data analytic workloads increasingly require handling multiple data models simultaneously. Two primary approaches meet this need: polyglot persistence and multi-model database systems. Polyglot persistence employs a coordinator program to manage several independent database systems but suffers from high communication costs due to its physically disaggregated architecture. Meanwhile, existing multi-model database systems rely on a single storage engine optimized for a specific data model, resulting in inefficient processing across diverse data models. To address these limitations, we present *M2*, a multi-model analytic system with integrated storage engines. *M2* treats all data models as first-class entities, composing query plans that incorporate operations across models. To effectively combine data from different models, the system introduces a specialized inter-model join algorithm called *multi-stage hash join*. Our evaluation demonstrates that *M2* outperforms existing approaches by up to 188× speedup on multi-model analytics, confirming the effectiveness of our proposed techniques.

## 1 Introduction

Unlike traditional workloads based on a single data model, modern data analytic workloads increasingly require the ability to handle multiple data models simultaneously. In the fields of machine learning and artificial intelligence, in particular, workloads typically involve processing array data for matrix computations or linear algebra algorithms together with structured and semi-structured data. Consider, for example, an e-commerce recommendation system that builds a non-negative matrix factorization model [18]. In this scenario, order and review data, both stored in a document-oriented model, are combined to construct a recommendation matrix that represents rating values for products by customers. This matrix is then fed into the factorization algorithm to generate recommendations. To extract insights such as identifying products that a customer would like, the reconstructed matrix is converted into a relational format for further analysis using relation operations. This data analysis pipeline requires handling three distinct data models, namely, relational, document-oriented, and array models. There are many use cases reported in the literature for combining an array data model with structured and semi-structured data models [10, 17, 37].

One way of processing such a multi-model analytic workload is to rely on a polyglot persistence system. Polyglot persistence enables us to adopt multiple database systems simultaneously for the processing of complex workloads [14]. In this approach, each database system is responsible for processing data in its own data

model, while a coordinator module typically orchestrates query execution across them. Unfortunately, however, the coordinator and the underlying database systems are physically disaggregated. Consequently, a substantial amount of communication overhead is incurred inevitably to transfer intermediate and final result data among the coordinator and the database systems [8, 15]. The overhead becomes aggravated as the volume of data involved in the workload increases.

Another approach has emerged that focuses on handling multiple data models within a single database system, which can be referred to as a single-engine multi-model database system [20]. Such a multi-model database system employs a single storage engine dedicated to a particular data model and attempts to accommodate all the supported data models with the storage engine. AgensGraph, for example, utilizes PostgreSQL to support its query processing for data not only in the relational model but also in the graph model. Evidently, this single storage engine approach suffers from non-trivial performance degradation due to the lack of direct support from the most relevant data model to each individual operation in a query plan. Several works reported that using non-native storage engines to process a data model shows inferior execution time compared to those with native storage engines [15, 16, 34].

This paper presents a prototype of a new multi-model database system called *M2* that can plan and execute a multi-model query with multiple underlying storage engines. *M2* addresses the limitations intrinsic to the polyglot persistence and single-engine multi-model database systems by tightly integrating multiple storage engines together. If a query involves operations defined in two or more distinct data models, then the query plan is segmented into partitions by models. Each query partition is then processed efficiently by a storage engine dedicated to and optimized for the corresponding data model. The current implementation of *M2* supports query processing across the relational, document-oriented, and array data models but can be extended to include additional data models such as a graph model. *M2* can be considered a true *multi-engine multi-model* database system as it treats every supported data model as a first-class entity with equal priority.

Figure 1 depicts the system architecture of *M2*. The system runs on two storage engines: an augmented version of DuckDB [23] for relational and document-oriented data and PreVision [16] for array data. DuckDB is an open-source relational database system, and we have augmented it with additional functionalities for document-oriented data processing and interoperability with *M2*. PreVision offers efficient multi-dimensional array processing capabilities with I/O optimized storage management. These two storage engines are tightly integrated, and they *share a common buffer pool* to improve the utilization of buffer space. Furthermore, a dedicated bridge module is added to facilitate the cooperation of the two storage

\*The authors contributed to this work when they were affiliated with Seoul National University

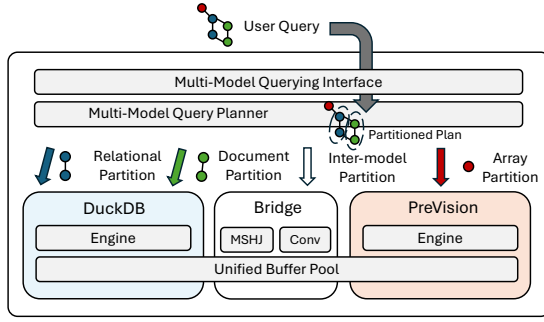


Figure 1: Architectural overview of M2

engines. This bridge module is responsible for data conversion and transmission between them for inter-model operations such as *multi-stage hash join* that will be described later.

The remainder of this paper is organized as follows. Section 2 describes the data models and operations currently supported by the system. Section 3 discusses the query execution flow and the unified buffer pool. Section 4 presents the multi-stage hash join algorithm. An experimental evaluation is demonstrated in Section 5, and related literature is summarized in Section 6. Lastly, Section 7 concludes this work.

## 2 Data Model and Operation

This section outlines the supported data models (Section 2.1) and the operations available for each data model, as well as the inter-model operations that work across different data types (Section 2.2).

### 2.1 Model Definition

We formally define the data models used in M2, beginning with their fundamental elements (record, document, and cell) and extending to their respective sets (relation, collection, and array).

**Definition 1 (Record).** A record  $r$  is a list of attribute values with a fixed number of attributes. That is,  $r = \{v_1, v_2, \dots, v_n\}$ , where  $v_i$  is the  $i$ -th attribute value and  $n$  is fixed.

**Definition 2 (Relation).** A relation  $R$  is a set of records where all records share the same number of attributes and consistent attribute types. That is,  $R = \{r_1, r_2, \dots, r_m\}$ , where  $\forall r_j \in R$ , the number of attributes  $n$  is fixed, and the type of each attribute remains consistent.

A record represents a row in a relational database table, and a relation corresponds to a table containing multiple rows with the same schema. A value type can be a list: it is often called an array, but this paper distinguishes those as the array model is defined later.

**Definition 3 (Document).** A document  $d$  is a single-attribute record whose value stores a list of key-value pairs. That is,  $d = \{(a_1, v_1), (a_2, v_2), \dots, (a_n, v_n)\}$ , where  $a_i$  is a key,  $v_i$  is the corresponding value, and  $n$  is the number of key-value pairs. Within a document, any attribute value may itself contain another document, allowing for nested structures.

