UNIVERSITY OF CALIFORNIA,
IRVINE

Enhancing Apache AsterixDB for Efficient Big Data Search and Analytics

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Taewoo Kim

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

2018

DEDICATION

To my family. I would not be who I am today without them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Professor Chen Li. We have been working together since January 2013. I still remember how our relationship started. After I took his data management class, I contacted him to get a sense of graduate level research. He kindly allowed me to work on a project called the iPubMed search engine. Since then, I have been asking his opinion on almost any topics during my Masters and Ph.D. program of study. In fact, he has been one of the most influential personnel during my stay at UCI for six years. I can't imagine how I could complete my Ph.D. program in a timely fashion without his guidance.

I also would like to thank Professor Michael J. Carey. We have been working together since June 2014 after I started my Ph.D. journey here at UCI. It has been my pleasure to meet him regularly with Professor Li. During those weekly meetings, we discussed a lot of things including research and other topics. Every discussion that we had helped and inspired me to grow as a Ph.D. student. Whatever complicated issues that I had, he could guide me to approach each issue in a concise and elegant way.

I also would like to thank Professor Sharad Mehrotra. One of Professor Mehrotra's lectures encouraged me to pursue Ph.D. program. I learned interesting concepts related to transactional data processing through his class. After taking the class, I started thinking that learning advanced topics about data management and applying this knowledge to application developments could be wonderful. He has been really supportive and offered helpful opinions on all the issues that I asked.

I was also helped by many other faculty, colleagues, and friends along my Ph.D. journey. I especially want to thank my friends and colleagues, Youngseok Kim, Jianfeng Jia, Xikui Wang, Ian Maxon, Shiva Jahangiri, Chen Luo, Gift Sinthong, Abdullah Alamoudi, Dmitry Lychagin, Till Westmann, Murtadha Hubail, Abdulrahman Alsaudi, Dhrubajyoti Ghosh, Primal Pappachan, Jeongheon Kim, Seulip Lee, Taemin Park, Minhaeng Lee, Eunjeong Shin, Myungseo Kim, Eunbae Yoon, Myungguk Lee, Minsoo Kim, Juhwan Kim, Dokyung Song, Yunho Heo, Hyungik Oh, and all other KGSA friends. I thank the Financial Supervisory Service of Korea for allowing me to have a long-term leave to pursue my Ph.D. degree.

Most importantly, I want to thank my wonderful family. My wife Soyhi, Mom, Dad, my brother Taehyung, his wife, and my nephew Seoha: without your support, I could not have completed this work. I am forever indebted to you all for that.

Some parts of Chapter 5 in this thesis/dissertation is a reprint of the material as it appears in the proceedings of EDBT 2018. The authors of this paper are Taewoo Kim, Wenhai Li, Alexander Behm, Inci Cetindil, Rares Vernica, Vinayak Borkar, Michael J. Carey, and Chen Li. 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 it benefits from ongoing software contributions from Couchbase.

CURRICULUM VITAE

Taewoo Kim

EDUCATION

Doctor of Philosophy in Computer Science **2018**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2014**
University of California, Irvine *Irvine, California*

Bachelor of Management in Management **2007**
Korea Open University *Seoul, Korea*

Bachelor of Science in Computer Science **2004**
Yonsei University *Seoul, Korea*

PUBLICATIONS

Supporting Similarity Queries in Apache AsterixDB **2018**
International Conference on Extending Database Technology (EDBT)

Caching Geospatial Objects in Web Browsers **2017**
International Conference on Advances in Geographic Information Systems (SIGSPATIAL)

A Comparative Study of Log-Structured Merge-Tree-Based Spatial Indexes For Big Data **2017**
IEEE International Conference on Data Engineering (ICDE)

Twitter Coverage of Climate Change and Health before and after the 2016 US Presidential Election **2017**
American Public Health Association

RILCA: Collecting and Analyzing User-Behavior Information in Instant Search Using Relational DBMS **2015**
VLDB Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)

Efficient Instant-Fuzzy Search with Proximity Ranking. **2014**
IEEE International Conference on Data Engineering (ICDE)

ABSTRACT OF THE DISSERTATION

Enhancing Apache AsterixDB for Efficient Big Data Search and Analytics

By

Taewoo Kim

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Chen Li, Chair

In a typical minute of a day in 2018, the Internet generates 3,138 terabytes of traffic [4], Twitter users send 473,000 tweets [4], and two million snaps are sent on Snapchat [4]. By 2020, it is estimated that for each person on earth, 1.7 MB of data will be created every second on the average [4]. Due to the large volumes of Big Data, efficient search methods and analytics are required to explore such data. Thus, there is a clear need for Big Data management system, such as Apache AsterixDB, to enable users and applications to search to explore Big Data.

Initiated in 2009, the AsterixDB project integrated ideas from three distinct areas - semi-structured data, parallel databases, and data-intensive computing - to create an open-source software platform that scales on large, shared-nothing commodity computing clusters. AsterixDB currently provides various types of index, such as B+-tree, R-tree, and inverted indexes to fetch data efficiently. Also, as the problem of supporting similarity queries has become increasingly important in many applications, AsterixDB also supports similarity query processing using various metrics and provides an efficient similarity join method [58]. It also provides various search and fundamental analytical functions. It can utilize external third-party libraries using user-defined functions to augment its functionalities. Following the release of the first public open-source version of AsterixDB in 2013, we identified several

optics that needed to be explored in depth to enhance AsterixDB further. Those topics are the focus of this thesis.

We first share our memory management experiences in AsterixDB. We describe the original implementation of the system's memory-intensive operations and a set of design flaws (oversights) related to memory management that we found later. We then discuss how we have addressed each of those oversights. We also discuss AsterixDB's memory management at the global level. We believe that future Big Data management system builders can benefit from our memory management experiences. With memory management under control, we next present the design and implementation of index-only query plans in AsterixDB. Use of these plans can boost the performance of an index-based search by several orders of magnitude compared to a scan-based or non-index-only approach. We discuss the challenges that we faced regarding the implementation of index-only query plans in AsterixDB and how we addressed these challenges.

Lastly, considering the importance of similarity query processing for Big Data searches and analytics, we evaluate the performance of similarity query processing in AsterixDB. We compare its approach to several other systems and report the efficacy and performance results.

Chapter 1

Introduction

In a typical minute of the day in 2018, the Internet generates 3,138 terabytes of traffic [4]. 3.8 million searches are conducted on Google [4]. About 73,249 online shopping transactions are made [4]. Twitter users send 473,000 tweets [4]. Two million snaps are sent on Snapchat [4]. By 2020, it is estimated that for each person on earth, 1.7 MB of data will be created every second on the average [4]. That is, about 143 GB of data will likely be generated for each person per day. Twitter also reported that there were 335 million monthly active users worldwide in the second quarter 2018 [12]. Facebook reported that they had 1,471 million daily active users in the second quarter of 2018 [6]; there were 2,234 million monthly active Facebook users in the same quarter.

Clearly, efficient searching and analytic methods are required to deal with large volume of data. Suppose that a data scientist in 2020 wishes to analyze the data generated by a group of 400 university students for an experiment in one month. The scientist will then need to process about 1,675 TB of data since each person will generate 4.3 TB of data (1.7 MB per second). The scientist could choose to hand-build an application to process this large amount of data. However, building an application that can store and analyze Big Data from

scratch requires a lot of effort and would take a large amount of time. It is better to use a Big Data management system, such as Apache AsterixDB, to search and analyze the data.

Initiated in 2009, the Apache AsterixDB [13] project had integrated ideas from three distinct areas – semi-structured data, parallel databases, and data-intensive computing – to create an open-source software platform that scales on large, shared-nothing commodity computing clusters [15]. AsterixDB provides various types of indexes, such as B+-tree, R-tree, and inverted indexes, to fetch data efficiently. As supporting similarity queries has become increasingly important in many applications, including search, record linkage [29], data cleaning [78], and social media analysis [21], AsterixDB also supports similarity query processing using various metrics. The system includes both searches and an efficient similarity join method [58]. Moreover, to the best of our knowledge, AsterixDB is the first Big Data management system that supports queries with multiple similarity joins. The first public open-source version of AsterixDB was released in 2013.

Following that, we identified several topics that needed to be explored in depth to enhance AsterixDB further.

1) Similarity queries and analytics require many database operators in a query execution plan, which can consume a large amount of memory, so the memory allocation and deallocation of each operator should be controlled properly via careful budgeting. That is, the system should be robust regarding its memory behaviors so as not to generate "Out Of Memory" (OOM) exceptions in the middle of a query execution for any complex query plan. Although the initial system assigned a memory budget for operators and tried to keep the operators under their budgets, the original release of Apache AsterixDB in 2013 could still generate OOM exceptions too easily under some conditions. This limitation motivated our attention to enhance the memory management in AsterixDB.

2) Index-based searches that utilize various types of indexes are very important and benefi-

cial. However, a limitation of index-based searches also exists. For instance, if the selectivity of an index-based search increases, the execution time increases as well since AsterixDB needs to look in the primary index to fetch the original records using keys found from an index search to generate final results. Thus, a full-scan actually becomes faster than an index-based search beyond a certain selectivity. Thus we need an efficient index-based search that can overcome this limitation when possible is required.

3) After the first release of AsterixDB, a study evaluated the performance of Big Data management systems including MongoDB, Hive, and AsterixDB [75] using various kinds of queries on a synthetic dataset. The overall performance of AsterixDB generally showed the lowest query execution time or at least comparable performance to the fastest system, for most of the benchmark queries. Among various kinds of queries, only one kind of similarity queries that utilized edit distance was included. Considering the growing importance of similarity query processing, we carefully evaluated the performance of similarity query processing in AsterixDB and related data management systems.

This thesis consists of three major parts, each of which is now previewed briefly.

Robust and Efficient Memory Management in AsterixDB: Chapter 3 presents memory management solutions in AsterixDB. Many of today's Big Data management systems would like to handle Big Data that cannot fit into main memory. Traditional relational database systems have done this by dividing the memory into a few regions such as a buffer cache and working memory. They assign a memory budget to each region and control the allocation of memory to each region to ensure the stability of the system's memory behaviors. In addition, they assign certain memory budgets to memory-intensive relational operators such as sort, group by, and hash joins, and control the allocation of memory to these operations. This control is required since these memory-intensive operators support both in-memory and disk-based operations to process large volumes of data. Each memory-intensive operator attempts to maximize its memory usage to conduct an operation rather

than a disk-based operation whenever. Implementing memory-intensive operations requires a careful design and the implementation of appropriate algorithms. In this chapter, we report our memory management experiences in AsterixDB. We describe the original implementation of its memory-intensive operations as well as the design limitations. These limitations are not uncommon in the research literature. We then discuss how to address those limitations. We also discuss AsterixDB's global memory management. We conducted an experimental study using several synthetic and real datasets to show the negative effects of the initial implementation. We have also conducted experiments to show that our current implementation is robust and scalable regardless of input data size.

Index-only Query Plans: Chapter 4 describes the design and implementation of index-only query plans in AsterixDB to further enhance index-based search. An index-only plan can improve the performance of an index-search by several orders of magnitude as compared to a scan-based or non-index-only approach. We discuss the challenges in the implementation of index-only plans and how we addressed these issues. We also explain how a scan-based plan that meets certain conditions can be transformed into an index-only plan during the optimization phase of query compilation. Lastly, we present a set of experimental results that compare scan-based, index-based, and index-only plans on a dataset with temporal and spatial fields using both B+-tree and R-tree indexes.

Performance Evaluation of Similarity Query Processing: Chapter 5 presents a performance evaluation of similarity queries in AsterixDB and two other systems. Most existing work has taken an algorithmic approach, to simplify queries, developing indexing structures, algorithms, and/or various optimizations. Here we take a different, systems-oriented approach. We describe the similarity query support in AsterixDB. We describe the lifecycle of a similarity query in the system, including the support provided at the query language, indexing, query execution plans (with and without indexes), and plan rewrites to optimize query execution. Our approach leverages the existing infrastructure of AsterixDB, including

its operators, parallel query engine, and rule-based optimizer. We conducted an experimental study using several large, real data sets on a parallel computing cluster to evaluate AsterixDB's support for similarity queries. We also conducted experiments with two other systems, and report the efficacy and performance results.

The thesis is organized as follows: We first describe the fundamentals of AsterixDB in Chapter 2. In Chapter 3, we present robust and efficient memory management in AsterixDB. In Chapter 4, we discuss index-only plans in AsterixDB. In Chapter 5, we evaluate the performance of similarity query processing in AsterixDB and two other systems. Chapter 6 concludes the thesis and outlines a few potential directions for future work.

Chapter 2

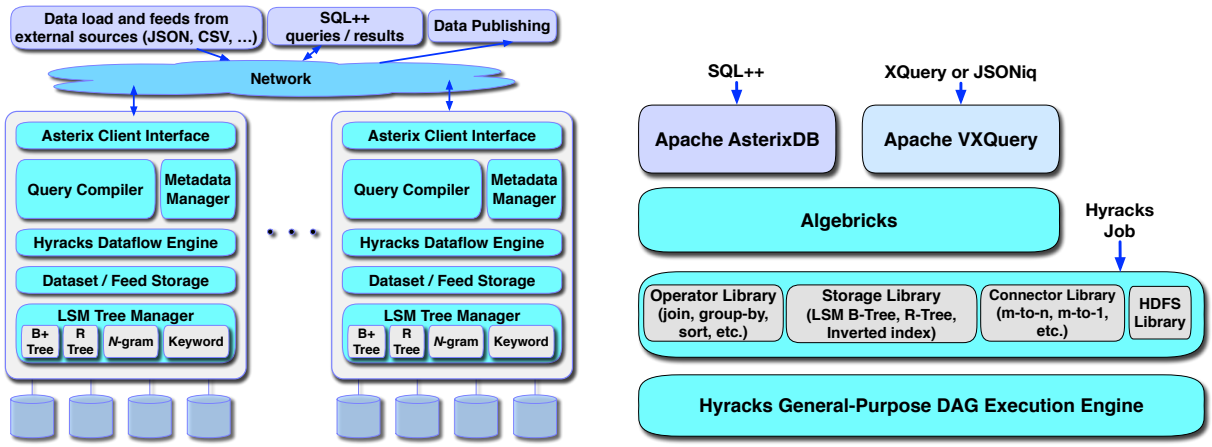
Preliminaries

2.1 Apache AsterixDB

Initiated in 2009, the AsterixDB [13] project integrated ideas from three distinct areas – semi-structured data, parallel databases, and data-intensive computing – to create an open-source Big Data management software platform that can scale horizontally on large, shared-nothing commodity computing clusters [15], as shown in the current architecture of AsterixDB as depicted in Figure 2.1(a). From the outset, the handling of truly large data volumes, exceeding the memory capacity of a cluster, has been one of the project's key objectives.

2.1.1 Software Architecture

As shown in Figure 2.1(b), Apache AsterixDB consists of several software layers. The topmost layer provides a full, flexible data model (ADM). The SQL++ [71, 26] query language for describing, storing, indexing, querying, and analyzing Big Data. The project's layered software architecture also allows for other projects (e.g., Apache VXQuery) to reuse the



(a) Architecture. (b) Layers.

Figure 2.1: AsterixDB Architecture and layers.

system's lower layers.

The second layer, a query compiler based on *Algebricks* [22], is used for parallel query processing. This algebraic layer receives a translated logical SQL++ query plan from the upper layer and performs rule-based optimizations. 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 30 rules for the physical optimization phase. After the physical optimization process is done, the Algebricks layer generates jobs to be executed on the *Hyracks* [23] layer.

The Hyracks layer includes storage facilities for datasets that are stored and managed by AsterixDB as partitioned LSM-based B+-trees with optional LSM-based secondary indexes [14]. AsterixDB translates a 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 output partitions produced by operators are then repartitioned by connectors to produce the input partitions for the next operator. An operator consists of one or more activities (sub-steps 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 co-scheduled, and Hyracks jobs are executed on a stage-by-stage-basis. Since data is represented as tuples of bytes at this level, Hyracks is an execution layer that is data-model agnostic. It means that data is accessed at the binary level rather than as language objects. This approach was used since creating and collecting language objects is often a cause of performance bottlenecks that can prevent systems from scaling [25].

2.1.2 Data Model

AsterixDB defines its data model (ADM) [13] targeting semi-structured 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 in which to create types, datasets, indexes, functions, and other artifacts. A datatype specifies the fields and 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 have only the specified fields. A dataset is a collection of data instances of a datatype. Figure 2.2 shows a simple example of some ADM DDL statements. The first two create a dataverse called exp and a data type for a dataset called the Reddit dataset. Its type, RedditType, is defined as an open type.

```
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;
```

Figure 2.2: ADM type and dataset.

Each record in an AsterixDB 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. Figure 2.2 also includes a SQL++ statement for creating an Reddit dataset. Each partition of an AsterixDB 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 index options; indexes are local, i.e., 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 in AsterixDB can be found in the AsterixDB storage management paper [14].

Chapter 3

Robust Memory Management in AsterixDB

3.1 Introduction

Jim Gray's now famous 5-minute rule [49] stated that a data item should be only in memory if it is referenced every 5 minutes or less. Although this rule has been revised over time [48, 43] to reflect changes in the cost and performance of memory and persistent storage, the principle that only a sufficiently frequently referenced item should be resident in memory still holds. We need to properly deal with data on disk (be it magnetic or solid state) since we cannot load a large amount of data into memory and keep it there.

Most robust data management systems handle data by using a disk buffer cache to hold the data items from disk and allocating a certain amount of memory to memory-intensive database operators such as sort, join, and group-by to ensure the stability of the system's memory behaviors. A database operator is memory-intensive if it supports both in-memory and disk-based operation to deal with the data size. For example, the operations listed above

can operate in memory when the memory budget is enough to hold and process the entire data. If the budget is not enough, they switch to disk-based (or "spilling") operation. Both in-memory and disk-based operations will generate the same logical result although their performance can be different. Supporting both types of operations means that an operator can process any amount of memory if the assigned budget is greater than the minimum memory requirement of the operator. Thus, systems need to carefully control the memory use of an in-memory 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 when a system allocates more memory to it. In contrast, other kinds of 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 a better performance. Thus, there is no reason to control a memory budget for such non-memory-intensive operators. A critical aspect of Big Data memory management is ensuring that the memory-intensive operators operate within a specified budget.

Despite the importance of good memory management, even today, searching "out of memory" together with a Big Data management system's name, such as MongoDB, Oracle, and PostgreSQL, in a web-search engine still yields many results. Furthermore, it is the developer's responsibility to manage the memory usage in some Big Data analysis engines, such as Apache Spark. From its inception, the AsterixDB system has sought to be effective for large data volumes that can far exceed memory. Although we assigned a memory budget for memory-intensive operators and tried to keep the operators perform its operation under the budget, the original release of Apache AsterixDB in 2013 could still generate "Out of Memory" exceptions in some cases.

In this chapter, we discuss the implementation of AsterixDB, focusing on its memory management, the design flaws (oversights) in the first implementation, and how we addressed

those oversights. We believe that our experiences can help other researchers understand the importance of considering the size of data structures, which have not been considered carefully in many research studies and systems.

The rest of the chapter is organized as follows. In Section 3.2, we present the architecture, the data model, and memory management model of AsterixDB. In Sections 3.3 through 3.5, we describe the implementations of the memory-intensive operators, the identified issues, and how we addressed them. Section 3.6 describes AsterixDB's global memory management. Section 3.7 shows experiments comparing the original implementation and the current AsterixDB implementation. Section 3.8 concludes the chapter.

3.1.1 Related Work

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

Relational Database Systems: Most open-source and commercial database management systems, such as DB2 [32], MySQL [52], Oracle [17], and PostgreSQL [81], divide their memory into a few sections and assign a memory budget to each of them. They control the allocation of memory to each section to ensure the stability of its memory behavior as shown in Figure 3.1. These systems also control the number of concurrent queries to properly manage memory, as shown in the figure.

These systems allocate a certain memory budget to memory-intensive operations, such as sort or join operations, to ensure the memory stability of the system. For example, DB2 [33] allocates memory buffers to each join or sort operation as specified by the "sortheap" parameter. Another DB2 parameter called "sortheapthres" controls the overall number of memory

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 3.1: Global memory components and concurrent query control in data management systems.

buffers that all concurrent join or sort operations can use. If the number of allocated pages becomes greater than this parameter, fewer memory buffers will be allocated to such operation. Similarly, PostgreSQL [76] allocates buffers to each sort or join operation using a "work_mem" parameter. MySQL [70] also has a similar parameter called "sort_buffer_size" to control the number of memory pages allocated per session. Oracle [72] has an area called the "SQL Work Area" to allocate memory pages to sort-based operators such as order-by, group-by, and roll-up; a hash join operator also gets its memory from this area.

Big Data analysis engines: Apache Spark uses two kinds of memory – execution and storage [16]. 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 is using Hadoop so that the heap sizes of mappers and reducers can be set. The maximum heap size for a Hive instance can be also set [92]. For Apache Tez, the maximum heap memory for a Tez instance can be defined [90]. Apache Impala similarly has parameters to set maximum

budgets for entire queries and an individual query [53].

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 [60] to perform their sort operations. Since the time when external sorting was first developed many decades ago, various methods have been developed to allow for memory fluctuations during an external sort process [73, 101, 102, 41]. 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 [59] and Hybrid Hash Join [36, 85]. 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 [100, 74, 31].

Text search using an inverted index: Another potentially memory-intensive operator today is text-search using an inverted index. A naive non-memory-intensive 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 algorithms and data structures have been proposed. Among them, inverted index [60] and its variants are widely used. Search engines such as Elastic Search [5] and Apache Solr [1] are based on Apache Lucene [64], which uses an inverted index. Lucene recommends using a large memory buffer [54] for a high indexing performance. It also recommends allocating more memory to the Java Virtual Memory (JVM) instance heap of the application that accesses a Lucene index. A heap size can be configured for Elastic Search and Solr JVM instances to load indexes from disk and necessary structures to perform searches.

3.2 Preliminaries - Memory Management in AsterixDB

The fundamentals of AsterixDB were discussed in Section 2.1. Thus, we discuss memory management in AsterixDB that is closely related to this Chapter. Figure 3.2 shows the memory structure of an AsterixDB partition in a Java Virtual Memory (JVM) instance. There are three main sections of memory, the section used for in-memory components (a.k.a. in-memory component memory), the buffer cache, and working memory [14].

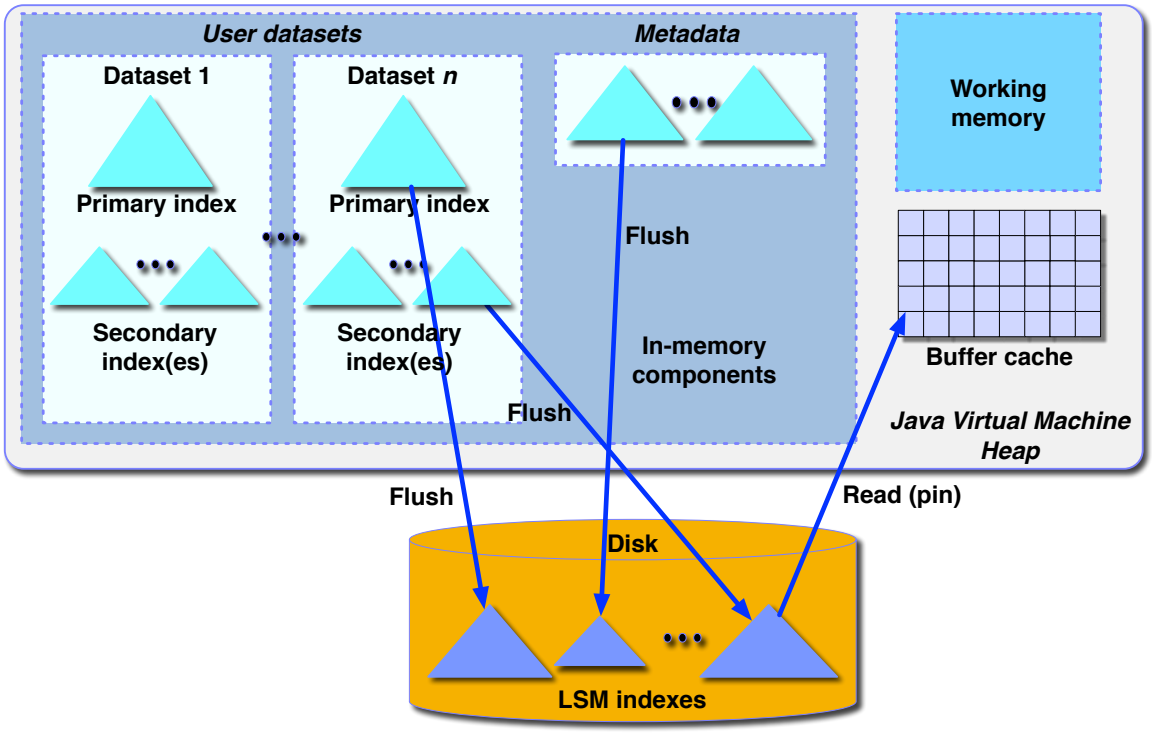


Figure 3.2: Memory structure in AsterixDB.

The first component, the in-memory components section, holds in-memory components of the currently active datasets. Due to the nature of LSM index structures, the results of every insert, upsert, and delete operation on records 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 an in-memory component 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 of a dataset. When the overall in-memory component size of a dataset

reaches this budget limit after an operation, the primary index and all secondary indexes of the dataset are flushed to the disk and become immutable disk components. The allocated in-memory component memory then becomes available again. AsterixDB also controls the maximum size of the overall in-memory component memory for the JVM instance. 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 size 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 caches [38] in other database systems, we will not discuss its detailed structure further.

The last memory section, working memory, honors the memory allocation requests from operators in an execution plan. An allocation request from an operator requires 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. In AsterixDB, an operator uses a set of memory pages as its input buffer, uses zero or more pages as its execution memory, and has a set of memory pages as its output buffer as shown in Figure 3.3.

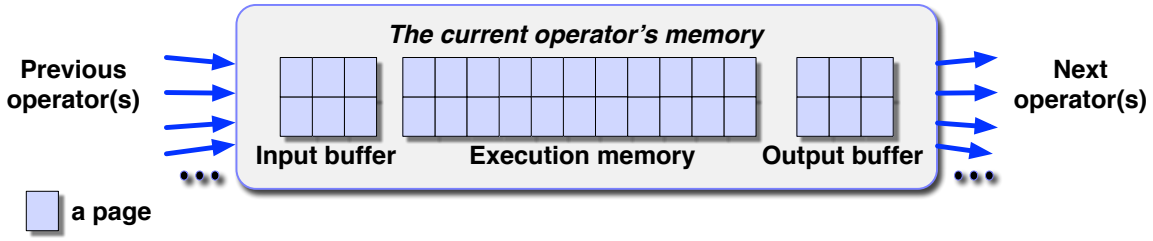


Figure 3.3: Operator model.

Since each operator uses input/output buffer pages, the difference among the operators re-

garding their memory requirements mainly comes from their execution memory requirement. Regarding the execution memory usage, there are three types of operators. The first type of operators 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 the given 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, the record is copied to the output buffer page. When the output buffer page becomes full, that page is pushed to the next operator. The second type of operators 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 operators 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 5.1. 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 then 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 to improve performance. The operators that we focus on in this chapter are these memory-intensive operators.

All memory-intensive operators share two common characteristics. First, when we allocate more memory to such an operator, its performance should be improved. Second, these 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 disk-based operation. For instance, a hash join operator spills some records to disk and deals with them later after processing records in memory. If they can allocate enough memory to process all records, an entirely in-memory operation can be performed. For instance, a sort operator can perform an in-memory sort to sort all of the records. As a consequence, the memory consumption of memory-intensive operators needs to be carefully controlled since they have memory allocation/deallocation logic inside to maximize their performance.

The budget for any memory-intensive operator is determined by operator-specific system parameters (e.g., 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 a maximum and uses these pages as its execution memory. 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.) 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 memory 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.

3.3 Memory-Intensive Operator: Sort

We released the first public open-source version of AsterixDB in 2013. Here we discuss the original implementations of memory-intensive operators in the release including the sort, hash group-by, hash join, and inverted-index search. In Sections 3.3 through 3.5, we will first discuss the implementation of each memory-intensive operator in detail. We will discuss how AsterixDB initially handled both in-memory and disk-based operations for each operator. We later found that there were significantly lingering memory management issues in each memory-intensive operator after the release of AsterixDB, so we will discuss those issues, too. We will then discuss how we have addressed those issues. In this first section, we describe the original implementation of the sort operator, its issues regarding memory management, and how we have addressed the issues.

3.3.1 Sort: Original Implementation

AsterixDB performs an external sort [60] when a sort operation is needed in a query execution plan. There are a few techniques to improve the performance of the sorting process. Since comparing binary-representations of field values is more efficient than comparing the original values of a field, the concept of *normalized key* can be found as far back as System R [20, 50]. Also, rather than moving an actual record during a sort, normally *an array of record pointers* is used in most implementations [42] 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 3.4. In the first phase, a sort operator gradually allocates memory pages to hold incoming records in each page. 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. Currently, AsterixDB utilizes merge sort as the in-memory sort algorithm. After this in-memory 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 the 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 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.

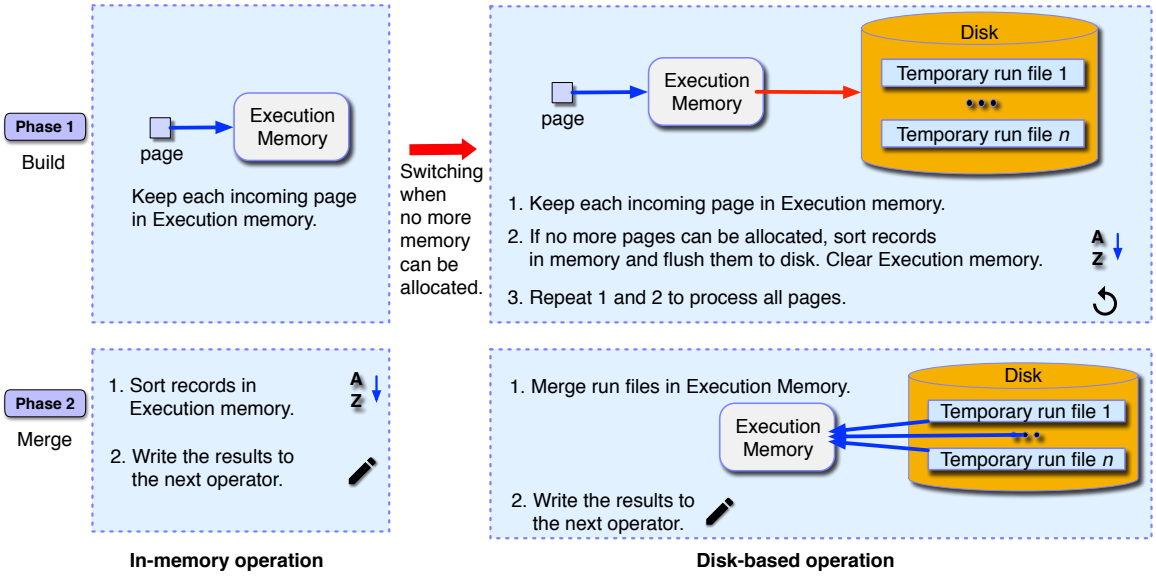


Figure 3.4: Two phases of the external sort.

In addition to utilizing pages, the sort operator also uses an additional data structure called *record pointer array*. In the first phase as shown in Figure 3.5, for an incoming page, the sort 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 field(s) that is being sorted. 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, using this array can improve the performance since comparing two normalized keys in the record pointer array using binary string comparison can be performed much faster than comparing the actual values between two records since the latter requires accessing the actual records in the pages. Another benefit of using the array is that its size is relatively small compared to 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. When reading an incoming page, when the sort operator cannot allocate any 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 are flushed to the file. The array will then be cleared and its pages will be deallocated. The sort operator then reads the next incoming page to fill in the execution memory with pages and to fill the array in again.

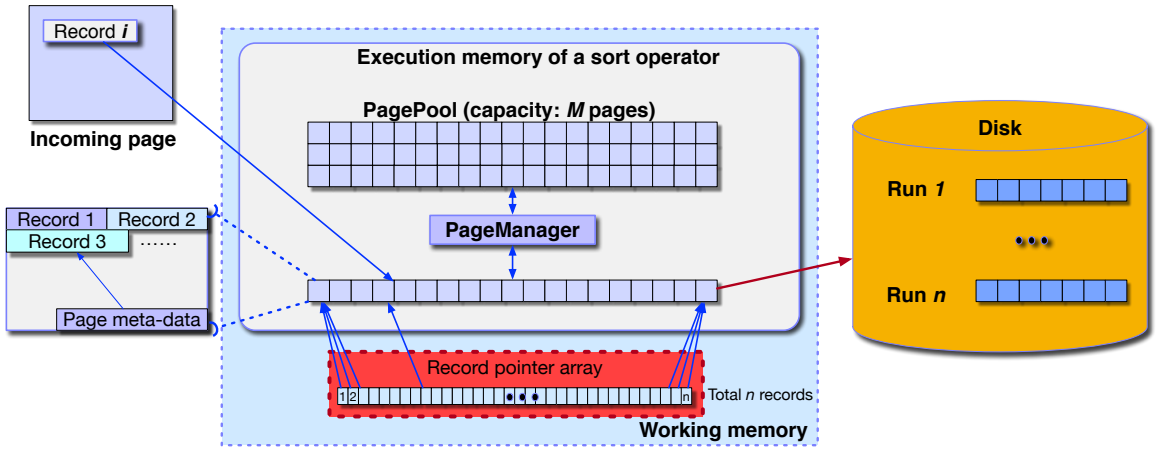


Figure 3.5: External sort: the original implementation.

