Check for updates

Received: 31 March 2019 | Revised: 13 October 2019 | Accepted: 11 January 2020

DOI: 10.1002/spe.2799

EXPERIENCE REPORT

WILEY

Robust and efficient memory management in Apache AsterixDB

Taewoo Kim[1] | Alexander Behm[2] | Michael Blow[3] | Vinayak Borkar[2] | Yingyi Bu[2] | Michael J. Carey[1] | Murtadha Hubail[2,3] | Shiva Jahangiri[1] | Jianfeng Jia[2] | Chen Li[1] | Chen Luo[1] | Ian Maxon[1] | Pouria Pirzadeh[2]

[1]Department of Computer Science, University of California at Irvine, Irvine California

[2]University of California at Irvine, Irvine, California

[3]Couchbase, Santa Clara, California

Correspondence
Taewoo Kim, Department of Computer Science, University of California at Irvine, Irvine, CA.
Email: taewok2@uci.edu

Funding information
Amazon; eBay; Facebook; Google; HTC; Infosys; Initial UC Discovery agent; Microsoft; NSF; Oracle Labs; Yahoo! Research

Summary

Traditional relational database systems handle data by dividing their memory into sections such as a buffer cache and working memory, assigning a memory budget to each section to efficiently manage a limited amount of overall memory. They also assign memory budgets to memory-intensive operators such as sorts and joins and control the allocation of memory to these operators; each memory-intensive operator attempts to maximize its memory usage to reduce disk I/O cost. Implementing such memory-intensive operators requires a careful design and application of appropriate algorithms that properly utilize memory. Today's Big Data management systems need the ability to handle large amounts of data similarly, as it is unrealistic to assume that truly big data will fit into memory. In this article, we share our memory management experiences in Apache AsterixDB, an open-source Big Data management software platform that scales out horizontally on shared-nothing commodity computing clusters. We describe the implementation of AsterixDB's memory-intensive operators and their designs related to memory management. We also discuss memory management at the global (cluster) level. We conducted an experimental study using several synthetic and real datasets to explore the impact of this work. We believe that future Big Data management system builders can benefit from these experiences.

KEYWORDS

Apache AsterixDB, big data management system, group by, hash join, inverted-index search, memory management, sort

1 | INTRODUCTION

Jim Gray's now-famous 5-minute rule[1] stated that a data item should be in memory only if it is referenced every 5 minutes or less. Although this rule has been revised over time[2-4] to reflect changes in the cost and performance of both memory and storage, the principle that only a sufficiently frequently referenced item should be resident in memory still holds. An example of a storage change is the appearance of nonvolatile memory express (NVMe) SSDs. These devices provide better performance than hard disk drives and previously released SSDs. In fact, it has previously been shown that NVMe

 © 2020 John Wiley & Sons, Ltd.

SSDs provide performance benefits up to eight times compared to non-NVMe SSDs in relational databases and big data management systems;[5] however, it was also shown that the access latency of DRAM is still three orders of magnitude lower than that of NVMe-based SSDs. Due to the latency difference between memory and persistent storage, systems need to properly deal with data on disk (whether magnetic or solid state) when they cannot load a large amount of data into memory and keep it there indefinitely. To reduce the latency when accessing data,* many data management systems use a buffer cache to hold the recently accessed data items from disk. They also allocate a certain portion of their memory to memory-intensive operators such as sorts, joins, and group-bys to ensure the stability of the system's memory behavior when processing complex queries.

A database operator is memory-intensive if its execution time is dependent on its given memory size. Such operators usually support both in-memory and disk-based operations for different data sizes. For example, the operators listed above can operate in memory when the memory budget is enough to hold and process the entire data. Otherwise, they gracefully migrate to performing a disk-based (or "spilling") operation. Both in-memory and disk-based operations will generate the same result, but their performance can be different. Supporting both types of operations means that an operator can proceed using any amount of memory if the assigned budget is greater than the minimum memory requirement of the operator. Systems need to carefully control the memory use of a memory-intensive operator to perform its operation under a budget since such operators can consume a lot of memory. For instance, a hash join can work with a small amount of memory, but its performance generally gets better with more memory. In contrast, other operators such as select or project do not have these characteristics. They only require a fixed amount of memory to operate, and allocating more memory usually does not yield better performance. Thus, there is no reason to control a memory budget for such nonmemory-intensive operators. A critical requirement of Big Data memory management is ensuring that the memory-intensive operators run within a specified budget.

Despite the importance of memory management, even today, some widely used Big Data management systems such as MongoDB or PostgreSQL can generate "out of memory" errors when there are too many concurrent connections to an instance of the engine. Furthermore, it is the developer's responsibility to manage the memory usage in some Big Data analysis engines, as is the case in Apache Spark.

From its inception, the AsterixDB system has sought to be effective for large data volumes that can far exceed memory. To achieve this, it divides its memory into sections, assigns a memory budget for memory-intensive operators, and ensures that each operator performs its operations within its given budget. In this article, we discuss the implementation of AsterixDB, focusing on its memory management. We believe that our experiences can help other researchers understand the importance of considering the size of data structures that have not been considered carefully in many prior research studies and systems.

The rest of the article is organized as follows. In Section 2, we present the architecture, data model, and memory management model of AsterixDB. In Sections 3 to 5, we describe the implementations of its memory-intensive operators. Section 6 describes AsterixDB's approach to global memory management. Section 7 shows a set of experimental results. Section 8 concludes the article.

1.1 | Related work

In this section, we briefly review how existing database systems and Big Data management systems deal with memory-intensive operators such as join, sort, and text (inverted-index) search. We also highlight some studies on memory-intensive operators related to our work.

Relational database systems: Most open-source and commercial database management systems, such as DB2,† MySQL,‡ Oracle,§ and PostgreSQL,¶ divide their memory into sections. These systems also control the number of concurrent queries to properly manage memory, as summarized in Figure 1.

*https://www.ibm.com/support/knowledgecenter/en/SSEPEK_10.0.0/intro/src/tpc/db2z_bufferpoolsanddatacaching.html
†https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.perf.doc/doc/c0059501.html
‡https://dev.mysql.com/doc/refman/8.0/en/memory-use.html
§https://docs.oracle.com/cd/B28359_01/server.111/b28318/memory.htm#CNCPT1221
¶https://www.postgresql.org/docs/9.4/static/runtime-config-resource.html

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Database	Global memory components	Concurrent query control
DB2	- DBMS (used for basic infrastructure purposes) - FMP (communication between agents and fenced mode process) - PRIVATE (used for general purposes) - DATABASE (buffer pools, catalog cache, shared sort heap) - APPLICATION (application-specific processing) - FCM (used exclusively by the fast communications manager)	The number of maximum concurrent queries is based on a heuristic calculation that factors in system hardware attributes such as the number of CPU sockets, CPU cores, and threads per core.
MySQL	- Buffer pool (holds cached data) - In-memory temporary tables - Thread memory - Cache (table and query) - Sort buffer	The maximum number of concurrent connections per database account is controlled by the system.
Oracle	- Software Code Area - System Global Area (data and control information for one DB instance) - Program Global Area (data and control information for server process)	The maximum number of concurrent connections to a database is controlled by the system.
Postgres	- Shared buffers - Temporary buffers - Work memory	The maximum number of concurrent connections to a database is controlled by the system.

FIGURE 1 Global memory components and concurrent query control in data management systems

Regarding the memory-intensive operators, DB2[#] allocates memory buffers to each join or sort operator as specified by the "sortheap" parameter. Another DB2 parameter called "sortheapthres" controls the overall number of memory buffers that all concurrent join or sort operators can use. If the number of allocated pages becomes greater than this parameter, fewer memory buffers will be allocated to such operators. Similarly, PostgreSQL[‖] allocates buffers to each sort or join operator using a "work_mem" parameter. MySQL[**] has a similar parameter called "sort_buffer_size" to control the number of memory pages allocated per session. Oracle[††] has a memory section called its "SQL Work Area" with memory pages for sort-based operators such as order-by, group-by, and roll-up; a hash join operator also gets its memory from this section.

Big Data management systems: Couchbase[‡‡] provides key-value (KV) store functionality via services. A service provides certain functionality such as data storage, querying, or indexing and a service can be hosted in multiple nodes (machines). Couchbase Server allows up to 80% of a node's total available memory to be allocated to the server and its services. Each service has a set of components. For instance, the data service has a dispatcher, a scheduler, and an KV engine. In the KV engine, there are managed cache, checkpoint manager, flusher, and other components that manage data in memory and on disk. Turning to Cassandra, Cassandra's memory management[5] is similar to that of AsterixDB. The results of insert/update/delete operations are kept in a memory-resident MemTable. The contents of a MemTable will be flushed to disk when it becomes full. MongoDB also uses a certain portion of memory to cache data in memory[§§]. In fact, about 50% of its memory, after deducting 1 GB from the total memory, will be allocated to MongoDB's cache.

Log-structured merge trees: A number of recent data management systems use log-structured merge (LSM) trees, like AsterixDB does, to store their data. LevelDB[¶¶] uses a MemTable to store the results of DML in memory and it uses SSTables to store the flushed components from MemTable on disk. It also utilizes caching much like AsterixDB does. RocksDB[##] utilizes main memory for a block cache, index and filter blocks, Memtables, and blocks pinned by iterators. Monkey[6] tunes the memory allocation between bloom filters and memory components to maximize the system throughput. It proposes a technique to set different false positive rates based on disk component sizes. It also includes the tuning of

[#] https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.config.doc/doc/r0000259.html
[‖] https://www.postgresql.org/docs/9.4/static/runtime-config-resource.html
[**] https://dev.mysql.com/doc/refman/8.0/en/memory-use.html
[††] https://docs.oracle.com/cd/B28359_01/server.111/b28318/memory.htm#CNCPT1221
[‡‡] https://docs.couchbase.com/server/current/learn/buckets-memory-and-storage/memory.html
[§§] https://docs.mongodb.com/manual/core/wiredtiger/
[¶¶] https://github.com/google/leveldb
[##] https://rocksdb.org/

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

memory component sizes. However, one important fact is that it does not utilize a buffer cache and assumes that each access will result in a separate disk I/O. In contrast, AsterixDB does not adjust memory allocation dynamically; however, AsterixDB utilizes a buffer cache for such operations.

Big Data analysis engines: Apache Spark uses two kinds of memory—execution and storage[‖‖]. Execution memory is used for sorts, joins, shuffles, and aggregations. Storage memory is used for caching and propagating internal data across the cluster. These two kinds of memory share a total amount. Execution memory can evict parts of the storage memory, but storage memory cannot evict the execution memory. Apache Hive uses Hadoop, and the heap sizes of mappers and reducers can be set. The maximum heap size for a Hive instance can be also set[***]. For Apache Tez, the maximum heap memory for an instance can be defined[†††]. Apache Impala similarly has parameters to set maximum budgets for entire queries and an individual query[‡‡‡].

External sorting: Since one cannot guarantee that data can be always loaded completely in main memory, most data management systems employ a memory-conscious external sorting method[7] to perform their sort operations. Ever since external sorting was first developed many decades ago, various methods have been developed to allow for memory fluctuations during an external sort process.[8-11] These methods assume that the system may wish to allocate or deallocate memory pages during a sort process based on a memory management policy for concurrent queries.

Hash joins: Various memory-conscious hash-based join algorithms were proposed in the early 1980s, such as Grace Hash Join[12] and Hybrid Hash Join.[13,14] Similar to sort algorithms that can deal with memory fluctuations, several methods have since been proposed to deal with memory fluctuations during a hash join.[15-17] Also, several methods haven been proposed to utilize modern processor architectures.[18-20]

Text search using an inverted index: Another memory-intensive operator today is text-search using an inverted index. A naive solution for text search is scanning the entire dataset and applying the search predicate to each record. To perform a more efficient text search, many data structures and algorithms have been proposed. Among them, the inverted index[7] and its variants are widely used. Search engines such as Elastic Search[§§§] and Apache Solr[¶¶¶] are based on Apache Lucene,[###] which uses an inverted index. Lucene recommends using a large memory buffer[‖‖‖] for a high indexing performance. It also recommends allocating more memory to the Java Virtual Memory (JVM) of the application that accesses a Lucene index. A heap size can be configured for the Elastic Search and Solr JVMs to load indexes from disk while performing searches.

2 | PRELIMINARIES—APACHE ASTERIXDB

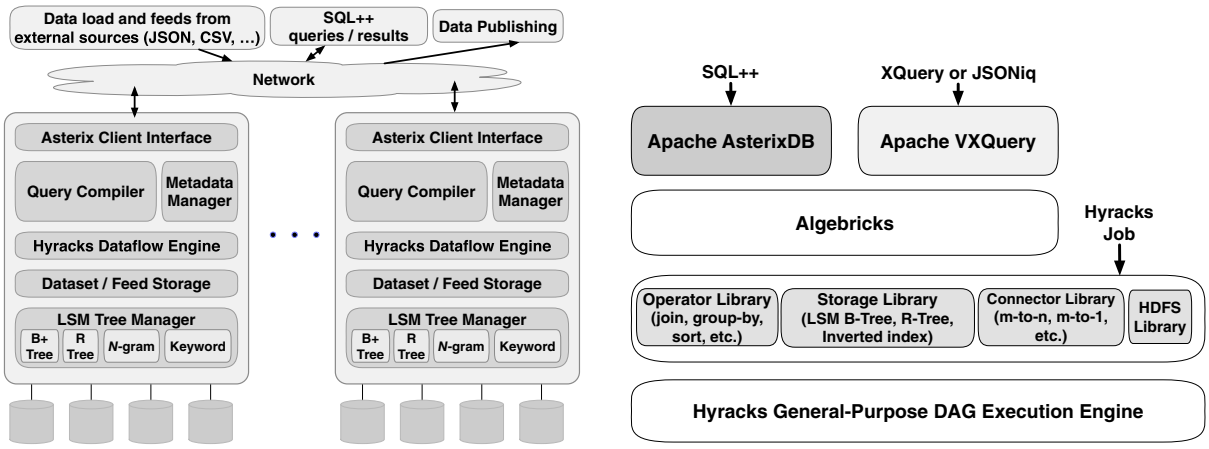
Initiated in 2009 (first public open-source release in 2013), the AsterixDB[21] project integrated ideas from three distinct areas—semistructured data, parallel databases, and data-intensive computing—to create an open-source Big Data management software that scales horizontally on large, shared-nothing commodity computing clusters.[****] The architecture overview is shown in Figure 2A. The handling of truly large data volumes that exceed the memory capacity of a cluster has been one of the project's main objectives.

2.1 | Software architecture

As shown in Figure 2B, AsterixDB consists of several software layers, and this architecture allows other projects (eg, Apache VXQuery) to reuse the system's lower layers. The top-most layer provides a full, flexible data model (ADM) and the SQL++[22,23] query language for describing, storing, indexing, querying, and analyzing Big Data.

[‖‖] https://spark.apache.org/docs/latest/tuning.html#memory-management-overview
[***] https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_hive_tuning.html
[†††] https://tez.apache.org/releases/0.8.2/tez-api-javadocs/configs/TezConfiguration.html
[‡‡‡] https://www.cloudera.com/documentation/enterprise/5-9-x/topics/impala_howto_rm.html
[§§§] https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html
[¶¶¶] http://lucene.apache.org/solr/
[###] https://lucene.apache.org
[‖‖‖] https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/index/IndexWriterConfig.html#setRAMBufferSizeMB-double-
[****] http://asterixdb.apache.org

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Architecture. (B) Software layers.

FIGURE 2 AsterixDB architecture and software layers. A, Architecture. B, Software layers [Colour figure can be viewed at wileyonlinelibrary.com]

The second layer, a query compiler based on *Algebricks*,[24] is used for parallel query processing. This algebraic layer receives a translated logical SQL++ query plan from the upper layer and performs rule-based optimization. A rule can be assigned to multiple rule sets; based on the configuration of a rule set, each rule can be applied repeatedly until no rule in the set can further transform the plan. For logical plan transformation, there are currently 15 rule sets and 171 rules (including multiple assignments of a rule to different rule sets). After performing logical optimization, the physical optimization phase of Algebricks selects the physical operators for each logical operator in the plan. For example, for a logical join operator, a hybrid-hash-join or nested-loop-join can be chosen based on the join predicate. There are 3 rule sets and 36 rules for the physical optimization phase as of the paper writing. After the physical optimization process is done, the Algebricks layer generates jobs to be executed by the *Hyracks*[25] layer.