**Definition 4 (Collection).** A collection  $C$  is a set of documents, defined as  $C = \{d_1, d_2, \dots, d_m\}$ .

Documents represent JSON objects in document databases, where each object contains flexible key-value pairs. For instance, the document below includes two key-value pairs. The second pair includes a “geometry” field with a nested document. Other documents in the same collection may or may not contain identical fields.

```
{
  "id": 1015,
  "geometry": {
    "type": "Point",
    "coordinates": [126.952511, 37.449527]
  }
}
```

**Definition 5 (Cell).** A cell  $c$  is a record in which a subset of the attributes represents dimensions. That is,  $c = \{a'_1, a'_2, \dots, a'_d, a_1, a_2, \dots, a_n\}$ , where  $a'_i$  is the  $i$ -th dimension value,  $d$  is the number of dimensions,  $a_i$  is the  $i$ -th non-dimension attribute value, and  $n$  is the number of non-dimension attributes. All dimension values are represented as unsigned integers.

**Definition 6 (Array).** An array  $A$  is a set of cells arranged in a  $d$ -dimensional structure, defined as  $A = \{c_1, c_2, \dots, c_m\}$ . The size of  $A$  is defined as  $AS = (l_1, l_2, \dots, l_d)$ , where each  $l_i = \text{MAX}(a'_i \in c)$ ,  $c \in A$  represents the length of the  $i$ -th dimension. Each cell  $c \in A$  is uniquely identified by its dimension attributes.

Cells are records containing both values and their dimensional coordinates. These coordinates may be conceptual rather than explicitly stored in the physical layout. For instance, in M2’s array storage engine, PreVision, cell coordinates in dense arrays are derived from the cell’s spatial location rather than stored explicitly. Arrays are multi-dimensional data such as vectors, matrices, or raster data used in machine learning and scientific computing. Each array maintains explicit size information for each dimension, as this is essential for determining the output shapes of array operations.

### 2.2 Interface and Operation

Listing 1 demonstrates how M2 can be used to predict the top 10 favorable products for a given customer during targeted marketing campaigns. The query begins by accessing order and review collections which are stored in a document-oriented model (Line 1). A document in the order collection includes a customer identifier, while a review document contains identifiers for an order and a product, and a rating value. To obtain rating values for each customer and product, these collections are merged using the join function, followed by the project function to filter only relevant dimensions and rating values (Line 2). Before being fed into the non-negative matrix factorization algorithm, the set of rating values is converted into a matrix by the `toArray` function with names for dimensions and a value (Line 3). By using the number of tuples in customer and product tables and the `rand` function, the factor matrices  $W$  and  $H$  are initialized (Lines 5-7), and the algorithm updates these matrices (Lines 8-10). To focus the analysis on specific customer-product pairs of interest (for targeted marketing campaigns), the interest table containing customer and product identifiers is opened (Line 12). Once the recommendation matrix is reconstructed (Line 13), the join function takes interest which is in a relational model and filled in an array model, producing equi-joined relational data as an output (this inter-model join operation will be discussed later). Among the joined rating values, the ones with the customer id = 3 are selected, and the top ten items

**Listing 1: Example pseudocode for illustration**

```

1 order, review = openCollection('order'), openCollection('review')
2 ratings = review.join(order, 'review.order_id =
   order.order_id').project('customer_id, product_id, rating')
3 X = ratings.toArray({'customer_id', 'product_id'}, {'rating'})
4
5 rank, num_iter = 10, 3
6 customer_cnt, product_cnt = openTable('customer').count(),
   openTable('product').count()
7 W, H = rand({customer_cnt, rank}), rand({rank, product_cnt})
8 for i in range(num_iter):
9   W = W * ((X @ H.T) / (W @ H @ H.T))
10  H = H * ((W.T @ X) / (W.T @ W @ H))
11
12 interest = openTable('interest')
13 filled = W @ H
14 result = filled.join(interest, RELATIONAL).filter('customer_id =
   3').sort('rating DESC').limit(10)
15
16 execute(result)

```

are extracted (Line 14). Finally, the execute function is called to get results from the described query above (Line 16).

As depicted in the example, an *M2* query consists of a series of operations, including model-specific operations and inter-model operations (Table 1 displays supported operations). Model-specific operations take a specific model as input and produce the same model as output. For the relational model, common relational operations, which can be expressed using Structured Query Language (SQL), are provided. For the document model, in addition to the fundamental relational operations, *M2* provides the dot processing functionality for accessing values in nested documents and supports the unwind operation to unnest lists, which are commonly used in semi-structured data analysis. The array model supports operations for linear algebra and raster processing.

In contrast to the model-specific operations, inter-model operations process data across model boundaries. Model conversion operations transform data from one model into a target model, and inter-model joins combine data from different models for integrated analysis. To precisely specify the semantics of the inter-model join in *M2*, we provide a formal definition.

**Definition 7 (Inter-model join).** *An inter-model join is a binary operation that combines two records from different models when they satisfy a given condition, producing results in one of the input models. That is,  $R \bowtie S = \{f(r \cup s) \mid r \in R \wedge s \in S \wedge g(r, s) \wedge m(R) \neq m(S)\}$ , where  $R$  and  $S$  are sets of records,  $f$  is a projection function to produce the output model,  $g$  is a join condition function, and  $m$  maps the given argument to its associated model.*

For example, in our earlier e-commerce analysis, we joined a table with an array using the join function, matching tuples and cells based on equal identifiers. Inter-model joins differ from traditional relational joins in two key aspects. First, they operate on heterogeneous data models rather than a single model, requiring distinct access methods for each model type. Second, the output can be one of the input models. Note that the cross-model join supported by Polypheny-DB [36] can be seen as an inter-model join with the fixed projection function producing a relational model.

Currently, *M2* utilizes two methods for inter-model joins: a *multi-stage hash join* for array-involving equi-joins and a method relying on a relational join operation for the rest. We will discuss the multi-stage hash join approach in Section 4.

**Table 1: Provided operations**

Model	Operation
Relational	Filter, Project, Sort, Limit, Aggregate, Union, and Join
Document	Filter (with dot processing), Project, Sort, Limit, Aggregate, Union, Join, and Unwind
Array	Element-Wise Arithmetic, Matrix Multiplication, Transposition, Aggregation, Window, Sub-Array, Spatial Join, and Array Build
Inter-Model	Model Conversion and Join

## 3 Query Processing with Integrated Engines

### 3.1 Query Execution Flow

When a user submits a multi-model query through *M2*'s querying interface, the system processes it through several stages. First, the query planner interprets the query, performing binding operations to create a logical plan represented as a directed acyclic graph (DAG). This logical plan consists of nodes and edges representing operations and producer-consumer relationships between nodes, respectively, capturing all operations across different data models and their relationships. The plan is then partitioned based on the data models involved. Each partition contains operations specific to a particular model (relational, document, or array) that will be executed by the corresponding storage engine. DuckDB handles both relational and document operations<sup>1</sup>, while PreVision processes array computations. A bridge module sits between the execution engines, handling inter-model conversions and join operations.