During the second phase of its operation, the sort 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. 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. In contrast, if the number of runs is greater than M, it first merges the number of runs equal to M 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 merge process until the run file list becomes empty.

An important issue in our initial implementation of this operator was that we did not consider the size of the record pointer array (depicted in red in Figure 3.5) within the budget M since we regarded it 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 3.7.3.1, 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 in some cases.

3.3.2 Sort: Current Implementation

To address this issue, AsterixDB now also considers and manages 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 size impact 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 of the current memory footprint and this size 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 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, as explained earlier.

3.4 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. That is, the hash table is used to guide searches to the actual records in memory based on hash values. We first present the hash group-by operation since its operation flow is somewhat simpler. We then describe the hash join operator.

3.4.1 Hash Group-by Operator

A group-by operation is used with aggregating functions, such as COUNT, MIN, MAX, SUM, or AVG, to group the results of an aggregation operation by one or more fields. The field(s) that are being grouped is the group field(s) and the field(s) that is being aggregated is the aggregate field. The group field(s) and aggregate field(s) 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 is age.

By default, when a user includes a group by clause in a SQL++ query, AsterixDB performs a sort-based group-by. This operation is quite similar to the external sort operation. The main difference is that the group-by operator generates an aggregate result per group field(s)

value. Thus, the sort-based group-by operator also shared the same memory accounting issue as the sort operator. Therefore, we will not discuss the sort-based group-by operation in further detail as its issue was similarly addressed.

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 first describe the operation flow of the hash group-by operator and the structure of the hash table and the data partition table in its original implementation. We then discuss the major memory issue that we subsequently identified. Lastly, we explain how we addressed the issue.

3.4.1.1 Hash Group-by: Original Implementation

The conceptual operation flow of the hash group-by is as follows. For each record i in an incoming page, the hash group-by 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 using the hash value $h(i)$ since the hash value $h(i)$ guides it to the actual location of the aggregate results for each group whose hash value is $h(i)$. If the partial aggregate result exists, the operator aggregates the record i into the aggregate result (e.g., adding the incoming salary to the running total) using the aggregate field value. If the aggregate 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 actual hash group-by operation flow consists of two phases, similar to the sort operator, as shown in Figure 3.6. In the first phase, the hash group-by operator first calculates the number of hash partitions that it will use based on the input data size and the given budget

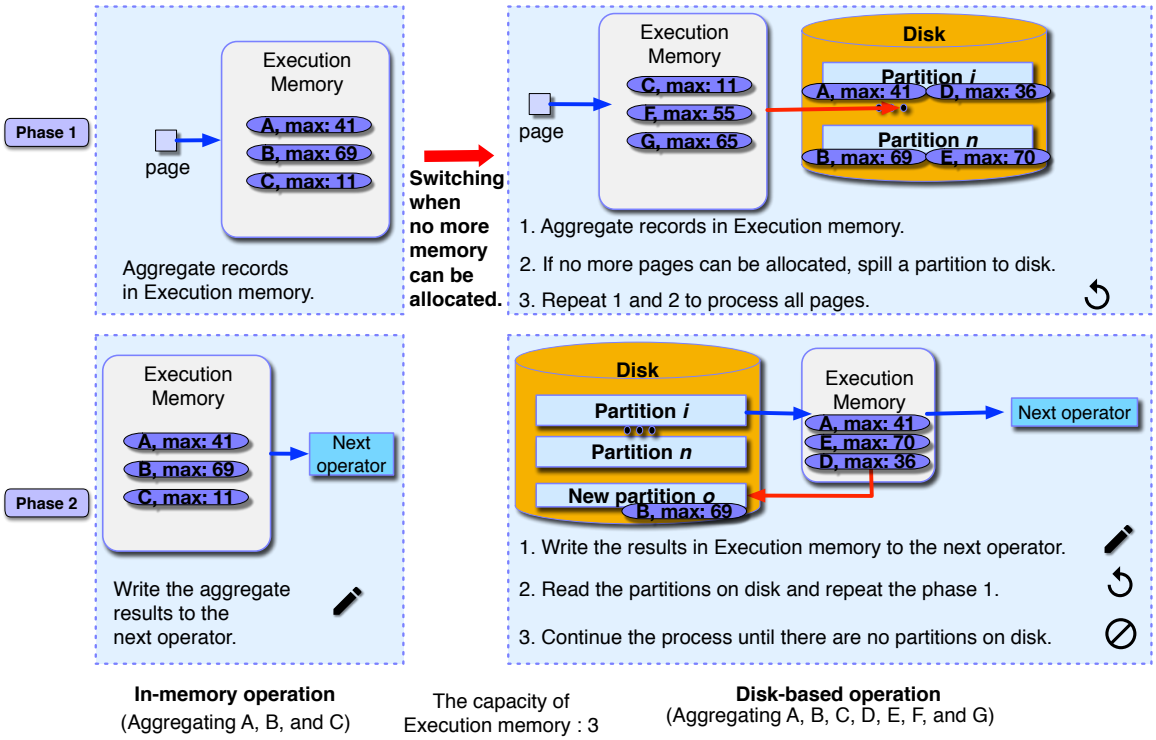


Figure 3.6: Two phases of hash group-by.

M. (This calculation aims to partition the aggregation assumption into pieces that can perhaps be performed in memory if the data size permits [40].) 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 that is estimated by the query optimizer, each partition covers the same number of contiguous hash values. For instance, if there are 1,000 expected hash values and 10 partitions, each partition covers 100 hash values. That is, the first partition stores the aggregate results for groups if the hash value of the group is between 0 and 99. 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 upon as the operator processes incoming pages.

After processing all incoming records, phase 2 of the hash group-by begins. At this point, some partitions are in memory and some partitions are on disk. The operator passes in-memory partitions to the next operator since it does not need to read 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 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 repeats phase 1 again for the remaining spilled partitions on disk. This recursive process continues until there are no spilled partitions remaining 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 3.7. The data partition table and the hash table are separated 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, 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 overhead than only moving the slot with pointers to the aggregate records.

The data partition table stores the records for groups and their aggregate results. It consists

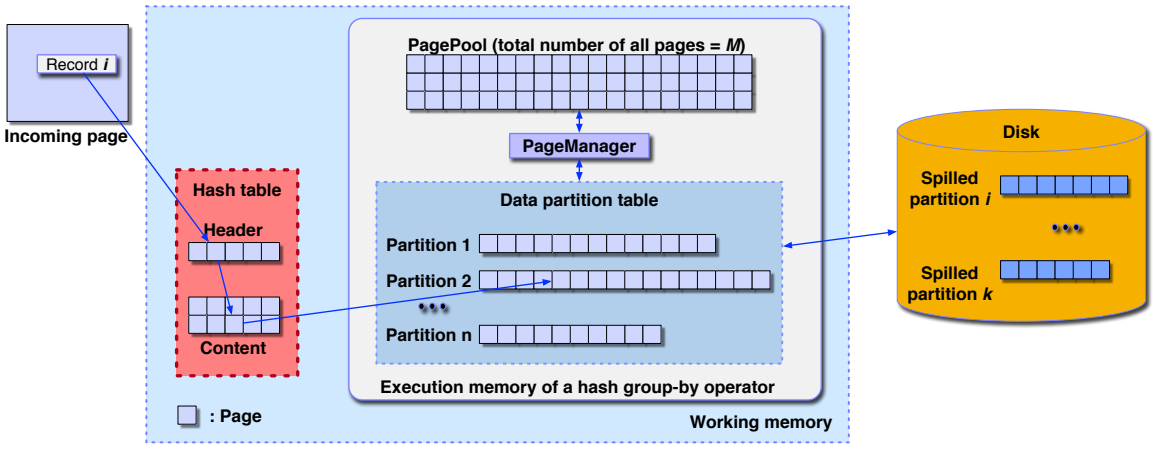


Figure 3.7: Hash Group-By: the original implementation.

of several partitions, and each partition contains zero or more pages to hold groups and their aggregate records. Each partition 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 3.7. The page manager is backed by the page pool whose maximum capacity is M.

The hash table holds pointers to the actual aggregate records to locate them efficiently as described before. Figure 3.8 shows an example instance of a logical hash table and 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 figure. 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 itself consists of two parts – *header* and *content* pages – as shown in Figure 3.9. 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 1,000 slots, the header slot for the 1,001st hash value will be the first slot in the

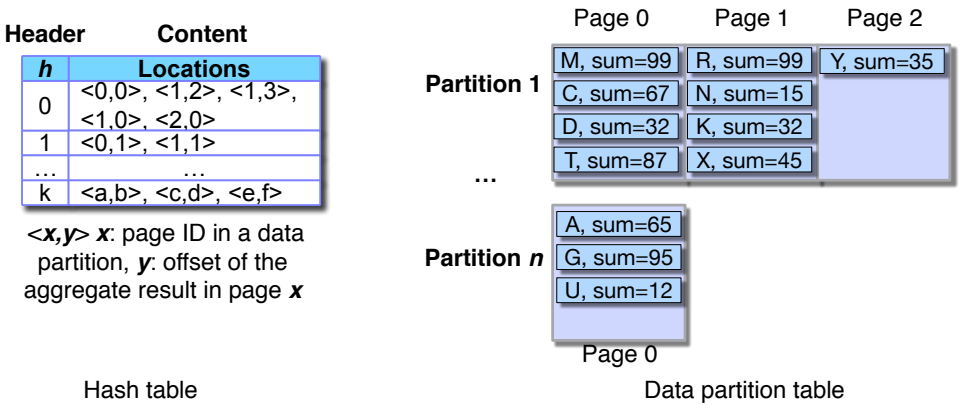


Figure 3.8: An example instance of a logical hash table and data partition table.

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 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 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. 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 also inserts the new aggregate record pointer that caused the overflow into the new slot. The content slot location will then be updated in the corresponding header slot in the header

page.

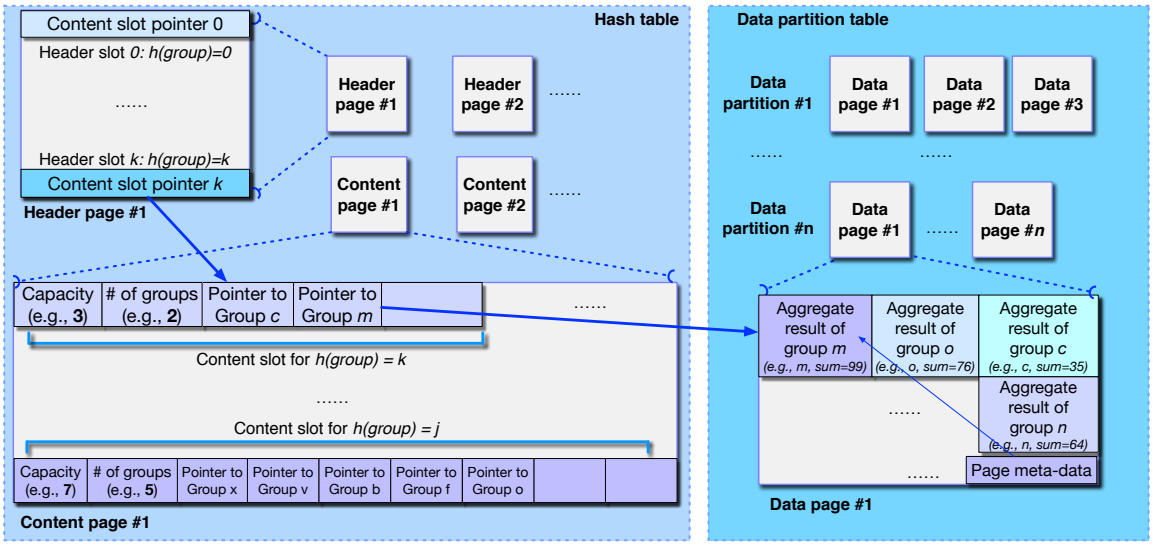


Figure 3.9: The detailed view of the hash table and the data partition table.

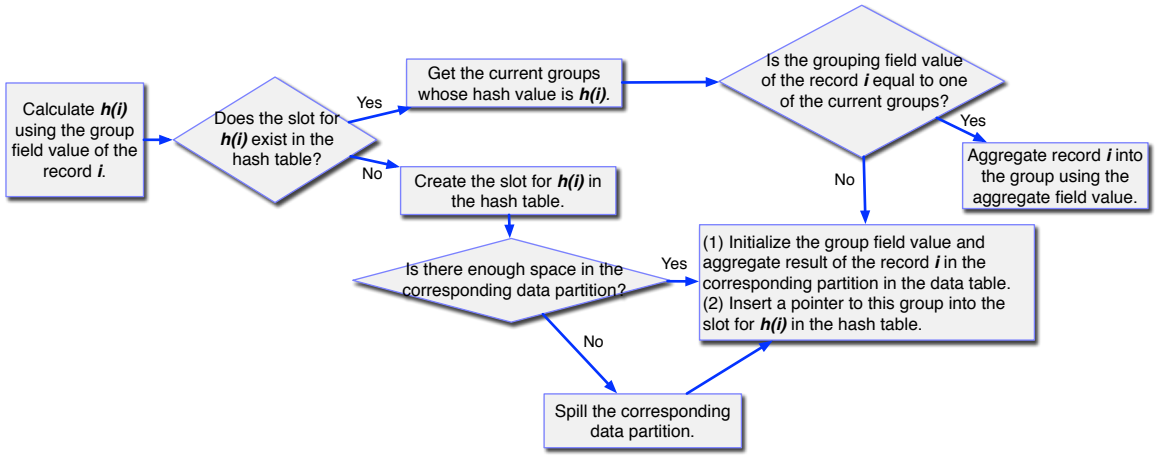


Figure 3.10: Aggregating a record.

Figure 3.10 shows how the data table and the hash table are used during the aggregation process for an incoming record i. This aggregation process allows the hash table and the data partition table 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. Therefore, after each iteration, the operator passes data partitions in memory to the next operator. The 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 3.11. (This incremental aggregation process that deals with the hash table and the data partition table at the same time is different from that of hash join, and the effect of this difference will be discussed in Section 3.7.3.3.)

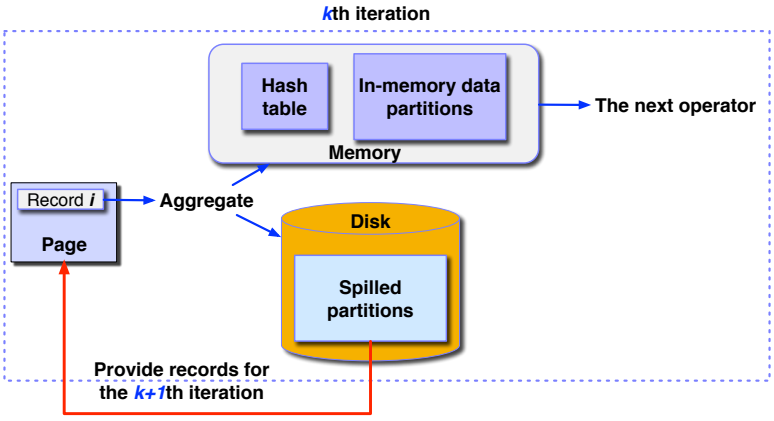


Figure 3.11: The data flow on each iteration of the hash group-by operation.

A major oversight in the original implementation was that the size of the hash table was not controlled within the budget M. Regarding this hash table size, most work including the original hybrid hash join paper [85] 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) [85]. There is one report that explained how hash join was implemented in DB2 that did consider the hash table as a part of the hash join memory footprint [63]. Following the popular literature, the original AsterixDB implementation also used the fudge factor to estimate the hash table memory footprint. Specifically, it used 20% of the budget M as the

fudge factor. However, the actual runtime hash table size was not constrained by this value. The operator could thus freely create header pages and content pages in the hash table as needed. When the working memory of the JVM instance was exhausted, an OOM could happen. We can see this issue (depicted in red) in Figure 3.7. 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. In fact, however, 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. We will see such a case in Section 3.7.3.2.

3.4.1.2 Hash Group-by: Current Implementation

To address the identified budgeting and control issue, we changed the design to 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. This new page manager shares the same page pool with the page manager of the data partition table. That is, the combined memory usage of both the data partition table and the hash table is bounded by the group-by budget M. (The reason why we do not use just one page manager is that the properties of the pages that these two page managers manage are different.) The current implementation is shown in Figure 3.12.

3.4.2 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 have discussed the detailed structure of these two tables, we focus on the operation flow of the hash join here.

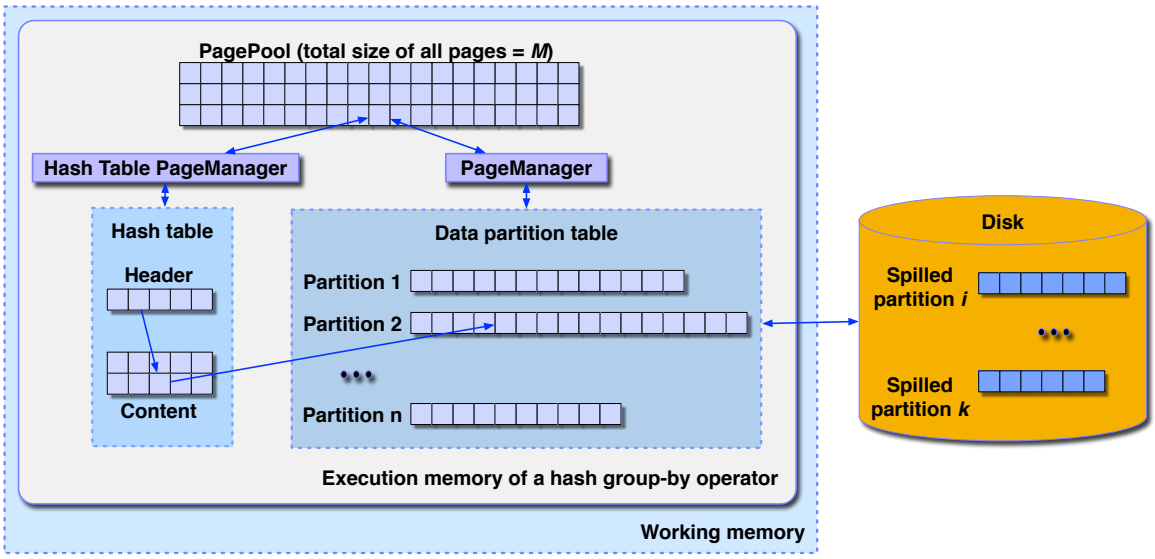


Figure 3.12: Hash Group-By: the current implementation.

3.4.2.1 Hash Join: Original Implementation

We have implemented a variation of a hybrid hash join [85] called *optimized hybrid hash join* in AsterixDB. The main difference between the hybrid hash join and our join technique is that the optimized-hybrid-hash-join operator does not pre-decide which data partitions should be kept in memory during the build phase of a hash join. Rather, these in-memory partitions are dynamically decided during the join process, similar to the hash group-by operation.

In AsterixDB, the hash join and the hash group-by operators use very similar hash table and data partition table structures. Therefore, their overall processing is similar except for a few places. We focus primarily on the differences between the hash group-by and the hash join here to avoid duplicated descriptions.

The conceptual operation flow for a hash join is shown in Figure 3.13. 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 each incoming page from the inner branch, the hash join operator

first calculates its hash value $h(i)$ using the join field(s) 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 will fit 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(s) 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.

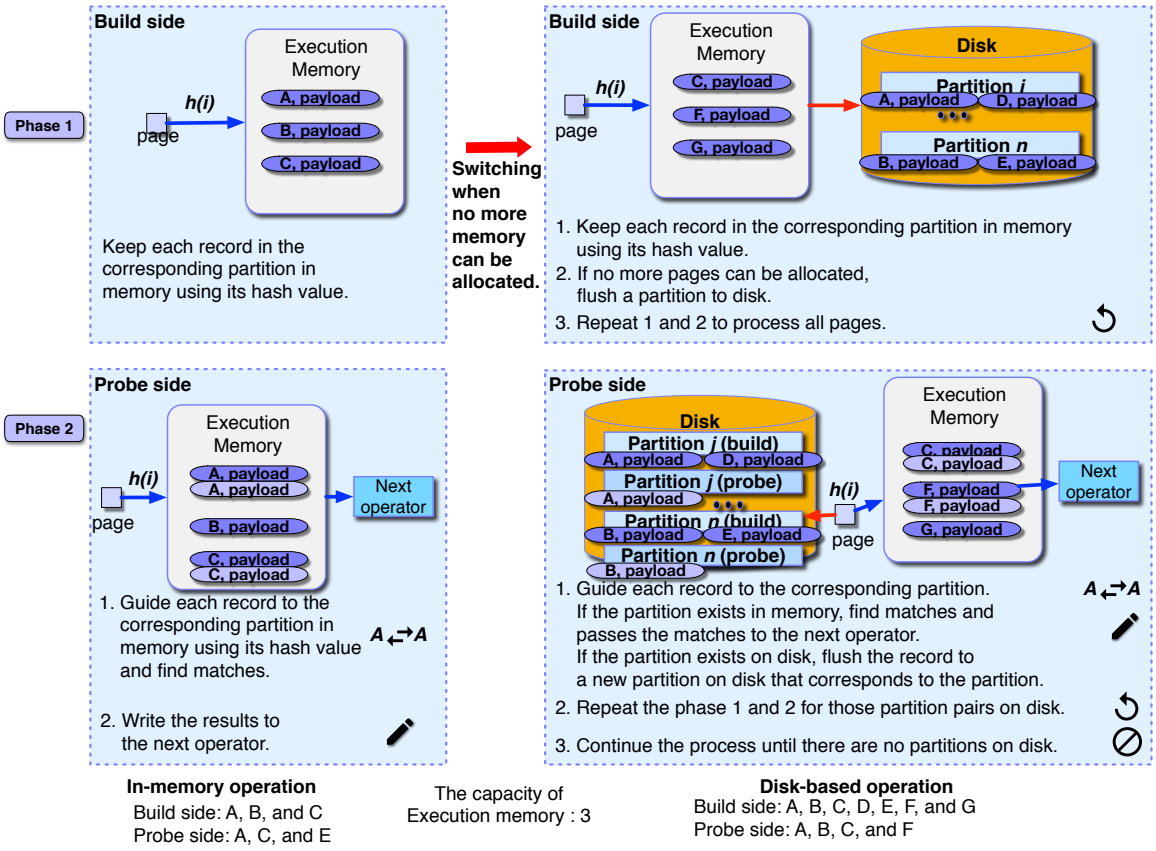


Figure 3.13: Two phases of the hash join.

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 hash values, the same number of contiguous hash values are 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 3.14. Note that, for a join, this operation is an insertion, not an aggregation. Thus, if there are two records whose join field(s) 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(s) values are the same.) As a consequence, the overall memory footprint of hash group-by is usually smaller.

Let us now consider the case of larger input data. 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, in-memory data partitions and spilled data partitions on disk are decided dynamically during this process. When the build phase is finished, some data partition's records are already spilled to disk and some data partition's records reside in memory. The operator then builds a hash table for the in-memory resident data partitions in order to perform an in-memory hash join for those partitions, and it then begins the probe phase as shown in Figure 3.15. 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 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 records in the corresponding data partition from the build side.

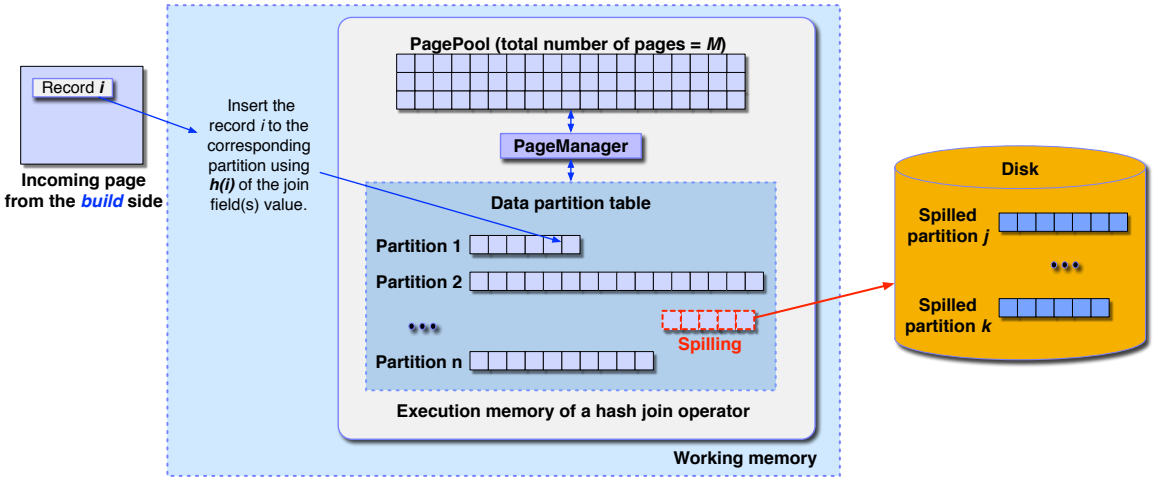


Figure 3.14: Hash join: the build phase.

During the probe phase, for each incoming record j from the probe side, the operator first calculates $h(j)$ and see whether the corresponding build partition has been spilled or not. If the build partition has been spilled, the operator just adds 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 add joined output records to the result output buffer page if the actual join condition holds. When the result output buffer becomes full, the page in the output buffer will be passed to the next operator. After the probe phase is done, there can be pairs of spilled data partitions from both build and probe side. For each pair, the hash join operator picks the smaller spilled partition of the pair as the build side and picks the other side as the probe (possibly involving *role reversal*) and begins a new hash join phase. This recursive process continues until there are no remaining spilled data partitions.

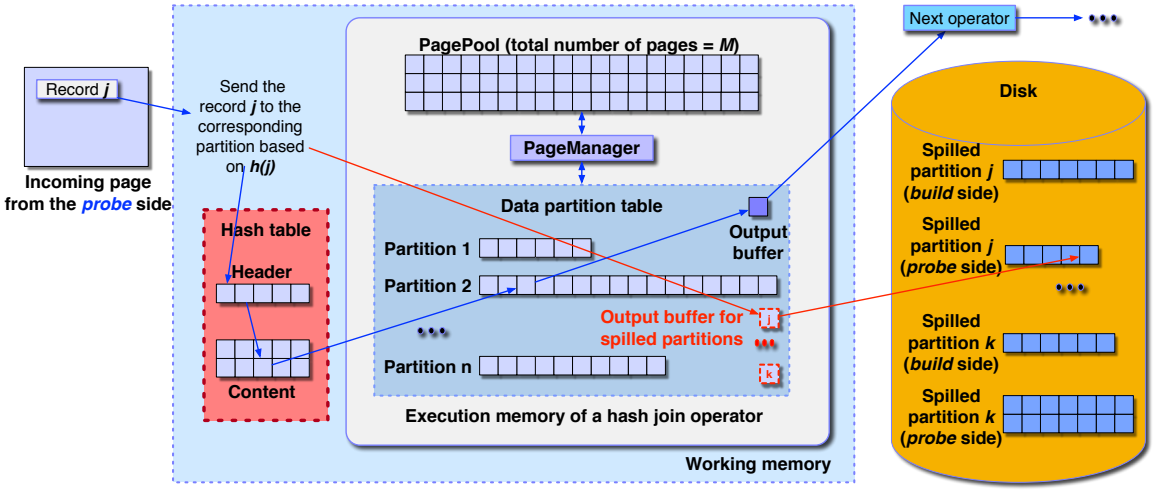


Figure 3.15: Hash join: the probe phase.

3.4.2.2 Hash Join: Current Implementation

Similar to the hash group-by case, the major budgeting oversight in the original implementation of the hash join was that the size of the hash table was not accounted for within the budget M, which was solely used for the data partitions. When building a hash table after processing all the records from the build side, the operator could 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 is depicted in Figure 3.15.

To address this issue, we changed the design of the hash join process to also consider the hash table size. However, a challenge 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 data partition table during the build phase of a join, there may not be enough space left for the hash table. In this case, the operator needs to 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. When this spilling

of partitions is done and there is enough space for the hash table, the operator builds the hash table for 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 100% utilized. As described before, the default capacity of a content slot 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 current implementation is shown in Figure 3.16. We can see there that the hash table allocates/deallocates pages from the hash table page manager and this manager shares the same page pool with the data partition page manager. (The reason why we use two page managers for one page pool is again that the properties of the pages that they manage are different.)

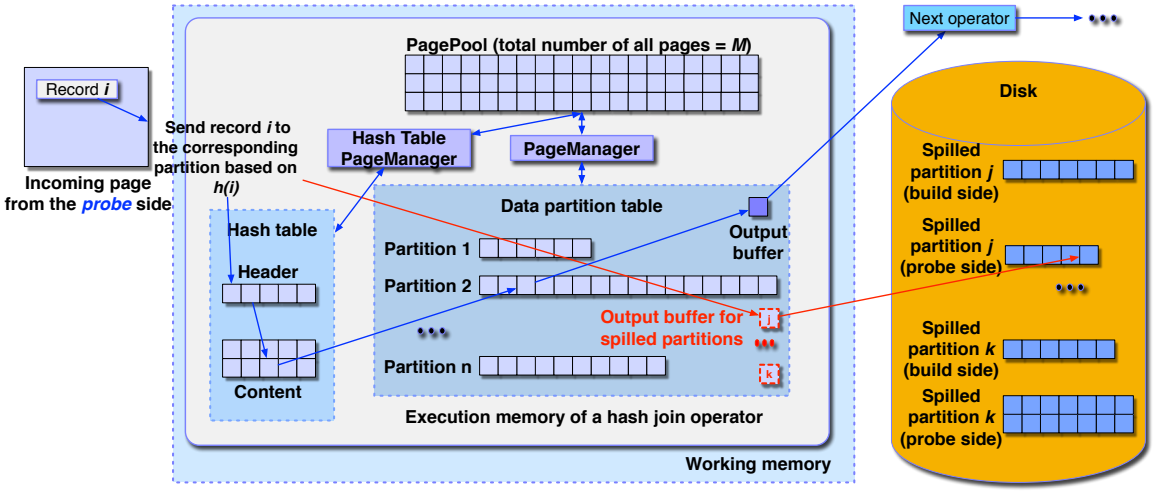


Figure 3.16: Hash join: the current implementation (probe step).

3.5 Memory-Intensive Operator: Inverted-index Search

In this section, we present the last memory-intensive operator – inverted-index Search. We first describe the original implementation of the inverted-index search operator and its memory management related issues. We then describe how we have addressed those issues in the current implementation.

3.5.1 Inverted-index Search: Original Implementation

An inverted-index edit-distance search in AsterixDB is performed using the T-Occurrence problem [57] approach. We can answer a text search query by computing the n-grams of the query string and retrieving the inverted lists of these grams. We then process the inverted lists to find all string ids that occur at least T times, since a string r within edit distance k of another string s must share at least $T = |G(r)| - k \times n$ grams with s [57]. Exact search can be treated as a special case of this formula where k is zero. Solving the T-occurrence problem yields a set of candidate string ids. The false positives are removed in a final verification step by fetching the strings of the candidate string ids and computing their real similarities to the query string. As an example, given a gram length $n = 2$, an edit distance threshold $k = 1$, and a query string $q =$ "memory", we would first compute the 2-grams of q as {"me", "em", "mo", "or", "ry"} and retrieve the inverted lists of these 2-grams. We consider the records that appear at least $T = 5 - 1 \times 2 = 3$ times on these lists as candidates. Last, we compute the similarity of these candidates to answer the similarity query. For the exact match case, we then compare the two strings to generate the final answer.

The conceptual operation flow for the inverted-index search operator is as follows. The operator first generates a token list from a query string. It orders the token list based on the length of the inverted lists from the shortest to longest. It then iterates over each token in the

token list. 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 it processes all tokens, the operator generates the final results, which are passed to the next operator. More details of this approach can be found in one of our previous papers [61].

As shown in Figure 3.17, in each iterative step, the operator processes one token, and its entire inverted list is loaded into the buffer cache. The operator then performs an intersection or union operation between the current inverted list and the previous intermediate search result to generate a new search result. 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. During each step, which set operation is used depends on the query predicate. If a query contains a disjunctive (OR) predicate, the operator performs a union operation. If a query contains a conjunctive (AND) predicate, the operator performs an intersect operation. For the next token, the 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 (the previous and the new) and one inverted list will all reside in memory. When processing the last token, the final search result is created to keep the final results, which will be passed to the next operator after the last set operation is finished.