The Hyracks layer includes storage facilities for datasets stored and managed as partitioned LSM-based B+-trees with optional LSM-based secondary indexes.[26] AsterixDB translates a query computation task into a Hyracks job—a directed-acyclic graph (DAG) of *operators* and *connectors*—and sends it to Hyracks for execution. In Hyracks, operators consume partitions of input data and produce partitions of output data. The produced output partitions can be repartitioned by connectors to produce the input partitions for the next operator. An operator consists of one or more *activities* (substeps or phases), and there may be control dependencies between two activities of certain operators. Using this information, one or more *stages* are created for the job. A stage includes a group of activities (an activity cluster) that can be coscheduled, and Hyracks jobs are executed on a stage-by-stage-basis. Since data are represented as tuples of bytes at this level, Hyracks is an execution layer that is data-model agnostic. This means that data are accessed at the binary level rather than as objects. This approach was used since creating and collecting language objects is often a cause of performance bottlenecks.[27]

2.2 | Data model

AsterixDB defines its data model (ADM)[21] for semistructured data. ADM is a superset of JSON, with support for multisets, arrays, nested types, and a variety of primitive types. A dataverse is the top-level organizing concept and provides a namespace of types, datasets, indexes, functions, and other artifacts. A datatype specifies the fields and their types that should be included in each data instance. An ADM datatype can be either an open type or a closed type. An open type means that instances must have all the specified fields but may also contain extra fields that can vary from instance to instance. In contrast, a closed-type enforces that its instances must have only the specified fields. A dataset is a collection of instances of a datatype. The following DDL shows a simple example of some ADM statements. The first two create a dataverse called exp and a data type for a dataset called Reddit. Its type, RedditType, is defined as an open type.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
create dataverse exp;
use exp;
create type RedditType as open {
 id: int, author: string, body: string, controversiality: int, created_utc: bigint,
 retrieved_on: bigint, subreddit: string?, subreddit_id: int?, subreddit_type: string? }
create dataset Reddit(RedditType) primary key id;
```

Each record in a dataset is identified by a unique primary key and records are hash-partitioned across the nodes of a cluster based on their primary keys. Each record in a dataset has to comply with its associated datatype. The above DDL also includes an SQL++ statement for creating a Reddit dataset. Each partition of a dataset is locally indexed by a primary key in an LSM B+-tree, a.k.a. the primary index, and resides on its node's local storage. AsterixDB also supports secondary indexing, including B+-tree, R-tree, and inverted indexes. These indexes are local, that is, they are partitioned in the same way as the primary index. Like the primary index, each secondary index also adopts an LSM-based structure. Further details of LSM-based index structures can be found in the AsterixDB storage management paper.[26]

2.3 | Memory management

Figure 3 shows the memory structure of an AsterixDB partition in a JVM instance. There are three main sections, including one for in-memory components (a.k.a. in-memory component memory), a buffer cache, and working memory.[26]

The in-memory component section holds the results of DML operations on the currently active datasets. Due to the nature of LSM index structures, the results of every insert, upsert, and delete operation of a dataset first go into an in-memory LSM component before being persisted to disk. The amount of memory used for this purpose is controlled by a budget parameter that limits the maximum amount of this memory that a dataset can occupy. This budget is shared by the primary and secondary indexes. When the overall in-memory component size of a dataset reaches this limit, its primary index and all secondary indexes are flushed to the disk and become immutable. The allocated in-memory component memory then becomes available again. AsterixDB also controls the maximum size of the overall memory for this section. If this limit is reached and an in-memory component of a new dataset needs to be created, an active dataset is chosen based on a policy to be flushed to disk to accommodate this new dataset.

The second major section of main memory, the buffer cache, is used to read disk pages from the LSM disk components. Since AsterixDB employs LSM index structures, a page of a disk index component is always immutable. Thus, there is never a dirty page in the buffer cache for a page that was read from a disk component. The maximum size of the buffer cache is limited by a buffer cache size parameter. Since the memory usage of the buffer cache stays within this budget, and its behavior is very similar to the buffer cache[28] in other database systems, we will not discuss its detailed structure further.

The last memory section, working memory, honors the memory allocation requests from the operators in an execution plan. These requests require careful management since there can be a large number of operators in a complex query plan and each operator has different characteristics regarding its memory usage and requirements.

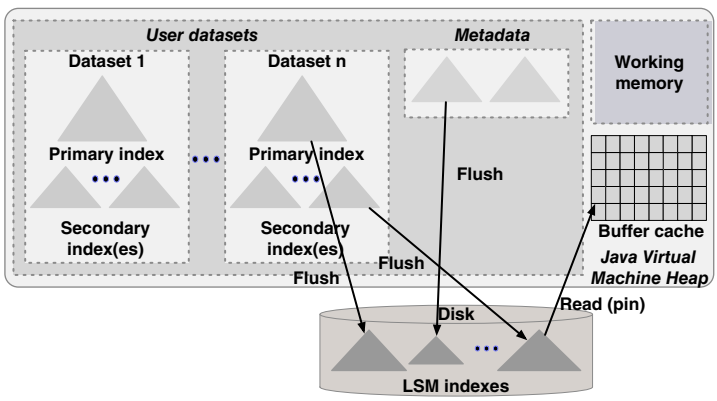


FIGURE 3 Memory structure in AsterixDB [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

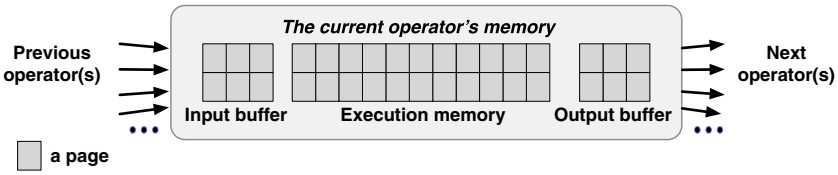


FIGURE 4 Operator model [Colour figure can be viewed at wileyonlinelibrary.com]

2.4 | Operators

As described before, an operator consists of one or more activities. To perform these activities, each operator in AsterixDB uses a set of memory pages as its input buffer, zero, or more pages as its execution memory, and a set of memory pages as its output buffer as shown in Figure 4. Each page can contain multiple records. (A page is the smallest data-transfer unit in Hyracks. For instance, an operator passes a page of records to the next operator, not just a single record.)

2.4.1 | Operator types regarding memory requirements

The difference the operators regarding their memory requirements mainly comes from their execution memory requirement. Regarding execution memory usage, there are three types of operators. The first type requires no intermediate execution memory for any amount of data. For example, a select operator uses one input buffer page to read records from the previous operator in a plan. If its operation is a simple scalar function, it does not require execution memory proportional to its overall input data size. If a condition such as c_name = "Smith" is provided, the operator can easily apply this predicate on each record in the incoming input buffer page. If a record satisfies this predicate, it is copied to the output buffer page. When the output page becomes full, that page is pushed to the next operator. The second type of operator requires a constant number of pages for its execution memory. Allocating more pages would not significantly accelerate its operation. For instance, a B+-tree index-search operator needs to keep the current leaf page per LSM disk component that contains the result of an index-search predicate. The third type of operator also requires a certain number of pages as its minimum execution memory. However, if more pages are allocated, its performance can be improved. This type of operators is memory-intensive operators, as briefly described in Section 1. For example, a sort operator requires at least three pages to hold the incoming records in a page, sort them using an in-memory pointer array, and generate the output for them in a page. If there is no available space to hold incoming records, the operator creates a temporary run file on disk after sorting the current records in the available working pages. After processing all records in this phase, it merges the run files on disk to generate the final results. However, if the number of available working pages is enough to accommodate all incoming records, the operator can instead perform an in-memory sort. The operators we focus on in this article are these memory-intensive operators.

2.4.2 | Memory-intensive operators

In AsterixDB, the budget for a memory-intensive operator is determined by operator-specific parameters (eg, 32 MB) and the system converts the budget into a number of pages using the system's page size parameter. Suppose that the assigned budget is M pages. This means the memory-intensive operator can request M pages from the working memory at maximum and uses these pages as its execution memory. Within an operator, a page pool whose maximum capacity is M is created and managed by a page manager. When a memory-intensive operator requires a page, it makes an allocation request to the page manager. The page manager then checks the page pool. If there are already available pages in the pool, one page is chosen and allocated. If there are no available pages but there is still enough space, a new page is created and allocated. When a memory-intensive operator releases a page, it issues a release request to the page manager, which returns the page to the page pool.

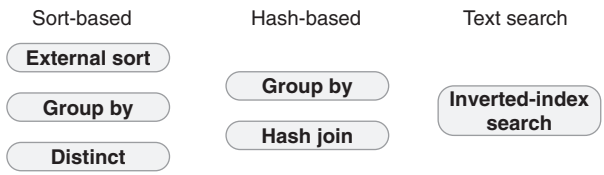
All memory-intensive operators support both in-memory operation and disk-based operation to deal with any volume of data. As these operators receive incoming records, they gradually allocate memory as necessary to process those records. When they cannot allocate more memory to process incoming records due to their budget limits, they switch to a disk-based operation. For instance, a hash join operator spills some records to disk and processes them later after processing records in memory. If they can allocate enough memory to process all records, an entirely in-memory operation

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 Memory-intensive operators in AsterixDB

Operator	Description	Category
StableSortPOperator	Performs external sort.	Sort
InMemoryStableSortPOperator	Performs in-memory sort.	Sort
SortGroupByPOperator	Performs external sort-based group-by.	Group-by
PreclusteredGroupByPOperator	Performs group-by on a pre-sorted collection of records.	Group-by
PreSortedDistinctByPOperator	Computes distinct values on a pre-sorted collection of records.	Group-by
ExternalGroupByPOperator	Performs external hash group-by.	Group-by
NestedLoopJoinPOperator	Performs block nested-loop join.	Join
HybridHashJoinPOperator	Performs optimized hybrid hash join.	Join
InMemoryHashJoinPOperator	Performs in-memory hash join.	Join
InvertedIndexPOperator	Searches an inverted-index using a predicate and returns index entries.	Inverted-index search

FIGURE 5 Classification of memory-intensive operators



can be performed. The memory consumption of memory-intensive operators needs to be carefully controlled since they have memory allocation/deallocation logic inside to maximize their performance.

In order to ensure that all memory-intensive operators run within a specified budget, we can analyze the memory usage of each physical operator and identify those that need to be budgeted. If an operator only requires a fixed number of pages, its memory usage is always fixed; thus, we do not discuss it. Currently, there are 52 physical operators in AsterixDB and most operators fall into this category. After excluding these non-memory-intensive operators, there are 10 memory-intensive operators as shown in Table 1. Some memory-intensive operators are related to other operators. For example, HybridHashJoinPOperator performs a hybrid hash join. During its probe phase, it initiates InMemoryHashJoinPOperator to perform an in-memory hash join for the records. Therefore, we can group the memory-intensive operators into four basic categories based on their functionalities—sort, group-by (distinct), join, and inverted-index search.

Let us consider each category to finalize the list of memory-intensive operators that will be discussed in this article, as shown in Figure 5. The first category is sort. We will discuss the sort category in Section 3. The second category, group-by, has two implementations in AsterixDB. The first one is sort-based group-by, which uses data structures similar to the sort operator. Thus, after discussing the sort operator, we will briefly discuss this operator. The second implementation is hash-based group-by, which uses a hash table and a data partition table. We will discuss hash-based group-by operator in Section 4.1. AsterixDB also supports a distinct operator that is very similar to the group-by operator. Per field value (group), it only produces one output, and internally it uses the same data structure as the sort-based group-by operator. Therefore, we will treat the distinct operator as a sort-based group-by operator in our discussion. The third category is join. There are three types of join in AsterixDB—index-nested-loop, nested-loop, and hash join. Index-nested-loop join utilizes index-search operators, which are not memory-intensive. The nested-loop join operator uses a block of memory pages to hold a chunk of pages from one input branch and traverses it by reading each page from the other branch. Other than using a block of memory, one page buffer, and one output-page buffer, the operator does not use any additional memory structures. Thus, we will not discuss this type of join in detail either. The hash join operator utilizes the same kind of hash table and data partition table as hash-based group-by operator. However, its operation flow is different. Therefore, we will discuss the hash join operator in Section 4.2. Finally, during our analysis, we find that the inverted-index search operator allocates/deallocates significant amounts of memory. Thus, we will discuss the inverted-index search operator in Section 5.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

3 | MEMORY-INTENSIVE OPERATOR: SORT

In this section, we describe the sort operator first since its structure is relatively simple compared with other memory-intensive operators. When a sort activity is needed in a plan (eg, due to an order by clause in a query), AsterixDB performs an external sort.[7] There are a few detailed techniques known to improve performance. Since comparing normalized binary-representations of field values is more efficient than comparing the original values, the concept of *normalized key* has been widely used.[29] Also, rather than moving an actual record during a sort, normally *an array of record pointers* is used in most implementations[29] to deal with variable-length records. AsterixDB adopts these two techniques, too.

An external sort consists of two phases—build and merge—as shown in Figure 6. In the first phase, a sort operator gradually allocates memory pages to hold incoming records. If it cannot allocate more pages because the budget is fully utilized, it will switch to disk-based operation. That is, it sorts the currently loaded records in its execution memory using an in-memory sort algorithm, for example, merge sort. After this sorting is done, it creates a temporary run file on disk to keep this partially sorted result. It then clears the memory pages and receives more incoming records. The build phase is done after processing all incoming records. The result of the first phase is either a set of generated run files on disk or all incoming records being resident in memory if the budget M was greater than the number of incoming pages. In the second phase, for the spilled case where there are run files on disk, the operator recursively multiway merges the run files and generates the final results. For the in-memory case, it performs an in-memory sort and generates the final results. In either case, the result will be passed (page by page) to the next operator.

In addition to utilizing pages, the sort operator also uses an additional data structure called the *record pointer array* that contains the location and the normalized key of each record in memory. This array is needed to avoid performing an in-place swap between two records during in-memory sorting since each record's length is generally different because of variable-length fields. Also, this array can improve the performance since comparing two normalized keys in the record pointer array using binary string comparison can be much faster than comparing the actual values between two records, which requires accessing the actual records in pages. Another benefit of using the array is that its size is smaller than actual records. Thus, there are higher chances that a record pointer array can be kept in CPU cache due to frequent accesses during in-memory sorting.

In the first phase as shown in Figure 7, for an incoming page, the operator allocates one memory page to store the records in the current incoming page. At the same time, it adds the information about each record such as the page and offset of the record in the page to the record pointer array. The array also contains the normalized key for the fields that are being sorted. When reading an incoming page, if the operator cannot allocate more pages from the working memory, it first sorts the record pointers in the array using the normalized keys in each pointer. After that, it creates a temporary run file on disk. It does so by adding records to the output page by scanning the array from the beginning. When an output page becomes full, the page is flushed to the temporary file on disk. This process continues until all sorted records

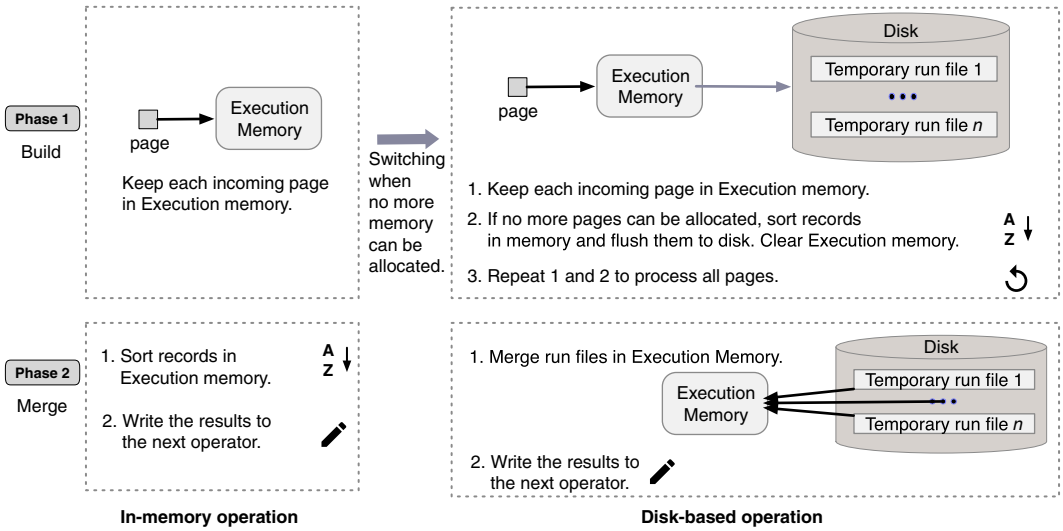


FIGURE 6 Two phases of the external sort [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

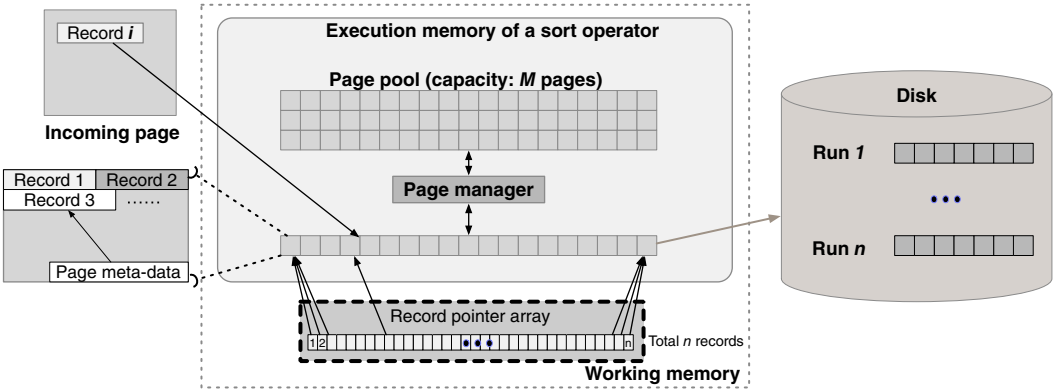


FIGURE 7 External sort: basic implementation [Colour figure can be viewed at wileyonlinelibrary.com]

are flushed to the file. The array will then be cleared and its pages will be deallocated. The operator then reads the next incoming page to fill in the execution memory with pages and to fill in the array again.

During the second phase, the operator merges the temporary sorted run files created in the first phase. By default, it attempts to allocate one working page per run file to use as an input buffer for that run in order to accommodate all run files. If the number of runs is smaller than M, this phase allocates more pages per run as an input buffer to allow it to read a series of pages from that run on disk at once to expedite the merge process. In contrast, if the number of runs is greater than M, it first merges M runs and generates an additional (M times longer) run file. This run file is added at the end of the current run file list. At the same time, it removes the newly merged run files from the list. The operator repeats this process until the run file list becomes empty.

Since a record pointer needs to be allocated per record, the size of a record pointer array grows as the number of records increases. Thus, it is evident that its size will increase as the operator deals with more data. However, the initial basic sort implementation of AsterixDB did not include the size of the record pointer array (depicted in the dashed rectangle in Figure 7) in the budget M since the structure was considered as a "small" auxiliary data structure that would "just" consist of a few integer values. Thus, the entire budget of M pages was made available to hold incoming data pages. The record pointer array was created and maintained separately and not considered for memory accounting purposes. In Section 7.2.1, we will show the actual size of the record pointer array during a sort operation and we will see that this can lead to significant memory allocation overages. This issue was solved by considering and managing the size of the record pointer array within the budget M. That is, every time the sort operator receives a new page in the first phase, it calculates the total memory footprint of this page by adding the page size and the size of the needed record pointers for records in that page. Space for the record pointer array is thus now a part of the operator's page pool. If the sum is less than the budget M, the incoming page is inserted and the size of the current memory footprint is updated. Otherwise, AsterixDB first sorts the currently loaded pages and flushes them to disk as a temporary run file. (Note that the record pointer array is no longer needed and does not need to be flushed.) This process continues until all incoming pages are processed. The second phase then merges the generated run files on disk to generate the final result.

4 | MEMORY-INTENSIVE OPERATORS: HASH-BASED

In this section, we describe two memory-intensive hash-based operators—hash group-by and hash join. Both operators utilize a *hash table* and a *data partition table* as their core data structures. The data partition table stores records and the hash table holds pointers to those records to locate them efficiently. We first present the hash group-by operator due to its simplicity. We then describe the hash join operator.

4.1 | Hash group-by operator

A group-by is used with aggregating functions, such as COUNT, MIN, MAX, SUM, or AVG, to group the results of an aggregation by one or more fields. The fields that are being grouped are called the "group fields" and the fields that are

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

being aggregated are called the "aggregate fields." The group fields and aggregate fields can be the same or different depending on the semantics of a query. For instance, suppose there is an employee dataset that includes empId, age, and salary fields. If a user wants to compute the average salary of employees per age, the group field is age and the aggregate field is salary. If the user wants to compute the number of employees per age, both the group field and the aggregate field are age.

By default, when a user includes a group by clause in an SQL++ query, AsterixDB chooses to use a sort-based group-by. This operator is similar to external sort with two phases—build and merge. The main difference is that group-by generates an aggregate result per group value. Specifically, during the build phase, after performing an in-memory sort, it generates a partially aggregated result per group value for that run. In the merge phase, it merges the partially aggregated results to generate the final result per group value. We will soon discuss how to generate the aggregate result per group. Since the external sort and the sort-based group-by operator use similar data structures, the sort-based group-by operator shares the same memory accounting issues as the sort operator covered in Section 3. In addition, the AsterixDB's sort-based distinct operator internally uses data structures similar to the sort-based group-by. Therefore, we will not discuss the sort-based group-by or the distinct in further detail.

AsterixDB also offers a second group-by operator. If a sorted result order after a group-by is not required, a user can optionally provide a "hash group-by" hint in a SQL++ query to instruct AsterixDB to instead perform a hash-based group-by computation. The budget for either group-by can be set using the "groupmemory" parameter. We now consider the hash-based group-by operator in more detail.