**Partitioning.** Given a logical plan, *M2* constructs a partition DAG comprised of partitions and their dependencies to group model-specific logical operations. A partition contains nodes with the same data model, and the only edges connecting the nodes. Inter-partition edges preserve the direction of the underlying operation graph, connecting partitions that produce data to their consumers.

*M2* uses a very simple heuristic approach to partition a logical plan. The query planner constructs partitions bottom-up; it combines compatible partitions into coarse-grained units. The partitioning process begins by creating an initial partition DAG where each operation resides in its own partition. The planner then performs bottom-up merging, starting from partitions with no producers. For each such partition, the planner traverses the partition graph using depth-first search. During traversal, it evaluates potential merging opportunities between the current partition and its consumers, performing merges when the following conditions allow:

- (1) Both partitions must contain operations of the same model.
- (2) The merged partition must have exactly one output node that does not have consumer nodes belonging to the partition, thus ensuring query executability.
- (3) The merging operation must not create cycles in the graph.

Once traversal completes, the resulting merged partitions are sorted topologically to determine the execution order between partitions. The ordered partitions are now ready for submission to their respective execution engines.

<sup>1</sup>We enhanced DuckDB with VelocityPack [3], a JSON serialization format, and added specialized JSON processing operations to optimize document processing in DuckDB.

**Query Submission.** *M2* constructs queries for storage engines from partitions using the querying interface provided by each engine. Since PreVision’s querying system is modeled as a DAG similar to *M2*’s approach, *M2* simply transforms its plan to PreVision’s query graph before submission. DuckDB requires a different approach, as its query shape is tree-based rather than DAG-based. To achieve this transformation, *M2* uses a simple materialization approach to decompose a DAG-based partition into a tree structure for proper DuckDB query submission.

When transforming a DAG query into a tree structure for the relational engine, *M2* begins by identifying nodes without producers and traverses the graph bottom-up from these starting points. During traversal, *M2* searches for nodes that have more than one consumer, as these nodes create the DAG structure. Upon finding a node with multiple consumers, *M2* detaches the entire subtree rooted at that node and stores it in a tree list maintained throughout query execution. *M2* then creates aliased nodes to represent this subtree and connects each aliased node to the original consumers. At the end of the traversal, the graph is ultimately transformed into a tree structure containing both regular and aliased nodes.

For query execution, *M2* submits each constructed query tree to the relational engine sequentially. First, *M2* processes the trees from the tree list, converting each into a DuckDB query tree. Since the roots of these trees originally had multiple consumers, their results are materialized. Any nodes that consume these materialized results are configured to reference them appropriately. Finally, *M2* submits the main tree to DuckDB, completing the query execution.

### 3.2 Unified Buffer Pool

The storage engines in *M2* are integrated into a unified buffer pool to optimize memory utilization. In the original versions of DuckDB and PreVision, each storage engine has its own buffer space, requiring dedicated allocations of the host’s memory. Since neither storage engine supports dynamic buffer space adjustment during query runtime, their buffer sizes must be determined before query execution. This limitation leads to lower buffer pool utilization by each storage engine, resulting in a poor buffer-hit ratio.

The buffer utilization issue becomes particularly problematic when an array model is involved. Consider the non-negative matrix factorization algorithm producing a significant amount of intermediate matrices [16]. The matrices take a large portion of buffer space, leading to buffer pool exhaustion. It results in high-volume disk spills, which are aggravated when the buffer size is limited (this trend is observable in Section 5). Thus, it is beneficial to allow storage engines to utilize the entire buffer space.

The unified buffer pool of *M2* provides a memory allocator and buffer pool utility functions. The memory allocator offers allocation functions such as `malloc` and `free`, and all buffer allocation requests are redirected to this allocator.

The utility functions include two essential buffer management operations: the `Add` function registers allocated objects with the buffer pool, and the `Evict` function requests the buffer pool to clear cached objects (under the LRU policy) until the requested amount of free memory space becomes available. Additionally, the buffer pool requires a storage engine to implement two callback functions:

`isEvictable` and `doEviction`. The `isEvictable` function determines whether a buffer object can be removed from the buffer pool (e.g., by checking that no readers are currently referencing the object). The `doEviction` function handles cleanup of the buffer object (e.g., writing array data to disk in the array file format). When the `Evict` function selects a buffered object for removal, it first calls the `isEvictable` function, and if the object is evictable, it then calls the `doEviction` function.

The buffer managers of DuckDB and PreVision are modified to attach to the unified buffer pool. The buffer managers of the engines invoke the `Add` and `Evict` functions when a buffer object is created and when insufficient space exists in the buffer pool, respectively. The engines register their callback functions with the unified buffer pool, enabling it to skip the currently used buffer objects and properly handle cleanup.

## 4 Multi-Stage Hash Join

In multi-model analytic workloads, data joins between an array model and another model (relation or collection) occur frequently. For example, in a benchmark for multi-model analytic tasks [15], four out of six array-involving tasks demonstrate joins between the array and another model. These cases utilize *spatial join*, which matches dimension values in the array model with attribute values in the other model.

A naive approach to performing such joins employs relational join operations, which necessitates converting array input data into relational format. This conversion incurs significant data copying costs, a problem that worsens as array volumes increase, given their inherent multi-dimensional nature with huge amounts of cells.

Array processing and management systems typically adopt tiled storage architectures to leverage data access locality [16]. Unlike traditional heap files in relational storage engines, which store records unordered, tiled storage organizes cells based on their coordinates, partitioning the array into rectangular tiles that function as pages or blocks. However, existing relational join operations do not exploit the spatial locality of tiled storage, leading to suboptimal performance with numerous data copies. Hence, a join method should leverage the array format structure to utilize the spatial locality to improve performance while minimizing data copies.

To improve performance by fully harnessing spatial locality, we propose *multi-stage hash join (MSHJ)*, a binary inter-model equi-join method that combines arrays and other models in their native storage, eliminating costly data transformation. *MSHJ* optimizes array access patterns to minimize disk I/O and, in some cases, produces output without accessing all array data. The joined result is produced in one of the input models.

Algorithm 1 describes the *MSHJ* procedure between array and relational data models, taking data from both models as input. Let  $AS$  denote the array size,  $TS$  the tile size, and  $D$  the number of dimensions. For clarity, we use a relation as the non-array input model, but a join between a collection and an array can be performed in the same way, as documents in collections can also contain dimension attributes.