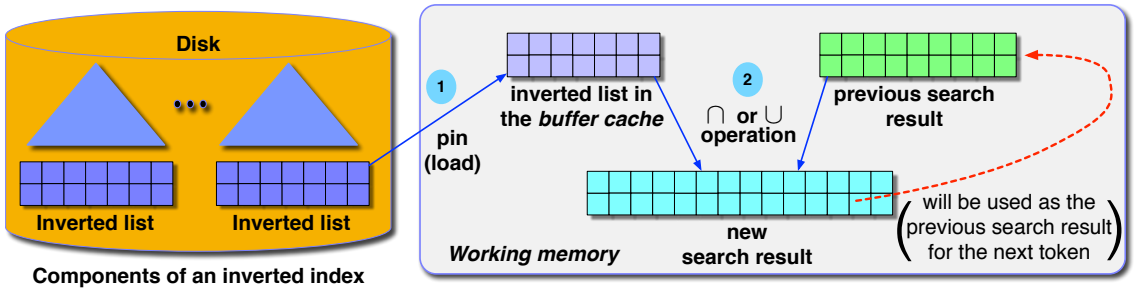


Figure 3.17: Inverted-index search: each iterative step.

Figure 3.18 shows the memory usage during this iterative search in the original implementation. The entire inverted list of the current token is loaded into the buffer cache. Also, the previous search result, the new search result, and the final search result can allocate memory pages as needed from the working memory.

There were several issues regarding memory management in this implementation. First of all, there was actually no defined budget M for an inverted-index search operation. (This operator was originally classified as an index lookup operation for budgeting purposes.) Therefore, the size of the previous search result and the size of the new search result were not controlled properly within a budget. This also means that there was no flexible disk-based operation for the inverted-index search. At any iterative step except the first step, working memory had to keep both the previous and the new search result. If an operation was a union, the size of the new search result could be the sum of the previous search result and the current inverted list. In other words, the size of these structures could easily increase, so an OOM could happen at any iterative step. The second issue was that the buffer cache was required to hold (pin) an entire inverted list for a token regardless of its size. If the inverted list size was greater than the buffer cache size, it would generate an exception after trying to find space to hold the entire inverted list in the buffer cache. Even if an inverted list could be loaded into the buffer cache, the inverted list had to be kept during the set operation. Unlike other page-level buffer cache related operations [38], this duration could be long depending on the size of the previous search result. In addition, the operator finished the entire calculation process before passing any result to the next operator. Thus, it needed to keep many pages for the final result as a consequence. (Keeping many pages for the final search result could be avoided if the operator had keep only one page as the final search result buffer and flushed the result each time this page became full.)

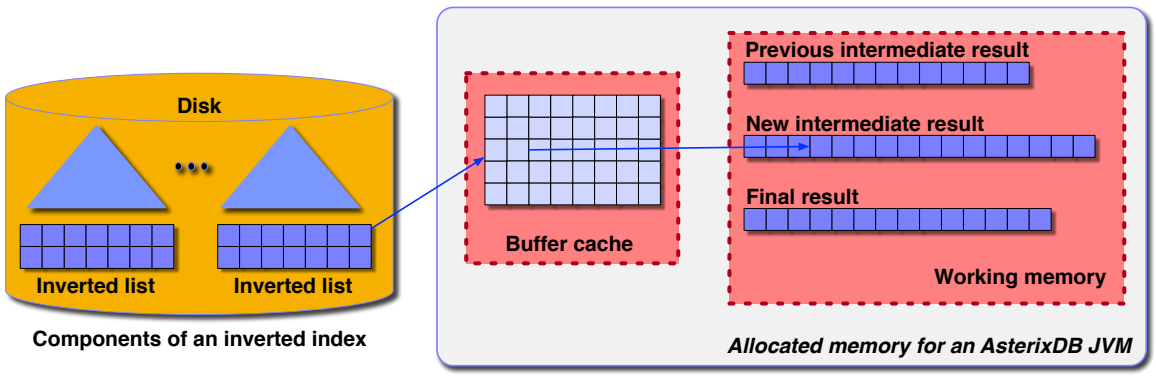


Figure 3.18: Inverted-index search: the original implementation.

3.5.2 Inverted-index Search: Current Implementation

To address the issues, we have introduced budget M for inverted-index searches. As a consequence, we changed the design of the inverted-index search operator to 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 it 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 should be allocated as shown in Figure 3.19. When there is not enough memory, these pages will be used as I/O buffers to read the current inverted list and the previous search 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 search result.

Since the essential part of the T-Occurrence problem is traversing a new inverted list using the previous search result, we put the highest memory allocation priority on the inverted list. Except for 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. The reason is to read a large chunk of the inverted list and use more efficient means to traverse the chunk (e.g., a binary search) to avoid an expensive full-scan that traverse all

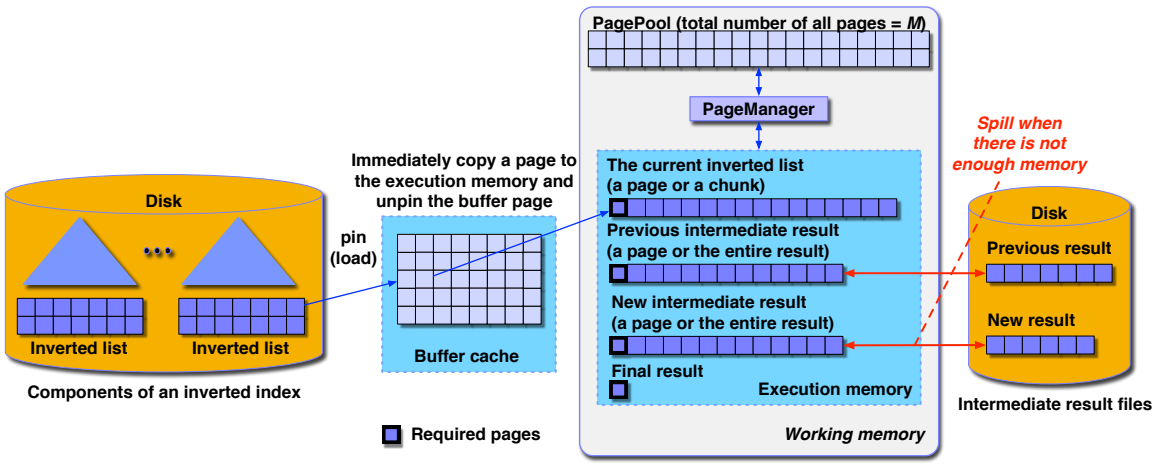


Figure 3.19: Inverted-index search: the current implementation.

keys in the chunk one by one. If the operator is indeed able to read the current inverted list in its entirety, it checks whether it can also read the previous search result entirely into memory if the result was written to disk. If there is still leftover memory, the new search result can also be written into memory. Thus, if the operator has a large enough budget M, the search behavior becomes exactly the same as the original implementation. The difference is that its memory usage is now properly controlled within the budget M: When there is not enough memory to hold the entire inverted list, the operator then only uses two buffer pages to read the previous search result from disk and write a new intermediate search result to disk and stays within its M-page limit.

In addition, to solve the issue where inverted list pages were pinned for a long time in the buffer cache, now 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 content 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.

One more change that we have made is that the operator no longer generates the entire final search result before passing it to the next operator. 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. Therefore, for the final search result, the operator only assigns one page as the output buffer. 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 receives the result, the operator continues the search process to fill the search result buffer page in again.

The final change that we have applied is that the inverted-index search operator avoids creating a separate final search result when the number of tokens in the query is just one. In the original implementation, even when the number of tokens was one, the operator loaded the token's inverted list into the buffer cache and generated the separate final search result. This was wasteful since if the operator was able to return the result directly from the inverted list, this copy operation was not necessary. In the current implementation, the operator 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 anymore.

3.6 Global Memory Management

So far, we have discussed the individual memory-intensive operators in detail. Even though each memory-intensive operator now 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 we explain how we handle query admission control. We then discuss another miscellaneous but important memory-related issue, namely, how we manage memory for records that may be larger than one disk page.

3.6.1 In-memory LSM Components

As described in Section 3.2, as a result of having the LSM index structure, we need to explicitly allocate some of the system's memory space to the LSM indexes to initially hold the results of insert/upsert/delete operations on a dataset. This section of memory is called the in-memory components area. To control the size of this section of memory, we originally had three parameters as illustrated in Figure 3.20. The globalbudget parameter determined the overall maximum size of all in-memory components. The numcomponent parameter set the number of in-memory components that each index could have. (The purpose of this parameter was 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 one needs to be flushed, the status would be changed to immutable and no more insert/upsert/delete operations could be made to the dataset. 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.) The numpages parameter set the maximum number of in-memory pages that any given dataset could allocate before triggering a flush operation for its components. All in-memory components of the primary and secondary indexes of a dataset shared this budget. For example, if this parameter was set to 1,000 and the page size was set to 128 KB, then each dataset could use about 128 MB of memory at a maximum.

Using the aforementioned parameters, the maximum number of concurrently active datasets was determined indirectly. For instance, if the maximum size of the in-memory components area was 1,000 MB, the maximum number of pages for a dataset was 200, and the page size was 0.1 MB, each dataset could occupy about 20 MB. That is, the number of maximum concurrent datasets was 50. A user of the original implementation had to perform this

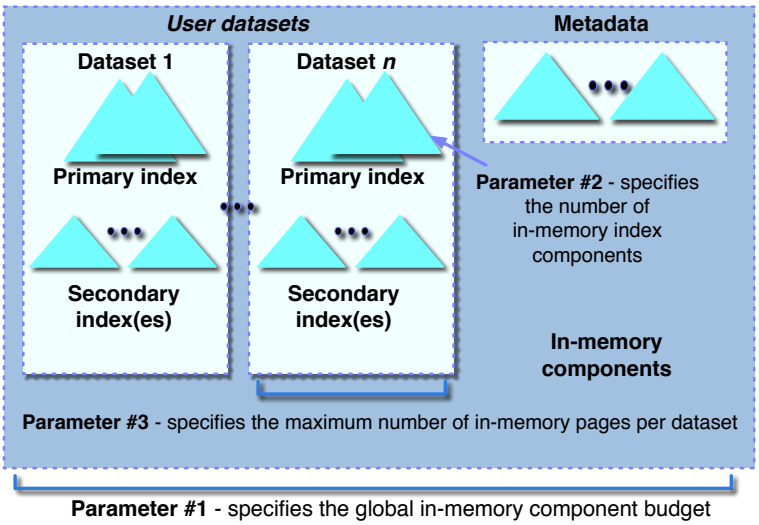


Figure 3.20: LSM component memory and two configurable parameters.

calculation to get the number of maximum concurrent user datasets supportable for their workload.

To make AsterixDB's controls more user-friendly, we have introduced a new parameter called maxdatasets to directly set the maximum number of concurrent active datasets in the in-memory components section of memory. Now, based on the overall size of the in-memory components area, a user can choose a number and it is easier for them to understand the maximum size that each dataset can have. The system then sets the parameter based on this number. For example, if the size of the overall in-memory components is 800 MB and maxdatasets is set to 10, each dataset can occupy 80 MB in the in-memory components section in memory.

3.6.2 Query Admission Control

The original implementation of AsterixDB did not have a query admission control feature. While controlling the amount of memory for memory-intensive operators, in-memory components, and the buffer cache size was sufficient to support the execution of concurrent queries,

an AsterixDB instance could generate an OOM in case of too many concurrent queries.

To address this issue, explicit query admission control logic was added to AsterixDB. This logic considers two factors in each query's plan to manage the number of concurrent queries in an AsterixDB instance. The two considered factors are the number of CPU cores desired and the required memory size for the query plan as shown in Figure 3.21. For memory, if a query requires more memory than the available memory, 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 parameter to a higher number (e.g., 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 at that time even though it can handle other operations. Thus, in order to fully utilize CPU cores on the system, when considering the number of CPU cores, AsterixDB's admission control policy also includes a tuneable coremultiplier parameter (which defaults to 3).

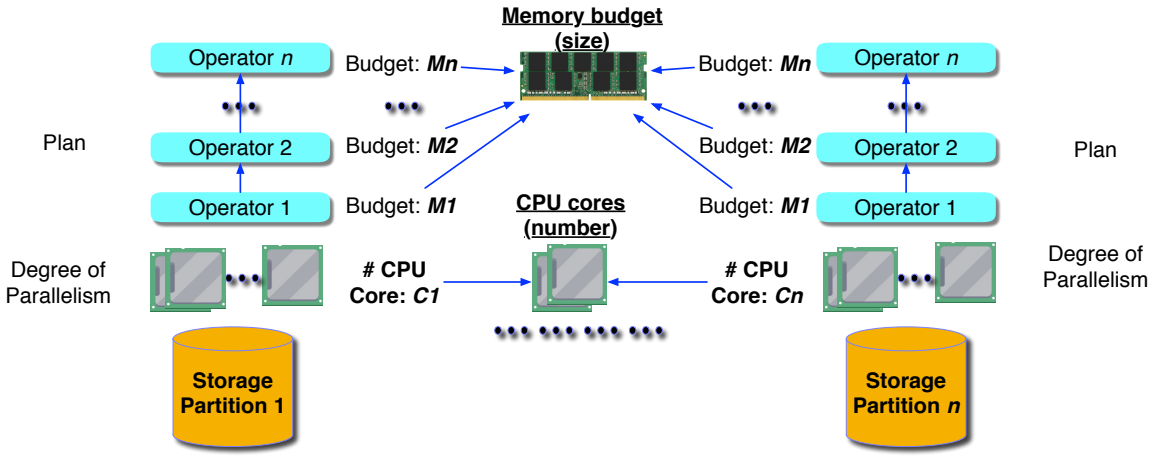


Figure 3.21: Factors in query admission control.

When an instance starts, AsterixDB collects the number of CPU cores and the available JVM heap memory that the instance can use by checking the JVM runtime. It uses the collected data as the maximum capacity, and 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 3.22, a query can be executed immediately if both the required number of CPU cores and the required memory size are available. A query will be queued if either the required number of CPU cores or the required memory size is less than the maximum capacity but one of them is greater than currently available capacity. 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 either the required number of CPU cores or the required memory size is greater than the cluster's overall maximum capacity.

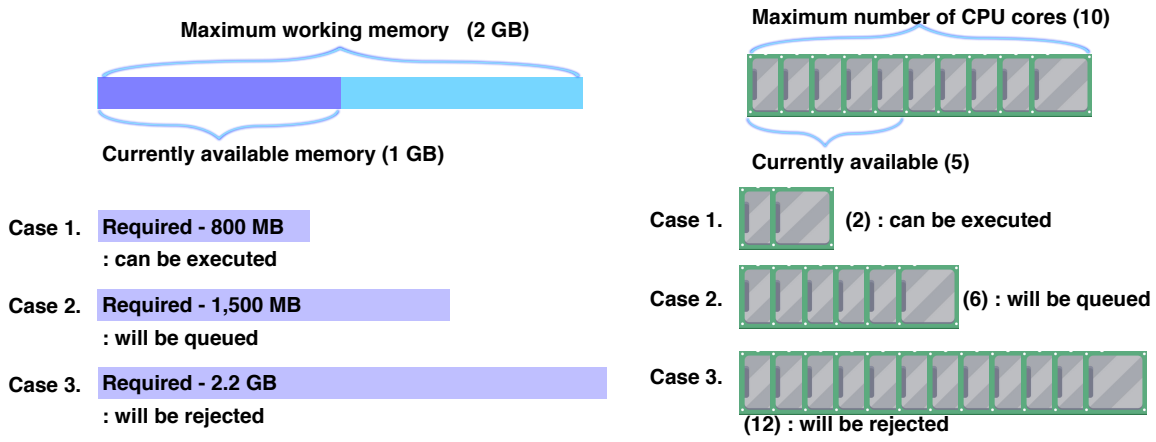


Figure 3.22: Some example query admission cases.

The degree of parallelism per query is controllable via a user-provided parallelism parameter called parallelism. Based on the number of physical dataset partitions and this user-provided 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 operators. If an operator is not memory-intensive, the memory control logic assumes that

the operator requires only one page. Memory requirements of memory-intensive operators can be computed by from their system configuration parameter value or a query-specific parameter. For instance, the per-operator sort memory size (e.g., 128 MB) can be set in the system configuration file. Its value will be used to compute 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.

Initially, AsterixDB's admission control logic conservatively assumed that all operators in the plan may execute at the same time, making the query's required memory estimate be the sum of the requested memory per operator. However, because of wait-for dependencies among the activities in operators, there are usually multiple execution stages in an query execution plan. It is not always the case that all operators can 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 after the join operator cannot be executed together with the build phase of the given join operator; they need to wait until the build phase of the hash join finishes and the probe phase has begun.

While the initial conservative admission control logic was sufficient to eliminate OOMs, we found that it was prone to underutilizing the system's resources. We have now improved the logic to allow more concurrent queries while still ensuring the stability of the memory behavior of the system. As described before, only operators (activities) in the same stage of a query plan can be executed in parallel. Based on this fact, the current admission control logic considers the generated execution stages (steps) in the query plan. The current 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 size of memory requirement and sets those as the required resources for the plan. (A comparison between the initial logic and the current logic will be discussed in Section 3.7.)

3.6.3 Handling Big Objects

When discussing the memory use of the memory-intensive operators so far, our implicit assumption has been that each record will fit into a page. All operations that we have described work based on this assumption. However, in practice, this assumption is not always true. Although the page size can be set by a user, 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 files 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 has been adopted to accommodate large records without exceeding its memory budget.

3.6.3.1 Adjusting the Storage layer

Let us first focus on the storage aspects of coping with large records. To make our design simple and to minimize the codebase impact, we chose to keep the basic page size identical (and fixed) since changing it 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 3.23. 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 storage consists of multiple sequential pages. To reflect this concept, we add new metadata on the

first page called the page multiplier and let it contain the number of actual physical pages. For instance, the page multiplier in the first page on the right side of Figure 3.23 is set to 6 since this logical page consists of 6 physical pages.

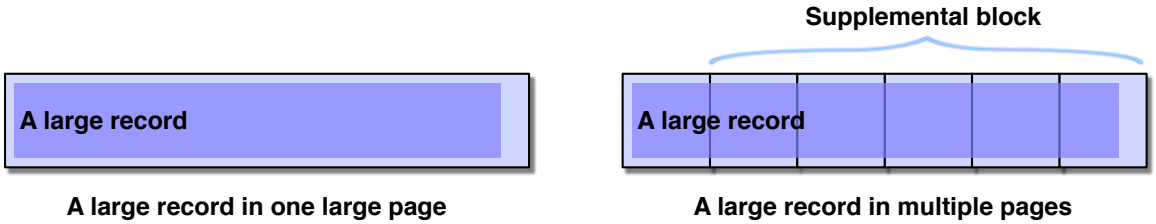


Figure 3.23: Two ways to store a large record.

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 is stored across all pages (including the first page) as shown in Figure 3.23. All pages after the first page consist entirely of the record data 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 fetched using the page multiplier and this page ID.

Another storage consideration is how to intermix a large record with non-large records. AsterixDB will attempt to separate (page-wise) a large record from other non-large 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 single small record is to avoid creating a page whose space utilization is very low.) However, if there are multiple records in the page, we insert 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 is executed as follows: For a write operation, AsterixDB checks the meta-data of a page in the buffer cache to get the actual number of pages and writes those pages sequentially to disk. When AsterixDB reads a page from disk, it reads the (first) 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.

3.6.3.2 Adjusting the Runtime Operators

Because of the logical page approach to handle big objects, when runtime operators read a logically large page, it can be regarded as "one page". Since we have encapsulated page access, all runtime operators can access a larger logical page as if it were accessing a regular single page. However, one thing needs to be adjusted. The size of an incoming page can be different now, and this fact should be reflected in 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 one 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

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 did 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 their 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.

Hash Join Operator: Similar to the hash group-by case, when a logical page that contains a large record is being inserted into a spilled data partition during the build or the 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 a regular fixed-size primary key. (It does not seem realistic to expect a primary key to be a large field that spans multiple pages, so we do not consider that case.)

3.7 Experiments

We have performed an experimental evaluation of our budget-driven memory management implementation using both synthetic and real datasets. We used a single-node cluster to host an 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 cluster environment. This node ran CentOS 6.9 with a Quadcore AMD Opteron CPU 2212 HE (2.0GHz), 8GB

RAM, 1 GB Ethernet NIC, and had two 7,200 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 5.3 shows the AsterixDB configuration parameters for the experiments.

Table 3.1: 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

3.7.1 Test Datasets

We used one variants of synthetic dataset called the Wisconsin benchmark dataset [47] 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 [91] to perform the query admission control experiment since it contained many memory-intensive operators in its queries. We also used a real dataset of Reddit comments [79] to test the behavior of the inverted-index search operator. Figure 3.24 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 TPC-H dataset was greater than that of the raw 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 name, 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 (i.e., 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 8 KB unused on such a record's page.

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 26K.
Wisconsin-Norm-M	1,000,000	28,645	37,733	A large string field was included. The field length was normally distributed. The average length was 26K and the standard deviation was 4.4K. (95% range: 17.8K ~ 35K)
Wisconsin-Norm-L	1,000,000	28,651	41,423	A large string field was included. The field length was normally distributed. The average length was 26K and the standard deviation was 8.8K. (95% range: 8.4K ~ 43.6K)
Wisconsin-Gam-1	1,000,000	28,603	39,417	A large string field was included. The field length followed a Gamma distribution (shape: 1.5, scale: 1). The average length was 26K.
Wisconsin-Gam-2	1,000,000	28,773	37,651	A large string field was included. The field length followed a Gamma distribution (shape: 1, scale: 0.5). The average length was 26K.
TPCH-Customer	1,500,000	234	326	TPC-H Dataset (scale factor:10)
TPCH-Lineitem	59,986,052	7,416	16,150	TPC-H Dataset (scale factor:10)
TPCH-Partsupp	8,000,000	1,150	1,619	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)

Figure 3.24: AsterixDB datasets.

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 the dataset to

make it meet our needs. Table 3.2 shows a part of the schema of our Wisconsin benchmark dataset. The italicized field represents an extra field that we added.

Table 3.2: 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
...			

For the experiments where no large string field was involved, we used the basic Wisconsin dataset in Figure 3.24. For performing experiments on records with a large string field, based on the schema in Table 3.2, we created five more datasets as shown in Figure 3.24 to see how the length distribution of a large string field affected 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 randomly among 16 HEX characters (0 to F). The only difference among the latter 5 Wisconsin datasets was that the length distribution of the large string field was different; the first three datasets were based on a normal distribution as shown in Figure 3.25 and the other two datasets were based on a gamma distribution as shown in Figure 3.26. 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 4,400, and that of the third dataset, Wisconsin-Norm-L was 8,800. 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

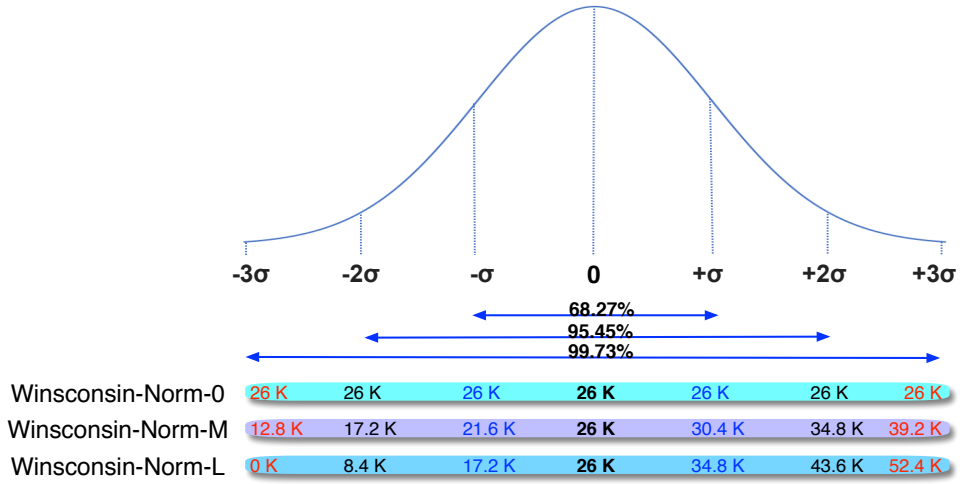
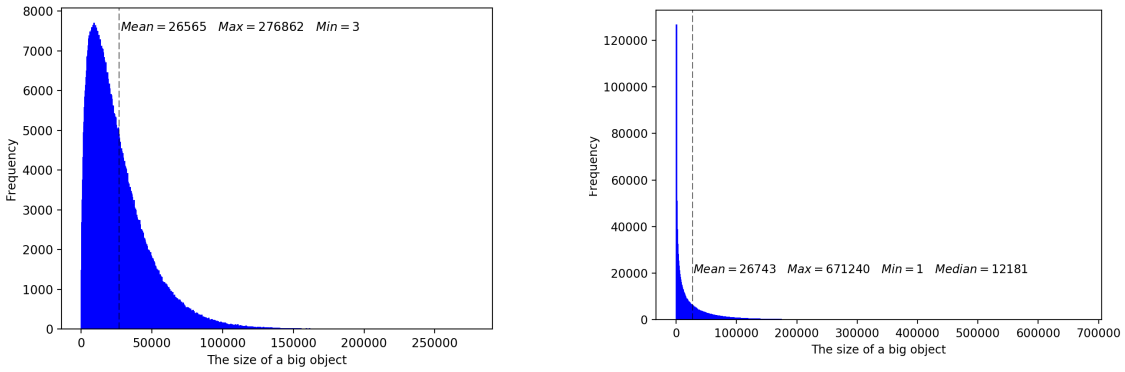


Figure 3.25: Length distributions of large string fields using Normal distribution.

that of the second dataset, Wisconsin-Gam-2, since the second dataset had a more skewed distribution toward zero.



(a) Gamma distribution 1.

(b) Gamma distribution 2.

Figure 3.26: Two Gamma distributions of large string fields used for Wisconsin datasets.

For the query admission control experiment, we used the standard TPC-H benchmark dataset with a scale factor of 10. In addition, we used one TPC-H query that had two joins, one group-by, and one order-by predicate to use multiple memory-intensive operators to emphasize 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 non-synthetic dataset that contained text data. It contained actual 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 it did not have a field that could be used as a primary key field, we instructed AsterixDB to assign an additional auto-generated 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.

3.7.2 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, we compared the original implementation with the current implementation of each of AsterixDB's memory-intensive operations. We measured the memory footprint of the data structures during memory-intensive operations.

We first considered a case where the referenced portion of each input record only consisted of one to three integer fields from 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 case where the referenced part of an input record consisted of one integer and one or two string fields whose length was 100, which seems more realistic. We used the same source dataset for that case, too. For all memory-intensive operators, we set the operator's memory budget to 128 MB.

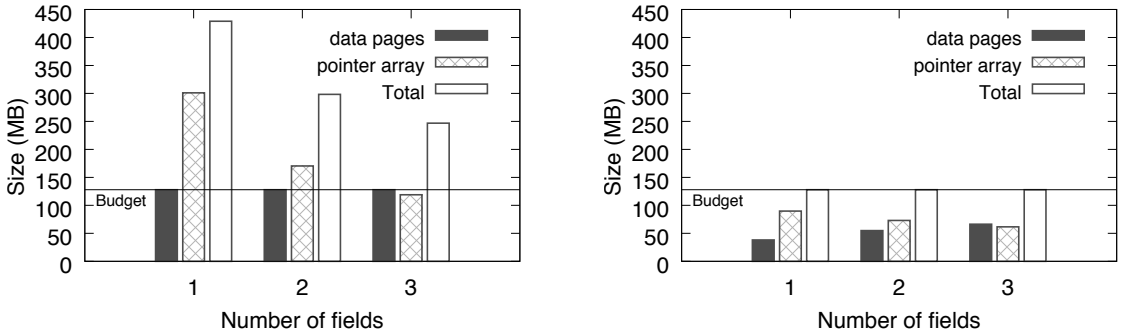
For each case, we first determined the input record cardinality where each memory-intensive operator used its budget at full capacity. To find this cardinality, we gradually increased the number of input records to each memory-intensive operator and found the exact operating

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 was 2,000,000, we used 1,999,999 records as the input cardinality and measured the operator's memory use at that operating point.

3.7.2.1 Sort Operator

To study the memory usage of the sort operator, we used query templates Q1 and Q2 shown in Figure 3.34 at the end of this section. The template shows the queries for the three-field cases for integers and strings. For the one-field case, we excluded two fields. For the two-field case, we excluded one field. 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.

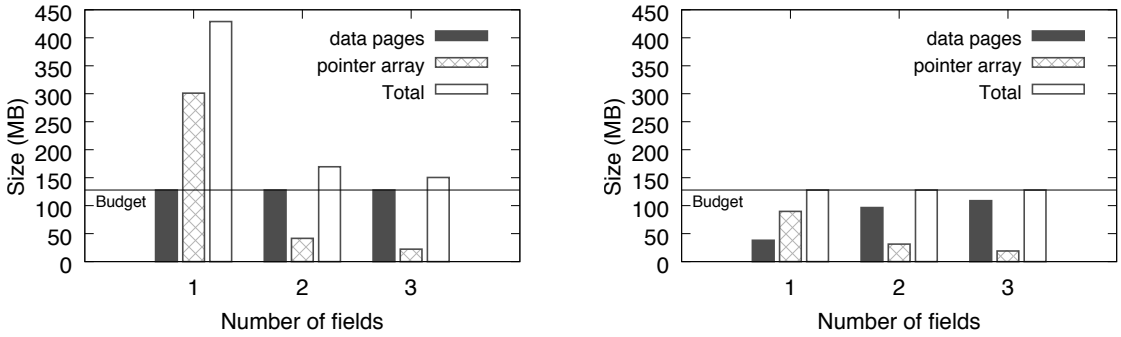
As described earlier, in the original implementation, AsterixDB's sort memory control logic did not consider the record pointer array size in the budget M, viewing it as negligible overhead. In reality, the size of the record pointer array during a sort operation is generally not negligible, as shown in Figure 3.27(a). When the record to be sorted is small, consisting of only one integer field, the pointer array size was 301 MB, and this was almost three times the allocated budget, which was 128 MB. For the first sort implementation, the budget M was solely used to track the memory used to load data pages. Thus, in this case, sort operation would actually use 430 MB in total even though the operator's budget was set to 128 MB. 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, and this alone is similar to the allocated budget M. In the current AsterixDB sort implementation, in Figure 3.27(b), we can see that both record pointer array and the data pages properly share the budget M, so their total size never exceeded the budget M.



(a) The first AsterixDB sort implementation. (b) The current AsterixDB sort implementation.

Figure 3.27: The size of data structures for a sort operation (integer fields).

For the second case, where 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 3.28. 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) The first AsterixDB sort implementation. (b) The current AsterixDB sort implementation.

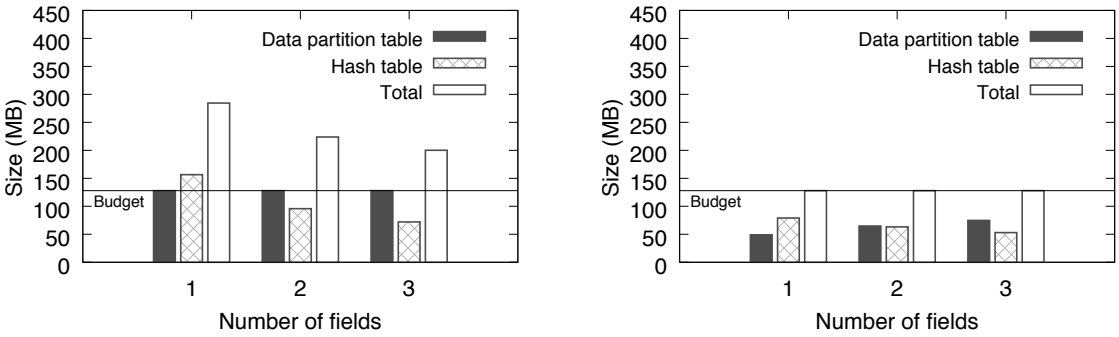
Figure 3.28: The size of data structures for a sort operation (string fields).

3.7.2.2 Hash Group-by Operator

Next, we evaluated the memory usage of the hash group-by operator in a similar fashion using query templates Q3 and Q4 in Figure 3.34. The figure shows the queries for the three-

field cases for integers and strings case. In the select value count clauses, we again included all fields so as not to let the optimizer project out the fields before the hash group-by operator.

In the first AsterixDB hash group-by implementation, the hash table size was not accounted for within the budget M. In this experiment, we first identified the maximum input record cardinality for which the operator would not spill any partition 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. As we can see in Figure 3.29(a), the hash table size for the one integer case field was 156.5 MB, 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.