The conceptual operation flow of the hash group-by is as follows. For each record i in an incoming page, the operator first calculates a hash value $h(i)$ for the record i by applying a hash function h to the group field value. It then checks whether a partial aggregate result for the group field value exists in working memory using the hash value $h(i)$. If the partial aggregate result exists, the operator aggregates the record i into the result (eg, adding the incoming salary to the running total) using the aggregate field value. If the result does not exist, it creates a new aggregate result record for the group field value. After the operator processes all incoming records, the accumulated aggregate results will be finalized and passed to the next operator.

The hash group-by operator consists of two phases, similar to the sort operator, as shown in Figure 8. This is because the operator needs to access all incoming records and not all aggregated results from the incoming records may fit into memory. Thus, partially aggregated results need to be spilled to disk as in the sorting case. In the first phase, the operator

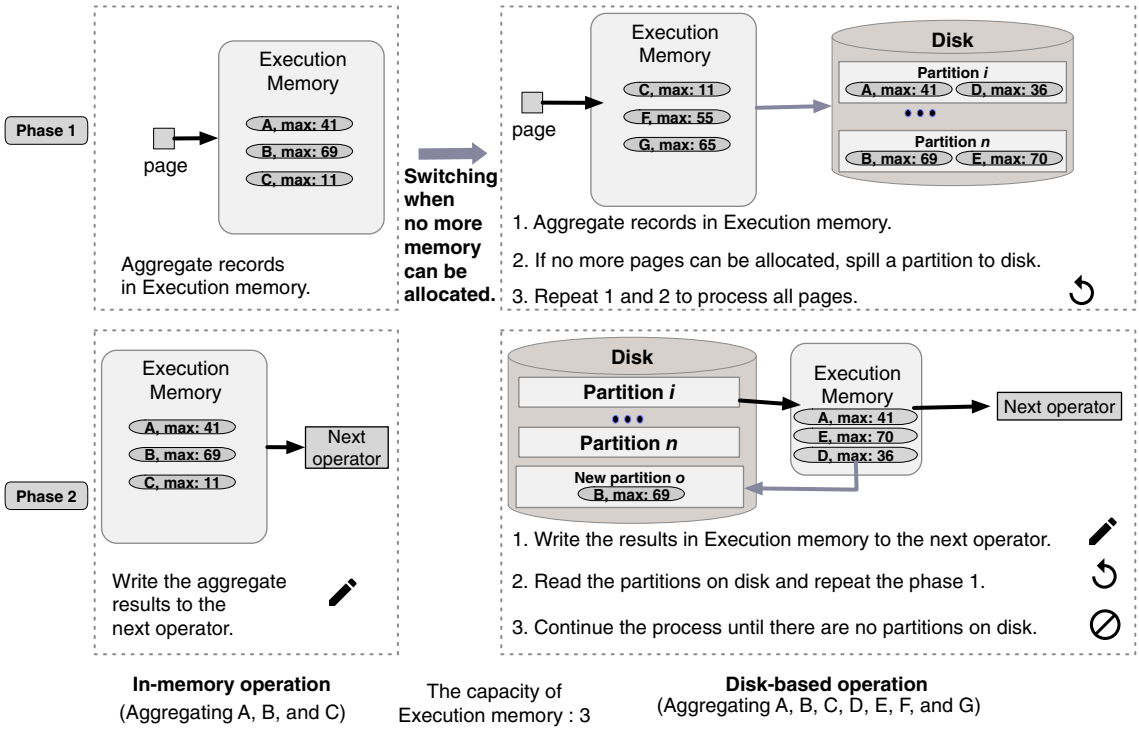


FIGURE 8 Two phases of hash group-by [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

calculates the number of hash partitions based on the input data size and the given budget M. (This calculation aims to partition the aggregation computation into pieces that can hopefully be performed in memory if the data size permits.[30]) Each partition consists of one or more pages, and each page contains one or more groups and their aggregate results. Based on the possible number of hash values estimated by the query optimizer, each partition covers the same number of contiguous hash values. For instance, if there are 1000 expected hash values and 20 partitions, each partition covers 50 values. That is, the first partition stores the aggregate results for groups if the hash value of the group is between 0 and 49. After the number of partitions and their hash value ranges are set, for each incoming record i, the hash group-by operator aggregates the record i using the $h(i)$ value of the group field value and the aggregate field value. It gradually allocates memory when a new page needs to be allocated in a partition to create a new group's aggregate result. When the operator cannot allocate more memory, it switches to disk-based operation. It spills an in-memory partition to disk based on a spill policy to make space in memory. Unlike sorting, the operator does not need to spill all partitions in memory to disk, as the purpose of spilling a partition is to create some space, and the aggregate result of one group does not depend on that of other groups. The memory pages occupied by the spilled partition are deallocated and the operator continues receiving incoming pages and spilling a partition whenever it cannot allocate more pages. Thus, in-memory partitions and spilled partitions are dynamically decided as the operator processes incoming pages. This dynamic adjustment is similar to the hash join operator that performs *optimized hybrid hash join*, which is our implementation of hybrid hash join.[14] We will discuss this operator shortly. The common property that these two operators share is that partitions that are being kept in memory during the first phase are dynamically decided. Thus, these operators can reflect the characteristics of input data on-the-fly. For instance, if these operators only keep the first five partitions in memory in the above example and spill all the other partitions to disk, the performance may be degraded in case these partitions do not have many associated records. This dynamic adjustment guarantees that the partitions that are kept in memory are read only once, unlike the spilled partitions.

After processing all incoming records, phase 2 of the hash group-by begins. At this point, some partitions are still in memory and some partitions are on disk. The operator passes the contents of the in-memory partitions to the next operator by copying their page-by-page into the output buffer since it does not need to read any spilled partitions on disk to generate the final results for in-memory partitions. The operator then goes through phase 1 again for the spilled partitions to generate the final results for these partitions. Again, when phase 1 for the spilled partitions is done, either all partitions are now in memory or some partitions are still in memory and some partitions are on disk. The operator passes the aggregate results in memory to the next operator and again repeats phase 1 for any remaining spilled partitions on disk. This recursive process continues until there are no spilled partitions on disk.

To implement the above operation flow, the hash group-by operator uses a *data partition table* to hold the aggregate records in partitions and a *hash table* to guide it to the location of the aggregate records that share the same hash value, as shown in Figure 9. We separated these two tables to increase the chance of CPU-caching the hash table entries by making its size smaller. This choice also gives the operator more flexibility when dealing with hash slot overflows due to hash value collisions. If the hash table contained the actual aggregate records, and not just pointers to aggregate records, when an overflow happens in a hash slot, the aggregate records would also need to be migrated to another expanded slot and could cause more time and space overhead than only moving the slot with pointers to the aggregate records.

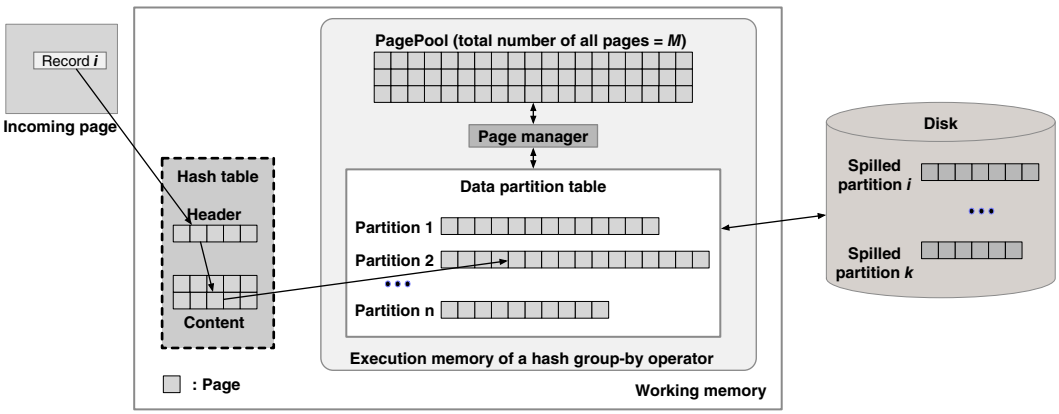


FIGURE 9 Hash group-by: basic implementation [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

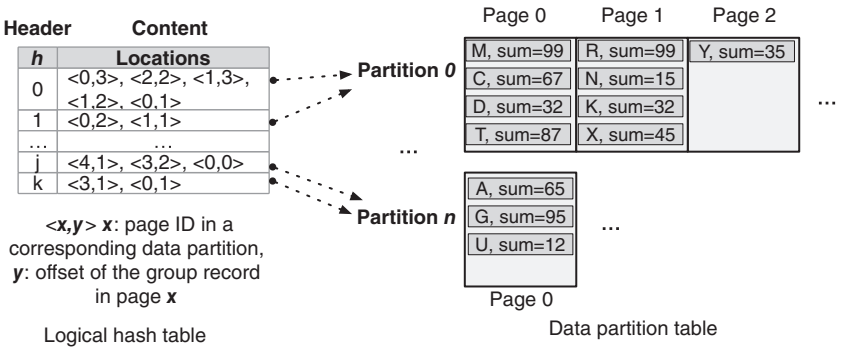


FIGURE 10 An example logical hash table and data partition table

The data partition table stores the records for groups and their aggregate results. Each partition in the table contains pages to hold groups and their aggregate records and covers an equal range of contiguous hash values as described earlier. Using the hash value of a group, the operator can easily find which partition the aggregate record for that group belongs to. When a page allocation/deallocation is needed, the operator makes a request to the page manager, as shown in Figure 9. The group-by page manager has a pool with a maximum capacity M.

The hash table for group-by holds pointers to the actual aggregate records to locate them efficiently as described before. Figure 10 shows an example instance of a logical hash table and the resulting data partition table. We can see that the locations of aggregate records that share the same hash value are stored in each hash slot. For instance, there are five aggregate records whose hash value is zero. In the data partition table, each aggregate record contains a group field value and its aggregate result. In this example, we use SUM as the aggregate function.

Physically, the hash table consists of two parts—*header* and *content* pages—as shown in Figure 11. Each slot in a header page indicates the location of a content slot in a content page and corresponds to one hash value. For instance, the first slot is for the first hash value (zero) and the second slot is for the second hash value (one). If each header page contains 1000 slots, the header slot for the 1001st hash value will be the first slot in the second header page. A content slot contains the actual in-memory location information for the aggregate records in the corresponding data partition that share the same hash value. Thus, locating aggregate results for a hash value k can be done by getting the location of the content slot in the kth slot in the header page, fetching pointers for the actual aggregate results from the content slot, and then accessing the actual aggregate results in the corresponding data partition. We have chosen this detailed design to let the group-by operator locate a hash slot quickly since the location for each hash value in a header page is fixed. The operator can also cope flexibly with hash slot overflows since it just needs to update the location of the content slot in the header page after migrating the old content slot to a newly extended content slot. This design choice also provides better space usage than equally dividing the hash table space per hash value since there can be data skewness. A content slot is always added to the last content page when the first aggregate record for the given hash value is created or when a slot is migrated. When a new aggregate record pointer is inserted into a content slot that is fully occupied, this causes an overflow of the content slot. In this case, the operator creates a new content slot whose capacity is twice that of the original slot and adds it to the last content page. (In order to reduce the number of content slot overflows, we set the initial capacity of a slot to three to prevent a small number of aggregate records that share the same hash value from causing an overflow.) It also migrates the current aggregate record pointers from the original slot into the new slot. It then inserts the new aggregate record pointer that caused the overflow into the new slot. Finally, the content slot location is updated in the corresponding header slot in the header page.

This aggregation process allows the header and data partition tables to grow gradually as incoming records are aggregated. When all records are processed, the partitions in memory will be passed to the next operator since they contain the final aggregate results for the corresponding hash value ranges. We define this process as the first iteration of the hash group-by operation since there can be multiple recursive operations for spilled data partitions when dealing with large input sizes. In the new iteration, the operator clears both the hash table and the data partition table in memory, and it processes each record in the spilled partitions one by one by aggregating it into the data partition table and inserting a pointer to the group into the hash table. After each iteration, the operator passes the contents of the data partitions in memory to the next operator. The remaining spilled data partitions are then processed in the next iteration. This recursive process continues until there are no remaining spilled data partitions. This process is depicted in Figure 12. (This incremental aggregation process is different from that of hash join, and the effect of this difference will be discussed in Section 7.2.3.)

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

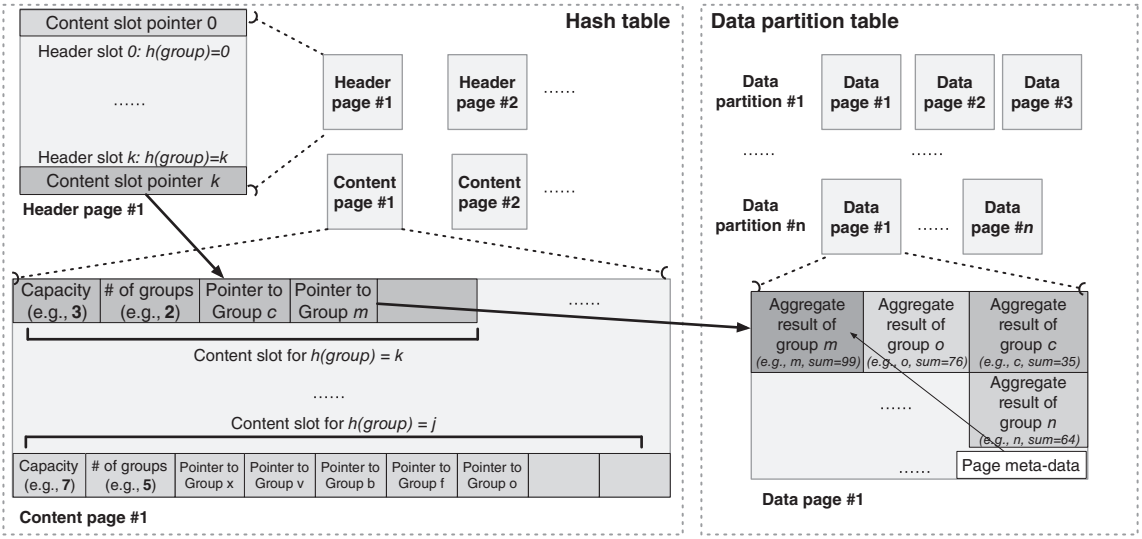
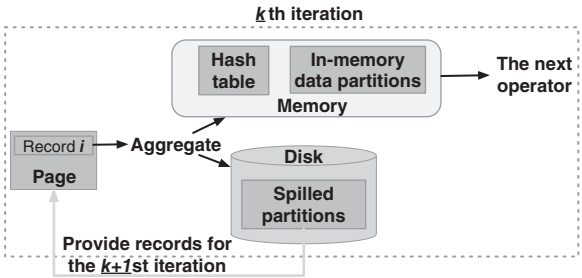


FIGURE 11 The detailed view of the hash table and the data partition table

FIGURE 12 The data flow on each iteration of the hash group-by operation



The actual hash table footprint can increase as the number of incoming records increases. However, regarding the hash table size, most work, including the original hybrid hash join paper,[14] has treated the actual footprint of the hash table as being negligible, using the concept of a *fudge factor F* during a hash-based operator's execution. For example, if the number of memory blocks that a dataset (table) R occupies is $|R|$, its hash table is assumed to occupy $F \times |R|$ and F is assumed to be small (e.g., 0.2).[14] In contrast, there is one report that explains how hash join was implemented in DB2 that did consider the hash table as a part of the hash join memory footprint[††††]. Following the literature, the initial basic AsterixDB implementation also used a fudge factor to estimate the hash table memory footprint. In fact, it used 20% of the budget M as the fudge factor. However, the actual runtime hash table size was not constrained by this value. Regarding the memory management of two tables, the data partition table was placed in the execution memory. In contrast, the hash table was simply allocated from the working memory of the JVM instance as was shown (depicted in the dashed rectangle) in Figure 9. This means that when the working memory of the JVM instance was exhausted, an OOM could happen. In fact, based on the size of the group-by field in a record, the size of the hash table can become even greater than that of the data partition table for some queries. We will see such a case in Section 7.2.2. This issue was subsequently resolved by changing the design to also account for the actual size of the hash table during the hash group-by operation. In short, the hash table now allocates/deallocates memory pages from/to a newly introduced hash table page manager, which shares the same page pool with the page manager of the data partition table. That is, the combined memory usage of both tables is now bounded by the group-by budget M.

4.2 | Hash join operator

The hash join operator also utilizes a data partition table and a hash table very similar to those just described. Since we already discussed the detailed structure of these two tables, we focus on the operation flow of the hash join next. We

[††††] https://pdfs.semanticscholar.org/presentation/38c4/bcd4fe05787c1c8af74981a4e182ff3411c2.pdf

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

have implemented a variation of a hybrid hash join[14] called *optimized hybrid hash join* in AsterixDB. The main difference between the traditional hybrid hash join and our join technique is that the latter does not predecide which data partitions should be kept in memory during the build phase of a hash join. Rather, the in-memory partitions are dynamically decided during the join process, similar to the hash group-by. This design choice was made to handle data from any source, such as an external data source or the result of a complex computation. The I/O cost of this method is essentially the same as that of a hybrid hash join[14] since it is the same as executing the hybrid hash join and setting the surviving partitions in our join technique as its in-memory resident partitions. When the allocated memory is large enough to process the incoming data, our technique performs an in-memory hash join. If the allocated memory is very small relative to the incoming data, our technique behaves like a Grace hash join. In between these two cases, it tries to use the allocated memory as well as it can. In AsterixDB, the hash join and the hash group-by operators use very similar table structures. Therefore, their overall processing is similar except in a few places. We focus primarily on the differences here.

The conceptual operation flow for a hash join is shown in Figure 13. A hash join has two input branches. The first input branch is called the *outer branch* and the second branch is called the *inner branch*. A hash join consists of two phases—*build* and *probe*. In the build phase, for each record i in an incoming page from the inner branch, the hash join operator first calculates its hash value $h(i)$ using the join field value. Based on $h(i)$, it places the record in the corresponding partition in memory for the hash value $h(i)$. For simplicity, let us first suppose that the inner input's data fits in memory. If so, the build phase finishes after processing all incoming records from the inner branch. In the probe phase, it starts reading records from the outer branch. For each record j, the operator calculates the hash value $h(j)$ of the record j using the join field value. It then finds the corresponding partition in memory for the hash value $h(j)$ and checks whether there are matches between the record j and the records from the inner branch whose hash value is $h(j)$. If there is a match, the pair will be joined and passed to the next operator. The probe phase finishes after processing all the records from the outer branch.

To implement the above operation flow, a data partition table and a hash table are used during the hash join. In the build phase, the operator first calculates the number of partitions to be used based on the budget M and the input data size. The overall number of possible hash values is set to the number of input records. Based on this number of possible

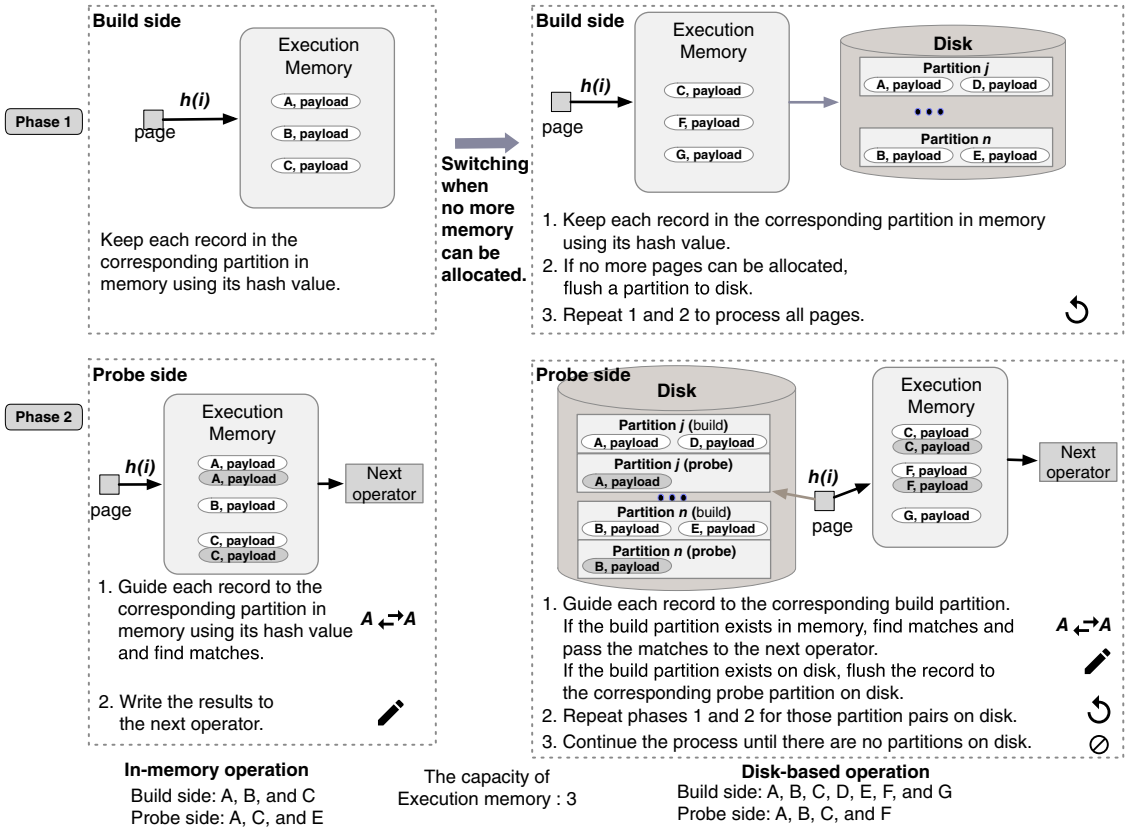


FIGURE 13 Two phases of the hash join [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

hash values, the same number of contiguous hash values is allocated to each partition, similar to the hash group-by case. For each incoming record i from the build side, after calculating $h(i)$ using the join field values of the record i, the operator inserts the record into a corresponding data partition using $h(i)$ as shown in Figure 14. Note that, for a join, this operation is an append, not an aggregation. Thus, if there are two records whose join field values are the same, they will be kept separately. (For the hash group-by case, these two records would be aggregated if their group field values are the same.) As a consequence, the overall memory footprint for hash group-by is usually smaller.

Let us now consider the case of larger input data that cannot fit into memory. A data partition gradually grows by adding a new page if the current page is full. If a data partition cannot grow because the budget M is fully utilized, the operator spills a data partition to disk based on a spill policy. It deallocates all pages in that partition to make space and continues the insertion process. Thus, similar to the hash group-by case, the in-memory data partitions and spilled data partitions on disk are decided dynamically during this process. When the build phase is finished, some data partitions' records are already spilled to disk and some data partitions' records reside in memory. The operator now builds a hash table for the in-memory partitions in order to perform an in-memory hash join, and it then begins the probe phase as shown in Figure 15. Note that the hash table is created and populated at this later point in time, unlike the hash group-by case where the hash table is created right at the beginning of phase 1. The hash join operator builds the hash table at the end of the build phase since the operator does not know which data partitions will still be in memory by the end of the build phase. In contrast, the hash group-by requires a hash table from the beginning since it needs to incrementally aggregate incoming records. In the hash join case, no aggregation is needed, so the operator can postpone building the

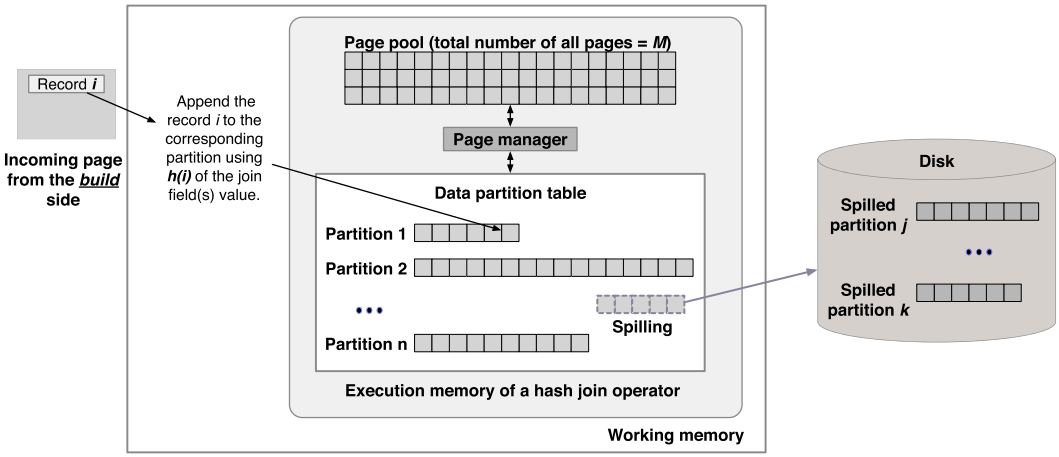


FIGURE 14 Hash join: build phase [Colour figure can be viewed at wileyonlinelibrary.com]

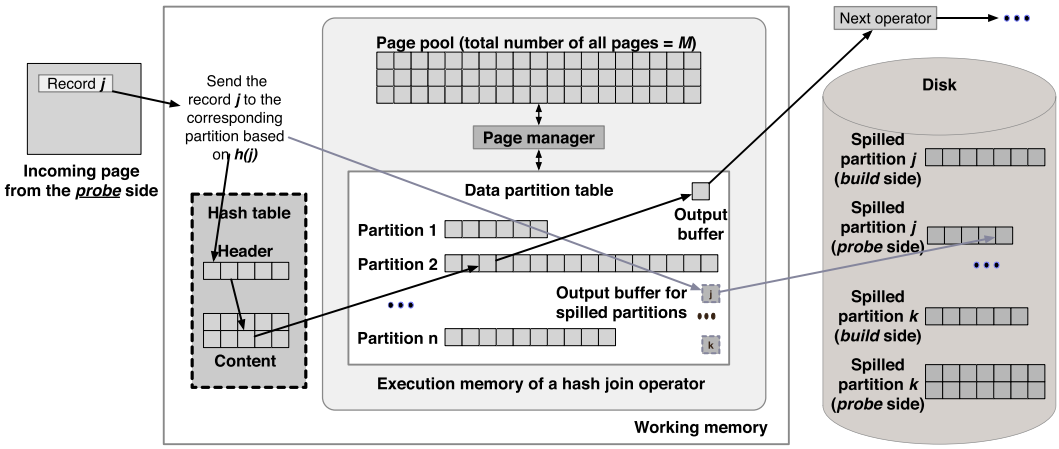


FIGURE 15 Hash join: probe phase [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

hash table until the end of the build phase. Its hash table is used to guide records from the probe side, based on their hash values, to the matching records in the corresponding data partition from the build side.

During the probe phase, for each incoming record j from the probe side, the operator first calculates $h(j)$ and sees whether the corresponding build partition has been spilled or not. If the build partition has been spilled, the operator just appends the record to the output buffer for the corresponding probe partition on disk to deal with spilled partition pairs later, as the operator cannot bring the spilled build partition into memory at this moment. When the output buffer for a probe partition becomes full, the page is added to the probe partition on disk. If the corresponding build partition is in memory, the operator fetches records whose hash value is equal to $h(j)$ and adds the joined output records to the result output buffer page if the actual join condition holds. When the buffer becomes full, the page in the buffer will be passed to the next operator. After the probe phase is done, there can be pairs of spilled data partitions from both the build and probe sides. For each such pair, the hash join operator picks the smaller spilled partition as the build side and the other side as the probe side (possibly involving *role reversal* of the original build and probe sides). The purpose of this pick is to reduce the size of partitions that would be spilled to disk in the next join. The operator then begins a new hash join phase for each pair. This recursive process continues until there are no remaining spilled data partitions.[‡‡‡‡]

As discussed before, the hash join operator uses the same data structures used in the hash group-by operator. Naturally, the initial implementation of AsterixDB considered the hash table as an auxiliary data structure, and the budget M was solely used for the data partitions. When building a hash table after processing all the records from the build side, the operator would thus allocate pages for the hash table from working memory without any limitation. An OOM could happen when building the hash table for in-memory partitions. This initial choice is depicted in the dashed rectangle in Figure 15. This issue was resolved by changing the design of the hash join operator to also consider the hash table size. However, a unique challenge regarding the hash table exists when building the hash table, unlike in the hash group-by case where the data partition table and the hash table grow gradually together from the beginning of the process. Since the budget M is already used to control the size of the data partition table during the build phase of a join, there may not be enough space left for the hash table in some cases. In such a case, the operator must spill some additional data partitions to disk to make space for the hash table. Each time an in-memory partition is spilled to disk, the operator needs to estimate the hash table size (based on the number of current records in memory) to find the point where it can stop spilling partitions to disk. Once this spilling of partitions is done and there is enough space for the hash table, the operator builds the hash table over all the in-memory partitions. A minor detail is that the estimated hash table size and the actual hash table size can be different, so the budget M may not be quite 100% utilized. As described before, the default capacity of a content slot in the table is three. However, since it is difficult to estimate the actual number of hash value collisions and hash slot overflows, the estimation logic conservatively assumes that a content slot in a hash table is occupied by only one record pointer to prevent the actual hash table size from possibly growing beyond the estimated size. However, due to some hash value collisions, the actual memory usage of the hash table is usually smaller than the estimate. The hash table currently allocates/deallocates pages from the hash table page manager, and this manager shares the same page pool with the data partition page manager.

5 | MEMORY-INTENSIVE OPERATOR: INVERTED-INDEX SEARCH

Text search in AsterixDB is implemented based on a technique that is covered in one of the project's previous papers.[31] In short, AsterixDB uses an inverted index to expedite text search. Here, we use a keyword search to illustrate how AsterixDB conducts an inverted-index search. The following SQL++ query asks AsterixDB to return each comment field value that contains the keywords "yosemite" and "park."