The initial step of *MSHJ* is reordering relation records to align with the tiled structure. This reordering is necessary because evaluating join conditions between unordered relation records and array

**Algorithm 1:** *MSHJ* between a relation and an array

---

```

MSHJ (Relation : R, Array : A)
1  S = R
   // Building phase
2  foreach d ∈ {1, ..., D} do
3      Bd = {0i | size(Bd) = ⌈ASd/TSd⌋}
4      foreach r ∈ S do
5          i = fd(vd ∈ r)
6          Bid.add(r)
7      end
8      S = Bd
   end
   // Probing phase
9  foreach r ∈ S do
10     TC = (f1(v1), ..., fD(vD))
11     CC = (v1 mod TS1 + 1, ..., vD mod TSD + 1)
12     if t.TC ≠ TC then
13         // Get a tile if new visiting tile
14         t = A.pin(TC)
15     end
16     // Get a cell and evaluate join result
17     emit(r ⋈ t.get(CC))
18 end

```

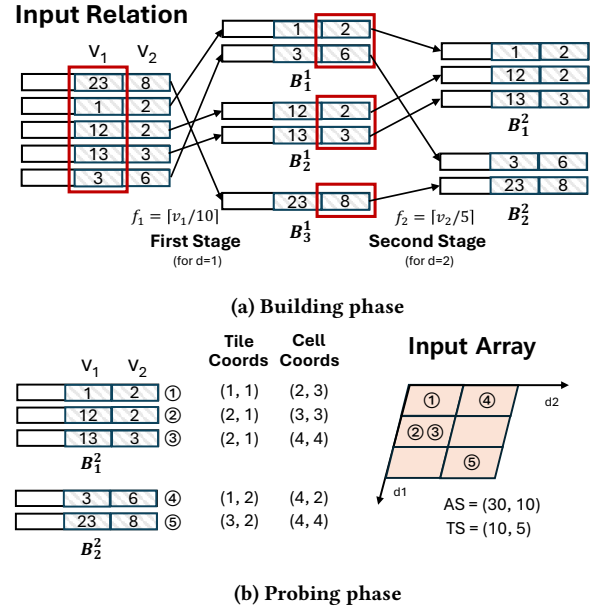
---

cells would require random access to array tiles, resulting in high disk I/O costs.

Similar to traditional hash joins, *MSHJ* operates in two phases: building and probing. However, it introduces key differences: the building phase consists of *D* sequential stages (one for each array dimension), and each stage employs a different hash function.

The building phase of *MSHJ* processes each dimension sequentially using its corresponding hash function, defined as  $f_d(v_d) = \lceil v_d / TS_d \rceil$ , where  $v_d$  is the relation attribute and  $TS_d$  is the tile size for dimension *d*. At each stage, order-preserving buckets  $B^d$  are initialized, with the number of buckets matching the number of tiles along that dimension (Line 3). The stage then scans all relation records, applying the hash function  $f_d$  to compute the target tile coordinate *i* for  $v_d$  (Line 5). Each record is inserted into the *i*-th bucket of  $B^d$  (Line 6). After each stage, the records are stored in buckets while preserving their original order. These buckets are then sequentially scanned in ascending order of *i* and fed into the next stage. This process continues through all dimensions until the final set of buckets corresponds to the number of tiles in the last dimension *D*. At the end of the building phase, records in each final bucket are ordered according to dimensions  $D-1, D-2, \dots, 1$ , similar to the radix sort.

In the probing phase, *MSHJ* sequentially scans records from each bucket in ascending order of *i*. For each record, the tile coordinates *TC* and the cell coordinates *CC* are calculated (Line 9 - Line 10). Using these coordinates, the corresponding tile *t* and cell *t*.get(*CC*) are accessed. If the tile differs from the previously accessed one, a new tile is pinned (Line 11 - Line 12). The final result is then produced by concatenating the record with the corresponding cell values (Line 13). In PreVision, tiles are stored in one of three formats: dense (cells in contiguous regions), sorted coordinate list (COO), or compressed sparse row (CSR). For dense tiles, probing uses direct



**Figure 2:** An example of the *MSHJ* procedure

cell access, whereas for sorted COO and CSR, probing is done in logarithmic time using binary search to locate matching cell coordinates. During the matching step, tiles are cached, and each tile is pinned *exactly once*. This is ensured by the bucket organization, which groups record accesses by tile coordinates.

**Example 1.** Figure 2 illustrates a working example of *MSHJ* for a two-dimensional spatial join. The input relation contains five records, each with  $v_1$  and  $v_2$  as dimension attributes. The input array has dimensions of  $30 \times 10$  with a tile size of  $10 \times 5$ .

As shown in Figure 2a, the building phase begins with bucketing records based on the first dimension ( $v_1$ ). Since the array and tile sizes in the first dimension are 30 and 10, respectively, the hash function  $f_1$  is defined as  $\lceil v_1 / 10 \rceil$ , and three buckets are initialized. Each record's  $v_1$  value is hashed to determine its bucket placement. For example, the first record with  $v_1 = 23$  is assigned to bucket  $B_3^1$  (calculated as  $\lceil 23 / 10 \rceil = 3$ ). After processing all records, the second stage initializes two buckets for the second dimension, and records are inserted while preserving the ordering established in the previous stage.

Once bucketing is complete, the probing phase begins. For each record, both tile coordinates and cell coordinates are computed. Tile coordinates are calculated as  $(\lceil v_1 / 10 \rceil, \lceil v_2 / 5 \rceil)$ , while cell coordinates within the tile are  $((v_1 \bmod 10) + 1, (v_2 \bmod 5) + 1)$ . For instance, the last record in Figure 2b yields tile coordinates of  $(\lceil 23 / 10 \rceil, \lceil 8 / 5 \rceil) = (3, 2)$  and cell coordinates of  $((23 \bmod 10) + 1, (8 \bmod 5) + 1) = (4, 4)$ . After identifying the corresponding tile, the algorithm retrieves the cell for the record's coordinates. If a valid cell exists, it is combined with the record and emitted as part of the join result.