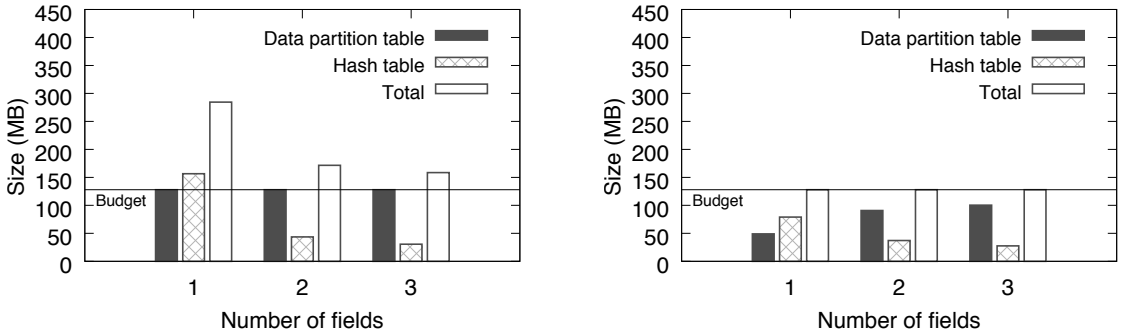


(a) The first AsterixDB hash group-by implementation.

(b) The current AsterixDB hash group-by implementation.

Figure 3.29: The size of data structures for a hash group-by operation (integer fields).

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 addressed this issue successfully. We can see in Figure 3.30(b) 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.



(a) The first AsterixDB hash group-by implementation.

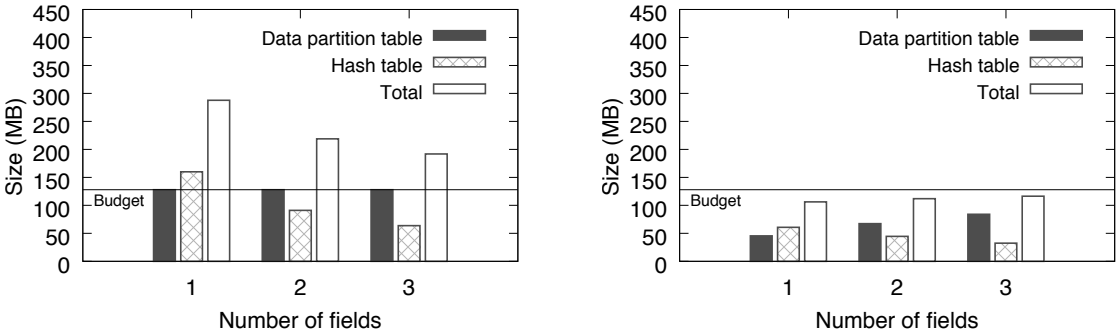
(b) The current AsterixDB hash group-by implementation.

Figure 3.30: The size of data structures for a hash group-by operation (string fields).

3.7.2.3 Hash Join Operator

We next measured the memory usage of the hash-based join operator which also uses a hash table and a data partition table. We used query templates Q5 and Q6 in Figure 3.34. The template shows the three-field cases for integers and strings. In the select value count clauses, we included all fields so as not to let the optimizer project out the fields before the hash join operator. For these queries, we selected 1,000 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.

Like the hash group-by operation, the hash table's size during a hash join operation was not considered within the budget M in the first AsterixDB implementation, instead being assumed to be negligible overhead. As we can see in Figure 3.31(a), 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 the assigned budget.

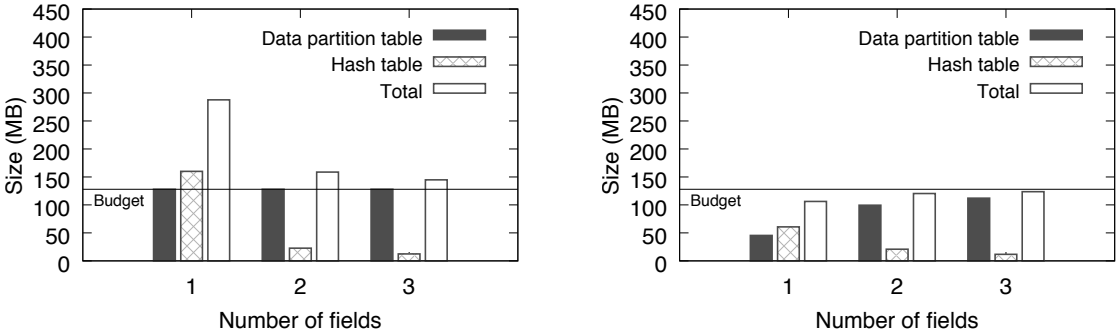


(a) The first AsterixDB hash join implementation.

(b) The current AsterixDB hash join implementation.

Figure 3.31: The size of data structures for a hash join operation (integer fields).

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 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), though less drastically so in this case.



(a) The first AsterixDB hash join implementation.

(b) The current AsterixDB hash join implementation.

Figure 3.32: The size of data structures for a hash join operation (string fields).

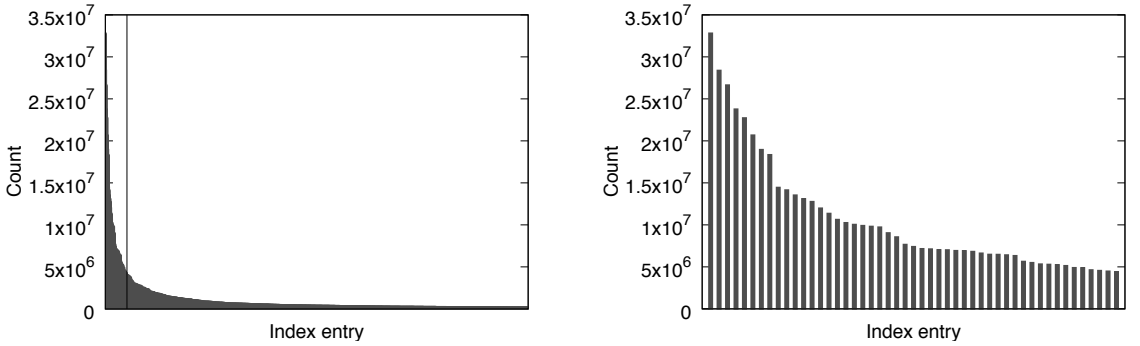
The current AsterixDB implementation successfully addressed the issue. We can see in the figures that the space utilization during a hash join operation is now always kept within the

operator's assigned budget. Note 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 100% of 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 3.4.2. 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 might reside in a separate hash slot. This estimation is a safe way to ensure that the hash table size will not exceed the estimated budget when constructing it. 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 near 100%.)

3.7.2.4 Inverted-index Search Operator

We now examine the memory use of the inverted-index search operator. The first implementation of AsterixDB's inverted-index search did not have a budget at all, so the size of its data structures was not accounted for. As described in Section 3.5.1, 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. We can estimate the footprint of these data structures by examining an actual inverted index. Figure 3.33(a) shows the size of the top 1,000 inverted lists in a full-text index on the `body` field of the Reddit comment dataset. We can clearly see that the distribution follows Zipf's law. Figure 3.33(b) shows the first 50 entries of the previous figure. Let

us examine the first entry of the inverted list. The frequency of occurrence 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's worth of the entire Reddit comment data; if more data were inserted, this size would increase proportionally as well. Clearly, based on these numbers, the first inverted-index search implementation could generate an OOM early for such real-world data sets. We do not include an actual comparison between the first implementation and the current memory-conscious implementation here since the first implementation was not a budget-based approach. Our current implementation has an assigned budget M and now properly supports disk-based operation as a consequence (as we will see shortly).



(a) Inverted list size (first 1,000 entries). (b) Inverted list size (first 50 entries).

Figure 3.33: The inverted list size of an inverted index on the body field of the Reddit comment dataset.

3.7.3 Living within The Budget

Our next experiment focused on the current AsterixDB operator implementations, which properly use the assigned budget M. We measured the average execution time of each of AserixDB's memory-intensive operations in the current implementation to verify that these operations are scalable since they need to work well regardless of the input data size. That is,

```
/* 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;
```

Figure 3.34: Query templates to measure the size of data structures during memory-intensive operations.

transitioning from in-memory to disk-based operation should work seamlessly. 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 effects from large string fields. (We will explore those effects separately later.) For each selectivity, we executed 100 random queries where each query picked a random contiguous input record range. For example, if the number of desired input records was 1 million, each query picked 1 million contiguous records from the dataset. However, the first record's starting position was randomly chosen.

3.7.3.1 Sort Operator

We first measured the average execution time of the current sort implementation using query template Q7 in Figure 3.38. We sent 100 random queries where each query had a randomly chosen unique2 range.

Figure 3.35 shows the average execution time of the sort operation as the input size is 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 operation 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 4,096 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 step, while the input dataset size was only 24 GB.

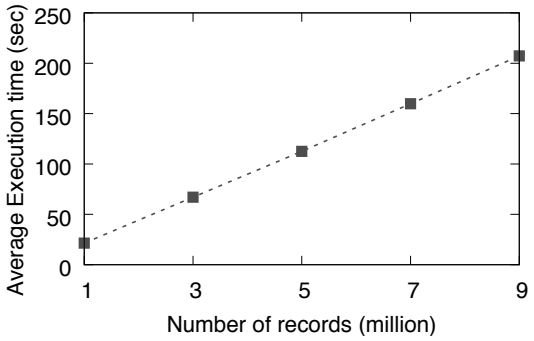


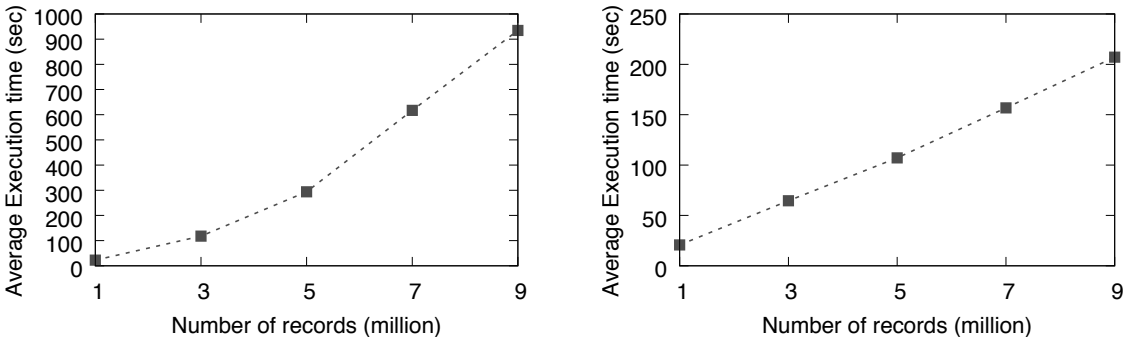
Figure 3.35: The average execution time of sort queries.

3.7.3.2 Hash Group-by Operator

Next, we checked how well the hash group-by operation scales using query template Q8 in Figure 3.38. We used 100 random queries where each query had a randomly chosen unique2 range. We used two fields to perform the hash group-by operation.

Figure 3.36(a) shows the average execution time for the hash group-by queries. 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 3.4.1.1, some data partitions may be spilled to disk in each iteration of the hash group-by operation. After processing all records, the in-memory partitions are passed to the next operator and the next iteration of the hash group-by operation then 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 3.36(a), where the number of records was 1 million, finished without spilling any data partitions to disk. That is, the group-by operation finished in the first iteration. 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 million, the operation finished only after the third iteration. Notice that we can see a linear trend within the same number of iterations (cases 5, 7, and 9).



(a) Hash Group-by.

(b) Hash Join.

Figure 3.36: The average execution time of hash group-by and hash join queries.

3.7.3.3 Hash Join Operator

Next, we checked how the hash join operator scales using query template Q9 in Figure 3.38 to measure the average execution time. From the outer branch (denoted by r), which was used as the probe branch, we randomly selected 1,000 records. For the inner branch (denoted by s) and used as the build branch), 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, 9 million records are 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 1,000 records from the probe branch were processed. With the datasets, during the build phase, only the first case (1 million records) did not spill any data partitions to disk. The other four cases spilled increasing more of the partitions to disk.

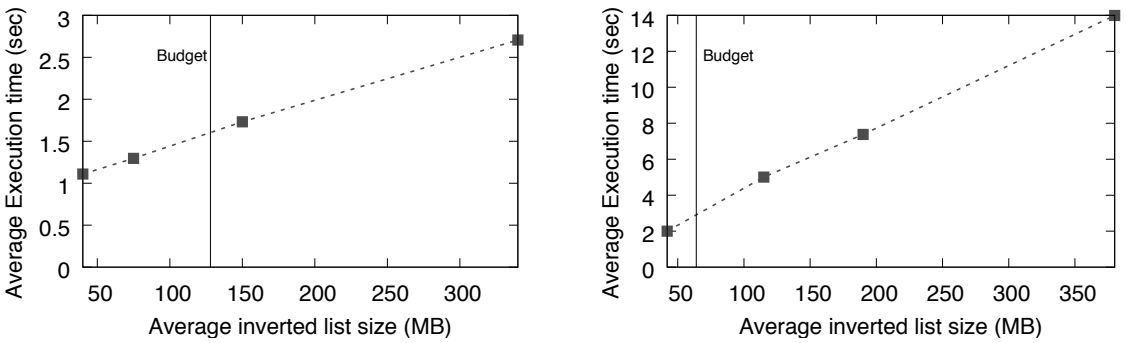
The average execution time trend for this experiment, shown in Figure 3.36(b), is different from Figure 3.36(a) (which showed the average execution time of hash group-by queries.). Although the hash join and hash group-by both utilize a hash table and data partition table, the trend for the same number of records is different because the 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-based 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.

3.7.3.4 Inverted-index Search Operator

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. When measuring the average execution time, we measured 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 operation, 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 we will describe shortly in Section 4.2.1. As a result, a large cardinality from an inverted-index search would cause the primary-index lookup to dominate the execution time of the overall query, shifting attention away from the operator of interest.

We first used query template Q10 in Figure 3.38 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. 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, 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

MB, 150 MB, 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 MB and 340 MB cases, the lists had to be divided into multiple chunks. Figure 3.37(a) shows the average execution time of an inverted-index search operator for query Q10 in Figure 3.38. We can see that the execution time was proportional to the average size of the inverted list and that the trend was linear.



(a) Two keywords. (b) Three keywords.

Figure 3.37: The average execution time of inverted-index search queries.

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 3.37(b) 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 were 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.

```
/* 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"});
```

Figure 3.38: Query templates to measure the average execution time of memory-intensive operations.

3.7.4 When Objects Get Large

So far, we have not included large string fields in our experiments. 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 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 in Table 4.2 to measure

the effect of having large string fields that result in different record size distributions.

3.7.4.1 Sort Operator

We used query template Q11 in Figure 3.41 to measure the average execution time of sort queries with large fields. 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 2,000 on the dataset Wisconsin-Norm-0, the sort process had to generate runs on disk.

Figure 3.39 shows the average execution time of sort queries with the 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 3.3, it takes more time to read all the records. In addition, as the percentage of large pages grows, we can see that the effective space utilization in the storage layer also drops (the second row of the table). 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. For these reasons, 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 lowest.

3.7.4.2 Hash Join Operator

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 Figure 3.41 to measure the average execution time of hash join queries with large fields. We again varied the cardinality and

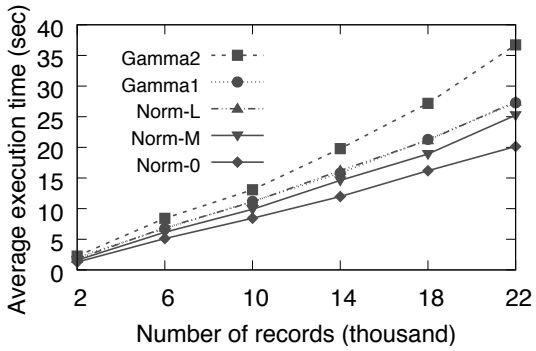


Figure 3.39: The average execution time of sort queries with large fields.

Table 3.3: Space utilization in the storage and during the runtime (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%

datasets as was done in the sort experiment. For the large field case, once the number of records was greater than 2,000 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 1,000 random records from the outer branch (probe side) and selected another random number of records from the inner branch (build side). Figure 3.40 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 3.39, and for similar reasons.

3.7.5 Query Access 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

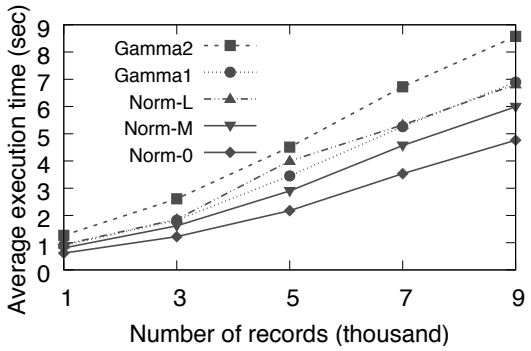


Figure 3.40: The average execution time of hash join queries with large fields.

```
/* 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;
```

Figure 3.41: Query templates to measure the average execution time of memory-intensive operations with large fields.

current stage-based admission control.

We used a variation of TPC-H query 3 for this experiment, shown later in Figure 3.47, 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. 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 that we have 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 3.42 shows the actual 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 the 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 jstat Linux command to sample the heap size of the AsterixDB JVM instance every two seconds. Since we allocate 6 GB to the AsterixDB JVM instance and we 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 is about 3 GB. Since this is a read-only query and the data is on disk, in-memory component region is not used. Thus, the maximum heap usage observed for each experiment should be lower than 5 GB (5,120 MB) if the system is behaving properly. We ran six workload-generator processes and each process sequentially issued five queries with a random predicate. Thus, thirty 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,

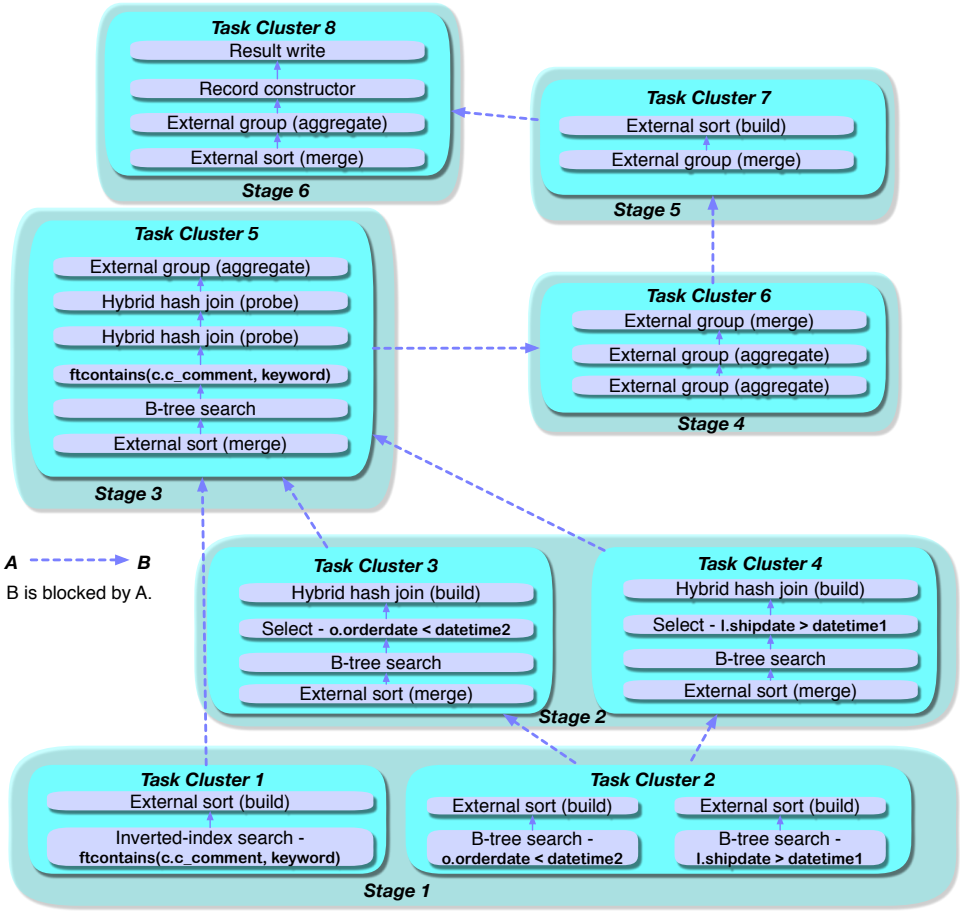


Figure 3.42: The runtime execution activity graph of the TPCH query 3.

we initially restarted the AsterixDB instance so that its memory usage always started from zero.

Before discussing the effect of the three query admission control policies, let us first examine the execution of a single query since this is the basic workload component. The memory usage of one query over time is shown in Figure 3.43. This query took about 1,000 seconds. (Thus, if we were to execute 30 such queries sequentially, it would take about 30,000 seconds to finish them in total.) 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 3.43. After this increase, we see a few zigzag patterns. These occur because when the execution of a task cluster finishes, an operator will release its resources if 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 pre-allocate 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.

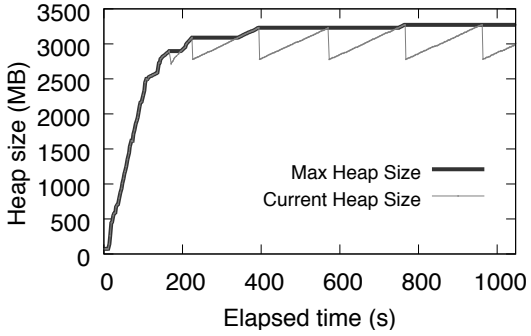


Figure 3.43: The heap size during an execution of one query.

3.7.5.1 Policy 1: No Query Admission Control

The first implementation of AsterixDB did not have any query admission control. Figure 3.44 shows the heap usage of six processes when there is no admission control in place. Thus, the first six queries were competing for getting the system resources. After 12,000 seconds of competing for getting resources, the system reached an OOM state and the AsterixDB instance became unstable. No query was able to finish its execution before the AsterixDB 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.

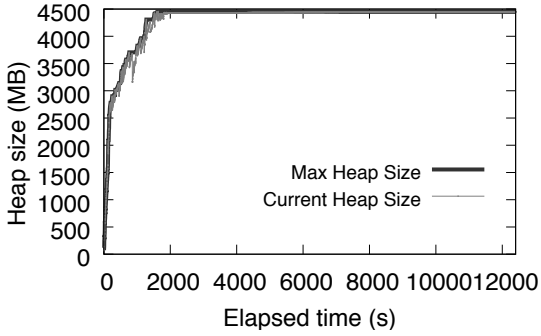


Figure 3.44: The heap size during an execution of multiple concurrent queries using no query admission control.

3.7.5.2 Policy 2: Conservative Query Admission Control

The first AsterixDB query admission control implementation assumed conservatively that all operators in the plan might execute at the same time. As we saw in Figure 3.42, 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 1,154 MB. Since about 2,400 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 3.42. However, the pattern is less clear than the pattern in Figure 3.43 since the execution of two concurrent queries whose starting times were different are overlapped. The heap usage was kept under 4,500 MB and it took about 25,000 seconds to finish the entire concurrent execution.

3.7.5.3 Policy 3: Stage Aware Query Admission Control

The current implementation of admission control in AsterixDB considers the execution stages of the plan and estimates the maximum memory usage per stage. Thus, each query requests

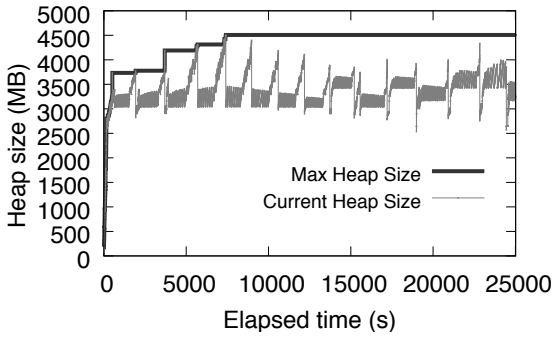


Figure 3.45: The heap size during an execution of multiple concurrent queries using the conservative query admission control.

at most 512MB 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 set to 128 MB. The available working memory was again about 2,400 MB. Here, four queries were allowed to execute at a given time. We can see in Figure 3.46 that the memory usage is 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 implementation. One major reason for this 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, there was only one physical storage partition, so 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.

3.8 Conclusion

In this Chapter, we have described the budget-driven approach to memory management in Apache AsterixDB, a parallel open source Big Data management system. We described how

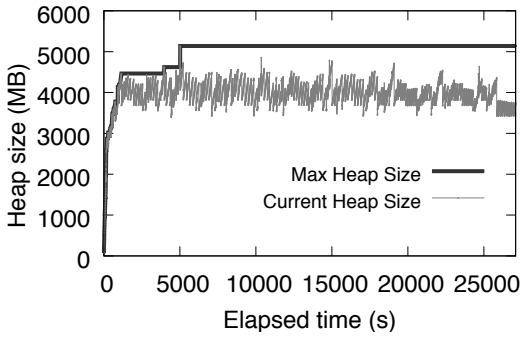


Figure 3.46: The heap size during an execution of multiple concurrent queries using the current query admission control.

```
/* 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;
```

Figure 3.47: A variation of TPC-H query no.3.

it divides memory into a few regions – in-memory components, the disk buffer cache, and working memory – and how it controls the memory usage of each region carefully. We then discussed how the system maintains a very carefully tracked budget in the context of its algorithms in order to keep the memory usage of its memory-intensive operators within a budget. Each memory-intensive operator's implementation requires careful attention regarding memory usage since a memory-intensive operator 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 original implementation of AsterixDB's memory-intensive operators and the memory details that they had overlooked. We then described how we modified these operators to truly operate within a budget. We

also discussed issues related to the global memory management in AsterixDB. Specifically, we discussed how it sizes in-memory components, implements query admission control, and incorporates large fields. We also presented experiments to empirically explore the 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 memory-intensive operators 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. Since the existing literature largely ignores these important details, we hope that future Big Data management system builders can benefit from our experiences.

Chapter 4

Index-only Query Plans in AsterixDB

4.1 Introduction

Most data management systems support various types of indexes to expedite their query execution. These systems utilize two types of indexes. A *primary index* either keeps a mapping between primary keys and records or stores the records themselves in it. A *secondary index* is an auxiliary structure built on a non-primary field. It maps from secondary keys to primary keys (or to record locations). If a search predicate is on the field, the system can utilize the secondary index to answer the query without performing a full scan of the records. This method can be efficient when the selectivity of the predicate is low. Its main limitation is that after performing the secondary-index search, the system needs to access the records to get the additional fields to generate the final results. For instance, the query in Figure 4.1 asks for three fields from a dataset called ds_tweet that contains a number of tweets. Suppose that userid is the primary key and that there is a secondary index on the username field. A secondary-index search on the username field can generate primary keys that satisfy the predicate. The system then needs to access the records to fetch the

additional fields needed to generate the final results. If the corresponding records are not physically contiguous, generating the final results may take more time than performing a full scan since each record lookup needs a random disk I/O. Sorting the primary keys first can help to reduce this cost [44], but there is still a full scan crossover at some point.

```
select t.user.userid, t.user.username, t.user.createat
from ds_tweet t where t.user.username like "Jerry%";
```

Figure 4.1: A query that can utilize a secondary index on the username field.

To solve this performance problem, most data management systems support *index-only query* plans that can generate the final results for some queries by only accessing a secondary index. For instance, if the query in Figure 4.1 asked to return only userid and username, the secondary index on the username field contains all the necessary data to answer this query. In this case, the system would not need to access the records to answer the query.

AsterixDB supports various types of secondary indexes including B+-tree, R-tree, and inverted indexes. In this chapter, we discuss how to support index-only plans in AsterixDB given the nature of its storage and transaction subsystems.

4.1.1 Related Work

Fetching declaratively-guided data using indexes was used in DBMS as early as System-R [27] and Ingres [51]. Since then, various types of indexes have been proposed. Most existing data management systems, such as DB2, Postgres, MySQL, and Oracle, support index-only plans. For example, DB2 implements a transactionally correct index-only plan by locking a data page or a record (or records) to guarantee the consistency between an index entry and the actual record in a data page [2]. In PostgreSQL, each heap page contains a visibility map bit to keep information about which of the records in a given page are visible to all active transactions [10]. If the visibility map bit is set, entries fetched from a secondary-index

search can be returned. Otherwise, the search process must check the heap page for each entry. After finding entries from a secondary-index search, a PostgreSQL search thus checks the visibility map bit for the corresponding data heap page(s) to verify whether the records in the data heap page are visible to the current search process [9]. MySQL reads a snapshot of the database and uses it in an index-only plan [7]. Oracle uses a multi-version consistency model to provide a snapshot of the database to an index-only plan [8].

4.2 Background

In this section, we discuss the current implementation of the relevant AsterixDB related to index-only query plans.

4.2.1 Index Search

A secondary-index search first traverses a secondary index to generate ⟨*secondary key, primary key*⟩ pairs that satisfy the predicate. AsterixDB then sorts these primary keys and looks in the primary index to fetch the actual records, retrieving all relevant fields including the secondary key field. It then verifies the search condition again using a select operator to generate the final results.

There are two reasons behind this process in AsterixDB. First, some types of secondary index searches may generate false positive results. For instance, a range-search condition on multiple fields on a composite B+-tree index may generate false positive results as we will explain shortly. Since an R-tree index stores a minimum bound rectangle (MBR) of a field value, not the field value itself, searching an R-tree may also generate false positive results. An inverted-index search may similarly generate false positive results when performing a similarity search. The second reason is to maintain concise (simplified) locking functionality.

AsterixDB only performs locking on the primary index. It does this in order to support a variety of secondary index structures in terms of concurrency control, since index-structure-specific aspects can then be ignored for locking [84]. However, with this scheme, there may be brief inconsistencies between the secondary index and the primary index, so the result of a secondary-index search is not "authoritative". Thus, AsterixDB needs to look up the primary index to generate a trustworthy result to be returned to the user.

If the selectivity of a query predicate is low, a secondary-index search outperforms a scan-based approach. As the number of results from the secondary index increases, however, the execution time also increases, and more primary keys are generated from the secondary index. To help mitigate this expense, once the secondary-index search generates ⟨*secondary key, primary key*⟩ pairs, AsterixDB sorts these primary keys and feeds them into the primary index to increase the buffer cache (I/O) cache performance. Since the resulting primary keys may not be on contiguous pages, AsterixDB performs a point lookup per primary key rather than performing a full scan.

4.2.2 Locking

During an index search, a lock is needed to ensure the consistency of a record as described in Section 4.1.1. Like most data management systems, AsterixDB supports two kinds of locks – shared (S) lock and exclusive (X) lock. An S lock is used when reading a record, and an X lock is used when inserting/updating/deleting a record. Regarding the duration of a lock, AsterixDB provides two lock types. The first one is an instant lock. As the name suggests, an instant lock holds a record for a very short amount of time. AsterixDB acquires a lock on a record and releases it instantly after fetching the record. The purpose of an instant lock is to see if locking a record in a given mode is currently possible and to hold a short-duration lock until it is being fetched. The second type is a non-instant lock. A transaction initially

holds and then releases such a lock only after its commit. AsterixDB also supports a third type, *try* lock, which is useful when a transaction wants to try to acquire a lock without being blocked if the request cannot currently be granted by the lock manager. Normally, when a transaction sends a lock request to the lock manager, the request is either granted by the lock manager or queued. A "try" lock first attempts to get a lock. If the request is granted, it holds the lock and returns true to the caller. If the request is queued, the locking request is disregarded and returns false without blocking the caller.

Figure 4.2 shows the possible combinations of locking methods in AsterixDB. For instance, if a transaction wants to request a shared lock for the normal duration, a non-instant-S-lock is created. If a transaction wants to request a shared lock instantly and does not want to wait if the request cannot be granted, an instant-try-S-lock can be created.

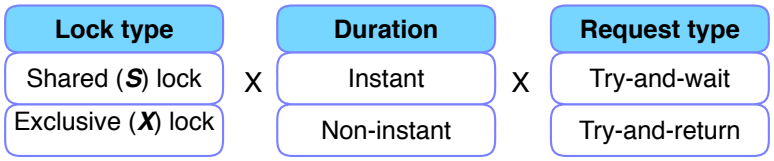


Figure 4.2: Combinations of locking methods.

A transaction that performs a primary index search without any modification requests an instant-S-lock for each primary key since it needs to get a shared lock before returning the current record. A transaction that performs insert/upsert/delete operations requests a non-instant-X-lock for each primary key since the lock should protect the record from being modified by other modification requests until the transaction commits. As described earlier, no lock will be placed for secondary indexes during modification/search operations.

4.3 Implementing Index-only Query Plans

In this section, we first discuss the required conditions of a scan-based query plan that can be correctly transformed into an index-only query plan. We then discuss two issues related to

the current secondary-index search that need to be addressed. We next present an index-only query plan and discuss how to perform the transformation.

4.3.1 Necessary Conditions

Not every secondary-index search plan can become an index-only plan since some plans may logically generate false positive results. Figure 4.3 shows how we can check if a query plan can be transformed into an index-only plan. AsterixDB first checks whether the plan can utilize a secondary index. If there is no secondary index on those fields in the predicates, AsterixDB simply performs a scan-based plan. If there are secondary indexes on the fields used in the predicates, AsterixDB checks the applicability of a corresponding index based on its type.