```
select t.comment from TravelBlog as t where ftcontains(t.comment, ["yosemite", "park"]);
```

To perform a keyword search using an inverted index, AsterixDB first extracts the query keywords and retrieves the inverted lists of these keywords from an applicable index. An inverted list for a keyword holds the primary keys whose indexed field contains the keyword. It then processes the inverted lists to find all primary keys that occur N times. This N can be calculated based on the predicate.[31-33] For example, for the above keywords q = "yosemite" and "park," AsterixDB would first fetch the inverted lists of these two keywords. It then considers the primary keys that appear on both lists and

[‡‡‡‡]After each iterative step, if the size reduction ratio of the partition is under the space threshold (currently 0.2), the hash join operator switches to perform a Block-Nested-Loop Join for the partition pair since the hash join is not effective due to the distribution of the records in the partition.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

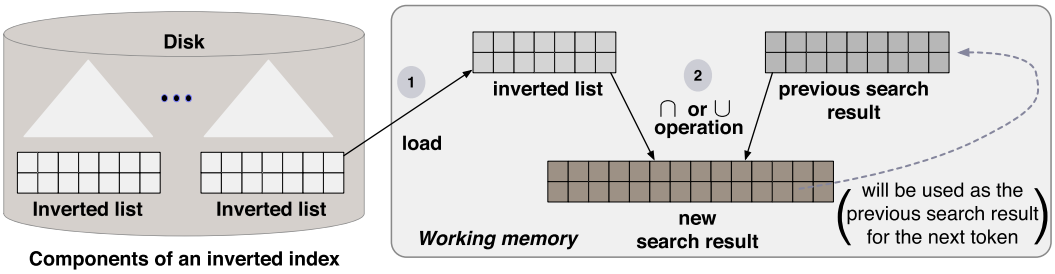


FIGURE 16 Inverted-index search: each logical iterative step [Colour figure can be viewed at wileyonlinelibrary.com]

passes them to the next operator. The records that correspond to these primary keys will be fetched and returned based on the predicate.

The conceptual operation flow for the inverted-index search operator is shown in Figure 16. The operator first generates a token list from a query string. It orders the tokens based on the length of their inverted lists from the shortest to longest. For each token, the operator fetches its inverted list and generates an intermediate result based on the previous result and the current inverted list. After processing all tokens, the operator generates the final results to be passed to the next operator. More details of this approach can be found in one of our previous articles.[31] Specifically, in each iterative step, after loading its inverted list for one token, the operator performs a set operation (intersection or union) between the current inverted list and the previous intermediate search result to generate a new search result. Which set operation (intersection or union) is used depends on the query predicate. If the query contains a disjunctive (OR) predicate, the operator performs a union operation. If the query contains a conjunctive (AND) predicate, the operator performs an intersect operation. Physically, each search result consists of zero or more pages, and a page consists of one or more entries. Each entry contains a primary key and its occurrence count during the search process. Initially, there is no previous search result. Thus, when processing the first token, the previous search result is an empty list. For the next token, this new search result for the previous token becomes the previous search result, and the operator reads the inverted list of the next token from the inverted index to generate a new search result. At any moment during this iterative step, two intermediate search results ("previous" and "new") and one inverted list will hopefully all reside in memory. When processing the last token, the final search result is created to keep the final results to be passed to the next operator after the last set operation is finished.

Since the size of an inverted list can range from only one primary key to any number of primary keys, the size of the intermediate search results can be any size, too. In addition, we saw that the operator needs to deal with two search results and one inverted list at a time. These observations suggest that it is crucial to control the operator's memory usage. A naive implementation might try to load the entire inverted list of the current token into the buffer cache to boost the performance if the buffer cache can hold it. The issue in this implementation is that the inverted list has to be accessible during the set operation. Unlike other page-level buffer cache related operations,[28] this duration can be long depending on the size of the previous search result. As a consequence, the operator needs to somehow support disk-based operation so that its memory usage can be properly controlled within the budget M regardless of the inverted list size.

To support efficient disk-based operation, we set the minimum number of required pages to four since the inverted-index search operator needs at least one page for the current inverted list, one page for the previous search result, one page for the new search result, and one page for the final search result, as shown in Figure 17. When there is not enough memory, these pages will be used as I/O buffers to read the current list and the previous result from disk. Also, one page will be used to incrementally write a new search result to disk. One page will be used to keep the final result. Since the next operator in an AsterixDB query plan only needs to access one incoming page at a time, passing one page to the next operator is enough. When this buffer page becomes full, the operator pauses the search operation and passes the content of the page to the next operator. After the next operator accepts the result, the operator continues the search process to fill the result buffer page in again.

Since the essential part of the inverted-index search in AsterixDB is traversing a new inverted list using the previous search result, we give the highest memory allocation priority on the inverted list. Except for the two buffer pages that are used for reading/writing the previous and the new search result from/to disk, the rest of the budget M is used to read a chunk of the current inverted list. Reading a large chunk of the inverted list allows the use of a more efficient means to traverse the chunk (eg, via a binary search) to avoid an expensive full-scan that would traverse all keys in the chunk one

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

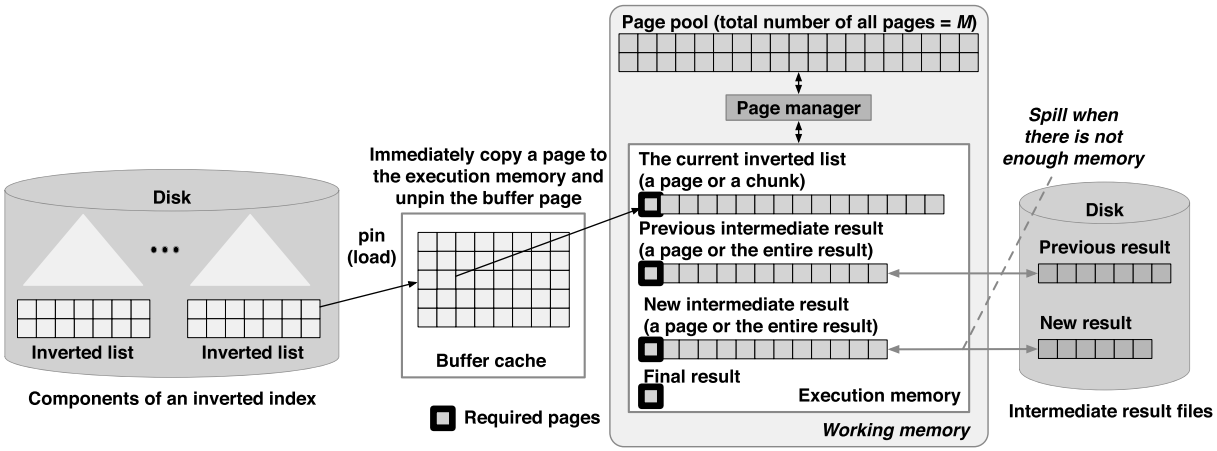


FIGURE 17 Inverted-index search: actual implementation [Colour figure can be viewed at wileyonlinelibrary.com]

by one. If the operator is able to read the current inverted list in its entirety, it then checks whether it can also read the previous result entirely into memory (if the result was written to disk). If there is still leftover memory, the new result can also be written into memory. Thus, if the operator has a large-enough budget M, the search can be completely performed in memory.

In addition, to avoid pinning inverted list pages for a long time in the buffer cache, when the operator reads a chunk of the current inverted list, as soon as a page from the current inverted list has been loaded into the buffer cache, the operator copies its contents to a page in the execution memory of the inverted-index search operator. After this copying of the page is done, the page in the buffer cache will be freed immediately. Also, as one additional optimization, the inverted-index search operator avoids creating a separate final result if the number of search tokens in the query is just one. The operator simply reads entries from the inverted list page by page and returns each page to the next operator in the one-token case (one inverted list). Thus, the redundant copying operation is not required in this case.

6 | GLOBAL MEMORY MANAGEMENT

So far, we have discussed the individual memory-intensive operators in detail. Even though each memory-intensive operator conforms to its budget properly, a higher level of memory management is still necessary. In this section, we discuss memory management at the global system level. We present a way of controlling the memory impact of the in-memory LSM index components and explain how to handle query admission control. We then discuss another important memory-related issue, namely, how to manage memory for records larger than one disk page.

6.1 | In-memory LSM components

As described in Section 2, as a result of having LSM index structures, we need to explicitly allocate memory to LSM indexes to initially hold the results of insert/upsert/delete operations. This section of memory is called the *in-memory components area*.

To control the size of this part of memory, AsterixDB has three parameters. The globalbudget parameter determines the overall maximum size of all in-memory components. The numcomponent parameter sets the number of in-memory components that each LSM index can have. The purpose of this parameter is dealing with the flush operation of an in-memory component. When an in-memory component is flushed to disk, the status of the component is changed to *immutable* to prevent further modifications during a flush operation. If the number of components for a dataset is one, when it needs to be flushed, its status would be changed to *immutable* and no more insert/upsert/delete operations could be made to the dataset until after the flush. To avoid immediate blocking operations, we set the default number of in-memory components for each index to two so that when flushing an in-memory component to disk, an additional component can be allocated to accept continued insert/upsert/delete operations. Finally, the maxdatasets parameter sets the maximum

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

number of concurrent active datasets in the in-memory components area. Based on the overall size of the in-memory components area, the maximum size that each dataset can have is then decided. All in-memory components of the primary and secondary indexes of a dataset share this budget. For example, if the size of the overall in-memory components area is 800 MB and maxdatasets is set to 10, each dataset can occupy 80 MB in the in-memory components area. Finally, if the number of in-memory components per index is two, 40 MB is shared among primary and secondary indexes at a time.

6.2 | Query admission control

Without a query admission control policy, an OOM could occur when there are too many concurrent queries, even if the amount of memory for the memory-intensive operators, in-memory components, and the buffer cache size are controlled. To solve this problem, AsterixDB has a query admission control feature that considers two factors in each query's plan to manage the number of concurrent queries. The factors are the number of CPU cores desired and the required memory size for the query plan (as shown in Figure 18). For memory, if a query requires more memory than the available amount, it should be immediately rejected. The other factor, the CPU core requirement, considers the degree of parallelism. By default, AsterixDB queries use one execution worker thread per physical storage partition. However, a user can request (via a query parameter called parallelism) to set a different number of worker threads. For instance, if there are four partitions and there is a sort operator in the plan, the default parallelism would use four sort operators in four execution worker threads. A user can request to increase the number of worker threads by setting the parallelism parameter to a higher number (eg, 16). The number of execution threads that contain the sort operator will then be increased, respectively. In addition, if the system were to only allow one execution thread per CPU core, when a thread is performing disk I/O, the CPU core would be idle even though it can handle other operations. Thus, in order to fully utilize CPU cores, when considering the number of CPU cores, AsterixDB's admission control policy also includes a tuneable coremultiplier parameter to control the number of maximum concurrent queries per core (which defaults to three admitted queries per core).

When an AsterixDB cluster instance starts up, it collects the number of CPU cores and available JVM heap memory that the instance can use by checking the JVM runtime. It uses the collected data as the maximum available capacity, and it considers each query's plan in their arrival order. Using the query plan, AsterixDB calculates the required number of CPU cores and the required memory size. (The details of this calculation will be discussed shortly.) As shown in Figure 19, a query can be executed immediately if both resources are available. A query will be queued if one of its required resources is not available. A query that is placed in the execution queue will be executed later, once the necessary resources become available. A query will be rejected immediately if one of the constraints is not satisfiable based on the system's maximum capacity.

As mentioned earlier, the degree of parallelism per query is controllable via the parameter called parallelism. Based on the number of physical dataset partitions and parameter, the number of CPU cores that a plan will request is decided. The required overall memory for a query plan is computed based on the plan's operators and their budgets. Each operator's memory requirement can be computed based on the characteristics of the operator. If an operator is not memory-intensive, the control logic assumes that the operator requires only one page. The memory requirements of memory-intensive operators can be computed from their system configuration parameter or a query-specific parameter. For instance, the per-operator sort memory size (eg, 128 MB) can be set in the system configuration file. Its value will be used to compute

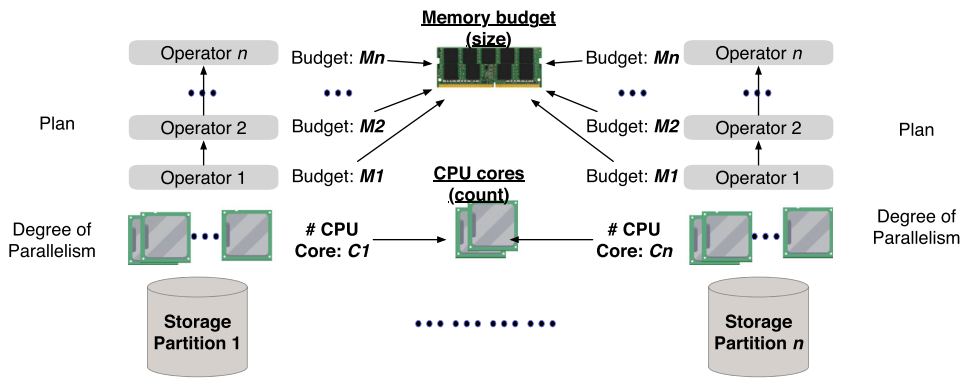


FIGURE 18 Factors in query admission control [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

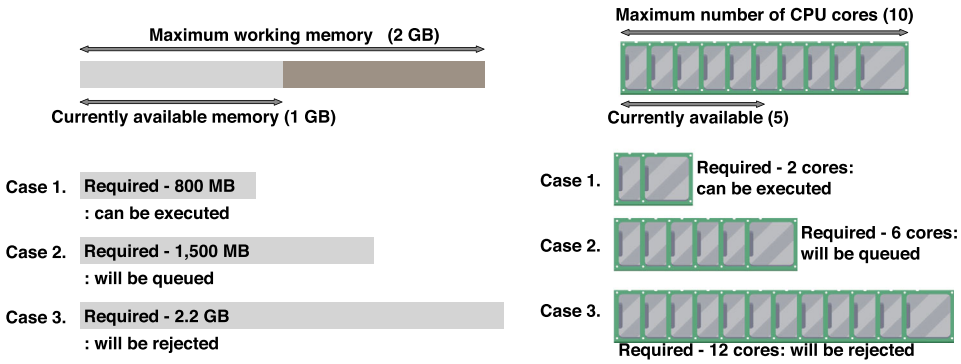


FIGURE 19 Example query admission cases [Colour figure can be viewed at wileyonlinelibrary.com]

the memory requirement of each sort operator unless it is overwritten by a query-specific parameter included in the given query (by using a set statement).

In AsterixDB, there are two query admission policies—conservative and execution-stage-awareness. The conservative admission control logic assumes that all operators in a query plan might execute at the same time, making the query's required memory estimate the sum of the requested memory per operator. This conservative admission control is sufficient to eliminate OOMs and give enough resources to each query. However, there may be a situation where allowing more concurrent queries while still ensuring the stability of the memory behavior of the system is desirable. That is, because of wait-for dependencies among the activities in operators, there are usually multiple execution stages in a query execution plan. It is thus not the case that all operators will always be executed in parallel. For example, as described earlier, there are two activities in the hybrid hash join operator—build and probe. The probe phase cannot start until the build phase finishes, so the operators that follow the join operator in the plan cannot be executed together with the build phase of the given join operator. Based on this fact, the execution-stage-awareness admission control policy considers the generated execution stages (steps) in the query plan. Its logic calculates the required number of CPU cores and the required memory size per stage. Among all stages, it then finds the maximum number of CPU cores and the maximum memory requirement and sets those as the required resources for the plan. (A comparison between the two policies will be discussed in Section 7.)

6.3 | Handling big objects

When discussing the memory usage of the memory-intensive operators so far, our implicit assumption has been that each record will fit into a page. All described operations work based on this assumption. However, this assumption is not always true since the length of some record fields is essentially unlimited in AsterixDB. We obviously cannot ask users to increase the page size each time they need to store records whose length is greater than the page size. Thus, we need to deal with the situation where a record's size is greater than one page. For example, in the second phase of the sort operator, the operator (as described earlier) assigns one page per run file on disk to merge them. If the run file has a large record that cannot fit into one page, the operator cannot read this record and thus the second phase cannot be finished. This section explains how AsterixDB accommodates large records without exceeding its memory budget.

6.3.1 | Adjusting the storage layer

Let us first focus on the storage aspects of large objects. To make the design simple and to minimize the codebase impact, we chose to keep the basic page size identical (and fixed), as allowing it to vary would affect many parts of the system. For example, we did not want to introduce a situation where one page is 100 KB and another page is 200 KB. Instead, our solution is to store a large record on multiple pages as shown on the right side of Figure 20. We can logically coalesce multiple pages into one sequential block of buffer cache pages (a "logical" buffer page) so that a large record still can fit into one logical page. Runtime operators, whose code assumes the bytes of an object to be in contiguous memory, will also treat this logical page as one page although its actual physical on-disk storage consists of multiple sequential pages. To reflect this concept, we add new metadata on such an object's first page called the *page multiplier* and let it contain the

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

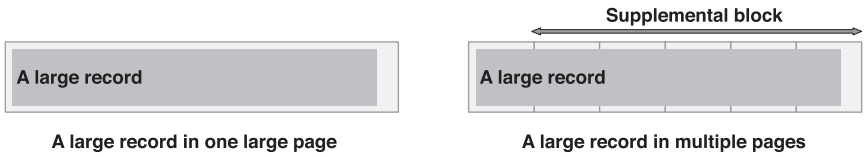


FIGURE 20 Two ways to store a large record

number of physical pages. For instance, the page multiplier in the first page on the right side of Figure 20 is set to 6 since the object consists of six physical pages.

At runtime, by checking the page multiplier upon reading the first page, the system can tell whether a page that has just been read from storage is logically a large page that contains a large record. After the metadata in the first page, the contents of a large record are stored across all pages including the first page, as shown in Figure 20. The second page through the last page only contain bytes of the record, and there is no metadata in those pages. We call those pages a supplemental block for this reason. Thus, a logical page consists of its first page and possibly a supplemental block that spans one or more additional pages. The beginning page ID of the supplemental block is also stored in the first page so that a logical page can be finally fetched using the page multiplier and this page ID.