The circled numbers on the records and tiles represent their access order. As the figure shows, the algorithm's sequential processing ensures each tile is accessed exactly once, eliminating the need for costly repeated tile buffer operations. Notably, this example demonstrates how *MSHJ* avoids scanning the entire array by accessing only

the necessary tiles, minimizing disk I/O while producing the correct output.  $\square$

With buckets implemented as hash tables, *MSH<sub>2</sub>* achieves  $O(DN)$  time complexity for the building phase, where  $D$  denotes the number of dimensions and  $N$  represents the number of records in a relation. The probing phase complexity varies depending on the tile format used on the array side. When using dense format, probing occurs in  $O(D)$  time, resulting in an overall time complexity of  $O(DN + DN) = O(DN)$ . Conversely, in PreVision, cells in sparse formats are stored in sorted order. Thus, binary searching on cell coordinates requires  $O(D \log M)$  time, where  $M$  represents the number of cells in the array side. This yields an overall time complexity of  $O(DN + DN \log M) = O(DN \log M)$ . For disk read, blocks on the table side must be read  $D$  times, while tiles on the array side are accessed only once. Consequently, the algorithm incurs  $O(DB + T)$  block accesses to read, where  $B$  denotes the number of blocks in the relation and  $T$  indicates the number of tiles in the array. The disk write cost is  $O(B)$  if a relational output is requested, while an array output incurs  $O(T)$  cost. Since  $D$  is generally small in practical applications, it can be treated as a constant factor.

Since inter-model joins are performed in the bridge component, which can access both DuckDB and PreVision, it can write join output directly to both data layouts. When the output is an array, cell writing processes for the output array also benefit from an optimized tile access pattern because input and output tiles have equal sizes in PreVision. In contrast, if inter-model joins producing array output are executed through relational joins, the join results must be converted to array format afterward.

## 5 Evaluation

We conducted an evaluation to assess the effectiveness of *M2*. All our evaluations were conducted on a desktop machine with an Intel i7-9700K CPU, 32GB RAM, and a 1TB SSD running Linux. All systems used in evaluations were configured to utilize 24GB of memory for their buffer spaces. Before each experiment, we reset both the system-specific buffer spaces and the operating system's page cache to eliminate any performance advantages from cached data. We configured the systems to utilize a single thread to see clearer performance characteristics of each storage engine. For systems using multiple storage engines, we configured them to avoid interleaving query executions and to use the same amount of buffer space.

### 5.1 Comparison with Other Systems

We used M2Bench [15], a database benchmark program for multi-model analytic workloads, as its model coverage aligns with *M2*'s supported models: relational, document-oriented, and array. From the complete M2Bench suite, we chose six tasks that require both execution engines of *M2*, ensuring a comprehensive evaluation of our multi-model architecture.

**Comparison Systems.** We selected several systems for comparison with *M2*: ArangoDB [2], AgensGraph [5], Polypheny-DB [36], a polyglot implementation provided with M2Bench [15] (M2Bench Polyglot), and another polyglot implementation developed by us (M2 Polyglot). SurrealDB [31] and OrientDB [27] were omitted

System	Polypheny-DB	AgensGraph	ArangoDB	MySQL
Version	93fbff6	2.14 <sup>1</sup>	3.12.3	9.2.0
System	MongoDB	SciDB	MonetDB	DuckDB
Version	8.0.4	19.11.5	11.53.9	1.0 <sup>2</sup>

Table 2: System versions

due to limited support for join operations, which are crucial for analytical processing.

Polypheny-DB was configured to utilize MonetDB [13] for the relational model and MongoDB [22] for the document-oriented model. Since Polypheny-DB did not support a connector for an array database system, MonetDB was used for array data processing.

M2Bench Polyglot was a hard-coded implementation specifically designed for M2Bench tasks. It utilized multiple database systems (one for each data model) with a client program. While model-specific operations were executed in their corresponding database systems, the client program handled inter-model joins and data conversions, serving as the coordination point between different data models. The M2Bench Polyglot was configured to use MySQL [25], MongoDB, and SciDB [30] for the relational, document-oriented, and array models, respectively. To observe the pure performance advantages of the *M2*'s architecture, we developed M2 Polyglot using the enhanced version of DuckDB and PreVision, each of which handled the relational and document models and the array model, respectively. Table 2 shows each system's version.

In both polyglot implementations, join operations between distinct storage engines were executed in the following ways. In array-involving joins, data from the non-array model was iterated, and the corresponding array cell was matched directly. Otherwise, the data were joined using the nested-loop method.

**Configuration.** For array tasks with iterative algorithms, we set the number of iterations to one, as it was enough to observe performance characteristics. The array data were initially chunked into about 5 megabytes tiles for the engines utilizing the tiled storage. As an exception, for workloads involving dense matrix multiplication operations, the array data for SciDB were split into  $1000 \times 1000$  tiles because SciDB imposed a size limit of 1024 for these operations. Since the benchmark includes queries for a property graph data model, which *M2* does not directly support, we handled the data with a relational storage engine. Graph data was stored in relational edge and node tables. For tasks requiring simple one-hop pattern matching, we substituted graph operations with equivalent relational selection queries on edge tables. We excluded tasks requiring more complex graph operations, such as multi-hop pattern matching or shortest path finding.

**Evaluation Results.** Figure 3 presents the execution times for benchmark tasks across all evaluated systems with scaling factor one. Following the profiling rules of M2Bench, we segment execution times by data model, shown as different colors in the stacked bars: blue for relational operations, green for document operations, and red for array operations. Inter-model operations executed by the bridge module of *M2* are represented in white. For the polyglot implementations, coordinator overhead and communication costs between the coordinator and underlying database systems are both

<sup>1</sup> AgensGraph 2.14 is based on PostgreSQL 14.

<sup>2</sup> The modified version is forked from this version.



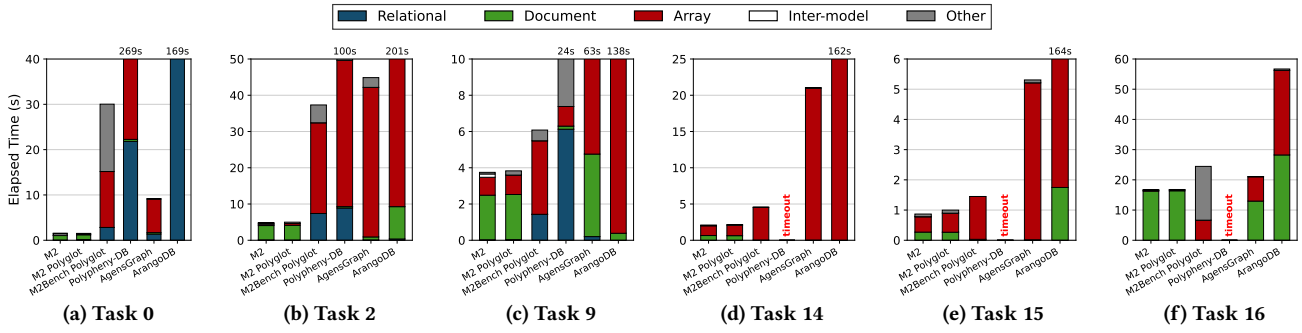


Figure 3: Overall comparison

categorized as “other.” Any remaining execution time not attributable to a specific model’s processing is also included in “other.” The overflowed bars display the elapsed times above the bars. We denote a timeout if the execution time exceeds one hour.