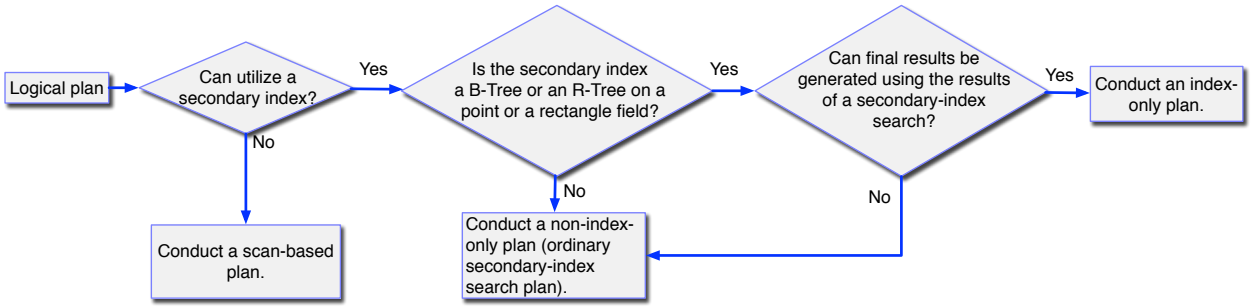


Figure 4.3: Index-only plan check.

1) B+-tree: If the index is a B+-tree, AsterixDB can apply the index-only plan optimization since a B+-tree search does not generate false positive results in general. However, there is one exception. A composite B+-tree can generate false positive results if both predicates contain range conditions. These false positive results can be removed by applying a SELECT operator after performing a secondary-index search in an index-only plan, as the composite B+-tree index contains the original values of both fields. For instance, the query in Figure 4.4 has two range predicates. Suppose there is a composite B+-tree on the user.create_at and user.posting_count fields. A secondary-index search in AsterixDB is currently done as a range

search that identifies the first ⟨*secondary key(s), primary key*⟩ entry and the last ⟨*secondary key(s), primary key*⟩ that satisfy the predicate range, and it returns all entries between them. As a result, the results from a secondary-index search can contain ⟨*user.create_at, user.posting_count, id*⟩ triples that may have false positives. For instance, in the query in Figure 4.4, if one of the user.create_at field values in the data is date("2017-03-31"), all values of user.posting_count associated with that value will be returned by the range scan.

```
select t.id, t.user.create_at, t.user.posting_count from ds_tweet t
where t.user.create_at >= date("2017-01-01") and t.user.create_at <= date("2017-12-31")
  and t.user.posting_count >= 10000 and t.user.posting_count < 99999;
```

Figure 4.4: A query that can utilize a composite B+-tree index on the user_create_at and user_posting_count fields.

2) R-tree: If the index is an R-tree, the plan can be transformed into an index-only plan only if the original field value can be reconstructed from the MBR in the index; otherwise, false positive results can be generated. For instance, the query in Figure 4.5 asks for all rectangles that overlap with a circle. When performing an R-tree search, the MBR of the circle will be used to search the R-tree on the field place.bounding_box. Since the circle's MBR can generate false positive results, they should be verified using the original circle. Since R-tree search results contain MBRs, if the original field cannot be generated, this verification cannot be done without looking at the original data. The only possible type of field that qualifies for an index-only plan is thus a point or a rectangle, since the R-tree index keeps the original point value and the MBR of a rectangle is equal to its shape.

```
select t.place.bounding_box from ds_tweet t
where spatial_intersect(t.place.bounding_box, circle(-126.76,54.44 7));
```

Figure 4.5: A query utilizing an R-tree index on the place_bounding_box field.

3) Inverted index: If the index is an inverted index, an index-only plan is not possible since an inverted index can have multiple secondary key entries per primary key. For example, suppose there is a full-text index on a field named reviewText and a user wants to execute

the full-text query in Figure 4.6. The query executes a full-text search that checks whether each instance of the reviewText field contains both "expected" and "more". If an instance of the field reviewText includes both keywords, two entries will be fetched for this field instance during the inverted-index search, namely ⟨expected, PK⟩ and ⟨more, PK⟩. Since an inverted-index search deals with one token plus its inverted list (primary keys) at a time, two entries from different tokens cannot be fetched at the same time. Thus, a secondary-index search will first fetch ⟨expected, PK⟩. When fetching the second pair ⟨more, PK⟩, the inverted-index search cannot transactionally guarantee that this entry refers to the same record version that was referred to the first primary key since the record may have changed during the gap between those two fetches.

```
select t.review_id, t.reviewText from ds_tweet t
where ftcontains(t.reviewText, ["expected","more"]);
```

Figure 4.6: A query utilizing an inverted (full-text) index on the reviewText field.

If the index type is potentially applicable, AsterixDB checks whether the index is a "covering index", i.e., whether it includes all of the fields in the predicate and all of the query's returned fields. If the index is not covering, AsterixDB still needs to access the primary index to fetch the relevant fields, so it cannot utilize an index-only query plan. Otherwise, an index-only plan can be generated.

4.3.2 Authoritative Secondary-index Search

As described above, the results from a secondary-index search need to be verified since the search is not transactionally authoritative. To make its search results trustworthy without re-verification, AsterixDB needs to ensure the consistency between the primary index and the secondary index during a secondary-index search. To ensure consistency, we can use a locking strategy similar to the primary-index search as explained below. Let us consider the simple query in Figure 4.7 that utilizes the primary index of the dataset ds_tweet. Here, id

is the primary key field. Since the predicate condition contains the id field, a primary-index search will be performed.

```
select t.id, t.user from ds_tweet t
where t.id > 1000;
```

Figure 4.7: A query that performs a range search on the primary index.

During the primary-index search, when AsterixDB fetches a primary key, it places an instant-S-lock on the key value as shown in Figure 4.8. Since it is not a try lock, the transaction waits until it gets a lock on the primary key. Once it gets the lock, there can be no concurrent modification access being made to this record. Thus, we can guarantee that the returned record is the latest as of the time when the lock request is granted.

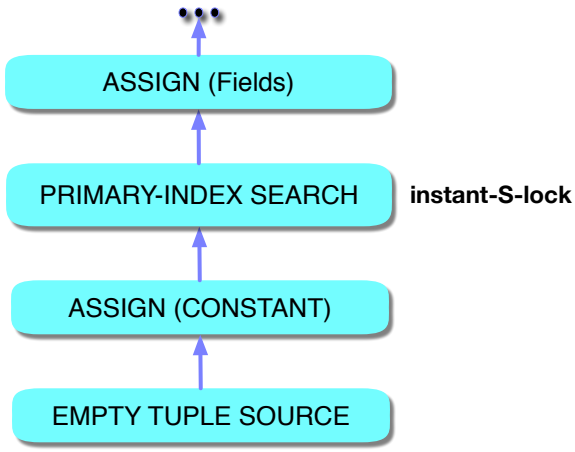


Figure 4.8: Primary-index search.

A similar locking strategy can be applied to a secondary-index search. During a secondary-index search, the transaction can also perform an instant-try-S-lock on the primary keys when seeing a ⟨secondary key, primary key⟩ pair from the secondary index. If this lock request is successful, then there was no ongoing concurrent modification being made on the record at the time of fetching the pair. In this case, the query can return the ⟨secondary key, primary key⟩ pair as part of the query result. If this instant-try-S-lock request fails, an ongoing concurrent modification was being made to the primary index. That is, the record

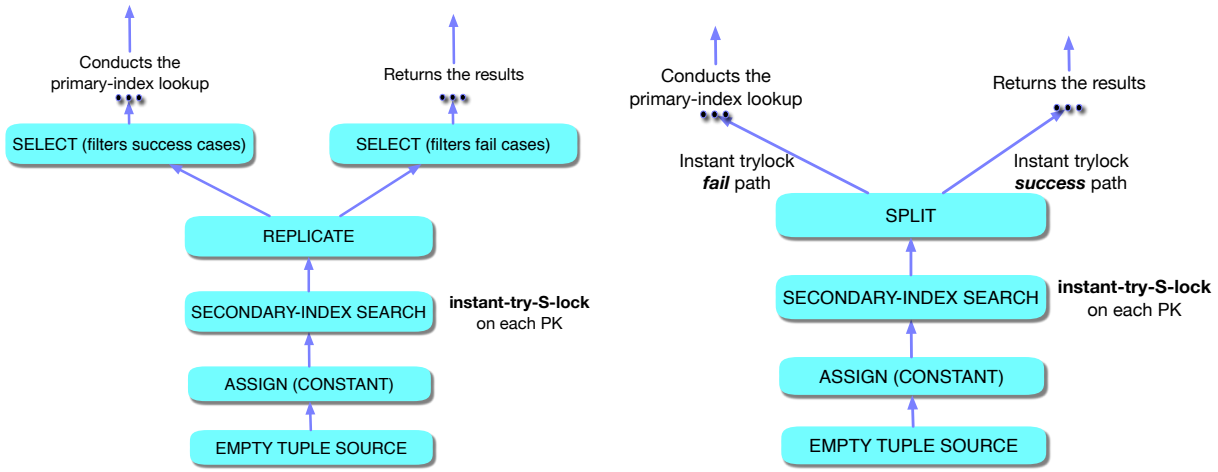
referred to by the primary key could have been deleted or the indexed field value could have been updated. In either case, the transaction can simply fetch the record from the primary index to verify the condition for the entry ⟨secondary key, primary key⟩ whose lock request was not granted. We have chosen this fallback approach for the following reasons. Suppose we want to resolve the instant-try-S-lock request failure case using the secondary index alone. A new secondary-index search with the same secondary key and primary key would need to be performed to make sure that the entry pair still exists. In addition, after seeing this pair again, a new lock request would need to be sent to the lock manager to ensure that there was no ongoing concurrent modification request and the current transaction would be unable to progress until this entry pair is processed. Instead, falling back to verification in this (hopefully rare) case seems simpler.

4.3.3 Implementing an Index-only Plan

As was just described, the secondary-index search can generate a trustworthy result by performing an instant-try-S-lock on each primary key. The result of this instant-try-S-lock is either false or true. If this lock request fails, the transaction still needs to perform a primary-index lookup on the primary key. If this lock request succeeds, however, the transaction can simply return the ⟨secondary key, primary key⟩ pair as the final result. Therefore, we need to create a logical query plan that has two paths after the secondary-index search based on the instant-try-S-lock result.

SPLIT operator: AsterixDB has a REPLICATE operator that can replicate the results of an operator to more than one output. Although the REPLICATE operator can propagate each result along multiple paths, each output receives the same sequence of ⟨secondary key, primary key⟩ pairs. In our case, however, AsterixDB also needs to filter the ⟨secondary key, primary key⟩ pairs in each search output path. One of the plan's output paths is for the

instant-try-S-lock failed case. In this path, AsterixDB needs to remove primary keys whose instant-try-S-lock request was successful, as it is not necessary to perform the primary-index lookup on these primary keys. In the other path, AsterixDB needs to remove the primary keys whose instant-try-S-lock request was not successful, as these primary keys still need to be checked in the primary index. Figure 4.9(a) illustrates this idea.



(a) Using REPLICATE operator.

(b) Using SPLIT operator.

Figure 4.9: Comparison between REPLICATE and SPLIT operators.

Using the REPLICATE operator is not as efficient as possible. Thus, we developed a new operator called SPLIT that can have multiple outputs with the additional functionality of sending a record to only one output. The SPLIT operator checks the value of an expression on each record and guides it to only one output. Figure 4.9(b) shows the revised plan using the SPLIT operator. We can see that the SELECT operators in the two paths are no longer needed.

Combining two output paths: The SPLIT operator makes it possible to propagate the primary keys from the secondary-index search to the appropriate paths. For the instant-try-S-lock success path, AsterixDB passes the ⟨secondary key, primary key⟩ pairs to the next operator directly. For the instant-try-S-lock failed path, it performs the primary-index lookup on each primary key and retrieves the latest record. It then fetches the secondary

key field and verifies the condition again. If the condition holds, that record can be passed to the next operator.

One issue with this approach, if done naively, is that the original path after the introduction of the SELECT operator should be copied to both paths because of the SPLIT operator. If this secondary-index search is the starting point of a complex query plan, then almost every operator in the plan would need to be duplicated. This approach is clearly not efficient. Instead, re-combining the two paths after the verification in the instant-try-S-lock fail path will be more efficient than creating two separate (largely duplicated) paths. AsterixDB has an operator called UNIONALL that can integrate two input paths into one output, as shown in Figure 4.10, and we can exploit that operator here for index-only plans.

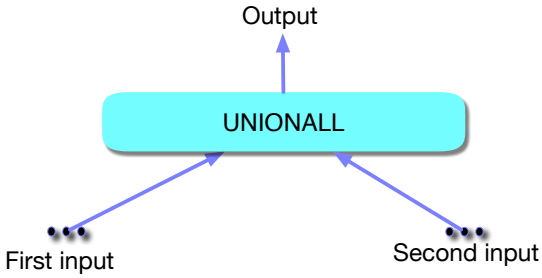


Figure 4.10: UNIONALL operator.

Using the SPLIT operator as the dividing point and the UNIONALL operator as the merging point, an index-only query plan now has a diamond shape as shown in Figure 4.11. The resulting plan can generate trustworthy results from a secondary-index search. The transactional consistency issue between the secondary-index search and the primary index is now resolved. Notice the PROJECT operators before the plan's UNIONALL operator; these are placed there to keep the field order of the ⟨secondary key, primary key⟩ pairs in the plan consistent. For instance, the order of the pairs in the left path will be ⟨primary key, secondary key⟩, as a primary-index search generates the primary key first and a secondary key field will be assigned. The order of the two keys in the right path will be ⟨secondary key, primary key⟩, as each pair from a secondary-index search generates them in this order.

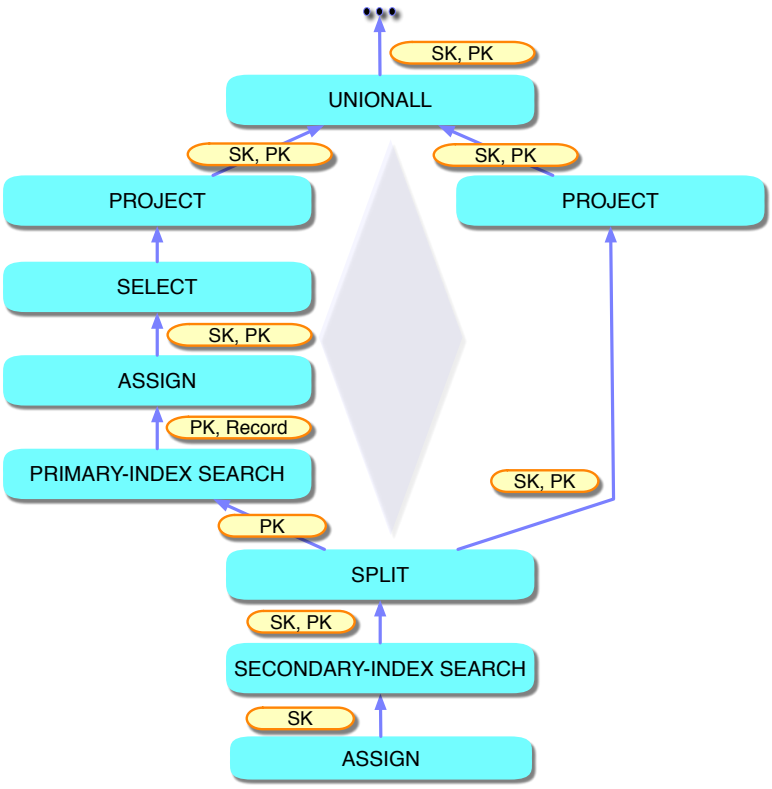


Figure 4.11: Index-only plan.

4.3.4 Rewriting Scan-based Plans into Index-only Plans

AsterixDB uses an extensive rule-based approach to query optimization [22]. An initial logical plan is constructed from a given query, and each optimization rule is tried on this plan. If a rule is applicable, the plan is transformed. A logical plan involving a dataset access always starts with a PRIMARY-INDEX-SCAN operator, followed by a SELECT operator if there are selection conditions. For queries that can utilize a secondary index, a non-index scan-based query plan is constructed first, and an index-based transformation then occurs during the optimization phase.

Figure 4.12 shows how a scan-based query is optimized to use an index. Figure 4.12(a) shows the original scan-based plan, and Figure 4.12(b) shows an index-utilization plan. Figure 4.12(c) shows an index-only plan. The index-utilization plan and index-only plan are

chosen based on the conditions described in Section 4.3.1.

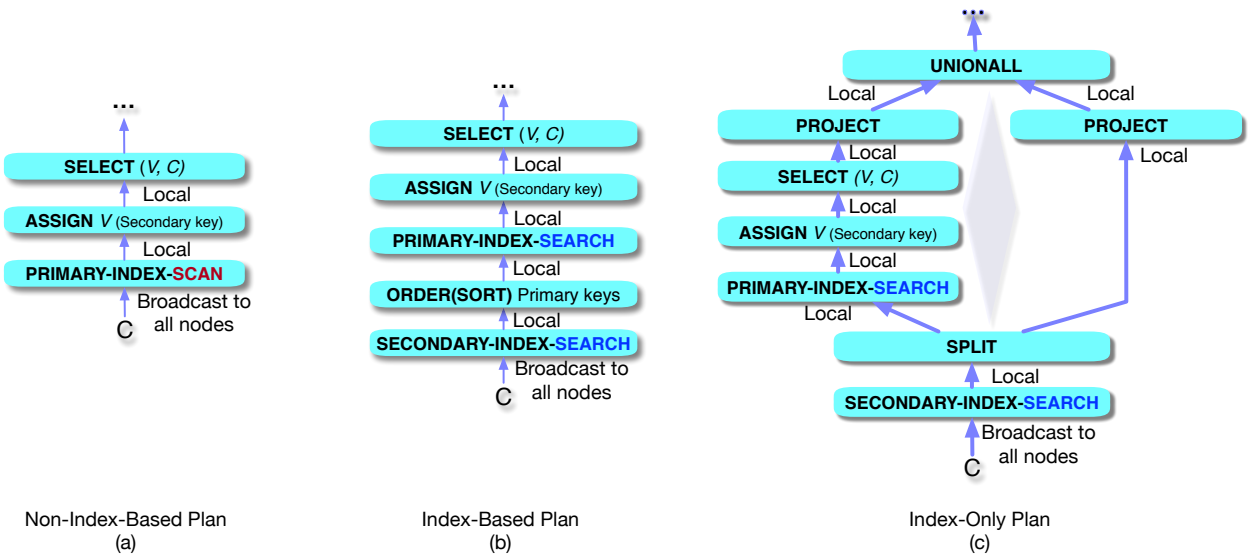


Figure 4.12: Rewriting a scan-based plan to an index-only plan.

Based on a SELECT operator with a condition, the optimizer tries to replace PRIMARY-INDEX-SCAN with a secondary-index-based search plan. To rewrite a query, the optimizer first matches an operator pattern consisting of a pipeline with a SELECT operator and a PRIMARY-INDEX-SCAN operator. Next, it analyzes the condition of the SELECT operator to determine whether the non-constant argument originates from the PRIMARY-INDEX-SCAN operator and whether the corresponding dataset has a secondary index on a field variable V. For each secondary index on V, the optimizer checks whether it is applicable based on the predicate. For example, an R-tree can be utilized for spatial queries that use the spatial_intersect function. The optimizer then checks whether the given index is a covering index and the necessary conditions described in Section 4.3.1 for index-only plans can be satisfied. If so, the plan can be transformed into an index-only query plan. If not, the plan can be transformed into a secondary-index utilization plan.

For a secondary-index utilization plan transformation, the optimizer builds a SECONDARY-INDEX-SEARCH operator whose parameter is the predicate used in the original SELECT operator. It then adds a SORT operator to sort the primary keys that are fetched from the

secondary-index search, and it adds a PRIMARY-INDEX-SEARCH operator to fetch the records. An ASSIGN operator is then placed to access the original secondary key field and the SELECT operator is placed to verify the original predicate.

For an index-only plan transformation, the optimizer builds a SECONDARY-INDEX-SEARCH operator as before. In addition, it instructs the operator to perform an instant-try-S-lock for each primary key that it fetches. It then places a SPLIT operator and builds the primary-index search path for the instant-try-S-lock failure case. For the instant-try-S-lock success case, if filtering false positive results are required, such as range searches in a composite index, it places a SELECT operator after the SPLIT operator. Once both paths are constructed, the optimizer creates a UNIONALL operator to merge the two paths.

4.4 Experiments

To quantify the impact of our approach to index-only plans, we performed an experimental evaluation of our implementation using a real dataset. The experimental setup was the same as Section 3.7. We used a single-node cluster to host an 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 and fine-tuned cluster environment. This node ran CentOS 6.9 with a Quadcore AMD Opteron CPU 2212 HE (2.0GHz), 8GB RAM, 1 GB Ethernet NIC, and had two 7,200 RPM SATA hard drives. We used one hard drive with one physical storage partition. Table 5.3 shows the AsterixDB configuration parameters.

4.4.1 Dataset

We used a real dataset collected from Twitter [93] utilizing their public streaming API. It had 1% of US tweets for 19 days in January 2017. It had spatial and temporal fields suitable

Table 4.1: AsterixDB parameters for the experiments.

Parameter	Value	Parameter	Value
Total memory allocated	6 GB	Sort memory	128 MB
In-memory-component size	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
Run-time page size	32 KB	Storage page size	32 KB

for B+-tree and R-tree indexes. We used the DDL in Figure 4.13 to create the AsterixDB dataset. Table 4.2 shows its characteristics. (Its size was smaller than the original JSON file since the field name did not have to be included in each stored record. In contrast, for each record in the JSON file, the field name is repeated.)

Table 4.2: AsterixDB Dataset.

Name	Cardinality	Raw data size (MB)	Dataset size in DB (MB)	Notes
ds_tweet	29,430,000	31,756	20,472	20 days of tweets

4.4.2 Index

We used the DDL in Figure 4.14 to create three B+-tree indexes for the B+-tree index-only plan experiments and two R-tree indexes for the spatial experiments. Table 5.6 shows the information about each index.

We can see that the index sizes of a B+-tree were around 4% of the original Twitter dataset. Thus, reading a B+-tree secondary index required fewer disk I/Os. The sizes of the R-tree indexes were about 7% of the original dataset size. The order of this field was the same as the primary key field id. Thus, the B+-tree index on the create_at field was used for a clustered single-field experiment. The B+-tree index on the user.screen_name field was used for an experiment with an unclustered single field since the order of this field and the order

```
create dataverse twitter;
use twitter;

create type typeUser if not exists as open {
    id: int64,  name: string,
    screen_name: string,  profile_image_url: string,
    lang: string,  location: string,
    create_at: date,  description: string,
    followers_count: int32,  friends_count: int32,
    status_count: int64
};

create type typePlace if not exists as open{
    country: string,  country_code: string,
    full_name: string,  id: string,
    name: string,  place_type: string,
    bounding_box: rectangle
};

create type typeGeoTag if not exists as open {
    stateID: int32,  stateName: string,
    countyID: int32,  countyName: string,
    cityID: int32?,  cityName: string?
};

create type typeTweet if not exists as open {
    create_at: datetime,  id: int64,
    text: string,  in_reply_to_status: int64,
    in_reply_to_user: int64,  favorite_count: int64,
    coordinate: point?,  retweet_count: int64,
    lang: string,  is_retweet: boolean,
    hashtags: {{ string }} ?,  user_mentions: {{ int64 }} ? ,
    user: typeUser,  place: typePlace?,
    geo_tag: typeGeoTag
};

create dataset ds_tweet(typeTweet) primary key id;
```

Figure 4.13: The DDL statements for the ds_tweet dataset.

```
use twitter;

create index create_at_idx on ds_tweet(create_at) type btree;
create index screen_name_idx on ds_tweet(user.screen_name) type btree;
create index user_composite_idx on ds_tweet(user.create_at,user.status_count) type btree;
create index coordinate_idx on ds_tweet(coordinate) type rtree;
create index place_bounding_box_idx on ds_tweet(place.bounding_box) type rtree;
```

Figure 4.14: The DDL statements for creating indexes on the ds_tweet dataset.

of the primary key field were unrelated. For an experiment with a composite B+-tree index, we built a composite index on the user.create_at and user.status_count fields. The order of these fields was also not related to the primary key field. For the spatial experiments, we built two R-tree indexes. One was on the coordinate field of type point, and its size was

small because only 14.6% of the records actually contained coordinate field values. We also built an R-tree index on place.bounding_box of type rectangle. All of the records had this field value, so this R-tree index was larger.

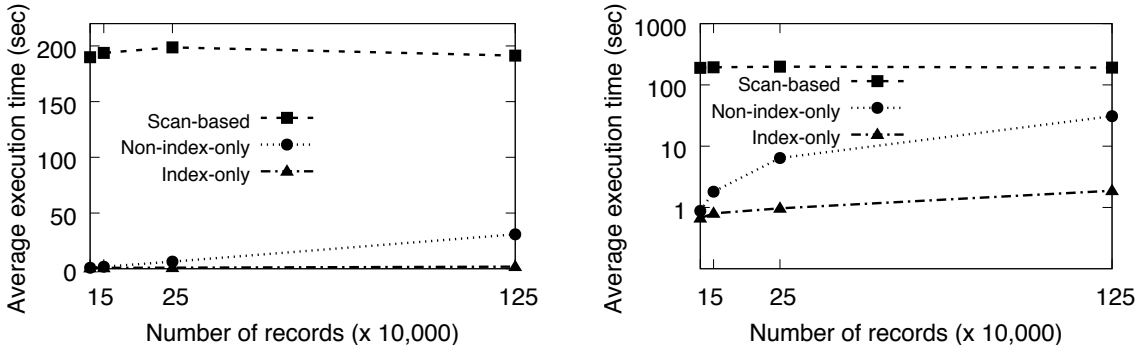
Table 4.3: Index size.

Field	Index Type	Size (MB)
Dataset itself	B+ tree (primary)	20,472
create_at	B+ tree (secondary)	648
user.screen_name	B+ tree (secondary)	793
user.create_at, user.status_count	B+ tree (secondary)	788
coordinate	R tree (point, secondary)	133
place.bounding_box	R tree (rectangle, secondary)	1,409

4.4.3 B+-tree: Single Field

Clustered field: We used a B+-tree on a single field create_at for this experiment. As described above, the order of this field and the order of the primary key field were correlated, making this index clustered relative to the data record's order. We used query Q1 in Figure 4.22 at the end of this chapter. This query returns the count of fields, which contains the secondary key field. We varied the number of returned records using a range condition and measured the average execution time of 30 queries with four selectivities, namely 10,000, 50,000, 250,000, and 1,250,000 records. We compared the average execution time of three types of access methods – index-only plan, non-index-only plan, and scan-based plan. As Figure 4.15 shows, the average execution time of an index-only plan was lowest, as expected. Furthermore, we can see that its average execution time did not vary much, as the secondary-index search alone generated the final results, and since searching the secondary index was a range search, not a point lookup. The average execution time of a non-index-only plan was greater than that of the index-only plan. The reason is that the system had to sort the primary keys, fetch the records from the primary index, assign the secondary key field, and verify the condition again. As the number of output records increased, we can see a

clear increase in the average execution time because of this reason. The average execution time of the scan-based plan was by far the largest. Each time, AsterixDB had to scan the entire dataset of about 20 GB and apply the predicate for each record. However, the average execution time did not vary much.

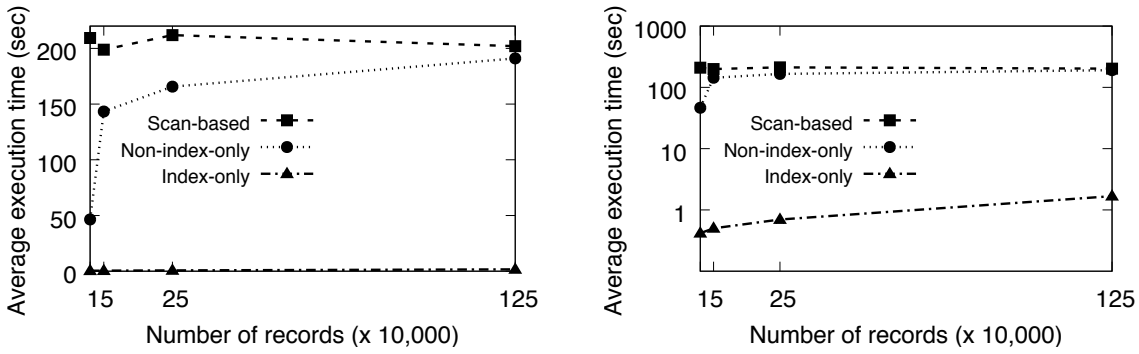


(a) Y-axis in linear scale.

(b) Y-axis in log scale.

Figure 4.15: Average execution time of a clustered-single-field query.

Unclustered field: We used a B-tree on a single field user.screen_name for this next experiment. As described above, the order of this field and the order of the primary key field were not correlated, making this index unclustered. Again, we used the same selectivities and measured the average execution time of 30 queries. We used query Q2 in Figure 4.22; it returns the count of fields, which contained the secondary key field. In Figure 4.16, we can see that the average execution times of the index-only plan and the scan-based query were similar to the average execution times of Q1. However, the average execution time of the non-index-only plan is different here. The average execution time of an unclustered-index query for 10,000 records was already 60 seconds due to point lookups on the primary index. Also, between 10,000 and 50,000 records, we can see a sharp increase in the average execution time; this is because the number of primary pages needing to be read increased sharply. However, after this point, the average execution time gradually increases as the selectivity increases. The reason is that the number of pages needing to be read cannot increase sharply after this point since the number of pages in the primary index was fixed.



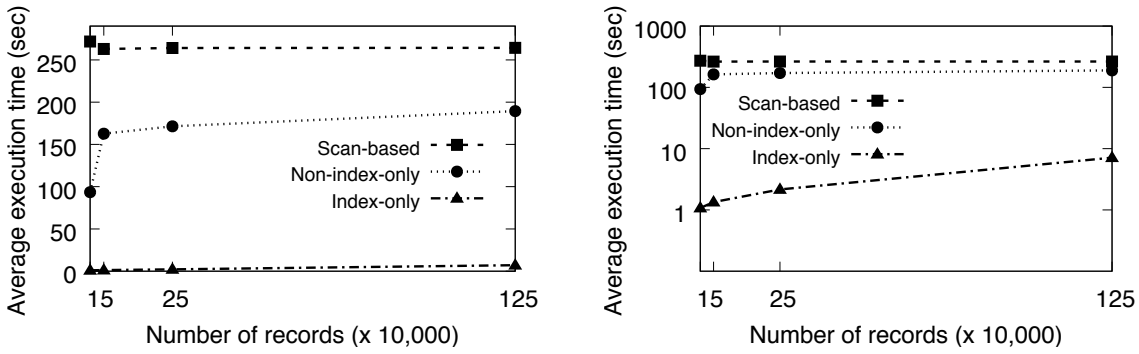
(a) Y-axis in linear scale.

(b) Y-axis in log scale.

Figure 4.16: Average execution time of an unclustered-single-field query.

4.4.4 B+-tree: Multiple Fields

We used a B-tree on two fields user.create_at and user.status_count for this experiment. As described above, the order of these fields and the order of the primary key field were not correlated, making this index an unclustered composite index. Again, we used the same selectivities and measured the average execution time of 30 queries. We used query Q3 in Figure 4.22, which returns the count of fields containing the secondary key fields. In Figure 4.17, we can see that the average execution time of the index-only plan and the scan-based query were similar to that of Q2. The difference is that we can see a slight increase in the average execution time of the index-only plan, as the potential logical false positive results are also being filtered on the instant-try-S-lock success path. Also, the average execution time of the non-index-only and the scan-based plan were greater than that of the single-field cases since the application of a predicate that contains two fields takes more time. For instance, the average execution time of the non-index-only composite plan for 10,000 records was 100 seconds. In contrast, the execution time for the unclustered-index single-field case was lower, at 60 seconds.



(a) Y-axis in linear scale.

(b) Y-axis in log scale.

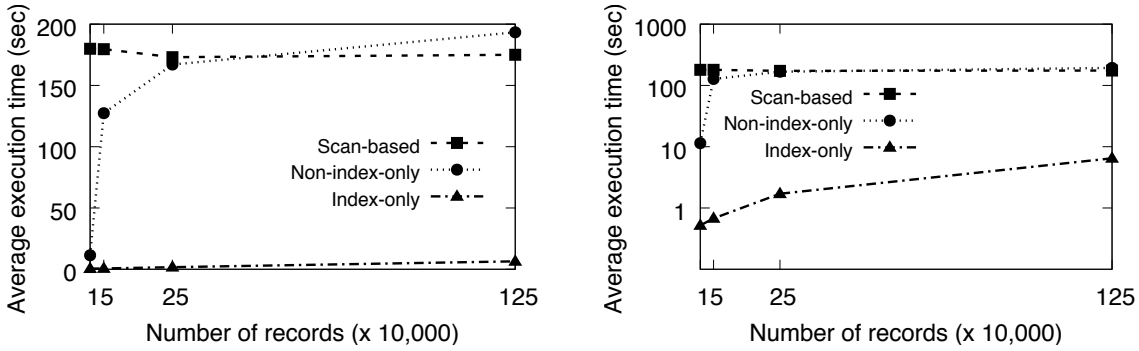
Figure 4.17: Average execution time of a query using an unclustered two-field index.