Another storage consideration is how to intermix a large record with nonlarge records. AsterixDB will attempt to separate pagewise a large record from other nonlarge records whenever possible. If a large record is being inserted into a page where the number of records in the page is less than two, it logically expands the page by setting the page multiplier to accommodate the large record. (The intuition behind not separating the new large record from the existing a small record is to avoid creating a page whose space utilization is very low.) However, if there are multiple records in the page, AsterixDB instead inserts the large record into a new page. The reason this is done is to prevent the costly need to fetch multiple physical pages when accessing most of the regular-size records. That is, if a large record and many regular-size records were allowed to reside in the same large logical page, even when accessing its regular-size records, multiple physical pages would need to be fetched to construct a logical page.

With this approach, the actual writing of a record to disk and reading it from disk are executed as follows. For a write operation, AsterixDB checks the metadata of a page in the buffer cache to get the actual number of pages and writes those pages sequentially to disk. When AsterixDB needs to read a logical page from disk, it reads the first physical page and checks its page multiplier. If it is greater than 1, the supplemental block of the page is then read into the buffer cache too. To make this possible, the buffer cache first allocates a contiguous space whose size is equal to the total number of physical pages. It copies the first page to that space and then loads the supplemental block into the allocated space.

6.3.2 | Adjusting the runtime operators

Because of the logical page approach to handling big objects, when runtime operators read a logically large page, it can be regarded as "one page." Thus, all runtime operators can access a logical page regardless of its actual size. However, one thing does need to be adjusted: the actual size of an incoming page can be different now, and this fact should be reflected in the memory management logic of each operator. The detailed considerations for each memory-intensive operation are as follows.

Sort operator: By default, when the operator merges runs, it allocates one buffer "page" to each run. This page should now be a logical page since if it were a regular page, there could be a case where the operator could load a logically large page from a run and exceed its memory budget in doing so. Thus, when loading each incoming page into the execution memory of a sort operator during its first phase, the operator keeps track of the current run's maximum number of physical pages. When it spills a run to disk, it also remembers the information about the maximum number of physical pages for that run. After creating all runs, when allocating an input buffer page for a run, the operator allocates the maximum logical page size on a per run basis. This way, any logical page in each run can be read without an issue during the merge phase.

Hash group-by operator: The hash table's memory logic does not need to be revised since the hash table only contains pointers to the actual groups and their aggregated results in the data partition table. However, we did need to take care of a possible aggregation situation where a large record is being aggregated in a spilled data partition. For this case, we extended the concept of a page to a logical page in the page manager for this operator as well.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Hash join operator: Similar to the hash group-by case, when a logical page that contains a large record is inserted into a spilled data partition during the build or probe phase, the operator still keeps only one (now logical) page in the spilled partition.

Inverted-index search operator: An inverted list only contains a list of primary keys. AsterixDB only allows an inverted index to be built on a regular fixed-size primary key. It is not realistic to expect a primary key to be a large field that spans multiple pages, so we do not allow this case.

7 | EXPERIMENTS

We have performed an experimental evaluation of our budget-driven memory management implementation using both synthetic and real datasets. First, we measured the potential consequences of not including the size of all supporting data structures in the budget for memory-intensive operators. Next, we conducted several experiments to verify that the current implementation of memory-intensive operators is scalable. We then measured the effect of having large records during sort and hash join operations. Finally, we explored the effect of the query admission policies in AsterixDB. For these experiments, we used a single-node cluster to host an Apache AsterixDB (0.9.4) instance. This instance had one physical storage partition since we wanted to observe and analyze the behavior of the system in a simple environment. This node ran CentOS 6.9 with a Quadcore AMD Opteron CPU 2212 HE (2.0 GHz), 8-GB RAM, 1 GB Ethernet NIC, and had two 7200 RPM SATA hard drives. We used only one hard drive since we created only one physical storage partition. (The node's other drive was reserved for the system's transaction log.) Table 2 shows the AsterixDB configuration parameters used for the experiments.

7.1 | Datasets

We used a variant of the Wisconsin benchmark dataset[34] to check the behavior of memory-intensive operators since we could control various aspects of the input data. We also used the TPC-H benchmark dataset[§§§§] to perform the query admission control experiment since TPC-H contains many memory-intensive operators in its queries. We also used a real dataset of Reddit comments[¶¶¶¶] to test the behavior of the inverted-index search operator. Table 3 shows the characteristics of all datasets. For TPC-H, the input format of the raw data file was CSV. Thus, it did not contain any additional information. In contrast, AsterixDB needed to store some metadata per record and per page. Thus, the size of the corresponding stored TPC-H dataset was greater than that of the raw input data file. For the basic Wisconsin dataset, the format of the raw data files was JSON. Since we declared the records' fields in the datatype, the AsterixDB records do not have to include the field names, so the size of the AsterixDB dataset was smaller than that of the original JSON file. However, for the five Wisconsin datasets with a large string field, the size of the AsterixDB dataset was greater than that of the original JSON file since some unused space in pages existed for each large string field instance (ie, due to internal fragmentation). For instance, for the Wisconsin-Norm-0 dataset, the page size was 32 KB and a typical large string field was 26 KB, leaving 6 KB unused on such a record's page.

TABLE 2 AsterixDB parameters for the experiments

Parameter	Value	Parameter	Value
Total memory allocated to the instance	6 GB	Sort memory	128 MB
LSM component memory	1 GB	Join memory	128 MB
Disk buffer cache	2 GB	Group-by memory	128 MB
Working memory	3 GB	Inverted-index search memory	128 MB
Runtime page size	32 KB	Storage page size	32 KB

Abbreviation: LSM, log-structured merge.

[§§§§] http://www.tpc.org/tpch/
[¶¶¶¶] https://files.pushshift.io/reddit/comments/

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 3 AsterixDB datasets

AsterixDB dataset	Cardinality	Rawdata size (MB)	Dataset size (MB)	Notes
Wisconsin	10 000 000	32 193	24 704	No large string field included
Wisconsin-Norm-0	1 000 000	28 647	31 277	A large string field was included. The length of all instances of the field was 26 K.
Wisconsin-Norm-M	1 000 000	28 645	37 733	A large string field was included (normal distribution—the average length: 26 K, the standard deviation: 4.4 K).
Wisconsin-Norm-L	1 000 000	28 651	41 423	A large string field was included (normal distribution—the average length: 26 K, the standard deviation: 8.8 K).
Wisconsin-Gam-1	1 000 000	28 603	39 417	A large string field was included (gamma distribution—the average length: 26 K, the shape: 1.5, the scale: 1).
Wisconsin-Gam-2	1 000 000	28 773	37 651	A large string field was included (gamma distribution—the average length: 26 K, the shape: 1, the scale: 0.5).
TPCH-Customer	1 500 000	234	326	TPC-H Dataset (scale factor: 10)
TPCH-Lineitem	59 986 052	7416	16 150	TPC-H Dataset (scale factor: 10)
TPCH-Orders	15 000 000	1669	2988	TPC-H Dataset (scale factor: 10)
TPCH-Partsupp	8 000 000	1150	1619	TPC-H Dataset (scale factor: 10)
TPCH-Part	2 000 000	233	361	TPC-H Dataset (scale factor: 10)
TPCH-Supplier	100 000	14	21	TPC-H Dataset (scale factor: 10)
Reddit-comment	91 558 594	59 817	103 049	Reddit comment (January 2018)

TABLE 4 A part of the Wisconsin dataset fields

Name	Type	Bytes	Remarks
unique1	int	8	unique, random order
unique2	int	8	unique, sequential
unique3	int	8	unique1
stringu1	string	100	random
stringu2	string	100	random
largeString	string	26 K	random, variable-length HEX string
…			

To control various aspects of the synthetic data such as the selectivity or size of a field, we used a variation of the Wisconsin dataset. We added some extra fields to make it meet our needs. Table 4 shows a portion of the schema of our version of the Wisconsin benchmark dataset. The italicized field represents the extra field we added.

For the experiments where no large string field was involved, we used the basic Wisconsin dataset in Table 3. For performing experiments on records with a large string field, based on the schema of Table 4, we created five more datasets as shown in Table 3 to see how the length distribution of a large string field affected the performance. Each record in those datasets had a large string field, and its average length was the same, 26 KB, which was slightly smaller than the page size (32 KB). This big-object field was a string field that contained a random HEX string. Each character was chosen

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

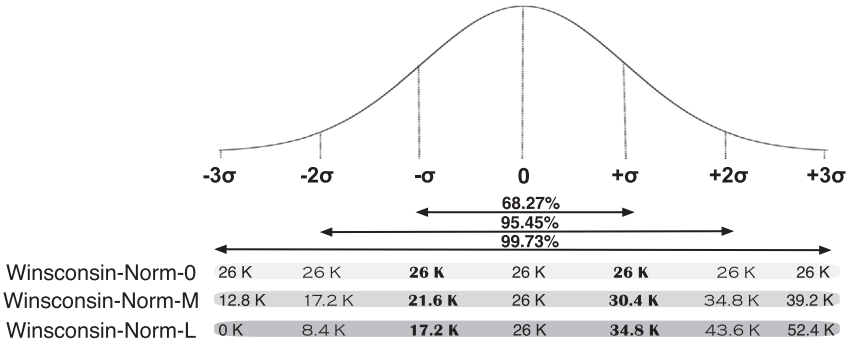
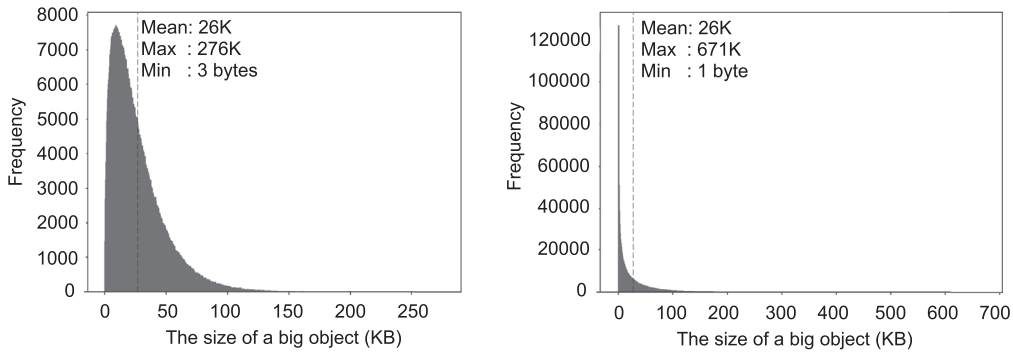


FIGURE 21 Length distributions of large string fields using normal distribution



(A) Gamma distribution 1. (B) Gamma distribution 2.

FIGURE 22 Two Gamma distributions of large string fields used for Wisconsin datasets. A, Gamma distribution 1. B, Gamma distribution 2

randomly among 16 HEX characters (0 to F). The only difference of the 5 latter Wisconsin datasets was that the length distribution of the large string field was different; the first three datasets were based on a normal distribution (Figure 21) and the other two were based on a gamma distribution (Figure 22). Since the standard deviation of the first dataset, Wisconsin-Norm-0, was zero, all large string field instances had the same size (26 KB). The standard deviation of the second dataset, Wisconsin-Norm-M, was 4400, and that of the third dataset, Wisconsin-Norm-L was 8800. Therefore, the third dataset had more records whose length was greater than 26 KB.

The last two datasets were based on a gamma distribution for the large string field's size. The length distribution range of the first gamma dataset, Wisconsin-Gam-1, was smaller than that of the second dataset, Wisconsin-Gam-2, since the latter had a more skewed distribution toward zero.

For the query admission control experiment, we used the standard TPC-H benchmark database with a scale factor of 10. In addition, we used one TPC-H query with two joins, one group-by, and one order-by predicate to use multiple memory-intensive operators to evaluate the memory usage. For the TPC-H dataset, other than creating a few indexes to expedite the query execution time, we did not add additional fields.

For the inverted-index search experiment, we used a nonsynthetic text dataset. This dataset contained Reddit comments collected in January 2018. We used the dataset's body field to perform a full-text search to check the behavior of the inverted-index search. Since the data did not have a field that could be used as a primary key field, we instructed AsterixDB to assign an additional autogenerated UUID primary key field when importing the data, as AsterixDB currently requires that the records of each dataset have a primary key. Other than this field, no additional fields were added to the Reddit records.

7.2 | Accounting for everything

To measure the potential consequences of not including the size of all supporting data structures in the budget M for memory-intensive operators, our first experiments compared a basic implementation of each operator that does not include the size of all data structures versus the current AsterixDB implementation of memory-intensive operators. We measured the memory footprint of the data structures during memory-intensive operations.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

We first considered a case where the portion of each input record referenced by a query only consisted of one to three integer fields from the Wisconsin dataset. Since the size of an integer field is relatively small (less than 10 bytes), the number of records that could fit into the execution memory of each memory-intensive operator was greatest in this case. Additionally, we measured a more relaxed (typical) case where the referenced part of an input record consisted of one integer and one or two string fields whose length was 100. We used the same source dataset for that case. For all memory-intensive operators, we set the operators' memory budget to 128 MB (Table 2).

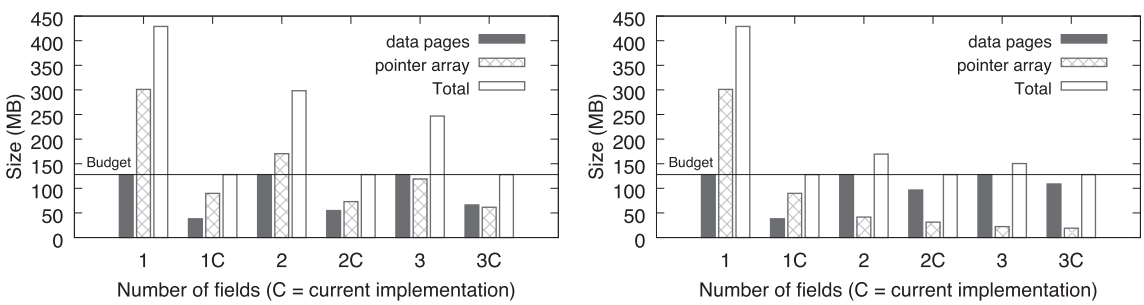
For each case, we first determined the input record cardinality where each operator would use its budget at its full capacity. To find this cardinality, we gradually increased the number of input records to each memory-intensive operator and found the exact operating point where adding any additional input records could cause the operator to switch to disk-based operation, and we used that cardinality to measure the size of the in-memory data structures. For instance, if the sort operator started generating run files on disk once the number of input records reached 2 000 000, we used 1 999 999 records as the input cardinality and measured the operator's memory use at that operating point.

7.2.1 | Sort operator accounting

To study the memory usage of the sort operator, we used query templates Q1 and Q2 shown in Appendix A. In Appendices A-D, we present the queries used in Section 7. Each template includes an order by clause that conducts a sort operation using an integer field where the record consists of three fields including this field. Q1 consisted of three integer fields and Q2 consisted of one integer field and two string fields. For the one-field and two-field cases, we adjusted the number of fields in the select clause. In the select value count clauses, we included all fields so as not to let the optimizer project out the fields before the sort operator.

By varying the number of fields in both templates, we measured the memory footprint of the data structures when the budget was fully utilized. The results are shown in Figure 23. The current AsterixDB sort operator implementation is displayed with the suffix C after the number of fields in the figure. As described earlier, the size of the record pointer array during a sort operation is generally not negligible. When the record to be sorted is small, consisting of only one integer field, the pointer array size was 301 MB, almost three times the allocated budget (128 MB). In the implementation where the budget M was solely used to track the memory used to load data pages, the sort operation would actually use 430 MB in total even though the operator's budget was set to 128 MB (on the left side of Figure 23A). As the number of integer fields increases, the record pointer array size becomes smaller since the number of records that fit in the execution memory becomes smaller. However, even for the three-integer-field case, the record pointer array size was 119 MB, which was similar to the allocated budget M. In the current AsterixDB sort implementation, we can see that both the record pointer array and the data pages properly share the budget M, so their total size never exceeds the budget M.

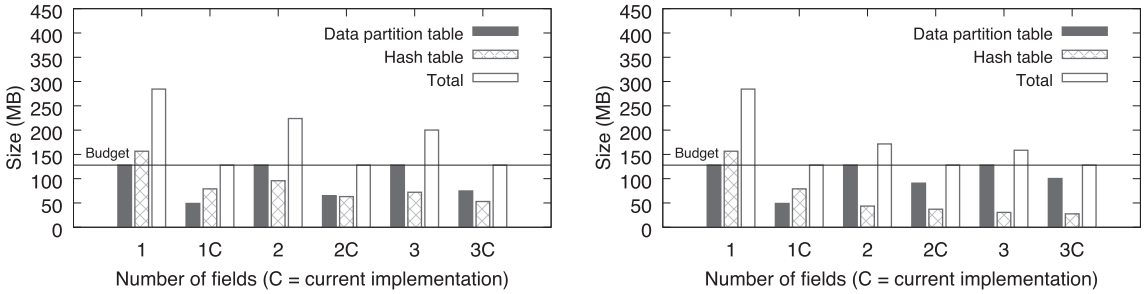
When a record consisted of one integer and one or two string fields whose length was 100, the record pointer array's relative overhead was significantly reduced since there were fewer records in the execution memory, as shown in Figure 23B. Still, the size of the record pointer array was not negligible. For example, in the case where a record consists of one integer and two string fields, the pointer array size was 19 MB, which was about 15% of the budget (128 MB).



(A) Integer fields. (B) String fields.

FIGURE 23 The size of data structures for a sort operation. A, Integer fields. B, String fields

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Integer fields. (B) String fields.

FIGURE 24 The size of data structures for a hash group-by operation. A, Integer fields. B, String fields

7.2.2 | Hash group-by operator accounting

Next, we evaluated the memory usage of the hash group-by operator in a similar fashion using query templates Q3 and Q4 in Appendix A. Each template included a group by clause on all fields that were returned with a hint that instructed the optimizer to generate a hash-based group-by operation. Q3 consisted of three integer fields and Q4 consisted of one integer field and two string fields. The other details were similar as in Section 7.2.1.

In this experiment, we first identified the maximum input record cardinality for which the operator would not spill any partitions to disk by gradually increasing the number of input records. We then measured the hash table size at this cardinality since the execution memory was utilized at full capacity at this point. Figure 24 shows the memory footprint of the data structures versus the number of fields. Again, the current AsterixDB implementation is displayed with the suffix C after the number of fields in the figure. As we can see in Figure 24A, the hash table size for the one integer case field was 156.5 MB in the basic implementation, and the budget M (128 MB) was entirely used for keeping the data partition table. The total memory used was thus 284 MB, which is more than twice the assigned budget.

For the string-field cases, the hash table size was smaller than the integer-field cases since the number of records that fit in memory was smaller. However, even for the two-string-field case, the hash table size was 30.6 MB, which was about 25% of the assigned budget. The current implementation addresses this issue successfully. We can see in Figure 24B that the data partition table and the hash table now utilize the budget together, so their total memory use was always within the assigned budget.