M2 accelerated query execution by up to 188× compared to all other systems across all benchmark tasks. Processing times for relational and array data models were consistently lower in M2 compared to the alternatives, excluding M2 Polyglot, attributed to its use of specialized engines: the DuckDB engine for relational and PreVision for array operations.

M2 delivered comparable performance to M2 Polyglot with speedups of 0.97–1.15×, attributed to using the same storage engines. Polypheny-DB and M2Bench Polyglot showed significant communication overhead by coordinator programs. In the case of Polypheny-DB, the overheads took about 20%, 50%, and 70% of the total elapsed times for Tasks 0, 2, and 9, respectively. While Polypheny-DB and M2Bench Polyglot were required to use inter-process communication between storage engines and coordinator programs, M2 Polyglot embedded DuckDB and PreVision in the same memory space, resulting in lower communication costs. This result supported our main idea: the integration of multiple specialized storage engines can effectively deliver performance while reducing communication overhead. Nevertheless, M2 Polyglot requires manual coordination between engines using their own interfaces. M2 provides an integrated interface with a unified buffer pool and *MSHJ*; their effectiveness is discussed later.

The M2Bench Polyglot and AgensGraph, which employed a row-oriented storage engine, exhibited inferior performance compared to M2 in relational processing. Polypheny-DB presented limited performance for relational processing, despite using a column-oriented relational engine. During its internal query rewriting step, Polypheny-DB split input queries into numerous sub-queries for storage engine submission, ultimately incurring significant query processing overheads. ArangoDB demonstrated poor performance when executing joins on relational datasets in Task 0, resulting in considerably longer execution times for relational operations.

For document processing, M2 did not demonstrate superior performance due to its currently partially optimized implementation. In Tasks 0, 2, and 9, M2 performed the VelocityPack serialization excessively, accounting for the majority of the elapsed time. Tasks 14 and 15 required nearest neighbor searches, but the lack of optimized geospatial indexing led to lengthy document processing

times, with geospatial searches consuming 95% of the document processing time. In Task 16, the scanning and filtering of a huge document collection accounted for almost all of the total elapsed time. As M2 is a prototype system still under development, we will further optimize the document processing capability of the system. ArangoDB showed inferior performance in document processing for Tasks 2, 15, and 16. In Task 2, the materialization of intermediate results was time-intensive, while in Tasks 15 and 16, index searching operations accounted for a significant portion of the execution time.

M2Bench Polyglot demonstrated inefficient array processing in Tasks 0, 2, 9, and 16. In Task 16, which involves spatial joins between a document collection and an array, SciDB’s performance suffered due to random access of the array data. In the rest of the tasks, linear algebra operations dominated the array processing requirements, resulting in poor performance by the array engine. Polypheny-DB exceeded the timeout threshold in Tasks 14, 15, and 16. Most of each execution time was spent processing window operations which are the bottleneck operations in the tasks. ArangoDB and AgensGraph stored array data in COO format, necessitating numerous join operations to match cell coordinates during processing. This led to poor performance for array-intensive tasks in these systems.

**Varying Scaling Factor.** To evaluate scalability, we conducted assessments using the M2Bench tasks with varying scaling factors. Figure 4 shows the performance results of multi-model solutions across all tasks, with the y-axis representing elapsed times on a logarithmic scale. Overall, elapsed times increased as the scaling factor went up, except for Task 9, which remained unaffected by the scaling factor. Except for M2 Polyglot, M2 consistently outperformed the other systems at higher scaling factors. Notably, M2 showed superior performance compared to M2 Polyglot at Task 2 with higher scaling factors. In Task 2, the non-negative matrix factorization algorithm was computed, producing a large volume of intermediate results. M2 was equipped with a unified buffer pool, offering better buffer space utilization compared to M2 Polyglot, where buffer pools for storage engines were physically separated. This difference resulted in fewer disk spills by M2, showing better performance. Polypheny-DB experienced out-of-memory errors in Task 0 with scaling factors over five. It also incurred timeouts in Tasks 14, 15, and 16 due to query times exceeding one hour.

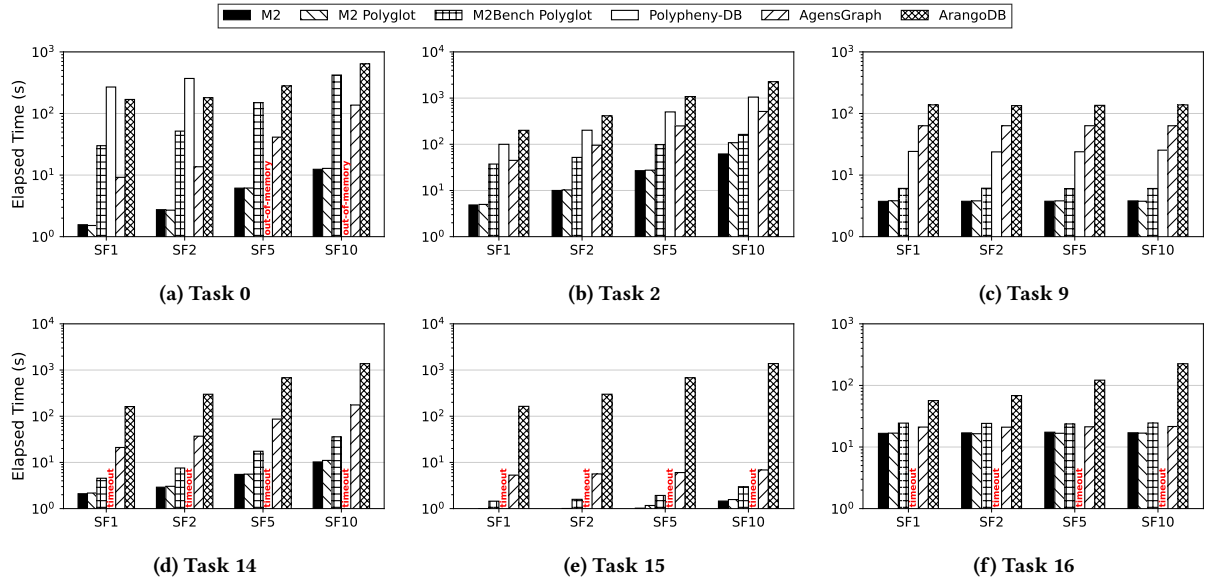
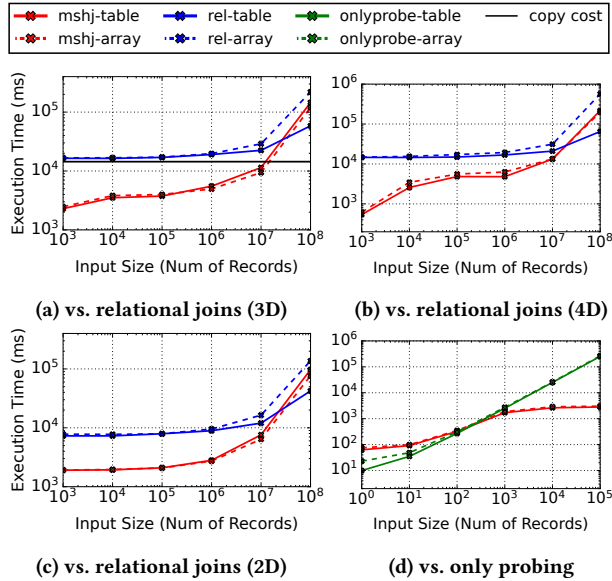