4.4.5 R-tree: Point Field

For this experiment, we used an R-tree index on the coordinate point field. Like the B+-tree experiment, we varied the selectivity and measured the average execution time of 30 queries. We created 1,000 random rectangles that satisfied the condition. The size and location of each rectangle were different. Each query randomly chose one rectangle as its predicate. We used the same selectivities and measured the average execution time of 30 queries. We used query Q4 in Figure 4.22 that returned the count of the secondary key field. Note that since the query shape was a rectangle, the index-search operator did not need to remove false positive results on the right path (instant-try-lock success path). The result is shown in Figure 4.18.

As the number of instances that had the coordinate field value was only 4.3 million, which was 14.6% of the entire collection of records, the average execution time of the scan-based method was the lowest among all scan-based queries in the experiments. This is because about 85% of records were skipped without checking the predicate. Also, after 250,000 records, the average execution time of the non-index-only plan became greater than that of the scan-based approach. The reason is that after this point, scanning entire primary pages took noticeably less time than randomly accessing the primary pages one by one.

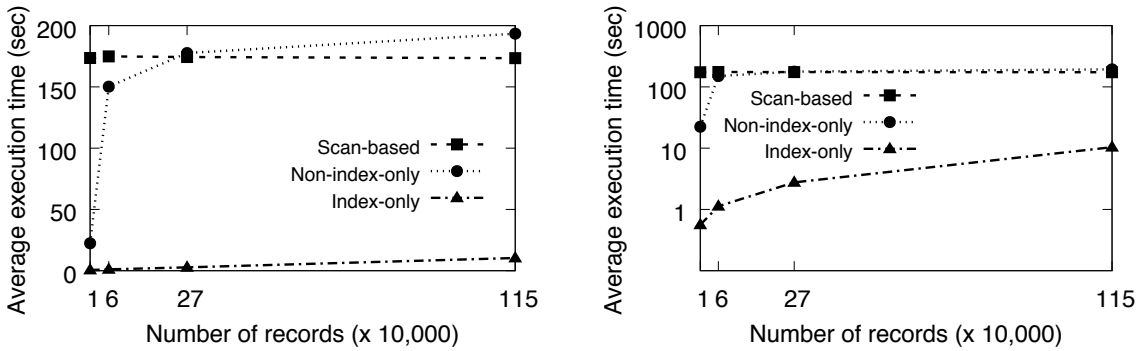
Next, we created 1,000 random circles whose area was the same as the corresponding rectangle in our 1,000 random rectangles. The center of each circle was the centroid of the corresponding rectangle and the radius was chosen to generate the same area as the rectangle. (Due to the different shapes and the data, the resulting selectivity was slightly different.) Figure 4.21 shows these results. As in the rectangle query case, the average execution time of the non-index-only plan became greater than that of the scan-based plan after 270,000 records, and for the same reason. Although false positive results were filtered on the right path (instant-try-lock success path) of the index-only plan, the average execution time of the index-only plan did not increase by much. In addition, we can see that the overall trends of two queries in Figures 4.18 and 4.19 were very similar.



(a) Y-axis in linear scale.

(b) Y-axis in log scale.

Figure 4.18: Average execution time of spatial queries (query shape: rectangle) on a point field.



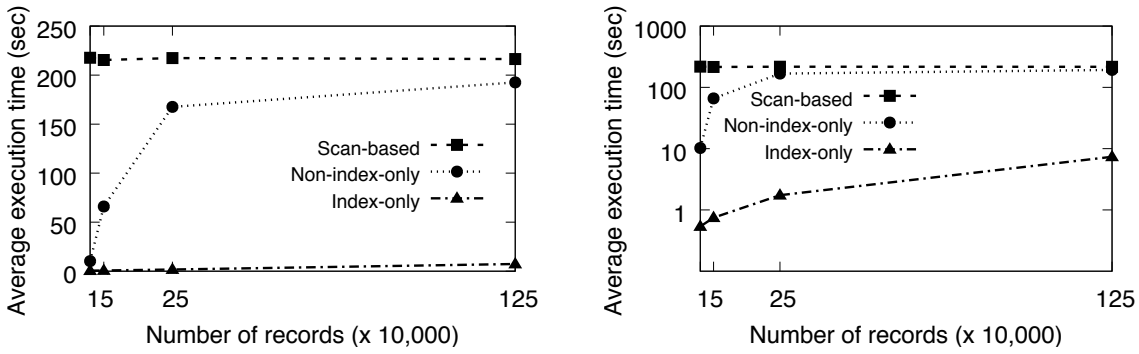
(a) Y-axis in linear scale.

(b) Y-axis in log scale.

Figure 4.19: Average execution time of spatial queries (query shape: circle) on a point field.

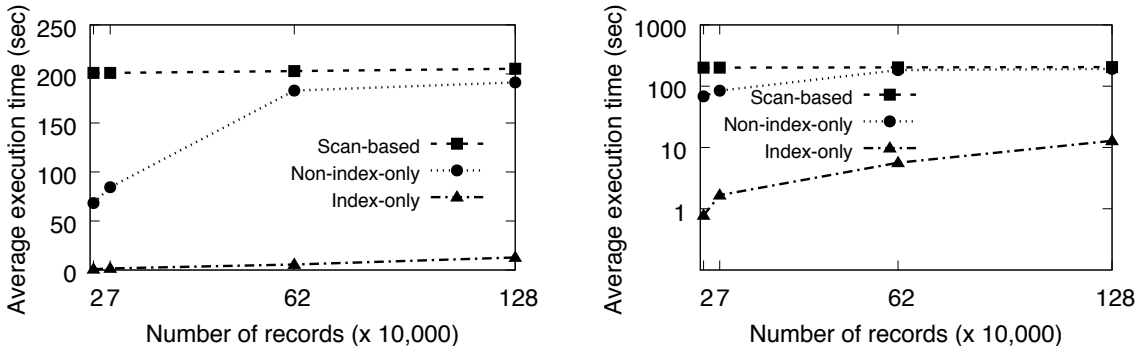
4.4.6 R-tree: Rectangle Field

For this last experiment, we used an R-tree index on the place.bounding_box rectangle field. We created 1,000 random rectangles that satisfied the same selectivity as the B+-tree experiments. The size and location of each rectangle were different. Since the distribution of rectangles and points were different, these random rectangles were different from the rectangles used for the point-field experiment. Each query randomly chose one rectangle as its predicate. We varied the selectivity and measured the average execution time of 30 queries. Note that since the query shape was a rectangle, the index-search operator did not need to remove any false positive results on the right path (instant-try-lock success path). The result is shown in Figure 4.20. We can see that the average execution time of the index-only plan was slightly higher than that of the B+-tree experiments since the rectangle field value needed to be generated again from its MBR in the instant-try-S-lock success path. The pattern of the average execution time of the non-index-only plan case was similar to that of the B+-tree experiments for the same reason. Next, we created 1,000 random circles whose area was the same as the corresponding rectangle in 1,000 random rectangles. The center of each circle was the centroid of the corresponding rectangle, and the radius was chosen to generate the same area as the rectangle. Thus, the resulting selectivity was different; the results of this experiment are shown in Figure 4.21. Because of the logical false positive records now being filtered on the instant-try-S-lock success path, we can see that the average execution time for the circle's index-only plan was greater than that of the index-only plan on the query that used a rectangle as the predicate. We can see that the overall trends of two queries in Figure 4.20 and 4.21 were similar, however.



(a) Y-axis in linear scale. (b) Y-axis in log scale.

Figure 4.20: Average execution time of spatial queries (query shape: rectangle) on a rectangle field.



(a) Y-axis in linear scale. (b) Y-axis in log scale.

Figure 4.21: Average execution time of spatial queries (query shape: circle) on a rectangle field.

4.5 Conclusions

In this chapter, we have described the index-only query plan implementation in AsterixDB. We discussed a few key transactional challenges that needed to be addressed in this context. We then described the required conditions for an index-only plan and we described how to create an index-only plan. We also showed how to transform a scan-based plan into an index-based plan and into an index-only plan during the query optimization process. We concluded with an experimental study on a real dataset using both spatial and temporal indexes. The study showed that the average execution performance of index-only queries

```
/* Q1. B+-tree: single field (clustered) */
select value count(first.create_at) from (
  select t.create_at, t.id from ds_tweet t
  where t.create_at  >= [datetime1] and t.create_at  < [datetime2]
) first;

/* Q2. B+-tree: single field (unclustered) */
select value count(first.screen_name) from (
  select t.user.screen_name, t.id from ds_tweet t
  where t.user.screen_name  >= [screen_name1] and t.user.screen_name  < [screen_name2]
) first;

/* Q3. B+-tree: multiples fields (unclustered) */
select value count(first.create_at) from (
  select t.user.create_at, t.user.status_count, t.id from ds_tweet t
  where t.user.create_at  >= [datetime1] and t.user.create_at  < [datetime2]
        and t.user.statues_count >= 0 and t.user.status_count < 1000000
) first;

/* Q4. R-tree: point field - query shape: rectangle */
select value count(first.coordinate) from (
  select t.coordinate, t.id from ds_tweet t
  where spatial_intersect(t.coordinate,
     create_rectangle(create_point(X1,Y1),create_point(X2,Y2)))
) first;

/* Q5. R-tree: point field - query shape: circle */
select value count(first.coordinate) from (
  select t.coordinate, t.id from ds_tweet t
  where spatial_intersect(t.coordinate,
     create_circle(create_point(X,Y), [radius])
) first;

/* Q6. R-tree: rectangle field - query shape: rectangle */
select value count(first.bounding_box) from (
  select t.place.bounding_box, t.id from ds_tweet2 t
  where spatial_intersect(t.place.bounding_box,
     create_rectangle(create_point(X1,Y1),create_point(X2,Y2)))
) first;

/* Q7. R-tree: rectangle field - query shape: circle */
select value count(first.bounding_box) from (
  select t.place.bounding_box, t.id from ds_tweet2 t
  where spatial_intersect(t.place.bounding_box,
     create_circle(create_point(X,Y), [radius])
) first;
```

Figure 4.22: Query templates used for the experiments.

was several orders of magnitude faster than scan-based and index-based queries.

Chapter 5

Performance Evaluation of Similarity Query Processing in AsterixDB

5.1 Introduction

Similarity queries compute answers satisfying matching conditions that are not exact but approximate. The problem of supporting similarity queries has become increasingly important in many applications, including search, record linkage [29], data cleaning [78], and social media analysis [21]. For instance, during a live phone conversation with a client, a call center representative might wish to immediately identify a product purchased by the customer by typing in a serial number. The system should locate the product even in the presence of typos in the search number. A social media analyst might want to find user accounts that share common hobbies or social friends. A medical researcher may want to search for papers with a title similar to a particular article. In each of these examples, the query includes a matching condition with a similarity function that is domain specific, such as edit distance for a keyword or Jaccard for sets of hobbies.

There are two basic types of similarity queries. One is *search*, or *selection*, which finds records similar to a given record. The other is *join*, which computes pairs of records that are similar to each other. There have been many studies on these two types of queries, both with and without indexes. A plethora of data structures, partitioning schemes, and algorithms have been developed to support similarity queries efficiently on large datasets. When the computation is beyond the limit of a single computer, there are also parallel solutions that support queries across multiple nodes in a cluster. (See Section 5.1.1 for an overview.) The techniques developed in the last two decades have significantly improved the performance of similarity queries and have enabled applications to support such queries on millions or even billions of records.

Most existing work has taken an algorithmic approach, focusing on the development of index structures and/or algorithmic optimizations. Our approach is different and systems-oriented – tackling the problem of supporting similarity queries *end-to-end* in a full, declarative parallel data management system setting. Here we explain how Apache AsterixDB, an open-source parallel data management system for semi-structured (NoSQL) data, supports such queries. By "end-to-end", we mean the whole lifecycle of a query, including the language support for similarity conditions, internal index structures, execution plans with or without an index, plan rewriting to optimize execution, and so on.

Achieving the end-to-end goal has involved several challenges. First, as similarity in queries can be domain specific, we need to support commonly used similarity functions and let users provide their own customized functions. Second, due to the complex logic of existing algorithms, we need to consider how to support them using existing parallel database operators without "reinventing the wheel" (i.e., without introducing new, ad hoc operators). Third, we need to consider how to leverage an existing query optimization framework to rewrite similarity queries to achieve high performance.

In earlier work, we extended the existing query language of AsterixDB to allow users to

specify a similarity query, either by using a system-provided function or specifying their own logic as a user-defined function. We also implemented state-of-the-art techniques using the existing operators in AsterixDB, both for index-based and non-index-based plans and for both search and join queries. Our solution not only allows query plans to benefit from the built-in optimizations in those operators but also to automatically enjoy future improvements in these operators. We revised the existing rule-based optimization framework to rewrite similarity queries. A plan for an ad hoc similarity join can be very complex. As an example, a three-stage-join plan based on the technique in [94] can involve up to 77 operators. To enable the optimizer to more easily transform such complex plans, we developed a novel framework called "AQL+" that takes a two-step approach to rewrite a plan. A major advantage of the framework is that it allows AsterixDB to support queries with more than one similarity join condition, making it the first parallel data management system (to our knowledge) to support similarity queries with multiple similarity joins.

This new optimization framework contains the AQL+ language, a super set of the AQL language with a few key extensions. As a result, any changes made over time to AQL should be also reflected in AQL+. As time progressed and AsterixDB evolved, we found that changes and improvements in the AQL language were often not properly reflected in AQL+ since they used two different language definition files, two different parsers, and two different translators. In short, AQL+ had a divergence issue. In addition, the AQL+ framework transforms a similarity join plan into a new three-stage-join plan by incorporating information from the incoming logical plan and an AQL+ similarity query template and compiling the resulting AQL query. As this transformation does not happen right at the beginning of the optimization phase, some of the query optimizations must be done again for this new logical plan since it is compiled again during the optimization. Note that this re-application process is not necessary for non-similarity queries since the plan generated for a non-similarity query has not been touched by the AQL+ framework. Therefore, for efficiency, the optimizer needs to ensure that the similarity-join rule set is only applied to

similarity-join queries. To apply some optimizations only to a new plan that is transformed using the AQL+ framework, we need to create a new rule set controller since the application of a rule set on a plan is controlled by a rule set controller and AQL+ framework is applied in a rule.

In this chapter, in addition to presenting the details of earlier work, we discuss how we addressed the AQL+ divergence issue and how we implemented a new rule set controller for the similarity queries that are transformed by the AQL+ framework. Also, we present an empirical study using several large, real data sets on a parallel cluster to evaluate the effects of these techniques. We also present results from comparative experiments with two other systems to explore the relative efficacy of AsterixDB's support for parallel similarity queries on large data sets. (Section 5.6).

5.1.1 Related Work

There are various kinds of similarity queries on strings and sets. Many algorithms (e.g., [61, 83, 19]) use a gram-based approach for for string similarity search. VGRAM [62] extends the approach by introducing variable-length grams. To optimize string similarity joins, filtering techniques are widely used. Length filtering uses the length of a string to reduce the number of candidates. An example algorithm is gram-count [45]. Prefix filtering [18, 99, 24, 82, 65, 98, 95, 77, 96, 34] utilizes the fact that two strings can be similar only if they share some commonality in their prefixes. Based on this fact, many algorithms have been proposed, such as AllPair [18], PPJoin/PPJoin+ [99], ED-Join [98], MPJoin [82], QChunk [77], VChunk [96], and Pivotal prefix [34]. Other related algorithms have been proposed such as M-Tree [30] and trie-Join [39].

There have been several evaluation studies of string/similarity [56] and set-similarity joins [66]. There is also a recent survey about string similarity queries [69]. The authors of [56] found

that AdaptJoin [95] and PPJoin+ [99] were best for Jaccard similarity. Meanwhile, the authors of [66] concluded that AllPair [18] was still competitive. The authors of [69] discussed prefix-filtering techniques. Many of these algorithms assume that the data to be searched or joined fits into main memory.

For parallel similarity joins, a number of studies have used the MapReduce framework [94, 88, 68, 97, 35]. There is one survey that discussed parallel similarity joins [37]. Vernica et al. [94] proposed a three-stage algorithm in such a setting. There are also studies on integrating similarity joins into database management systems [45, 46, 28, 86, 87]. Some of these adopted the similarity join as a UDF or expressed a similarity join in a SQL expression; others have introduced a relational operator to support similarity joins.

Our focus is different, as it is about supporting similarity queries in a general-purpose parallel database system. We need to address various systems-oriented challenges when adopting existing techniques in this context. A parallel similarity query processing system called Dima [89] was proposed recently. Dima is an in-memory-based system, unlike AsterixDB. There are some search systems and DBMSs that support similarity queries, including Elasticsearch, Oracle, and Couchbase. Unlike AsterixDB, Elasticsearch is middleware and it focuses on search, not join. Oracle supports edit distance via an extension package if a specific type of index is created. Couchbase supports edit distance searches on NoSQL data in its new full-text search service, but only via a separate full-text API (not its N1QL query language). In contrast, AsterixDB provides a general class of similarity functions for strings that work for both select and join operations, and a similarity predicate can be part of a general declarative query along with non-similarity predicates.

5.2 Preliminaries

5.2.1 Similarity Functions

A similarity measure is used to represent the degree of similarity between two objects. An object can be a string or a bag of elements. There are various types of similarity measures available depending on the objects that are being compared. In this chapter, we focus on two widely used classes of measures, namely string-similarity functions and set-similarity functions.

String-Similarity Functions: One widely used string similarity function is edit distance, also known as Levenshtein distance. The edit distance between two strings r and s is the minimum number of single-character operations (insertion, deletion, and substitution) required to transform r to s. For instance, the edit distance between "james" and "jamie" is 2, because the former can be transformed to the latter by inserting "i" after "m" and deleting "s". There are other string-similarity functions such as Hamming distance and Jaro-winkler distance.

Set-Similarity Functions: These are used to represent the similarity between two sets. There are many such functions, such as Jaccard, dice, and cosine. In this chapter, we focus on Jaccard similarity, which is one of the most common set-similarity measures. For two sets r and s, their Jaccard similarity is $Jaccard(r, s) = \frac{|r \cap s|}{|r \cup s|}$. For example, the Jaccard similarity between $r = \{$"Good", "Product", "Value"$\}$ and $s = \{$"Nice", "Product"$\}$ is $\frac{1}{4}$. Such set-similarity functions can also be utilized to measure the similarity between two strings by tokenizing them (i.e., into n-grams or words) and measuring the set similarity of their token multisets. Dice and cosine values can be calculated similarly.

Similarity Search: Similarity search finds all objects in a collection that are similar to a given object based on a given similarity metric. Let sim be a similarity function, and δ

be a similarity threshold. An object r from a collection R is similar to a query object q if $sim(r, q) \geq \delta$.

Similarity Join: Joins find similar $\langle r, s \rangle$ pairs of objects from two collections R and S, where $r \in R$, $s \in S$, and $sim(r, s) \geq \delta$.

5.2.2 Answering Similarity Queries

For similarity queries, using a brute-force, scan-based algorithm is computationally expensive, so there have been many studies in the literature on how to support similarity queries more efficiently. One widely used method is the gram-based approach, which utilizes the n-grams of a string. An n-gram of a string r is a substring of r with length n. For instance, the 2-grams of string "james" are {"ja", "am", "me", "es"}.

review_id	username	review_summary
1	james	This movie touched my heart!
2	mary	The best car charger I ever
3	mario	Different than my usual but good
4	jamie	Great Product - Fantastic Gift
5	maria	Better ever than I expected

Figure 5.1: Example data of Amazon Review dataset (simplified).

String-similarity queries can be answered by utilizing an n-gram inverted index. For each gram g of the strings in a collection R, there is an inverted list l_g of the ids of the strings that include this gram g. Figure 5.2 shows the inverted lists for the 2-grams of the `username` field of the little sample Amazon Review dataset in Figure 5.1.

gram	am	ar	es	ia	ie	io	ja	ma	me	mi	ri	ry
inverted list	1 4	2 3 5	1	5	4	3	1 4	2 3 5	1	4	3 5	2

Figure 5.2: Inverted lists for 2-grams of the `username` field.

We can answer a string-similarity query by computing the n-grams of the query string and retrieving the inverted lists of these grams. We then process the inverted lists to find all

string ids that occur at least T times, since a string r within edit distance k of another string s must share at least $T = |G(r)| - k \times n$ grams with s [57]. This problem is called the *T-occurrence problem*. Solving the T-occurrence problem yields a set of candidate string ids. The false positives are then removed in a final verification step by fetching the strings of the candidate string ids and computing their real similarities to the query. As an example, given a gram length $n = 2$, an edit distance threshold $k = 1$, and a query string $q =$ "marla", Figure 5.3 illustrates how to find the similar usernames from the data in Figure 5.1. We first compute the 2-grams of q as {"ma", "ar", "rl", "la"} and retrieve the inverted lists of these 2-grams. We consider the records that appear at least $T = 4 - 2 \times 1 = 2$ times on these lists as candidates, which have review_ids 2, 3, and 5. Last, we compute the real similarity for these candidates, and the review_id 5 is the final answer. Note that if the threshold $T \leq 0$, then the entire data collection needs to be scanned to compute the results; this is called a *corner case*. In the above example, if the threshold is 3, then $T = 4 - 2 \times 3 = -2$, causing a corner case.

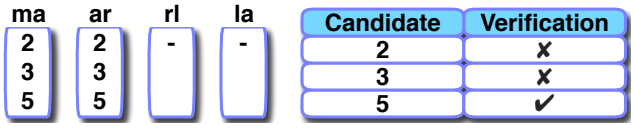


Figure 5.3: Answering an edit-distance query for "q"=marla and T=2.

5.3 Using Similarity Queries

In this section, we discuss the similarity measures supported in AsterixDB and show how users can express similarity queries in SQL++. We also show how users can specify indexes to expedite query processing. Please note that this is earlier work.

5.3.1 Supported Similarity Measures

AsterixDB currently supports two built-in similarity measures, Jaccard and edit distance, to solve set-similarity and string queries. Both measures can be processed with or without indexes. Let us focus on edit distance first. This measure can be calculated on two strings. As an extension in AsterixDB, edit distance can also be computed between two arrays of scalar values. For example, the edit distance between ["Better", "than", "I", "expected"] and ["Better", "than", "expected"] is 1. This generalization is possible since a character in a text string can be viewed as an element in an array if we think of the string as a collection of ordered characters.

The other supported measure, the Jaccard value, can be computed on two arrays or multisets of elements. If a field type is string, a user can use a tokenization function to first make an array of elements from the string. For example, it is possible to calculate the Jaccard similarity between two strings by tokenizing each string into an array of words.

If a user wishes to use their own similarity measure, they can create a user-defined function (UDF). A UDF can be expressed in SQL++ or AQL (the two query languages supported by AsterixDB) or implemented as an external Java class. If the desired UDF can be expressed in SQL++ or AQL, the user can create such a function using the following syntax and use it like a native function.

```
create function similarity-cosine(x, y) {
 ......
}
```

5.3.2 Expressing Similarity Queries

AsterixDB provides two ways to express a similarity query in a SQL++ or AQL query, both illustrated by the example SQL++ queries in Figure 5.4. These equivalent queries find the record pairs from the Amazon review dataset that have similar summaries. In Figure 5.4(a) before the actual query, the similarity function and threshold are defined with set statements. The query then uses a similarity operator ~=, which is syntactic sugar defined for similarity functions. This similarity operator computes the similarity between its two operands according to the simfunction and simthreshold and returns the records that are similar. The same query can also be written without using the similarity operator. The similarity query in Figure 5.4(b) uses a Jaccard function named similarity_jaccard(), and this query is equivalent to that in Figure 5.4(a). The first syntax can be easier to use because the simfunction and simthreshold also have the default settings and a user is not required to provide the two set statements. In addition, the user does not need to remember the exact function name with that syntax. Also, if the user wants to change the similarity function, they only need to change the set statements without changing the query itself. During query parsing and compilation, it is easy for the optimizer to replace this syntactic sugar and generate a desired optimized plan. On the other hand, the second form gives the user more direct control. There are a few variations of similarity functions in AsterixDB, e.g., one that can do early termination during the evaluation. A user can freely choose any of them.

5.3.3 Using Indexes

Without an index, AsterixDB scans the whole dataset in the query to compute the result for the given query. To expedite query execution, AsterixDB supports two kinds of inverted indexes to support the two similarity measures efficiently.

The first index type, called keyword index, uses the elements of a given multiset as keys and

```
use TextStore;
set simfunction 'jaccard';
set simthreshold '0.5';
select element {"summary1":t1, "summary2": t2}
from AmazonReview t1, AmazonReview t2
where word_tokens(t1.summary) ~= word_tokens(t2.summary);
```

(a) ~= Notation

```
use TextStore;
select element {"summary1":t1, "summary2": t2}
from AmazonReview t1, AmazonReview t2
where similarity_jaccard(word_tokens(t1.summary),
                         word_tokens(t2.summary)) >= 0.5;
```

(b) Function Notation

Figure 5.4: SQL++ join on the summary field of the Amazon review dataset using Jaccard similarity.

maps those keys to their corresponding primary ids. For example, it is possible to tokenize a string and use each token as a key. This index is suitable for Jaccard similarity. The two queries in Figure 5.4 could utilize a keyword index on the summary field. A keyword index can be created using the following DDL statement, where summaryidx is the index name:

```
create index summaryidx on AmazonReview(summary) type keyword;
```

The second index type is called the n-gram index and is suitable for edit distance. An n-gram index uses the extracted n-grams of a string as the keys and maps those keys to their corresponding primary ids. For example, we can use the following DDL statement to create a 2-gram index on the reviewerName field:

```
create index reviewernameidx on AmazonReview(reviewerName) type ngram(2);
```

5.4 Executing Similarity Queries

In this section, we explain how similarity queries are internally executed in AsterixDB. We first describe the execution flow for a similarity query in the presence of an index and then

describe the execution flow in the case where no index is available. Please note that this is earlier work.

5.4.1 Inverted Index

An inverted index in AsterixDB is an LSM-based secondary index that consists of a mutable in-memory component and multiple immutable on-disk components, as illustrated in Figure 5.5. This design choice was made to support high-frequency insertions, as LSM-based indexes amortize the cost of writes by consolidating updates in memory before writing them to disk [14]. The in-memory component consists of two B^+-trees, one for the in-memory inverted index and one to store the primary keys of deleted records. On-disk components are immutable, so AsterixDB denotes the deleted records of the on-disk components using this B^+-tree instead of deleting them from the inverted index itself. This design choice also implies that primary keys obtained from the inverted index may have already been deleted, so they need to be verified by checking their existence in the deleted-key B^+-tree. An in-memory index component grows with inserted/deleted records until the memory budget allocated for the component is exhausted. It is then flushed to disk as a new immutable on-disk component. The multiple index components on disk must be searched besides the in-memory component during a given index-search operation. To mitigate this, AsterixDB periodically merges on-disk components based on a configurable merge policy.

To improve its search performance, AsterixDB employs a length-based technique to partition the inverted index. This technique is useful since it allows the use of length filtering prior to a search, which eliminates records that are not similar based on the length required for the similarity threshold of a query. We use the number of tokens in the given field as the length. Figure 5.6(a) shows the details of an in-memory inverted index. The secondary key field is first tokenized based on the type of the index (n-gram or keyword), and each token is inserted

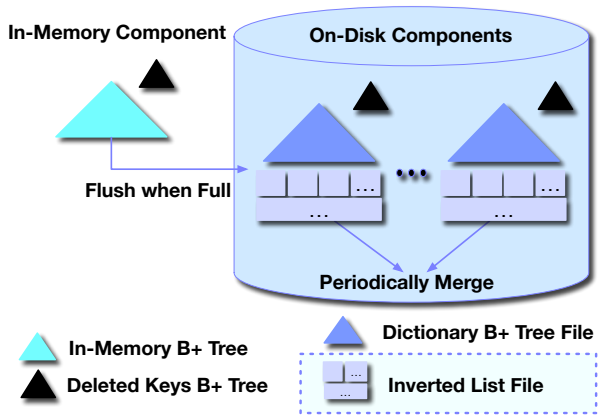
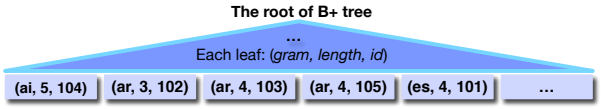
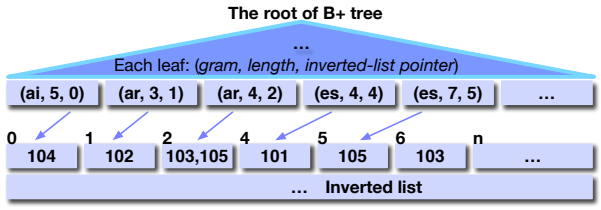


Figure 5.5: The structure of an inverted index.

into the in-memory inverted index along with the length of the secondary key field and the primary key of the record. The in-memory index component is a B$^+$-tree with keys consisting of triples. Each triple contains $\langle token,\ length,\ primary\ key \rangle$. For example, in Figure 5.6(a), the leftmost entry is $\langle ai,\ 5,\ 104 \rangle$. Its secondary key token is ai and the number of tokens for the given field is 5. The triple also tells us that this entry comes from the record whose primary key is 104. Once the in-memory index is flushed to disk, it becomes immutable, and it is finalized by merging the primary keys with the same token and length into a sorted list in the inverted list file and using the resulting $\langle token,\ length,\ inverted\ list\ pointer \rangle$ triples as B$^+$-tree keys as shown in Figure 5.6(b). The pointer there indicates the starting offset of the associated list of primary keys for the given *token* and *length* pair.



(a) In-memory inverted index.



(b) On-disk inverted index.

Figure 5.6: An example instance of an n-gram inverted LSM index.

5.4.2 Executing Similarity Selections

We first present the execution strategy that AsterixDB uses for selection queries. We use the example query in Figure 5.7 to explain the execution flow; this SQL++ query computes the edit distance between a field title of a dataset called Reddit and a constant search key good competitions where the edit-distance-threshold is 2.

```
select r.id, r.title
from Reddit as r
where edit_distance(r.title, "good competitions") < 2;
```

Figure 5.7: A similarity-selection query.

5.4.2.1 Index-Based Search Execution

When running the above query on a cluster with multiple nodes, the query coordinator (*a.k.a.* cluster controller) sends a request containing the constant search key (C) to each participating node (*a.k.a.* node controller). Figure 5.8 illustrates how such a similarity-selection query is executed using a secondary inverted index on a 3-node cluster. In the figure, C is good competitions and V refers to the title field in Figure 5.7. Each cluster node contains a partitioned primary index and a local inverted index. That is, the contents of each inverted index are generated from the local primary index. Thus, the nodes do not need to communicate with each other to execute a selection query.

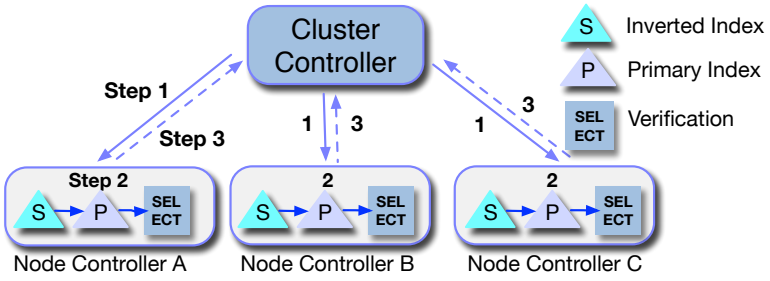


Figure 5.8: Parallel execution of a similarity-selection query.

If an index is available, AsterixDB runs an index-based selection plan at each node. It first gives the constant value (C) to the secondary inverted index. The secondary-inverted-index search generates $\langle Secondary\ Key, Primary\ Key \rangle$ pairs that satisfy the T-occurrence condition, which may include false positives. It then looks up these primary keys in the primary index to fetch their corresponding records. The primary keys are sorted prior to this lookup to increase the chance of page cache hits in the buffer. After fetching the actual field value from the primary index, a SELECT operator is applied to remove false positives and generate the final results. If the similarity condition is selective enough, such an index-based search plan can be much more efficient than the non-index-based plan that uses DATASET-SCAN and SELECT operators in the absence of an index. Once the local results are generated at each node, they are sent to the coordinator to be combined into the final query result.