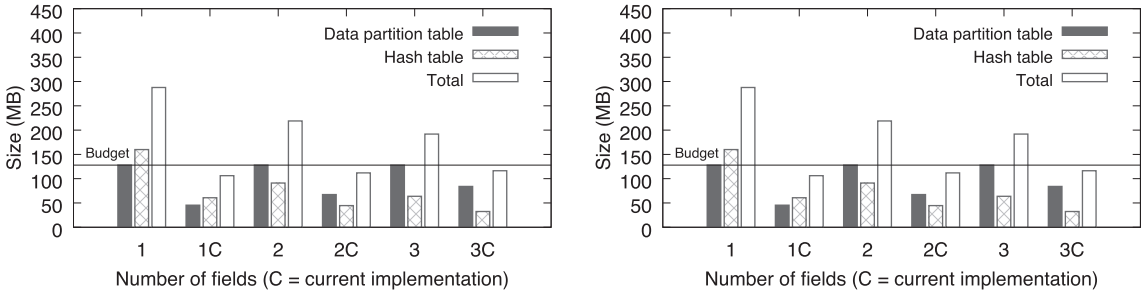
7.2.3 | Hash join operator accounting

We next measured the memory usage of the hash join operator, which uses a hash table and a data partition table. We used query templates Q5 and Q6 in Appendix A. Each template contained hash join conditions on all the returned fields, where the record consisted of three fields. Q5 consisted of three integer fields and Q6 consisted of one integer field and two string fields. The other details were similar as in Section 7.2.1. For these queries, we selected 1000 records from the outer (probe) branch. Similar to the previous experiments, we identified the maximum number of records from the inner (build) branch where the join operator would not spill any data partitions to disk by gradually increasing the number of records. We used this cardinality for the inner branch so that the memory usage would be at a maximum. Figure 25 shows the memory footprint of data structures per number of fields. Again, the current implementation is displayed with the suffix C after the number of fields in the figure.

Like the hash group-by operation, when the size of the hash table during a hash join operation was not considered within the budget M, as we can see in Figure 25A, the hash table size for the one integer field was 159.9 MB, while the budget (128 MB) was entirely used for the data partition table. The resulting total amount of memory used by the query was 287.81 MB, which was more than twice its assigned budget.

For the string-field cases, as before, the number of records used was smaller since the record size was larger. We can see that the hash table overhead was smaller compared to the integer-field cases. For instance, when there were one integer field and two string fields, where the length of each string field was 100, the hash table size was 12.4 MB, while

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Integer fields. (B) String fields.

FIGURE 25 The size of data structures for a hash join operation. A, Integer fields. B, String fields

the entire budget M was used to hold the data partition table. In total, the hash join operator used 140.4 MB, which was greater than the assigned budget (128 MB).

We can see in the hash join figures that the space utilization during a join operation was always kept within the operator's assigned budget in the current implementation. Note, however, that the utilization of the hash join memory budget is a bit lower than for the sort and hash group-by operations. Those operators utilized almost 100% of their budget, while the hash join operator did not fully utilize its allotted budget. For instance, in the three-integer-field case, the total execution memory usage was 116.3 MB, which was 90.8% of the assigned budget (128 MB). This is because the hash table for the hash join operator is built after processing all records from the build side, as described in Section 4.2. Based on the number of records in the in-memory data partitions, the operator estimates the expected hash table size by assuming that each record resides in a separate hash slot. This estimation is a safe way to ensure that the hash table size will not exceed the estimated budget. In reality, because of hash value collisions, not all hash slots were allocated and used. That is why the space utilization was lower than 100%. (We did not observe this behavior in the hash group-by operation since its hash table gradually grows with the data partition table, and the spilling of an aggregate data partition only happens when its space utilization reaches nearly 100%.)

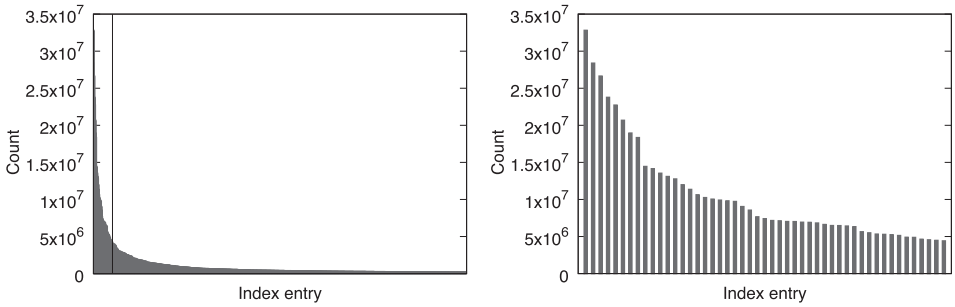
7.2.4 | Inverted-index search operator accounting

We now examine the memory use of the inverted-index search operator. As described in Section 5, there can be two intermediate search results and one final search result in memory during an inverted-index search. If the operation is a union, the inverted list for the current token will be added to the new search result. Thus, the memory footprint of these data structures can easily be significant if every operation is performed in memory. We can estimate the footprint of these data structures by examining an actual inverted index. Figure 26A shows the size of the top 1000 inverted lists in a full-text index on the body field of the Reddit-comment dataset. We can clearly see that the distribution follows the Zipf's law. Figure 26B zooms in to show the first 50 entries of the previous figure. Let us examine the first entry of the inverted list. The frequency of the first entry was about 34 million. With the primary key size being 16 bytes, the size of this inverted list was about 518 MB. Therefore, processing this inverted list would require about 1 GB of memory even before considering the previous search result size. Note that this dataset contained only one month of the entire Reddit comment data. If more data were inserted, this size would increase proportionally as well. We do not include an actual comparison between an implementation that can only operate in memory and the current memory-conscious implementation since the former is not a budget-based approach at all. Our current implementation has an assigned budget M and properly supports disk-based operation.

7.3 | Conforming to the budget

Our next experiments focused on the current AsterixDB operator implementations, which properly use the assigned budget M. We measured the average execution time of each of the memory-intensive operators in the current implementation to verify that they are scalable. That is, transitioning from in-memory to disk-based operation should work seamlessly.

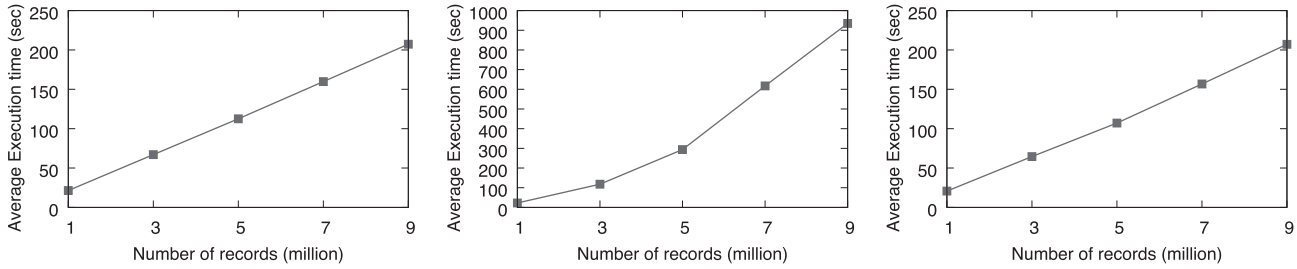
1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Inverted list size (first 1,000 entries).

(B) Inverted list size (first 50 entries).

FIGURE 26 The inverted list size of an inverted index on the body field of the Reddit comment dataset. A, Inverted list size (first 1000 entries. B, Inverted list size (first 50 entries)



(A) Sort.

(B) Hash Group-by.

(C) Hash Join.

FIGURE 27 The average execution time of (A) sort, (B) hash group-by, and (C) hash join

To measure both in-memory and disk-based operations, we varied the number of input records fed to these operations and measured the average execution time of a query. We used the basic (unmodified) Wisconsin dataset to avoid any effect from large string fields. (We will explore those effects separately later.) For each selectivity, we executed 100 random queries where each query used a random contiguous input record range. For example, if the number of desired input records was 1 million, each query used 1 million contiguous records from the dataset. However, the first record's starting position was randomly chosen.

7.3.1 | Sort operator scalability

We first measured the average execution time of the current sort implementation using query template Q7 in Appendix B, which conducts a sort operation. Specifically, the template contains two integer fields. The first field, unique2, is used to limit the cardinality, and the other field, unique1, is the sort key field. We sent 100 random queries where each query had a randomly chosen range on the unique2 field. Figure 27A shows the average execution time of the sort operation as the input size was varied. Based on the operator's budget size of 128 MB, only the 1-million-record case was able to stay within in-memory based operation. The other four cases included varying degrees of disk-based operation. The figure shows that the average execution time of the sort operator scaled linearly with the data size. Note that the linearity is because all of the intermediate temporary runs can be merged in one merge step for the other four cases. For example, since the page size was 32 KB, up to 4096 runs can be merged in one step given a 128-MB budget. Thus, up to 512 GB of data could potentially be merged in one merge step, while the input dataset size was only 24 GB (three times the machine's overall main memory size).

7.3.2 | Hash group-by operator scalability

Next, we checked how well the hash group-by operation scales using query template Q8 in Appendix B, which conducts a hash group-by operation on two integer fields. The first field, unique2, is used to limit the cardinality, and both fields

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

are the group-by aggregate fields. We used 100 random queries where each query had a randomly chosen range on the unique2 field.

Figure 27B shows the average execution time of the hash group-by operation as the input size is varied. As shown in the figure, the average execution time of the hash group-by query is not linear in the input size because inserting aggregate record pointers into groups in the hash table from some spilled data partitions can occur multiple times. As described in Section 4.1, some data partitions may be spilled to disk in each iteration. After processing all records, the in-memory partitions are passed to the next operator, and the next iteration of the hash group-by operation starts by processing the records from the spilled partitions again. Thus, spilled records will be inserted into the hash table at least two times. That is, if a record is spilled k times, this record is inserted into the hash table $k + 1$ times. The first case in Figure 27B, where the number of records was 1 million, finished without spilling any data partitions to disk. When the number of records was 3 million, some partitions were spilled to disk and the operation finished in the second iteration. When the number of records was 5, 7, and 9 millions, the operation finished only after the third iteration. Note that we can see a linear trend within the same number of iterations (cases 5, 7, and 9).

7.3.3 | Hash join operator scalability

For the hash join operator, we used query template Q9 in Appendix B, which conducts a hash join operation on one integer field where the record consists of two integer fields. From the outer probe branch (denoted by r), 1000 random records were chosen for each query. For the inner build branch (denoted by s), we used five selectivities, namely, 1, 3, 5, 7, and 9 million records. For example, if 9 million records were selected from the build branch, they were inserted into the corresponding partitions using the hash value of the join field during the build phase. The hash table was built after processing all records. In the probe phase, the randomly chosen 1000 records from the probe branch were processed. During the build phase, only the first case (1 million records) did not spill any data partitions to disk. The other four cases spilled some partitions to disk. As we increased the number of records, the number of spilled partitions increased as well.

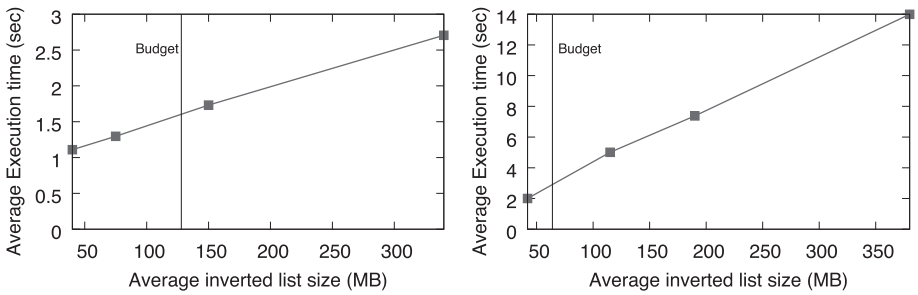
The average execution time for this experiment (Figure 27C) is different from Figure 27B, which showed the average execution time of hash group-by queries. Although both the hash join and hash group-by utilize a hash table and data partition table, the trend for the same number of records is different because a pointer to an actual record is inserted into the hash table only once for all records in the hash join case. That is, when a data partition from the build side is spilled to disk, the operator spills the corresponding partition from the probe side to disk, too. The operator chooses the smaller spilled file as the build side in the next phase and recursively repeats the build and probe phase for each spilled partition pair. Recall that the hash join operator builds the hash table for the in-memory partitions. Thus, if records are spilled to disk, the operator does not insert those records into the hash table until they are among the in-memory records for a phase. (In contrast, in the hash group-by operation, when a record is being processed, it is always first aggregated in the data partition table and the pointer for its group is inserted into the hash table. Therefore, the hash table in each phase of the hash group-by processes all incoming records.) This is a significant difference between the hash join and hash group-by.

7.3.4 | Inverted-index search operator scalability

The next experiment explored the scalability of the inverted-index search operator. We used the Reddit comment dataset and built a full-text index on the body field. We collected the average execution time of the inverted-index search operator, not the overall query execution time. We did this because we wanted to measure the average execution of both in-memory and disk-based operations, and the experiment required accessing a large number of primary keys on an inverted list to switch from in-memory operation to the disk-based operation. For text searches, AsterixDB needs to access the primary index to fetch each actual record to verify the predicate since it does not do locking during a secondary-index search. As a result, a large cardinality from an inverted-index search would cause primary-index lookups to dominate the execution time of the overall query, shifting attention away from the operator of interest.

We first used query template Q10 in Appendix B with two keywords in the predicate and using a conjunctive (AND) search to measure whether the inverted-index search operator indeed reads the inverted lists within its budget M. This query returns the count of records that satisfy the full-text search predicate. When the assigned budget is greater than the size of an inverted list, the list can be loaded into the execution memory all at once. When the inverted list size is greater than the budget, the operator must divide the inverted list into chunks and read a chunk at a time. For this experiment,

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Two keywords (AND). (B) Three keywords (OR).

FIGURE 28 The average execution time of inverted-index search queries. A, Two keywords (AND). B, Three keywords (OR)

we first created a sorted keyword list containing all keywords based on their inverted list size (MB). To generate each query, we used this list to randomly choose the first keyword where the frequency range was between 100 000 and 120 000 (about 2 MB). The purpose of the first keyword was to generate the search result that would then serve as the previous search result so that each primary key in this result will be used to traverse the inverted list for the second keyword. We then randomly picked the second keyword based on its average inverted list size. We used three distinct ranges where the average sizes of the inverted list for the keyword were 75, 150, and 340 MB, respectively. Based on an operator budget of 128 MB, inverted lists whose average size was 75 MB could be read as one chunk. For the 150- and 340-MB cases, the lists had to be divided into multiple chunks. Figure 28A shows the average execution time of an inverted-index search operator for query Q10 in Appendix B. We can see that the execution time was proportional to the average size of the inverted list and that the trend was linear.

The case above focused on reading an inverted list. Next, we increased the number of keywords to three and performed a disjunctive search (OR) to union all three inverted lists so that spilling of an intermediate result to disk would be needed for large inverted lists. Figure 28B shows the average execution time for the three-keyword query. We varied the inverted list size of each keyword so that, except for the first case (total inverted list size = 42 MB), the other three sizes generated intermediate results on disk. Here, we set the budget to 64 MB to cause the switch from in-memory operation to disk-based operation to occur sooner. Also, one additional reason to adjust the budget size was that the number of the inverted lists whose size was similar or greater than the original budget (128 MB) was small in the Reddit data. In the figure, we can see that the inverted-index search indeed handled the data linearly regardless of the size of the inverted list. Switching from in-memory operation mode to disk-based operation occurred seamlessly. For instance, in Figure 28B, we can see that the inverted-index search handled about 23 MB of the inverted list per second regardless of its operation mode.

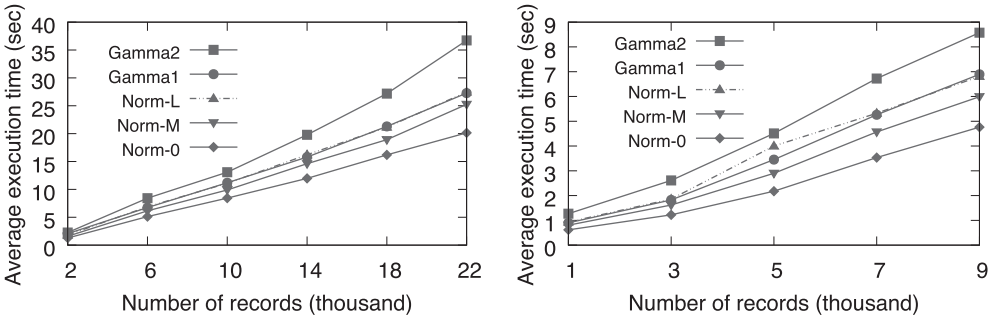
7.4 | Impact of large objects

Next, we investigate the average execution time of the different memory-intensive operators when we include a large string field in each record (a somewhat extreme case). We focus on the sort and hash join operations since grouping based on a large string field would be a rare case. In addition, an inverted-index search is not normally related to large string fields since an inverted index only contains information about secondary and primary index keys. Even if we issued a query that returned a large string field, the inverted-index search process itself would not see these fields. In contrast, for the sort and hash join cases, if a query wants to return a large string field, this field instance is included as a field in a record that will flow through the operation. For these experiments, we used the five Wisconsin variant datasets from Table 3 to measure the effect of having large string fields that result in different record size distributions.

7.4.1 | Sort operator impact

We used query template Q11 in Appendix C to measure the average execution time of sort queries in the presence of large fields. The query returns large string fields ordered by an integer field. For each dataset, we varied the cardinality and issued 100 random queries as we did before. For the large-field case, when the number of records was greater than 2000 on the dataset Wisconsin-Norm-0, the sort operation had to generate runs on disk.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Sort. (B) Hash join.

FIGURE 29 Average execution time of queries with large fields. A, Sort. B, Hash join

TABLE 5 Space utilization in the pages read by the query (22 K records)

Dataset	Norm-0	Norm-M	Norm-L	Gamma1	Gamma2
The total volume of all pages read (MB)	575.19	580.92	571.34	491.35	393.56
Used space percentage	89%	69%	56%	56%	54%
The total volume of all large pages read (MB)	0.00	118.67	197.44	193.25	167.27
Large page percentage	0%	20%	35%	39%	43%