Figure 4: Performance with varying scale factors

Figure 5: Performance comparison for *MSHJ*

## 5.2 Evaluation for Multi-Stage Hash Join

While the overall system evaluation in Section 5.1 demonstrates *M2*'s superior performance, it is difficult to identify the effectiveness of *MSHJ* due to the small overhead of inter-model queries in the evaluation. To directly investigate the effectiveness of *MSHJ*, we conducted a focused comparison between *MSHJ* and two alternative approaches for relational-array joins: an indirect method using conventional relational joins and a naive probing-only approach in the bridge component. For the relational join method, conversion of array data to relational format was necessary before applying

standard relational join operations. If the output model was specified as an array, it transformed the result relation back to the array. The probing-only approach was implemented to demonstrate the impact of tile access optimization in *MSHJ*. This method skipped the bucket building phase from *MSHJ*.

All data used in evaluations were synthetically generated. Arrays were initialized with cells containing a double attribute, with the following specifications: a 2D array of size  $10000 \times 10000$  with a tile size of  $1000 \times 1000$ , a 3D array of size  $500 \times 500 \times 500$  with a tile size of  $50 \times 50 \times 50$ , and a 4D array of size  $100 \times 100 \times 100 \times 100$  with a tile size of  $10 \times 10 \times 10 \times 10$ . All arrays were initialized in dense format to avoid confusion from cell distribution. Relations were prepared with varying record counts ranging from 1 to 100 million. Each relation was initialized with dimension attributes matching the domains of the array, plus one double attribute serving as a payload.

Figure 5 illustrates the performance comparison between *MSHJ* and the other join methods as the number of records on the relational side varies. In the results, “mshj”, “rel”, and “onlyprobe” prefixes represent the performance of *MSHJ*, relational joins, and our implemented naive approach, respectively. The suffixes “table” and “array” indicate the output model of the join results.

Figure 5a shows that *MSHJ* outperformed relational joins when processing fewer than approximately 30 million records, while relational joins with relation output achieved superior performance with larger record volumes. This performance crossover occurred because *MSHJ* required multiple scanning processes during its bucketing phase. As record numbers increased, these building costs grew proportionally, impacting overall performance. Meanwhile, the relational join approach incurred significant copy costs when converting arrays to relations (represented by the black line in the figure). When array output was required, relational joins faced additional conversion overhead to transform results back to array



format, further affecting their performance. The output format had a negligible impact on *MSHJ* performance. The experimental results in Figure 5b and Figure 5c demonstrate trends similar to those in the 3D array experiments, confirming *MSHJ* maintains its performance resilience even as the number of processing stages increases.

To evaluate the tile access optimization of *MSHJ*, we compared it with the probing-only version under buffer-exhaustive conditions where only one input array tile could be buffered in memory. Figure 5d presents the performance comparison between these methods using the 3D array. The probing-only approach performed faster with fewer than about 200 records, but *MSHJ* demonstrated superior performance in all other cases. With small record counts, the random access costs for tiles in the probing-only method did not exceed *MSHJ*'s bucketing phase overhead. However, as record numbers increased, random access costs became increasingly dominant, significantly degrading the probing-only method's performance. *MSHJ* maintained consistent performance patterns across both buffer-exhaustive and non-buffer-exhaustive environments, highlighting its I/O efficiency regardless of memory constraints.

## 6 Related Work

As analytic workloads increasingly span multiple data models, various approaches have emerged to meet these demands. Multi-model database systems handle multi-model workloads within a single database system [20]. Polyglot persistence combines different specialized database systems to cope with workloads with multi-model data [14]. Data lakes [6, 32] serve as centralized repositories for storing heterogeneous data, usually in open formats like Apache Parquet [33]. Lakehouse architecture [4] combines the scalability of data lakes with the reliability of data warehouses.

Research has extensively explored in processing multiple data types. Multicategory [35] demonstrates a category-theoretic approach to multi-model query processing. AIDA [7] and HADAD [1] optimize hybrid complex queries that combine relational and linear algebra operations. A line of studies integrates diverse query interfaces using a unified intermediate representation (IR) to optimize query performance. PyTond transforms NumPy and Pandas queries into an IR and generates SQL from it to efficiently perform the query in the relational database system [28]. Weld converts library operations into WeldIR and provides a code-generation runtime to execute its IR efficiently [26]. SDQL introduces a functional collection programming language and executor for hybrid relational algebra and linear algebra workloads with semi-right dictionaries [29].

Fundamental approaches for joining multi-dimensional data include grid files [24] and spatial hash joins [19]. However, these methods cannot be directly applied to inter-model joins between relational and array data structures.

Partitioning and optimization for query graphs have been widely studied. A prominent research direction is acyclic partitioning for graphs, which aims to partition vertices into blocks without cycles among them [12]. Several works have applied graph partitioning and optimization techniques to queries structured as graphs [9, 21]. However, none of these approaches are directly applicable to the graph query optimization framework employed in our system.

M2Bench [15] is a comprehensive benchmark designed to evaluate multi-model database systems that support relational, document-oriented, array, and property graph data models. UniBench [38] focuses on e-commerce workloads and supports relational, key-value, graph, XML, and JSON formats. BigBench [11] initially supported relational and document models for big data analysis and was later extended to include the key-value model. In this paper, we used M2Bench because UniBench and BigBench do not support the array model.

## 7 Conclusion and Future Work

This paper presents *M2*, a multi-model analytic system with specialized storage engines for relational, document-oriented, and array data models. The *M2* query plan is partitioned by data models and processed in storage engines optimized for each specific model. *M2* features two key techniques: *multi-stage hash join*, which handles inter-model joins commonly found in multi-model workloads, and *unified buffer pool*, which integrates buffer pools of the storage engines to utilize more buffer space.

Developing *M2* represents the first step toward a standalone multi-model database system that treats all data models as first-class entities and processes them using optimized storage engines. Throughout the development, we explored fundamental approaches to query execution and operation optimization for multi-model query processing. Several opportunities remain for future work, including sophisticated query optimization techniques, multi-model query languages, and further optimization of inter-model join and conversion operations.