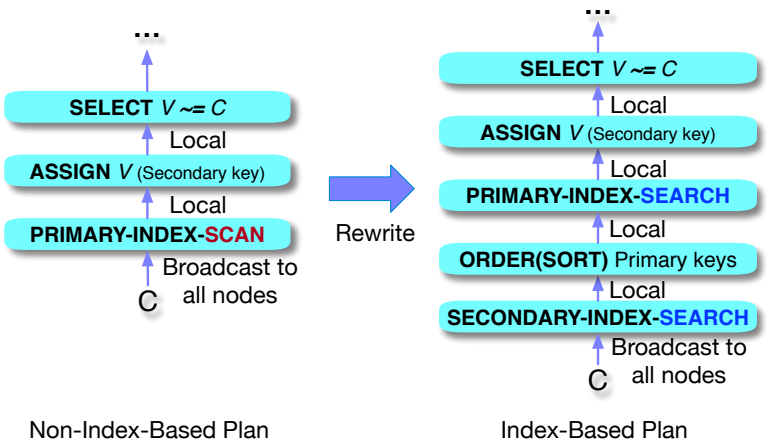


Figure 5.9: Similarity-selection query plans

To process a similarity-selection query, the SQL++ compiler first generates a simple non-index-based selection plan (the left part of Figure 5.9) from a user query. The optimizer then transforms the initial plan into an index-based selection plan if there is an applicable index during the logical optimization process. We will discuss this rewriting process further in Section 5.5.1.

5.4.2.2 Non-Index-Based Search Execution

Similar to index-based execution, when there are multiple nodes, the coordinator sends a request containing the search key C to all the nodes. At each node, as there is no index on the field in the given similarity condition, AsterixDB scans the primary index, fetches all records, and verifies the similarity condition on the given field for each record. The left part of Figure 5.9 depicts this process. Finally, the results will be returned to the coordinator.

5.4.3 Executing Similarity Joins

A similarity join operator has two branches as its input. We call the first one the outer branch and the second one the inner branch. For example, in Figure 5.10, the SQL++ alias *ar* refers to the outer branch and *r* refers to the inner branch. This query fetches three fields from each dataset based on a Jaccard join condition with a threshold of 0.5.

```
select ar.reviewer_id, ar.id, ar.summary,
       r.author_id , r.id, r.title
from AmazonReview ar, Reddit r
where similarity_jaccard(ar.summary, r.title) > 0.5;
```

Figure 5.10: A similarity-join query.

5.4.3.1 Index-Based Join Execution

Similar to the similarity-selection case, where the search predicate value was broadcast to all nodes, in the similarity-join case, the records coming from the outer join branch of each node are broadcast to all nodes. Figure 5.11 depicts how a similarity-join query is executed using a secondary inverted index on a 3-node cluster.

The coordinator first sends the query execution request to each participating node. Each node of an outer-branch partition scans its portion of the outer-branch data. While doing

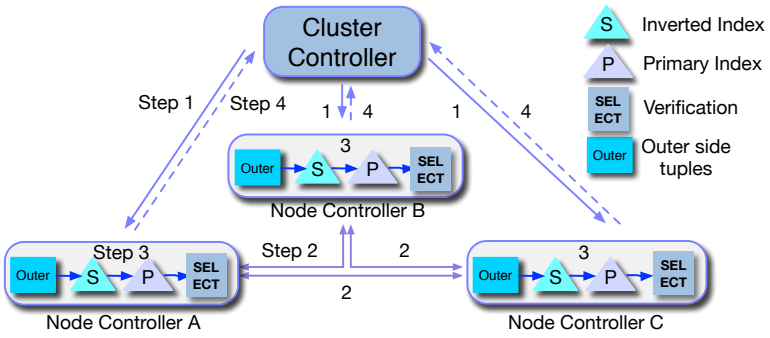


Figure 5.11: Parallel execution of a similarity-join query.

so, it broadcasts the resulting records to all nodes with a partition of the inner branch's dataset. This replicates all records of the outer-branch on each node, which then perform a secondary-index search. Each node with an index-side partition uses the incoming outer-branch records (as well as its local ones) to search its local inverted index. Once each secondary-index partition has processed all the records from the outer branch, the resulting primary keys from the search will be fed into the inner dataset's primary index and a primary-index search will be conducted. As discussed before, these primary search keys are sorted before the primary-index search to increase the chance of page cache hits. As before, we need to remove false positives from the index-based subplan using a SELECT operator based on the original similarity condition, which is taken from the JOIN operator. The right side of Figure 5.12 depicts this process. Finally, the results are sent to the coordinator to be combined.

5.4.3.2 Non-Index-Based Join Execution

When there is no index, a simple nested-loop join could be performed for a similarity join query. The outer branch would feed the predicate from each record to the inner branch. The complexity of this solution would be quadratic. To avoid such a costly nested-loop join, we instead adopt a three-stage-similarity-join algorithm [94] that we review here for ad hoc similarity join processing in AsterixDB.

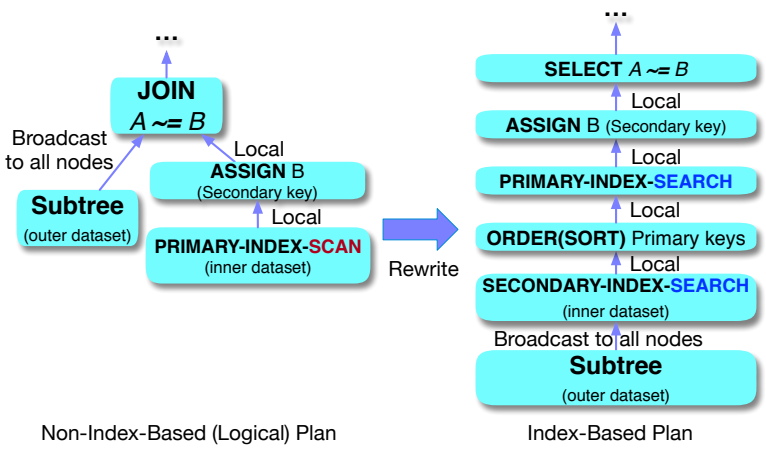


Figure 5.12: A similarity-join query plan.

The three-stage algorithm uses a prefix-filtering method, so a global token order needs to be computed to generate a prefix for each field value. This global token order can be arbitrary; we choose the increasing token-frequency order, which tends to generate fewer candidate pairs [94]. The first stage computes this global token order by counting the frequency of each token in the tokenized data and sorting the tokens based on their frequencies. In the second stage, the algorithm computes a short prefix subset for each set based on the global token order produced in the first stage. The record id and (only) the join attribute of each record are then replicated and repartitioned by hashing on these prefix tokens. After the repartitioning step, candidate pairs are generated by grouping the pairs by their ids, and the similarity is computed for each pair to filter out the dissimilar ones. This stage produces only similar record id pairs. Finally, the third stage of the algorithm rescans the inputs to fetch the rest of the query's desired record fields for these id pairs.

To apply this three-stage algorithm in AsterixDB, rather than implementing new query operators and complex query plans, we chose to describe the algorithm by using existing AQL constructs such as for, let, group by, and order by since this approach would be potentially more extendable in the future. In addition, if/as we improve AsterixDB's existing operators, we would not need to modify the AQL description to utilize the improved operators. For

example, if a new sort algorithm becomes available for the sort that generates a global token order, its benefit will be applied without any alteration of the AQL. Figure 5.13 shows an AQL query that captures the three stages for a self-similarity join on the summary field of the Amazon Review dataset using Jaccard similarity with a threshold. Note how each step is implemented using basic AQL constructs and functions. We now discuss the details of these three stages.

Stage 1: *Token Ordering* is expressed in lines 11-18 of Figure 5.13. In this subquery, we iterate over the records in the dataset. For each record, we retrieve the tokens in the summary field and count the number of occurrences of each token using a group-by clause. To expedite this calculation, we use a compiler hint in line 15 that suggests using hash-based aggregation instead of the default sort-based aggregation for the group-by statement since the order of tokens at this particular step is not meaningful. Finally, we order the tokens based on their count using an order-by clause. The same subquery is repeated later, in lines 30-37, in the context of the second dataset. During optimization, the optimizer will detect this common subquery and execute the subquery only once by using a replicate operator to send the results to both outer plans.

Stage 2: *Record ID (RID)-Pair Generation* is expressed in lines 5-50. We scan the dataset in line 6 and then retrieve each token from the summary field. We order the tokens by the rank computed in the first stage (lines 12-23) by joining the set of tokens in one summary with the set of ranked tokens. We use a hint in line 19 that advises the compiler to use a broadcast join operator to broadcast the ranked-tokens. Next, we order the join results by rank, stored in the variable $\$i$. We then extract the prefix tokens in line 22 and use the prefix-len-jaccard() built-in function to compute the length of the prefix for Jaccard similarity with a threshold of 0.5. The built-in subset-collection() function extracts the prefix subset of the tokens. The same process of tokenizing, ordering the tokens, and extracting the prefix tokens is done in lines 25-42 for the second dataset. We then join the two streams on their

```
1 // -- - Stage 3 - --
2 for $ARevLeft in dataset AmazonReview
3 for $ARevRight in dataset AmazonReview
4 for $ridpair in
5  // -- - Stage 2 - --
6  for $ARevLeft in dataset AmazonReview
7  let $lenLeft := len($ARevLeft.summary)
8  let $tokensLeft :=
9   for $tokenUnranked in $ARevLeft.summary
10  for $tokenRanked at $i in
11   // -- - Stage 1 - --
12   for $t in dataset AmazonReview
13   let $id := $t.ARev_id
14   for $token in word-tokens($t.summary)
15   /*+ hash */
16   group by $tokenGrouped := $token with $id
17   order by count($id), $tokenGrouped
18   return $tokenGrouped
19  where $tokenUnranked = /*+ bcast */ $tokenRanked
20  order by $i
21  return $i
22 for $prefixTokenLeft in subset-collection($tokensLeft, 0,
23 prefix-len-jaccard($lenLeft, .5f) - $lenLeft + len($tokensLeft))
24
25 for $ARevRight in dataset AmazonReview
26 let $lenRight := len($ARevRight.summary)
27 let $tokensRight :=
28  for $tokenUnranked in $ARevRight.summary
29  for $tokenRanked at $i in
30   // -- - Stage 1 - --
31   for $t in dataset AmazonReview
32   let $id := $t.ARev_id
33   for $token in word-tokens($t.summary)
34   /*+ hash */
35   group by $tokenGrouped := $token with $id
36   order by count($id), $tokenGrouped
37   return $tokenGrouped
38  where $tokenUnranked = /*+ bcast */ $tokenRanked
39  order by $i
40  return $i
41 for $prefixTokenRight in subset-collection(
42 $tokensRight, 0, prefix-len-jaccard($lenRight, .5f))
43
44 where $prefixTokenLeft = $prefixTokenRight
45 let $sim := similarity-jaccard($tokensLeft, $tokensRight, .5f)
46 where $sim >= .5f and $ARevLeft.ARev_id < $ARevRight.ARev_id
47 group by $idLeft := $ARevLeft.ARev_id,
48  $idRight := $ARevRight.ARev_id with $sim
49 return {'idLeft': $idLeft, 'idRight': $idRight, 'sim': $sim[0]}
50
51 where $ridpair.idLeft = $ARevLeft.ARev_id and
52  $ridpair.idRight = $ARevRight.ARev_id
53 order by $ARevLeft.ARev_id, $ARevRight.ARev_id
54 return {'left': $ARevLeft, 'right': $ARevRight, 'sim': $ridpair.sim}
```

Figure 5.13: Three-stage set-similarity algorithm expressed in AQL for a self join on the Amazon Review dataset using Jaccard similarity with a threshold of 0.5.

prefix tokens in line 44, and compute and verify the similarity of each joined pair using the built-in similarity-jaccard() function. Since a pair of records can share more than one token in

their prefixes, duplicate pairs can be produced, and they are eliminated by using a group-by clause in line 47.

Stage 3: *Record Join* is expressed in lines 1-4 and 51-54, which consist of two joins. The first join adds the record information for the first RID of each RID pair, while the second join adds the record information for the second.

The logical query plan resulting from this large AQL query is shown in Figure 5.14. Hash repartition in the figure means that a tuple will be repartitioned to a corresponding node based on its hashed value. Sort merge repartitioning on a node merges incoming tuples based on their sort field values. To transform a logical query plan generated from a user's SQL++ or AQL query into a similarity join query to the three-stage-similarity query plan similar to the explicit AQL in Figure 5.13, we developed a new framework called AQL+, which will be discussed in Section 5.5.2.

5.5 Optimizing Similarity Queries

In this section, we discuss how the AsterixDB query processor optimizes SQL++ (or AQL) similarity queries and we describe the AQL+ framework in more detail. Note that rewriting a similarity query and AQL+ framework earlier works. We then present how we have addressed the issues in AQL+ framework described in Section 5.1.

5.5.1 Rewriting a Similarity Query

AsterixDB uses rule-based optimization approach [22] as described in Section 2.1.1. An initial logical plan is constructed from a given query, and each optimization rule is tried on this plan. If a rule is applicable, the plan is transformed. A logical plan involving a dataset

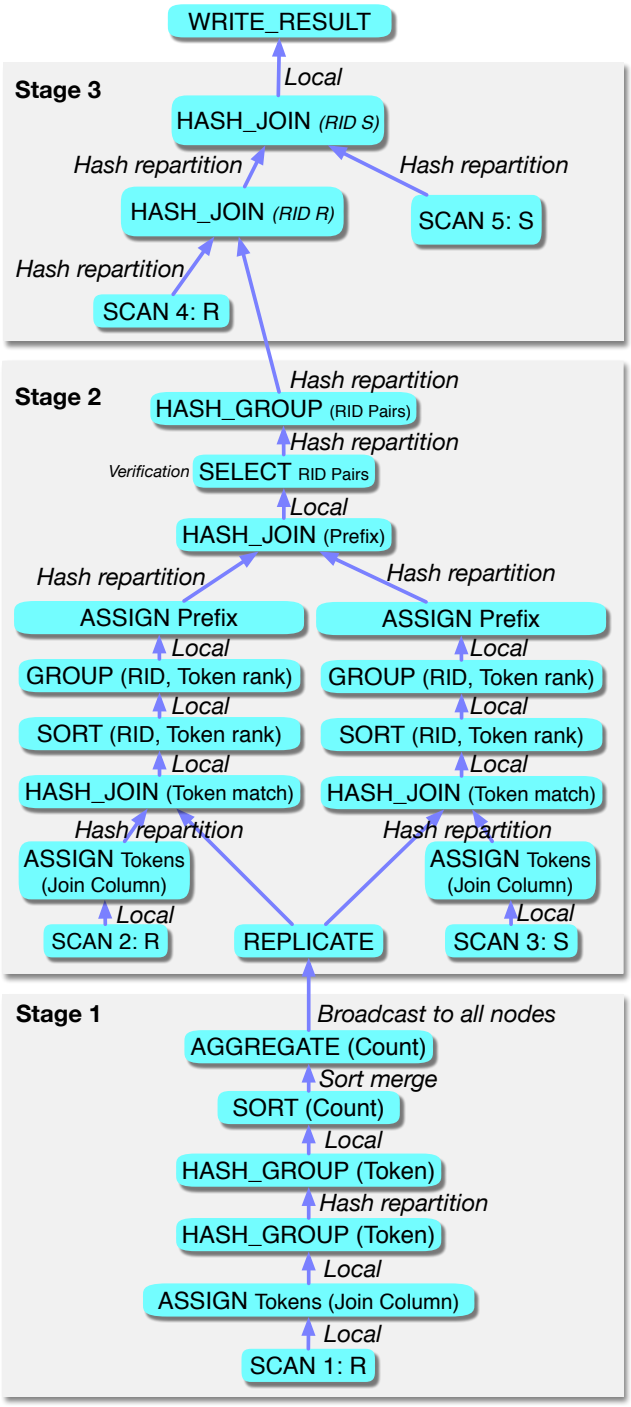


Figure 5.14: A plan of a three-stage-similarity join qeury.

always starts with a PRIMARY-INDEX-SCAN operator, followed by a SELECT operator if there are one or more conditions. For similarity queries, a non-index similarity query plan is constructed first, and an index-based transformation or a three-stage-similarity join can

be introduced during the optimization.

5.5.1.1 Rewriting a Similarity-Selection Query

Figure 5.9 from Section 5.4 shows how a similarity-selection query is optimized to use an index. The left-hand side shows the original scan-based plan, and the right-hand side shows the optimized plan. Based on a SELECT operator with a similarity condition, the optimizer tries to replace the PRIMARY-INDEX-SCAN with a secondary-index-based search plan.

To rewrite a similarity-selection query, the optimizer first matches an operator pattern consisting of a pipeline with a SELECT operator and a PRIMARY-INDEX-SCAN operator. Next, it analyzes the condition of the given SELECT operator to see if it contains a similarity condition and if one of its arguments is a constant. If so, it determines whether the non-constant argument originates from the PRIMARY-INDEX-SCAN operator and whether the corresponding dataset has a secondary index on a field variable V. For each secondary index on V, the optimizer checks an index-function-compatibility table (Figure 5.15) to determine its applicability. For example, an n-gram index can be utilized for the edit_distance() function. The final SELECT operator in the figure filters out false positives.

Index Type	Supported Functions
n-gram	edit-distance(), contains()
keyword	similarity-jaccard()

Figure 5.15: Index-function compatibility table.

Corner cases: Recall that for queries using edit distance, the lower bound on the number of common q-grams (or tokens) may become zero or negative. For such a corner case, the optimizer must revert to a scan-based plan even if an index is available since an index cannot be used for non-positive T-occurrence values. For selection queries, the optimizer can foresee such cases at compile time when applying the corresponding index-rewrite rule by analyzing the constant argument in the similarity condition. When detecting a corner case, it simply

stops rewriting the plan. Note that no such corner cases are possible for similarity queries based on Jaccard, because if two sets have no elements in common, then they can never reach a Jaccard similarity greater than 0. In contrast, two strings could be within a certain (large) edit distance even if the n-gram sets of the (short) strings have no common elements.

5.5.1.2 Rewriting a Similarity-Join Query

The basic rewriting of a similarity-join query using an index is shown in Figure 5.12 from Section 5.4. The optimized query plan on the right-hand side uses an index-nested-loop join strategy. Similar to the rewrite for selection queries, the optimizer replaces the PRIMARY-INDEX-SCAN of the inner branch with a secondary-index search followed by a primary-index search. Thus, it is required that the inner branch of the join is a PRIMARY-INDEX-SCAN, while the outer branch could be an arbitrary operator subtree (shown simply as Subtree in the figure). In the optimized plan, the outer branch feeds into the SECONDARY-INDEX-SEARCH operator, i.e., every record from Subtree will be used as a search key to the secondary index. As in the similarity-selection case, the optimizer needs to remove false positives from the index-based subplan using a SELECT operator based on the original similarity condition, which is taken from the JOIN operator. Notice the broadcast connection between the outer subtree and the secondary-index search. This connection tells the Subtree to broadcast its output stream's records to all of the inner dataset's secondary-index partitions.

The optimizer first matches the join operator's required pattern, which is a PRIMARY-INDEX-SCAN in the inner branch, since this operator fully scans the dataset rather than using a secondary index on some other condition. Also, the optimizer checks the inner branch since it considers using a secondary index only from the inner branch, not from the outer branch. Next, it analyzes the join condition to make sure the similarity function has two non-constant arguments and checks if an argument of the similarity condition is produced by

the PRIMARY-INDEX-SCAN operator in the inner branch, and whether the corresponding inner dataset has an applicable secondary index to support the required similarity lookups. The optimizer then consults the index compatibility matrix to decide whether it can rewrite the query using an index.

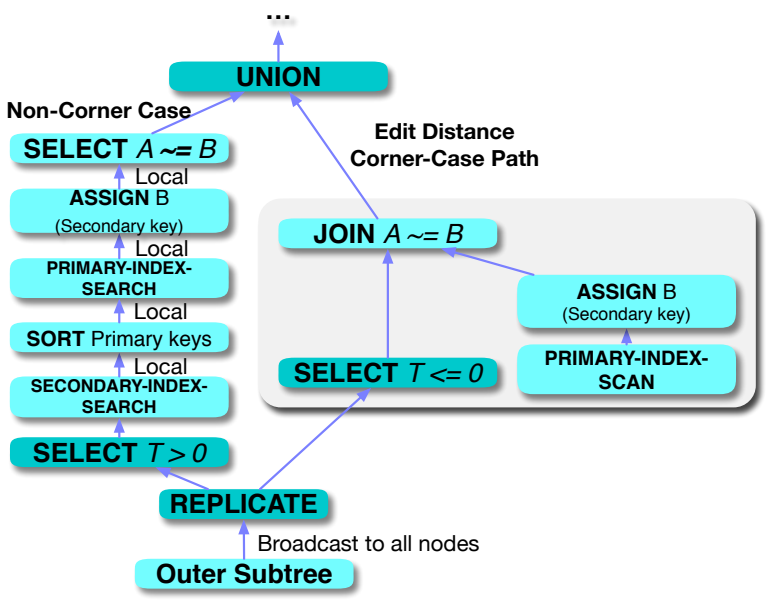


Figure 5.16: An optimized similarity-join query plan with the corner case.

Corner cases: For string-similarity joins using edit distance, we must modify the basic index-nested-loop join plan in Figure 5.12 to handle corner cases. Unlike selection queries, where the secondary-index search key is a constant, the secondary-index search keys for an index-nested-loop join are produced by the outer branch (Subtree). Join corner cases must, therefore, be dealt with at the query runtime, as opposed to the query compile time as for selection queries. Figure 5.16 shows the modified index-nested-loop plan for handling corner cases for edit distance. The main difference lies in separating the records produced by the outer subtree into two sets, one containing non-corner-case records ($T > 0$), and one containing corner-case records ($T \leq 0$). We do this by using a REPLICATE operator above the outer subtree, followed by SELECT operators on each of its two outputs to filter out the corner-case and non-corner-case records, respectively. As before, the non-corner-case records are fed into the secondary-to-primary index plan. The corner-case records participate in a

non-index nested-loop join plan. The final query answer is the union of the results of those two joins.

5.5.2 AQL+ Framework

As discussed in Section 5.4.3.2, for non-index-based similarity joins, the optimizer needs to transform a nested-loop-join plan generated from a user's query into a three-stage join plan to accelerate similarity-join-query execution. A challenge is that, unlike the index-nested-loop-join optimization that adds or replaces a few operators from a nested-loop join plan, a three-stage-similarity join plan contains a large number of operators as illustrated by the AQL query in Figure 5.13. Figure 5.17 shows the number of operators in a three-stage-similarity join.

Operator	Count
ASSIGN	12
SCAN	2
JOIN	1
Total	**15**

Nested-loop join plan

Operator	Count	Operator	Count
AGGREGATE	6	JOIN	3
ASSIGN	44	ORDER	3
SCAN	6	SELECT	4
GROUP	3	UNNEST	8
		Total	**77**

Three-stage-similarity join plan

Figure 5.17: Number of operators for a nested-loop join and three-stage-similarity join plan for the same query.

Due to the complexity of the tree-stage-join query plan, it would be rather difficult to build and maintain an optimization rule that manually constructs the DAG of operators that transform a simple nested-loop join plan into a three-stage join plan. Instead, we developed a novel rewrite framework called AQL+ that converts a simple logical plan generated from a user's similarity-join query into a three-stage join plan. The flow of the AQL+ framework is depicted in Figure 5.18. The essential part of the AQL+ framework is the use of an AQL+ query template to express sophisticated query expressions, integrate the information from the incoming logical plan into it, and finally transform the plan during the optimization process. This way the optimizer does not need to have a complex rule that manually translates a

simple nested-loop-join plan into a three-stage-similarity-join plan. What we need instead is an AQL+ query template that expresses the three-stage-similarity join and for the AQL+ framework to combine the incoming plan with the query template.

When the SQL++ (or AQL) optimizer receives a logical similarity-join plan in AQL+, it extracts the information from the plan and integrates it with an AQL+ query template that expresses the three-stage-similarity join. The generated AQL+ query is then parsed and compiled again using the AQL+ parser and translator since the generated query itself is also a query. The result of this process is a transformed logical plan. The resulting plan is then processed by the rest of the query plan optimization process.

To combine the information from an incoming logical plan and the three-stage-similarity-join AQL query template, we need ways to refer to relevant portions of the surrounding logical plan from within the AQL+ query template. Therefore, the AQL+ framework consists of a few AQL language extensions and the compilation of these language extensions during the optimization process. As a result, the AQL+ language is a superset of AQL, the first AsterixDB query language. (Note that we developed AQL+ when AQL was the primary language of AsterixDB, so AQL+ was based on AQL. We later adopted SQL++ in AsterixDB and SQL++ is now mainly used. However, the differences between SQL++ and AQL do not affect the AQL+ framework since the AQL+ framework works on the logical plan level.) The AQL+ language has three AQL extensions: Meta Variable (denoted as $$), Meta Clause (##), and Explicit Join (join). We use these extensions to refer to the logical variables and operators in the incoming logical plan during the optimization process, as the AQL+ transformation of a given plan happens during the optimization process. Note that the optimizer sees only the logical plan and physical plan, not the original query. Since AQL itself does not have an explicit join clause, AQL+ adds one in order to express a join of two branches. We use meta-variables to refer to the primary keys of the input records or variables in the similarity predicate. The usage of meta-clauses is to refer to the inputs of the AQL

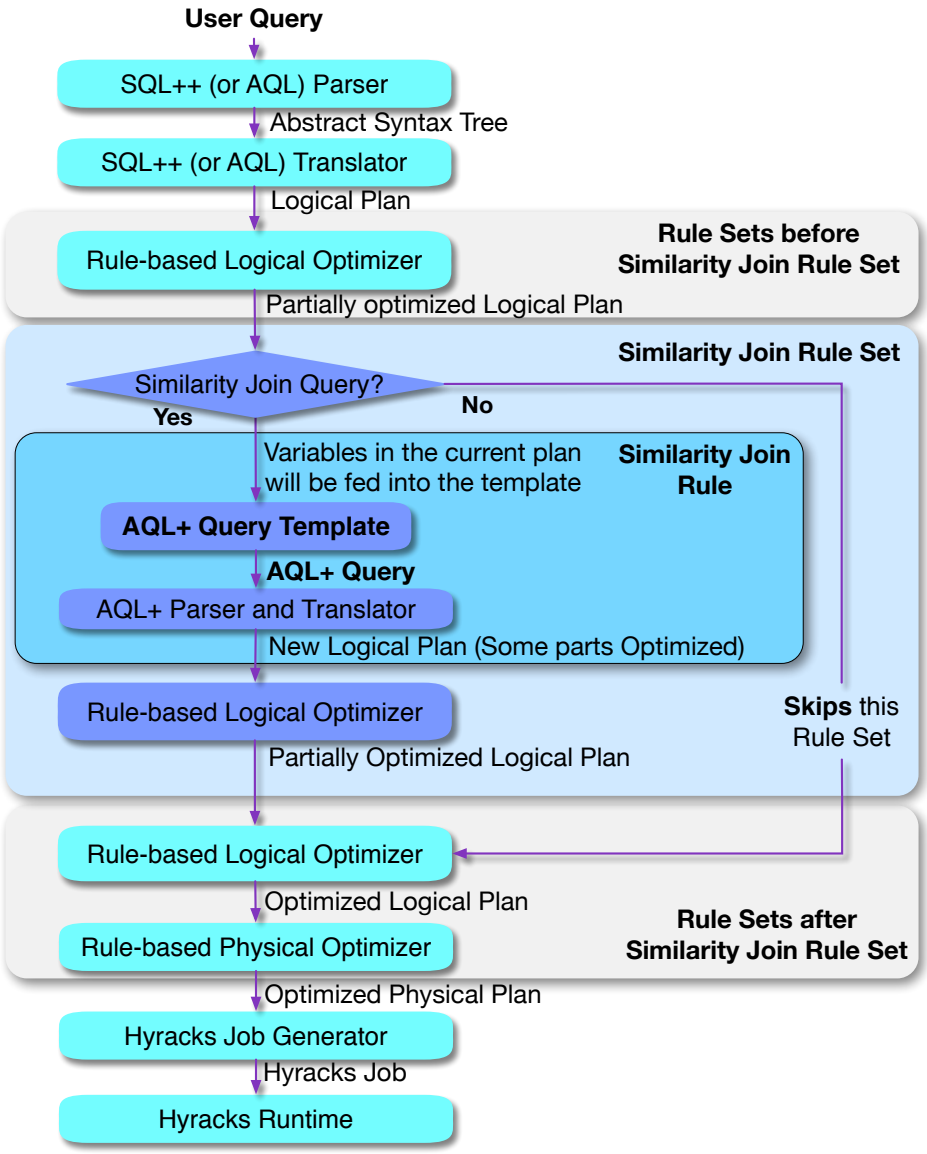


Figure 5.18: Execution of a similarity-join query using AQL+.

query and to refer to logical constructs that cannot be directly specified in AQL, such as operators in the plan. In this way, any AQL+ template can be combined with any join input branches, where the inputs can be from any kinds of subplans made up of other algebraic operators. In addition, to support various types of data, similarity functions, and thresholds, the similarity-join rule template uses placeholders that are parts of the AQL+ query and are unknown until runtime. These are used for data types, similarity-specific functions, or values. For example, a SIMILARITY placeholder is used for built-in AQL functions, and a

THRESHOLD placeholder is used for numerical similarity values.

Specifically, for the three-stage-similarity join, the optimizer needs to identify a similarity JOIN operator that contains a Jaccard similarity join and its threshold. It also needs to get the information about the two branches of this JOIN operator. Using this information, the logical plan fed into the AQL+ template can be transformed into the equivalent three-stage-similarity-join plan. Again, rather than doing this transformation by introducing operators by hand, we rely on the existing compilation path to generate a revised plan. This process is depicted in Figure 5.18; the details of this optimization flow will be discussed in the next subsection.

Table 5.1: AQL+ extensions (to AQL).

Extension	Symbol	Functionality
Meta Variable	$$	Refer to a variable in the plan
Meta Clause	##	Refer to an operator in the plan
Join Clause	join, loj	Express an explicit inner join or left-outer join

The optimizer uses the AQL+ three-stage-similarity-join query template shown in Figure 5.19 to transform the incoming user query during the rule-rewrite phase. In this way, the simple user-written query of Figure 5.4(a) can be transformed into the query of Figure 5.13 during the optimization process. The details of this AQL+ template are as follows. (We mostly focus on the AQL+ constructs here.)

Stage 1: *Token Ordering.* This stage is expressed in lines 14-21 of Figure 5.19. The first meta-clause ##RIGHT_3 refers to the left input operator of the given join. Since the same branch can be used several times in each stage of the three-stage-similarity-join, if there are dependencies between the reused branches, deep copies of the given branch will be created. The suffix _3 here denotes that this is the third copy of the given branch. In the next line, $id is assigned to the primary key of the left branch, which is denoted by a meta-variable, $$RIGHTPK_3. The suffix _3 means the third copy of the branch as described. TOKENIZER is a template placeholder that will be replaced by an actual tokenizer function. For example,

```
1 // --- Stage3 ---
2 join(
3  (##LEFT_0),
4  (join(
5    (##RIGHT_0),
6    // --- Stage 2 ---
7    (join(
8  (##RIGHT_1
9  let $tokensUnrankedRight := TOKENIZER($$RIGHT_1)
10 let $lenRight := len($tokensUnrankedRight)
11 let $tokensRight :=
12  for $token in $tokensUnrankedRight
13  for $tokenRanked at $i in
14   // --- Stage1 ---
15   ##RIGHT_3
16   let $id := $$RIGHTPK_3
17   for $token in TOKENIZER($$RIGHT_3)
18   /*+ hash */
19   group by $tokenGrouped := $token with $id
20   order by count($id), $tokenGrouped
21   return $tokenGrouped
22  where $token = /*+ bcast */ $tokenRanked
23  order by $i
24  return $i
25 for $prefixTokenRight in subset-collection($tokensRight, 0,
26     PREFIX_LEN(len($tokensRight), THRESHOLD))
27   ),
28
29   (##LEFT_1
30 let $tokensUnrankedLeft := TOKENIZER($$LEFT_1)
31 let $lenLeft := len($tokensUnrankedLeft)
32 let $tokensLeft :=
33  for $token in $tokensUnrankedLeft
34  for $tokenRanked at $i in
35   // --- Stage1---
36   ##RIGHT_2
37   let $id := $$RIGHTPK_2
38   for $token in TOKENIZER($$RIGHT_2)
39   /*+ hash */
40   group by $tokenGrouped := $token with $id
41   order by count($id), $tokenGrouped
42   return $tokenGrouped
43  where $token = /*+ bcast */ $tokenRanked
44  order by $i
45  return $i
46 let $actualPreLen:=PREFIX_LEN(len($tokensUnrankedLeft), THRESHOLD)
47                                - $lenLeft + len($tokenLeft)
48 for $prefixTokenLeft in subset-collection(
49  $tokensLeft, 0, $actualPreLen)
50 ),
51 $prefixTokenLeft = $prefixTokenRight)
52   let $sim := SIMILARITY($tokensRight, $tokensLeft)
53   where $sim >= THRESHOLD
54   /*+ hash */
55   group by $idLeft:=$$LEFTPK_1, $idRight:=$$RIGHTPK_1 with $sim),
56  $$LEFTPK_0 = $idLeft)),
57 $$RIGHTPK_0 = $idRgiht)
```

Figure 5.19: Three-stage-similarity join algorithm expressed in AQL+.

the string tokenizer will be used for a string field. Note that if $$RIGHT_3 is an array or a multiset, no tokenizer will be added.