Figure 29A shows the average execution time of sort queries with different large-field-size distributions. The average execution time of dataset Wisconsin-Gamma2 was the highest and that of Wisconsin-Norm-0 was the lowest. The reason is that the sort operator has to access the disk twice to read a record with a large field if it does not fit into a regular page. Recall that it first reads a single disk page and checks the page multiplier to see whether it is a logically large page; if so, it needs to read the accompanying supplemental block separately. Therefore, as the percentage of large pages increases, as shown in Table 5, it takes more time to read all of the records. In addition, as the percentage of large pages grew, the effective space utilization in the storage layer also dropped. Thus, in order to read the same number of records, the number of physical pages that need to be read is larger when the space utilization drops. The average execution time for the dataset Wisconsin-Norm-0 was the lowest since its effective space utilization ratio was the highest and its percentage of large pages was the lowest.

7.4.2 | Hash join operator impact

The next experiment measured the effect of having large fields with different size distributions in the case of a hash join. We used query template Q12 in Appendix C to measure the average execution time of hash join queries in the presence of large fields. The query returns large string fields after a hash join is conducted on an integer field. We again varied the cardinality and datasets as was done in the sort experiment. For the large field case, once the number of records was greater than 2000 on dataset Wisconsin-Norm-0, the hash join process needed to spill some data partitions to disk. As in the previous hash join experiment, we selected 1000 random records from the outer branch (probe side) and selected another random number of records from the inner branch (build side). Figure 29B shows the average execution time of the above query on the five datasets as a function of the build-side cardinality. The trend is similar to that of the sort queries in Figure 29A, and for similar reasons.

7.5 | Query admission control

We now explore the effect of the query admission policy in AsterixDB using the TPC-H dataset with a scale factor of 10 and a representative TPC-H query. The explored policies are no admission control, the system's initial conservative admission control, and the system's current stage-based admission control.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

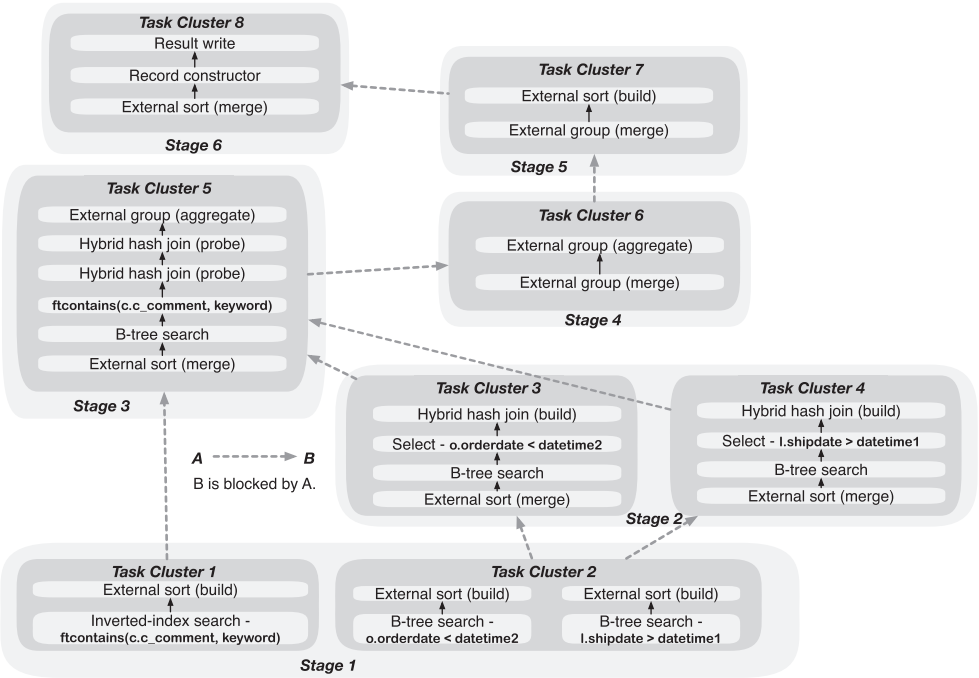


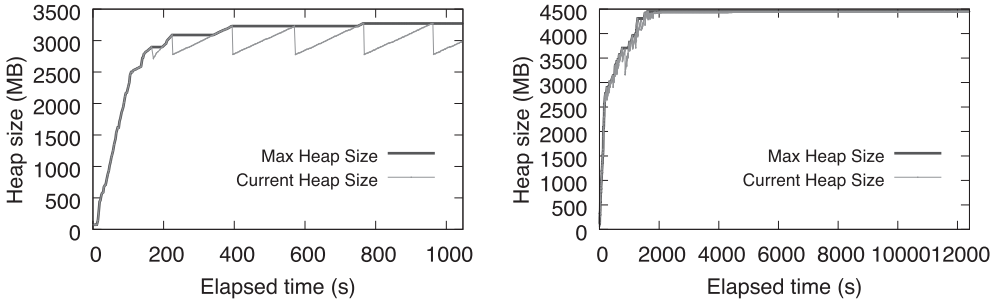
FIGURE 30 The runtime execution activity graph of the TPCH query 3

We used a variation of TPC-H query 3 for this experiment, shown in Appendix D, since this query included various memory-intensive operations, including two joins, one group-by with three fields, and one sort (order by) with two fields. This query returns the unshipped orders with the highest value. We changed the query slightly to also include an inverted-index search. Specifically, we changed the query's string equality comparison predicate to be a full-text search condition predicate. Thus, this query was able to utilize all of the memory-intensive operations discussed here. To expedite the query's execution, we created two B-tree indexes on the TPC-H o_orderdate and l_shipdate fields. We also created a full-text index on the c_comment field.

Figure 30 shows the activity execution graph of the modified TPC-H query. Each task cluster in the execution graph consists of multiple activities that can be pipelined and executed together. For instance, in task cluster 1, an inverted-index search is executed and its result will be pipelined to the build phase of an external sort operator. A dashed arrow in the graph means that the cluster at the end of the arrow is blocked by the cluster at the beginning of the arrow. (That is, the cluster at the end of the arrow cannot start until the cluster at the beginning of the arrow finishes.) For instance, task cluster 5 cannot start until task clusters 1,2,3, and 4 finish. If there are no dashed arrows (including transitive arrows) between two clusters, such as for task clusters 1 and 2, they can be executed in parallel. Therefore, the overall activity graph indicates the possible execution order for the query's activities. It also determines that the maximum amount of working memory that the AsterixDB instance may need to use for this query at any given moment. For example, task cluster 7 is the only cluster that can be executed after task cluster 6 is finished, and it has a sort operator and a hash group-by operator. Thus, task cluster 7 can use up to 256 MB of working memory if the budget of each operator is set to 128 MB. Task cluster 5 will use the largest amount of working memory in this query since it has a sort operator, two hash join operators, and a hash group-by operator.

To observe the memory usage of the AsterixDB instance, we used the Linux jstat command to sample the heap size of the JVM instance every two seconds. Since we allocated 6 GB to the JVM instance, and set the size of the buffer cache to 2 GB and the size of the in-memory components region to 1 GB, the working memory space was about 3 GB. Since this was a read-only query and the data was on disk, the in-memory component section was not used. Thus, the maximum heap usage observed for each experiment should be lower than 5 GB (5120 MB). We ran six workload-generator processes and each process sequentially issued five queries with a random predicate. Thus, 30 queries were sent to the instance in total. We also set a five-minute interval between the start of each process to avoid a convoy effect by staggering their starts. In each experiment, we initially restarted the AsterixDB instance so that its memory usage always started from zero.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Single query. (B) No admission control (multiple concurrent queries).

FIGURE 31 Heap size during an execution of TPC-H query 3. A, Single query. B, No admission control (multiple concurrent queries)

Let us first examine the execution of a single query. The memory usage of one query over time is shown in Figure 31A. This query took about 1000 seconds. When the query starts, an inverted-index search and two B-Tree search operators are executed in stage 1 to fetch the primary keys that satisfy the conditions. Also, three sort operators are initialized and receive the primary keys pipelined from these secondary-index search operators. During this time, the buffer cache is being filled in and the three sort operators are fully utilized. This explains the first sharp increase in the heap usage in Figure 31A. After this increase, we see a few zigzag patterns because when the execution of a task cluster finishes, an operator will release its resources once all activities of the operator are finished. For instance, a hash join operator releases the pages from its execution memory back to the working memory pool after the probe phase is finished (not the build phase). When a new task cluster starts, all of the memory-intensive operators in the task cluster are initialized and start their operations. Note that the memory usage of a memory-intensive operator will increase gradually since it does not preallocate its maximum number of pages when it is initialized. The resulting behavior is thus a series of gradual increases and sharp drops in the query's heap-size graph.

7.5.1 | Policy 1: No query admission control

Figure 31B shows the heap usage of six processes when there is no query admission control in place. Thus, the first six queries were all competing for getting the system resources. After 12 000 seconds of competing for resources, the system reached an OOM state and the instance became unstable. No query was able to finish its execution before the instance reached the OOM state. In fact, during this competition, the system's CPU utilization was high but its disk utilization was low since each query execution thread was struggling to get more memory from the JVM instance.

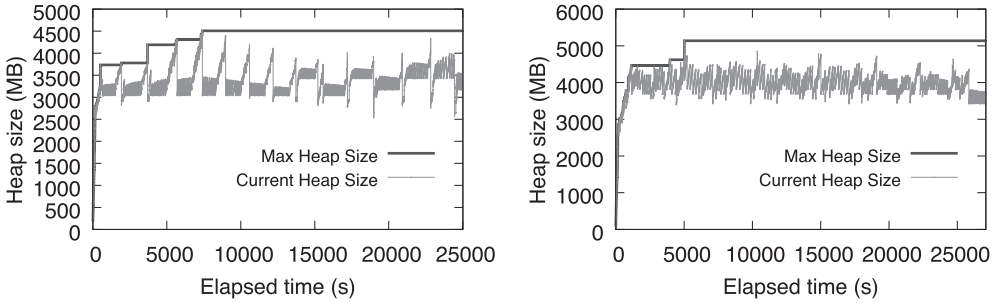
7.5.2 | Policy 2: Conservative query admission control

The second experiment was performed using the conservative query admission policy that assumes that all operators in the plan can execute at the same time. As shown in Figure 30, there are many memory-intensive operators in the plan. As a result, this implementation estimated the potential memory requirement of the plan for our version of TPC-H query 3 to be 1154 MB. Since about 2400 MB was available as working memory, only two queries were allowed to execute at any given time. We can again see a zigzag pattern of memory usage in Figure 32A. However, the pattern is less clear than the pattern in Figure 31A since the execution of two concurrent queries whose starting times were different are overlapped. The heap usage was kept under 4500 MB and it took about 25 000 seconds to finish the entire concurrent execution.

7.5.3 | Policy 3: Stage-aware query-admission control

The last experiment was performed using the query admission control that considers the execution stages of the plan and estimates the maximum memory usage per stage. Each query requested at most 512 MB of working memory at any given time since task cluster 5 had four memory-intensive operators and the budget of each of memory-intensive operator was

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(A) Conservative admission control.

(B) Stage-aware admission control.

FIGURE 32 The heap size during an execution of TPC-H query 3 (multiple concurrent queries). A, Conservative admission control. B, Stage-aware admission control

set to 128 MB. The available working memory was again about 2400 MB. Here, four queries were allowed to execute at a given time. We can see in Figure 32B that the memory usage was controlled below 5 GB. Interestingly, it took slightly longer to execute the 30 queries in this case than it did with the more conservative initial admission control. One major reason is that with more execution worker threads, only two of the system's resources (memory and CPU cores) were enough to accommodate these threads. However, because there was only one physical storage partition, accessing disk was actually contentious among all worker threads. If we could increase the number of physical storage partitions by creating several of them on separate disks, we could reduce this disk contention. This suggests that while AsterixDB's current admission control policy does a good job with respect to CPU and memory management, I/O resources should also be considered at admission time to limit I/O contention.

8 | CONCLUSION

In this article, we have described the budget-driven approach to memory management in Apache AsterixDB, a parallel open source Big Data management system. We described how it divides memory into several sections—in-memory components, the disk buffer cache, and working memory—and how it controls the memory usage of each section carefully. We then discussed how the system maintains a carefully tracked budget in the context of its algorithms in order to keep the memory usage of its memory-intensive operators within budget. Each memory-intensive operator's implementation requires careful attention regarding memory usage since it needs to perform both in-memory and disk-based operation to cope with any volume of data, and each operator has a different algorithm to allocate/deallocate memory pages. We described the implementation of AsterixDB's memory-intensive operators and discussed several issues related to global memory management. Specifically, we discussed how the system manages its in-memory LSM components, implements query admission control, and handles large records. We also presented a series of experiments to empirically explore the potential effect of not considering the size of the data structures used in memory-intensive operators. We used both synthetic and real datasets and showed that the current implementations of operators in AsterixDB are well-controlled and scalable. We also showed that these operators worked as expected both in a single query environment and in multiple concurrent query environments. We believe that future Big Data management system builders can benefit from our experiences.

ACKNOWLEDGEMENTS

The AsterixDB project has been supported at UCI and UCR by an initial UC Discovery grant, by NSF IIS awards 0910989, 0910859, 0910820, 0844574, and by NSF CNS awards 1305430 and 1059436. The project has received industrial support from Amazon, eBay, Facebook, Google, HTC, Infosys, Microsoft, Oracle Labs, and Yahoo! Research and benefits from ongoing software contributions from Couchbase.

ORCID

Taewoo Kim https://orcid.org/0000-0003-2841-6385

REFERENCES

1. Gray J, Putzolu F. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. Paper presented at: Proceedings of the 1987 ACM SIGMOD; 1987:395-398; San Francisco, CA.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2. Gray J, Graefe G. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Rec*. 1997;26(4): 63-68.
3. Graefe G. The five-minute rule twenty years later, and how flash memory changes the rules. Paper presented at: Proceedings of the 3rd International Workshop on Data Management on New Hardware DaMoN '07; 2007:6:1-6:9; Beijing, China.
4. Appuswamy R, Borovica-Gajic R, Graefe G, Ailamaki A. The five-minute rule thirty years later and its impact on the storage hierarchy. Paper presented at: Proceedings of the 7th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures; 2017:1-8.
5. Xu Q, Siyamwala H, Ghosh M, et al. Performance analysis of NVMe SSDs and their implication on real world databases. Paper presented at: Proceedings of the 8th ACM International Systems and Storage Conference; 2015:6:1-6:11.
6. Dayan N, Athanassoulis M, Idreos S. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Trans Database Syst*. 2018;43(4):16:1-16:48. https://doi.org/10.1145/3276980.
7. Knuth DE. *The Art of Computer Programming*. Sorting and Searching. Vol 3. Reading, MA: Addison-Westley Publishing Company; 1973.
8. Pang HH, Carey MJ, Livny M. Memory-adaptive external sorts and sort-merge joins. Paper presented at: Proceedings of the 19th International Conference on VLDB; 1993:618-629; San Francisco, CA.
9. Zhang W, Larson PÅ. A memory-adaptive sort (MASORT) for database systems. Paper presented at: Proceedings of the 1996 CASCON; 1996:41; Toronto, Canada / Ontario, CA.
10. Zhang W, Larson PÅ. Dynamic memory adjustment for external mergesort. Paper presented at: Proceedings of the 23rd International Conference on VLDB; 1997:376-385; San Francisco, CA.
11. Graefe G. Sorting and indexing with partitioned B-trees. Paper presented at: Proceedings of the CIDR 2003, First Biennial Conference on Innovative Data Systems Research; January 5-8, 2003; Asilomar, CA.
12. Kitsuregawa M, Tanaka H, Moto-Oka T. Application of hash to data base machine and its architecture. *New Generat Comput*. 1983;1(1):63-74.
13. DeWitt DJ, Katz RH, Olken F, Shapiro LD, Stonebraker MR, Wood DA. Implementation techniques for main memory database systems. Paper presented at: Proceedings of the 1984 ACM SIGMOD international conference on Management of data; 1984; ACM.
14. Shapiro LD. Join processing in database systems with large main memories. *ACM Trans Database Syst (TODS)*. 1986;11(3):239-264.
15. Zeller H, Gray J. An adaptive hash join algorithm for multiuser environments. Paper presented at: Proceedings of the 16th International Conference on VLDB; 1990:186-197; San Francisco, CA.
16. Pang HH, Carey MJ, Livny M. Partially preemptible hash joins. Paper presented at: Proceedings of the 1993 ACM SIGMOD; 1993:59-68; Washington, DC.
17. Davison DL, Graefe G. Memory-contention responsive hash joins. Paper presented at: Proceedings of the 20th International Conference on VLDB; 1994:379-390; San Francisco, CA.
18. Balkesen C, Teubner J, Alonso G, Özsu MT. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng*. 2015;27(7):1754-1766. https://doi.org/10.1109/TKDE.2014.2313874.
19. Jha S, He B, Lu M, Cheng X, Huynh HP. Improving main memory hash joins on intel xeon phi processors: an experimental approach. *PVLDB*. 2015;8(6):642-653. https://doi.org/10.14778/2735703.2735704.
20. Cheng X, He B, Du X, Lau CT. A study of main-memory hash joins on many-core processor: a case with intel knights landing architecture. Paper presented at: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management; 2017:657-666.
21. Alsubaiee S, Altowim Y, Altwaijry H, et al. AsterixDB: A scalable, open source BDMS. *PVLDB*. 2014;7(14):1905-1916.
22. Ong KW, Papakonstantinou Y, Vernoux R. The SQL++ semi-structured data model and query language: a capabilities survey of SQL-on-hadoop, NoSQL and NewSQL databases. *CoRR*. 2014;1-13. https://arxiv.org/abs/1405.3631.
23. Chamberlin D. *SQL++ For SQL Users: A Tutorial*. Santa Clara, CA: Couchbase; 2018.
24. Borkar VR, Bu Y. Jr Carman EP, et al. Algebricks: a data model-agnostic compiler backend for big data languages. Paper presented at: Proceedings of the 6th ACM Symposium on Cloud Computing; 2015:422-433; Kohala, HI.
25. Borkar VR, Carey MJ, Grover R, et al. Hyracks: a flexible and extensible foundation for data-intensive computing. Paper presented at: Proceedings of the 27th ICDE International Conference on Data Engineering; 2011:1151-1162.
26. Alsubaiee S, Behm A, Borkar VR, et al. Storage management in AsterixDB. *PVLDB*. 2014;7(10):841-852. https://doi.org/10.14778/2732951.2732958.
27. Bu Y, Borkar V, Xu G, Carey MJ. A bloat-aware design for big data applications. Paper presented at: Proceedings of the 2013 International Symposium on Memory Management; 2013:119-130.
28. Effelsberg W, Härder T. Principles of database buffer management. *ACM Trans Database Syst*. 1984;9(4):560-595.
29. Graefe G. Implementing sorting in database systems. *ACM Comput Surv (CSUR)*. 2006;38(3):10.
30. Graefe G. Query evaluation techniques for large databases. *ACM Comput Surv*. 1993;25(2):73-170.
31. Li C, Lu J, Lu Y. Efficient merging and filtering algorithms for approximate string searches. Paper presented at: IEEE 24th International Conference on Data Engineering; April 7-12, 2008:257-266; Cancún, Mexico.
32. Jokinen P, Ukkonen E. Two algorithms for approxmate string matching in static texts. Paper presented at: Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science MFCS; 1991:240-248; Kazimierz Dolny, Poland.
33. Kim T, Li W, Behm A, et al. Supporting similarity queries in apache AsterixDB. *EDBT*. Konstanz, Germany: OpenProceedings.org; 2018:528-539.
34. Gray J, ed. *The Wisconsin Benchmark: Past, Present, and Future*. San Francisco, CA: Morgan Kaufmann; 1993.

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

How to cite this article: Kim T, Behm A, Blow M, et al. Robust and efficient memory management in Apache AsterixDB. *Softw Pract Exper*. 2020;50:1114–1151. https://doi.org/10.1002/spe.2799

APPENDIX A QUERIES USED IN THE "ACCOUNTING FOR EVERYTHING" EXPERIMENTS