## References

- [1] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. 2021. HADAD: A Lightweight Approach for Optimizing Hybrid Complex Analytics Queries. In *Proceedings of the 2021 International Conference on Management of Data*. Xi'an, Shaanxi, China, 23–35.
- [2] ArangoDB, Inc. 2025. *ArangoDB: Multi-Model Database for Your Modern Apps*. Retrieved August 4, 2025 from <https://www.arangodb.com/>
- [3] ArangoDB, Inc. 2025. *arangodb/velocypack: A fast and compact format for serialization and storage*. Retrieved August 4, 2025 from <https://github.com/arangodb/velocypack>
- [4] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research*. Online, 1–8.
- [5] Bitnine Global Inc. 2025. *AgensGraph*. Retrieved August 4, 2025 from <https://bitnine.net/agensgraph/>
- [6] Mandy Chessell, Ferd Scheepers, Nhan Nguyen, Ruud van Kessel, and Ron van der Starre. 2014. *Governing and Managing Big Data for Analytics and Decision Makers*. IBM Redbooks.
- [7] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. 2018. AIDA: abstraction for advanced in-database analytics. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1400–1413.
- [8] Adam Dziedzic, Aaron J. Elmore, and Michael Stonebraker. 2016. Data transformation and migration in polystores. In *2016 IEEE High Performance Extreme Computing Conference*. Waltham, MA, USA, 1–6.
- [9] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN computation graph using graph substitutions. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2734–2746.
- [10] Reza Ferdousi, Reza Safdari, and Yadollah Omid. 2017. Computational prediction of drug-drug interactions based on drugs functional similarities. *Journal of Biomedical Informatics* 70 (2017), 54–64.
- [11] Ahmad Ghazal, Todor Ivanov, Pekka Kostamaa, Alain Crolotte, Ryan Voong, Mohammed Al-Kateb, Waleed Ghazal, and Roberto V. Zicari. 2017. BigBench V2: The New and Improved BigBench. In *33rd IEEE ICDE Conference*. Anaheim, CA, USA, 1225–1236.
- [12] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. 2017. Acyclic Partitioning of Large Directed Acyclic Graphs. In *17th IEEE/ACM*

- International Symposium on Cluster, Cloud and Grid Computing*. Madrid, Spain, 371–380.
- [13] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.
  - [14] Felix Kiehn, Mareike Schmidt, Daniel Glake, Fabian Panse, Wolfram Wingerath, Benjamin Wollmer, Martin Poppinga, and Norbert Ritter. 2022. Polyglot data management: state of the art & open challenges. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3750–3753.
  - [15] Bogyong Kim, Kyoseung Koo, Undraa Enkhbat, Sohyun Kim, Juhun Kim, and Bongki Moon. 2022. M2bench: a database benchmark for multi-model analytic workloads. *Proceedings of the VLDB Endowment* 16, 4 (2022), 747–759.
  - [16] Kyoseung Koo, Sohyun Kim, Wonhyeon Kim, Yoojin Choi, Juhee Han, Bogyong Kim, and Bongki Moon. 2024. PreVision: An Out-of-Core Matrix Computation System with Optimal Buffer Replacement. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–25.
  - [17] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37.
  - [18] Daniel D. Lee and H. Sebastian Seung. 2000. Algorithms for non-negative matrix factorization. In *Proceedings of the 14th International Conference on Neural Information Processing Systems*. Denver, CO, USA, 535–541.
  - [19] Ming-Ling Lo and Chinya V Ravishanker. 1996. Spatial hash-joins. In *Proceedings of the 1996 International Conference on Management of Data*. Montreal, Canada, 247–258.
  - [20] Jiaheng Lu and Irena Holubová. 2019. Multi-model databases: a new journey to handle the variety of data. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–38.
  - [21] Ruben Mayer, Christian Mayer, and Larissa Laich. 2017. The TensorFlow Partitioning and Scheduling Problem: It’s the Critical Path!. In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. Las Vegas, NV, USA, 1–6.
  - [22] MongoDB, Inc. 2025. *MongoDB: The World’s Leading Modern Database* | MongoDB. Retrieved August 4, 2025 from <https://www.mongodb.com/>
  - [23] Hannes Mühleisen and Mark Raasveldt. 2025. *duckdb: DBI Package for the DuckDB Database Management System*. <https://r.duckdb.org/> R package version 1.1.3.9029, <https://github.com/duckdb/duckdb-r>.
  - [24] Jürg Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)* 9, 1 (1984), 38–71.
  - [25] Oracle. 2025. *MySQL*. Retrieved August 4, 2025 from <https://www.mysql.com/>
  - [26] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A common runtime for high performance data analytics. In *8th Conference on Innovative Data Systems Research*. Chaminade, CA, USA.
  - [27] SAP. 2025. *OrientDB Document & Graph Database*. Retrieved August 4, 2025 from <https://orientdb.dev/>
  - [28] Hesam Shahrokhi, Amirali Kaboli, Mahdi Ghorbani, and Amir Shaikhha. 2024. Pytnd: Efficient python data science on the shoulders of databases. In *40th IEEE ICDE Conference*. Utrecht, Netherlands, 423–435.
  - [29] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–33.
  - [30] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The architecture of SciDB. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*. Portland, OR, USA, 1–16.
  - [31] SurrealDB Ltd. 2025. *The ultimate multi-model database* | SurrealDB. Retrieved August 4, 2025 from <https://surrealdb.com/>
  - [32] Ignacio Terrizzano, Peter Schwarz, Mary Roth, and John E. Colino. 2015. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *7th Conference on Innovative Data Systems Research*. Asilomar, CA, USA.
  - [33] The Apache Software Foundation. 2025. *Apache Parquet*. Retrieved August 4, 2025 from <https://parquet.apache.org/>
  - [34] Anthony Thomas and Arun Kumar. 2018. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2168–2182.
  - [35] Valter Uotila, Jiaheng Lu, Dieter Gawlick, Zhen Hua Liu, Souripriya Das, and Gregory Pogossians. 2021. MultiCategory: multi-model query processing meets category theory and functional programming. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2663–2666.
  - [36] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2018. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *2018 IEEE International Conference on Big Data*. Seattle, WA, USA, 3364–3373.
  - [37] Jinhong Xian, Dongsong Sun, Wenjing Xu, Yuli Han, Jun Zheng, Jiancao Peng, and Shaochen Yang. 2020. Urban air pollution monitoring using scanning Lidar. *Environmental Pollution* 258 (2020), 113696.
  - [38] Chao Zhang and Jiaheng Lu. 2021. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases* 39, 1 (2021), 1–33.