```
/* Q1. Sort - three-fields (int) */
select value count(first.unique1 + first.unique2 + first.unique3) from (
select unique1, unique2, unique3 from Wisconsin where unique2 < [end] order by unique1 ) first;

/* Q2. Sort - three-fields (string) */
select count(first.unique1 + len(first.stringu1) + len(first.stringu2)) from (
select unique1, stringu1, stringu2 from Wisconsin where unique2 < [end] order by unique1 ) first;

/* Q3. Hash Group-by - three-fields (int) */
select value count(first.unique1 + first.unique2 + first.unique3) from (
select t.unique1, t.unique2, t.unique3, count(t.unique3) from Wisconsin t
where t.unique2 < [end] /* +hash */ group by t.unique1, t.unique2, t.unique3 ) first;

/* Q4. Hash Group-by - three-fields (string) */
select value count(first.unique1 + length(first.stringu1) + length(first.stringu2)) from (
select t.unique1, t.stringu1, t.stringu2, count(t.stringu2) from Wisconsin t
where t.unique2 < [end] /* +hash */ group by t.unique1, t.stringu1, t.stringu2 ) first;

/* Q5. Hash Join - three-fields (int) */
select value count(first.unique2 + first.unique1 + first.unique3) from (
select r.unique2, r.unique1, r.unique3 from
( select unique3, unique2, unique1 from Wisconsin where unique2 >= 1000 and unique2 < 2000 ) r,
( select unique3, unique2, unique1 from Wisconsin where unique2 >= 0 and unique2 < [end] ) s
where r.unique3 = s.unique3 and r.unique2 = s.unique2 and r.unique1 = s.unique1 ) first;

/* Q6. Hash Join - three-fields (string) */
select value count(first.unique2 + length(first.stringu1) + length(first.stringu2)) from (
select r.unique2, r.stringu1, r.stringu2 from
( select unique2, stringu1, stringu2 from Wisconsin where unique2 >= 1000 and unique2 < 2000 ) r,
( select unique2, stringu1, stringu2 from Wisconsin where unique2 >= 0 and unique2 < [end] ) s
where r.unique2 = s.unique2 and r.stringu1 = s.stringu1 and r.stringu2 = s.stringu2 ) first;
```

APPENDIX B QUERIES USED IN THE "CONFORMING TO THE BUDGET" EXPERIMENTS

```
/* Q7. Sort */
select value count(first.unique1) from (
 select unique1, unique2
 from Wisconsin where unique2 >= [start] and unique2 < [end] order by unique1 ) first;

/* Q8. Hash Group-by */
select value count(first.unique1) from (
 select unique1, unique2, count(unique2) from Wisconsin
 where unique2 >= [start] and unique2 < [end] /* +hash */ group by unique1, unique2 ) first;

/* Q9. Hash Join */
select count(first.unique1) from (
 select r.unique2, s.unique1 from
 ( select unique2 from Wisconsin where unique2 >= [start1] and unique2 < [end1] ) r,
 ( select unique2, unique1 from Wisconsin where unique2 >= [start2] and unique2 < [end2] ) s
 where r.unique2 = s.unique2 ) first;

/* Q10. Inverted-index search */
select count(*) from reddit r where ftcontains(["keyword1", "keyword2", "keyword3"], {"mode":"all"});
```

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

APPENDIX C QUERIES USED IN THE "LARGE OBJECTS" EXPERIMENTS

```
/* Q11. Sort: large fields */
select value count(largeString) from (
 select unique1, largeString from [dataset] where unique2 >= [start] and unique2 < [end] order by unique1
) first;

/* Q12. Hash Join: large fields */
select count(first.largeString) from (
 select r.unique2, s.largeString from
 ( select unique2 from [dataset] where unique2 >= [start1] and unique2 < [end1] ) r,
 ( select unique2, largeString from [dataset] where unique2 >= [start2] and unique2 < [end2] ) s
 where r.unique2 = s.unique2 ) first;
```

APPENDIX D QUERIES USED IN THE "QUERY ADMISSION CONTROL" EXPERIMENTS

```
/* Q13. TPC-H Query */
select value count(*) from (
  select l.l_orderkey, sum(l.l_extendedprice*(1-l.l_discount)) revenue, o.o_orderdate, o.o_shippriority
  from customer as c, orders as o, lineitem as l
  where ftcontains(c.c_comment, [keyword]) and c.c_custkey = o.o_custkey and l.l_orderkey = o.o_orderkey
     and o.o_orderdate < [date] and l.l_shipdate > [date]
  /* +hash */ group by l.l_orderkey, o.o_orderdate, o.o_shippriority
  order by revenue desc, o.o_orderdate
) first;
```

1097024x, 2020, 7, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2799 by Universitatsbibliothek, Wiley Online Library on [06/10/2026]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License