Stage 2: *Record ID (RID)-Pair Generation.* This stage is expressed in lines 6-51. This stage starts with a join meta-clause. This meta-clause has three arguments, and each argument is separated by a comma. The first argument is the left input branch of the join. The second argument is the right input branch of the join. The third argument describes the join condition. Since AQL does not have an explicit join clause, the join meta-clause in AQL+ provides a join expression that can be directly translated into a logical JOIN operator. In this join, the left branch is expressed in lines 8-27. Lines 29-50 denote the right branch of the join meta-clause. Line 51 of the template denotes the join condition itself. Specifically, the left branch starts with ##RIGHT_1, which means the first copy of the branch. In the next line, tokens will be generated from the right variable of the original JOIN operator. In line 25, the prefix tokens for the right branch are calculated. Here, PREFIX_LEN denotes the function that calculates the prefix length based on the similarity type and the threshold. THRESHOLD contains the similarity threshold. In line 46, the prefix length for the left side is calculated. This calculation is different from that of the right branch, as a token that is in the right branch may not exist in the left branch. This calculation is needed since the template will be applied for both *R*-*S* joins and self-joins (*R*-*R*). The join condition is in line 51.

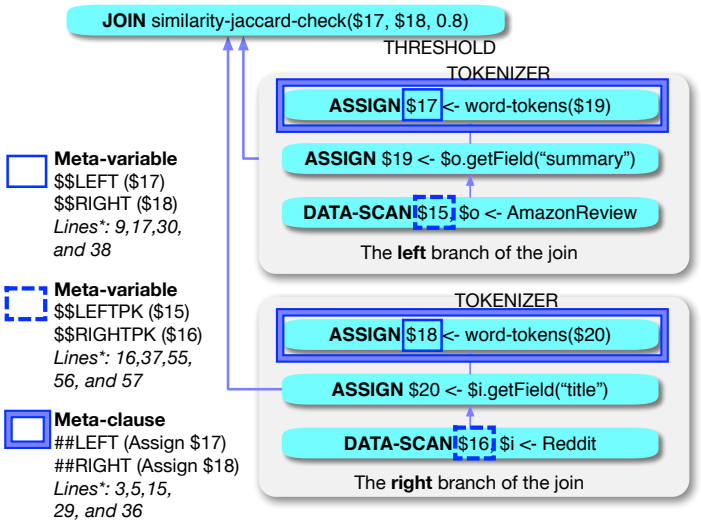
Stage 3: *Record Join.* This stage, which fetches the fields for similar records for the final result, is expressed in lines 1-5 and 56-57, each of which consists of two join meta-clauses. The first join meta-clause adds the record information for the right branch and the second join meta-clause adds the record information for the left branch. The conditions for the two joins are in lines 56-57.

To apply this AQL+ template to transform a nested-loop-similarity-join plan into the three-stage-similarity-join plan, some preparation must be done. All meta-variables, meta-clauses, and placeholders need to be replaced by actual operators, variables, and logical JOIN operators in the three-stage-similarity-join optimization rule. For example, in the AQL+ template,

the primary key of each branch are referred using ##RIGHTPK and ##LEFTPK variables. These meta-variables are replaced with the actual primary keys in that optimizer rule. Figure 5.20 shows an example similarity-join query and its logical plan including the AQL+ constructs used in the AQL+ template. We can see that the top-most operators in both join branches form meta-clauses. Also, the primary keys from the datasets in both branches are regarded as meta-variables. The variables used in the similarity-join condition are also meta-variables in the plan. These AQL+ constructs appear in the AQL+ template in Figure 5.19. For example, the ##LEFT meta-clause is used in line 3.

```
select element {"rid":o.id, "iid":i.id}
from AmazonReview o, Reddit i
where similarity_jaccard(word_tokens(o.summary),
      word_tokens(i.title)) >= 0.8
```

(a) A simple similarity-join query



Lines* indicates the lines in the AQL+ template where the given AQL+ constructs are used.

(b) A logical plan for the query and AQL+ constructs.

Figure 5.20: An example query and the corresponding logical plan that AQL+ template receives.

In addition to two-way similarity joins, the AQL+ framework can be applied to transform multi-way-similarity join plans as well because the optimizer can transform a logical plan iteratively. Similar to non-similarity-join cases, multi-way-similarity joins can be transformed sequentially. For instance, Figure 5.21 shows a similarity-join plan involving four datasets.

The join between the first two datasets, R and S, has already been transformed into a three-stage-similarity join plan. This branch will act as the outer branch when the optimizer processes the next JOIN operator on the third dataset T.

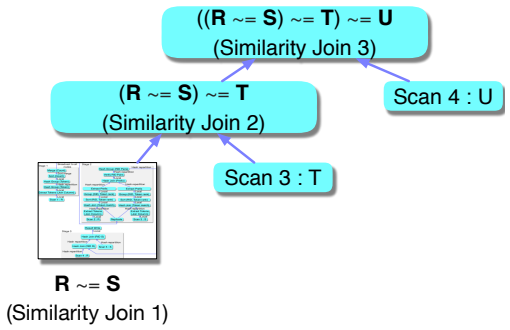


Figure 5.21: Rewriting a multi-way-similarity-join plan on four datasets.

It is worth noting that AQL+ is a general extension framework, not only for similarity queries, that in principle can be used to support other transformations expressed via AQL during the compilation process. (This was part of the original vision for AQL+.)

5.5.3 The Optimization Rule For Similarity Queries

As discussed in Section 2.1.1, the optimization process in AsterixDB is rule-based. Once the Algebricks layer receives a compiled plan from a SQL++/AQL query, it optimizes the plan both logically and physically. It first optimizes the given plan logically using several rule sets.

To apply the similarity-query optimization framework to the current optimization path, we created a new rule set for the AQL+ framework and similarity queries, as was shown in Figure 5.18. The new rule set includes a similarity join rule (SJR) along with a handful of other rules that need to be applied after SJR is applied. As described earlier, the main functionality of AQL+ is to perform a transformation using a complex AQL+ template in order to re-generate a logical plan while maintaining the current surrounding query plan as

part of the new plan. SJR first analyzes the conditions of a JOIN operator. If its condition includes a similarity predicate, it applies the AQL+ template to the plan to generate an AQL+ query. It then compiles the query into a new logical Algebricks plan. During this process, all meta-variables, meta-clauses, and placeholders are replaced by actual variables, operators, and logical JOIN operators.

At this point, some parts of the overall query plan will have already been optimized if they belonged to the original incoming plan. However, much of the plan will not have been optimized yet, as the three-phase plan has just been compiled and had not gone through the optimization process before the application of the SJR rule set. Therefore, the newly generated plan needs to go through some of the earlier optimization rules again to ensure that those rules have a chance to process all of the newly added plan fragment's constructs. Note that this re-application process is not necessary for non-similarity queries since the plan generated for a non-similarity query has not been touched by the SJR rule set. Therefore, for efficiency, the optimizer needs to ensure that the similarity-join rule set is only applied to similarity-join queries. A benefit of this approach is that the optimization steps for similarity queries can be executed without interfering with those for non-similarity queries; this approach also gives the newly generated similarity-query plan a chance to reach the same level of transformation once the similarity rule set has finished its work.

To apply the similarity-join rule set only to similarity join queries, we needed to create a new rule set controller since the application of a rule set on a plan is controlled by a rule set controller as shown in Table 5.2. For instance, the Prioritized rule set controller executes each rule until that rule does not generate any changes to the plan. After the rule controller goes through all rules in the rule set, it repeats the process until all rules do not generate any changes to the plan. Since non-similarity join queries should not be affected by the similarity join rule, we developed a new rule set controller called Sequential-first-rule-gatekeeper to ensure this behavior. This new rule set controller executes the first rule in

the rule set. If the application of the first rule succeeds, the rule set controller executes the other rules in the rule set. If the application of the first rule fails, the entire rule set will be ignored. Therefore, we place SJR as the first rule in the similarity join rule set and use this new rule controller. If SJR is not fired, none of the rules in the rule set will be executed.

Table 5.2: Rule controllers for a rule set.

Rule Controller	Functionality
Prioritized	Executes each rule until it produces no changes. Then the whole collection of rules is executed again until no change is made.
Sequential-fix-point	Executes rules sequentially in a round-robin fashion until one iteration over all rules produces no change.
Sequential-once	Executes all rules sequentially only once.
Sequential-first-rule-gatekeeper	If the first rule in the rule set is fired during the first iteration, it executes the other rules sequentially in a round-robin fashion until one iteration over all rules produces no change. Except to the case where the first rule in the first iteration fails, all rules will be executed during each iteration.

5.5.4 Maintaining the AQL+ Framework

Like AQL, AQL+ needs to have its language grammar definition, the parser generated from this grammar, and the translator to translate the parsed expression into a logical plan. Using the grammar file, we could generate the parser using JavaCC [55] like we did for AQL. Once the parser generates a parsed expression tree from the AQL+ query template, the AQL+ compiler translates the expression to a logical plan.

Since the AQL+ language is a superset of AQL, it needs to include all parts of the AQL language grammar plus three AQL+ extensions in its grammar file. Our original design choice was keeping a separate grammar file, as shown in Figure 5.22, since we did not want to expose AQL+ to the user level. Another reason for that choice was that we wanted to minimize the interference between the two languages by separating them. Thus, the separate AQL+ grammar file included the entirety of the AQL grammar file and it also contained the AQL+ extensions. The AQL+ parser was then generated from this AQL+ grammar file.

The AQL+ compiler also included the entirety of the AQL compiler plus its three extensions to AQL+.

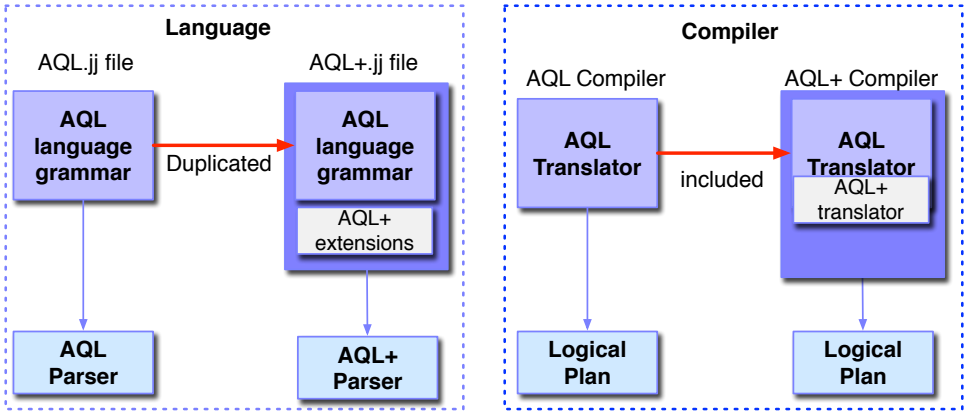


Figure 5.22: The AQL and the original implementation of the AQL+.

One issue with the original fully separated design was that we needed to manually maintain the AQL+ grammar file and AQL+ translator. That is, whenever an update was made to the user-level AQL grammar and translator, a corresponding update had to be applied to the AQL+ grammar file and AQL+ translator as well. Over time, the grammar files and translator files diverged as there was no enforcement. Since AQL+ was implemented in 2012, as time passed, we found that not all AQL updates were applied to AQL+. This divergence issue was the worst in the AQL+ translator; a number of the newer optimizations made to the AQL translator were not applied to the AQL+ translator.

To address these issues, we developed a method to automatically maintain the AQL+ grammar file and AQL+ translator. For the AQL+ grammar, we extended the grammar's pattern match and replacement module to recognize AQL+. (This module was originally implemented as earlier work to extend AQL.) This module reads both the user-level AQL grammar file and the AQL+ extension grammar file and can integrate them into the needed AQL+ grammar file. Thus, whenever the AQL grammar is changed, the AQL+ grammar file is now automatically updated by this module, as shown in Figure 5.23.

We have addressed the divergence issue in the compiler by using inheritance. Previously, the

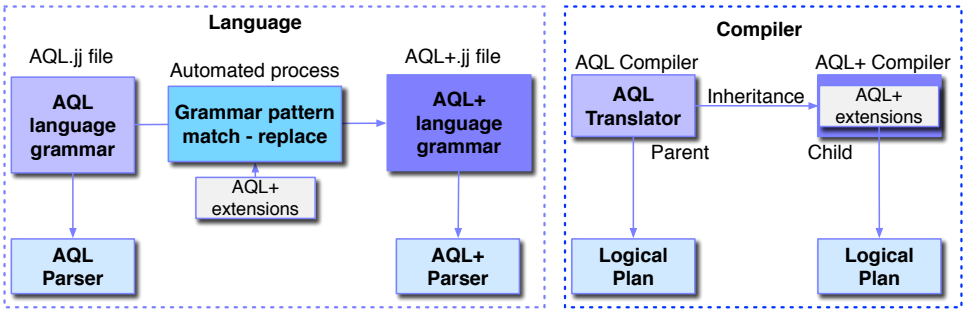


Figure 5.23: The revised relationship between AQL and AQL+.

AQL+ translator and AQL translator were independent classes in the codebase. We have now designated the AQL translator class as the parent of the AQL+ translator class. Thus, in the AQL+ translator class, we only need to keep the logic required to deal with the three AQL+ extensions, as shown in Figure 5.23. Because of this inheritance between the two compiler classes, we no longer need to manually update the AQL+ translator when AQL is changed.

5.6 Experiments

We have conducted an experimental evaluation of our approach in AsterixDB using large, real data sets. We used an 8-node cluster to host an AsterixDB (0.9.3) instance, where each node ran Ubuntu with a Quadcore AMD Opteron CPU 2212 HE (2.0GHz), 8GB RAM, 1 GB Ethernet NIC, and had two 7,200 RPM SATA hard drives. Each dataset was horizontally partitioned into 16 partitions (2 per node) based on their primary keys to provide full I/O parallelism. Table 5.3 shows the AsterixDB configuration parameters.

5.6.1 Datasets

We used several similarity functions to experiment with different types of data. Edit distance is more suitable for short string fields, while Jaccard is more suitable for long fields with

Table 5.3: AsterixDB parameters for the experiments.

Parameter	Value
Global memory budget per node	6 GB
Budget for in-memory components	3 GB
Data page size	128 KB
Disk buffer cache size	2 GB
Sort buffer size	128 MB
Join buffer size	128 MB
Group-by buffer size	128 MB

many elements. To evaluate AsterixDB with different similarity functions, we used the three datasets with different characteristics shown in Table 5.4. The Amazon Review dataset, discussed in earlier sections, included Amazon product reviews from [67]. The Reddit Submission dataset contained about eight years of postings on Reddit from [80]. The Twitter [93] dataset had 1% of US tweets for three months that we obtained ourselves via Twitter's public API. When imported into AsterixDB, each data set had an additional auto-generated primary key field, as AsterixDB requires that each dataset must have a primary key. Other than this field, we did not define more fields in the pre-declared test schemas. This gave us a lot of flexibility to import any datasets into AsterixDB. The dataset size in AsterixDB was greater than the raw data size since each stored record contained additional information about each non-pre-declared field such as the field name, field type, and value. For example, for a string field named summary, each instance of the field summary will contain the field name summary, its type as string, and its value. (In contrast, if the field was explicitly defined in the schema, the field name and its type would not be required to be stored in each record.)

Table 5.5 shows the characteristics of the search fields of the three datasets. The minimum character length and minimum word count of the fields were 0. The first three fields in the table were used for edit distance, while the latter three fields were used for Jaccard.

Table 5.4: Dataset characteristics.

Dataset	**Amazon Review [67]**	**Reddit [80]**	**Twitter [93]**
Content	Amazon product reviews	Reddit postings	Tweets
Number of Records	83.68M	196M	155M
Data Period	1996 - 2014	01/2006 - 08/2015	06/2016 - 08/2016
Raw Data Format	JSON	JSON	JSON
Raw Data Size	55 GB	252 GB	465 GB
Dataset Size in AsterixDB	60.6 GB	305 GB	582 GB
Fields used	summary, reviewerName	title, author	text, user.name

Table 5.5: Characteristics of the search fields.

Field	**Avg char count**	**Max char count**	**Avg word count**	**Max word count**
AmazonReview.reviewerName	10.3	49	1.7	14
Reddit.author	24.3	275	4.1	32
Twitter.user.name	10.6	20	1.7	10
AmazonReview.summary	22.8	361	4.0	44
Reddit.title	1,056.2	400K	1,173	20K
Twitter.text	62.5	140	9.7	70

5.6.2 Index Size

We built a keyword index for Jaccard similarity queries and a 2-gram index for edit distance queries. To measure the execution time for basic exact-match queries on the same fields to serve as a baseline, we also built a B+ tree index on each of the search fields. Table 5.6 shows the index sizes for the Amazon Review dataset and the time to create each index. An n-gram index took much more space than a B+ tree index or a keyword index, as it had more secondary keys per record. For instance, a 2-gram index on the reviewerName field needed 15.6GB of disk space, which was about 25% of the original dataset size. The size of a keyword index was also greater than a B+ tree index on the same field since it had multiple secondary keys per record. For each type of index, the construction time was roughly proportional to the size of the index. In each case, the dataset itself was also stored in a primary B+ tree index.

Table 5.6: Index size and build time for Amazon Review dataset.

Field	Index Type	Size (GB)	Build Time (sec)
Dataset itself	B+ tree (primary)	60.6	1,563
reviewerName	B+ tree (secondary)	2.7	223
reviewerName	2-gram (secondary)	15.6	1,441
summary	B+ tree (secondary)	3.5	275
summary	keyword (secondary)	5.4	573

5.6.3 Selection Queries

To measure the performance of similarity-selection queries, we first created a search value set that contained 10,000 random unique values that we extracted from the search field. For Jaccard queries, we ensured that the minimum number of words in each value in the search set was 3. For edit distance queries, the minimum length of characters in each value was 3. For each similarity threshold, we randomly chose search values from the set for a query and sent 100 such queries to the cluster, and measured their average execution time. The

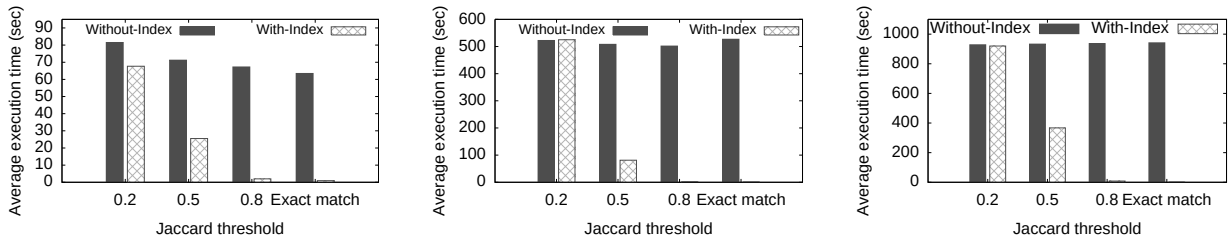
performance baseline for comparison purposes was an equality-condition query that used the same values for the given field. Figure 5.24 shows an example query that we used to measure the average execution time of the Jaccard similarity queries. In this example, we used similarity_jaccard as the similarity function on the summary field with the threshold of 0.5. The second parameter of the function was a random value from the above search value set.

```
select value count(*) from (
  select ar.id
  from AmazonReview as ar
  where similarity_jaccard(ar.summary,
        "should have tried them at the store") >= 0.5
) as first_select;
```

Figure 5.24: An example SQL++ similarity-selection query.

5.6.3.1 Jaccard Similarity

For each of the three datasets, we ran similarity queries using Jaccard similarity on suitable fields using different thresholds: 0.2, 0.5, and 0.8. Figure 5.25 shows the results. We see that the average execution time for similarity selection queries decreased as the threshold increased in the case of index-based plans. For example, it took the index-based method 67.6 seconds to conduct a Jaccard query with a threshold of 0.2, while it took only 25.5 seconds to execute a query with a threshold of 0.5 on the Amazon Review dataset. If there was no applicable index, both similarity and exact-match queries showed a high execution time as each record had to be read from the primary index and the data scan time was a dominant factor in the overall execution time. We can also see the overhead of the similarity query versus the exact-match query for all the thresholds since it took more time to calculate a Jaccard value than to get the result of an exact match. This overhead decreased as the threshold increased because we applied optimizations such as early termination and pruning based on string lengths, which significantly reduced the cost of computing the similarity. The trend is similar in the other two datasets.



(a) Amazon Review (b) Reddit (c) Twitter

Figure 5.25: Execution time of Jaccard selection queries on the three datasets.

When the threshold was low, the execution times were similar for both index-based and non-index-based queries. This is because the candidate set size using T-occurrence for index-based queries was large when the threshold was low, as shown in Table 5.7. As the number of candidates increased, the search time increased due to the need for a primary-index lookup and a verification for each candidate.

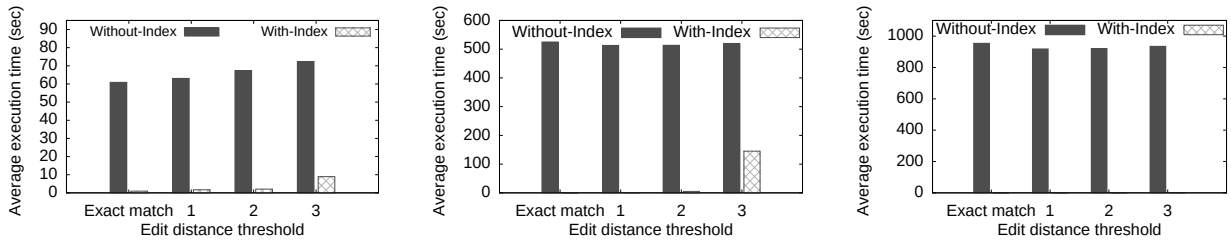
Table 5.7: Candidate size and the final result size for the indexed-Jaccard-selection query for Amazon Review dataset in Figure 5.25.

Jaccard Threshold	**Actual Result Record Count (B)**	**Candidate Set Record Count (C)**	**Ratio (B/C)**
0.2	559,167	8,298,473	6.7%
0.5	12,260	660,016	1.9%
0.8	36	12,420	0.3%

5.6.3.2 Edit Distance

We measured the average execution time of an edit distance selection query using different thresholds, namely 1, 2, and 3. Figure 5.26 shows the results. As the threshold increased, the execution time increased. The reason is similar to the case of Jaccard queries; the candidate set size using T-occurrence increased as the threshold increased, as can be seen in Table 5.8. It took the index-based method 2 seconds to run a selection query with a threshold of 2; it took 8.9 seconds to run a query with a threshold of 3. We can also see that the execution time of non-index-based edit distance queries increased as the threshold increased for the

same reason as described above.



(a) Amazon Review (b) Reddit (c) Twitter

Figure 5.26: Execution time of edit-distance selection queries on the three datasets.

Table 5.8: Candidate size and the final result size for the indexed-edit-distance-selection query for Amazon Review dataset in Figure 5.26.

Edit Distance Threshold	Actual Result Record Count (B)	Candidate Set Record Count (C)	Ratio (B/C)
1	52	64	81.25%
2	297	3,477	8.54%
3	4,185	239,166	1.75%

5.6.4 Join Queries

To measure the performance of similarity join queries, we first ran similarity-self-join queries on the three datasets. Figure 5.27 shows an example query that was used to measure the average execution time as in the similarity-selection query case. Here, summary is the field on which we applied a similarity function and id is the primary key field. After conducting the self-join experiments, we conducted an additional multi-way join experiment that included both similarity and non-similarity joins. Finally, we conducted a multi-way similarity join experiment that had two similarity joins involving all three datasets in one query.

```
select value count(*) from (
  select element {"oid":o.id, "iid":i.id}
  from AmazonReview as o, AmazonReview as i
  where similarity_jaccard(o.summary, i.summary)
        >= 0.8
  and o.product_id = "B00103DCIZ" and o.id < i.id
) as first_join;
```

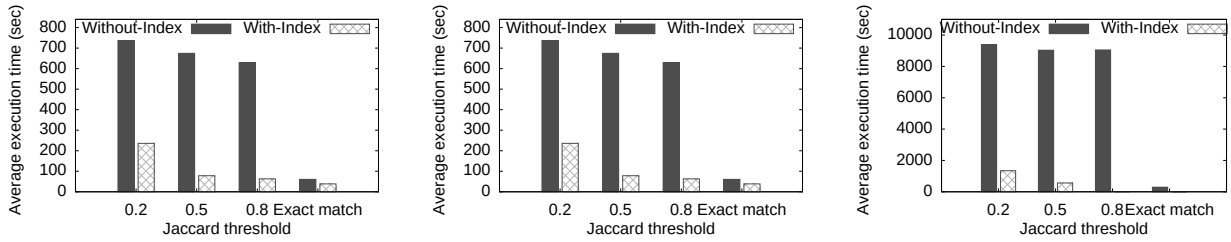
Figure 5.27: An example SQL++ similarity-join query.

5.6.4.1 Varying Threshold

We first extracted a certain number of records from the outer branch of the join to limit the size of its input. For each query, we chose 10 random records from the outer branch. In the example query in Figure 5.27, the field named product_id was used to impose this limit. For Jaccard join queries, we used three similarity thresholds, namely 0.2, 0.5, and 0.8. For edit distance, we used distance thresholds of 1, 2, and 3.

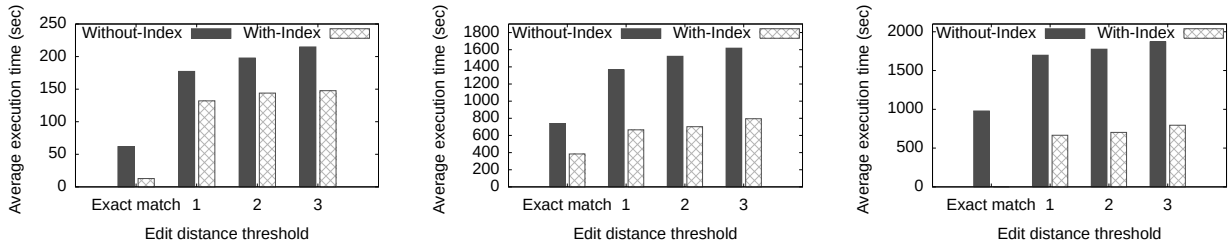
When there was no applicable index, AsterixDB used the three-stage-similarity-join plan for the Jaccard queries. The results are shown in Figures 5.28 and A.1. The trends were similar to those of selection queries except for the exact-match join, which significantly outperformed both the Jaccard and edit distance joins since it used a hash join in which the join keys were broadcast to multiple nodes. For the index-nested-loop join case, all three datasets showed a similar trend on both the Jaccard and edit distance joins. For instance, for the Jaccard queries, as the threshold increased, the average execution time decreased as well.

Regarding the compilation overhead of AQL+, we observed that the average overhead of generating a new logical three-stage-similarity-join plan using AQL+ for the queries of Figure 5.28 was around 50 ms, and it took around 500 ms to optimize that plan. The overall compilation time of the three-stage-similarity-join query was around 900 ms, which was small relative to the time required to actually execute the resulting query plan.



(a) Amazon Review (b) Reddit (c) Twitter

Figure 5.28: Execution time of Jaccard join queries on the three datasets.



(a) Amazon Review (b) Reddit (c) Twitter

Figure 5.29: Execution time of edit distance join queries on the three datasets.

5.6.4.2 Varying Input Size

To further explore the relative performance of the join methods, we conducted a Jaccard join with a fixed threshold and varied the number of records to be joined. For a Jaccard join query, its execution time was smallest when the threshold was 0.8. In this experiment, we varied the number of records coming from the outer branch and fixed the threshold at 0.8. The times for the non-index-nested-loop self-join, index-nested-loop self-join, and three-stage-similarity self-join on the Amazon Review dataset are shown in Figure 5.30. We increased the number of output records from the outer branch and measured the resulting execution time of each join. First, we see that the execution time of non-index-nested-loop self-join was already the highest by far at 200 records and that it increased drastically compared to the other two types of joins. Once the number of output records from the outer branch reached around 400, the three-stage-similarity join began to outperform the index-nested-loop join. This is because the time for the index-nested-loop join is proportional to the number of records fed

to its secondary-index search, as it deals with each record one at a time. For the three-stage-similarity join, the time spent on global-token-order generation in the first stage was the same for all cases, since the order was generated from the inner branch and we only varied the number of records from the outer branch. The hash joins utilized in stage 2 and 3 can deal with the incoming records efficiently since each join key (a token) is sent to only one node. In fact, the average execution time increased slightly for the three-stage-similarity-join case as the number of records that need to be processed in stage 2 and 3 was increased as well. This slight increase of the average execution time of the three-stage-similarity join can be verified in the figure. For instance, the time for the three-stage-similarity join for 800 records was 619 seconds, while it was 674 seconds for 1,000 records. This result shows only 55 seconds of increase, whereas the execution-time difference for index-nested-loop joins when going from 800 to 1,000 input records was 384 seconds.

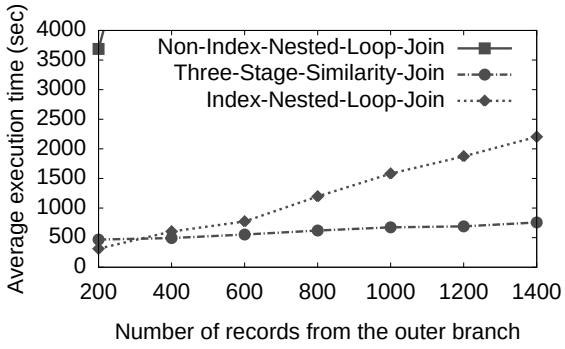


Figure 5.30: Similarity joins on the Amazon Review dataset.

5.6.4.3 Multi-Way Join Queries

So far we have used only one similarity condition per query. Next, we added one more similarity condition to the query and varied the order of the conditions. A similarity join is conducted with the first condition and then a SELECT operator with the other predicate is applied after the join. These similarity conditions were a Jaccard condition with a threshold of 0.8 and an edit distance condition with a threshold of 1. We also added an initial equijoin

to control the number of records being fed into the similarity join. This join is applied first to generate a fixed number of records. Figure 5.31 shows an example query that we used to measure the average execution time. As we see in this query, there is one similarity join and one equijoin. The dataset ProductID and the field product_id were what we used to limit the number of initial records from the outer branch.

```
select value count(*) from (
  select element {"oid":o.id, "iid":i.id} from
  from ProductID as pr, AmazonReview as o,
       AmazonReview as i
  where pr.product_id = "B00103DCIZ"
  and pr.product_id = o.product_id
  and similarity_jaccard(o.summary, i.summary) >= 0.8
  and edit_distance(o.reviewerName, i.reviewerName)
      <= 1 and o.id < i.id
) as first_join;
```

Figure 5.31: An example SQL++ multi-way-join query.

For the first equijoin, we used an index-nested-loop join to fetch the initial records quickly to avoid a full-scan of the dataset. The Jaccard similarity and edit distance conditions were then applied. In the cases where we applied the Jaccard condition first, the Jaccard join was followed by the edit distance condition in a SELECT operator. For both conditions, we used an index-based method for the first and a non-index-based method for the second. That is, we tried three types of queries in total. The first query initially used the indexed Jaccard similarity join. The second query used the indexed-edit distance join first. The last query used the non-indexed Jaccard join first. Figure 5.32 shows that the performance was the best when the index-based-Jaccard join was conducted first, as then there were no corner cases for Jaccard similarity. This similarity predicate order also generated fewer candidates than applying the index-based edit distance predicate first. In contrast, for the edit distance case, the optimizer needed to augment the corner-case path in the logical plan, and thus it generated more candidates.

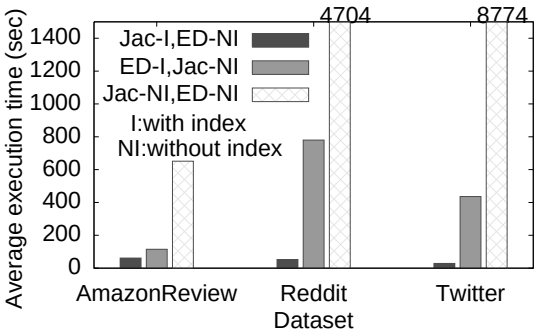


Figure 5.32: Multi-way-join queries on the three datasets.

5.6.4.4 Multi-Way Three-Stage-Similarity Join Queries

The previous join experiment used a non-similarity-index-nested-loop join and a similarity join. After two joins, a second similarity predicate was applied via a SELECT operator. To test the performance of a query with multiple three-stage-similarity-joins, next we used all three datasets in one query as shown in Figure 5.33. First, we fetched ten random records from the Amazon Review dataset and conducted a three-stage-similarity-join between the summary field of the dataset and the title field of the Reddit dataset. The result was used to conduct a join with the text field of the Twitter dataset. We ran this multi-way join query three times with ten different records each time. The resulting average execution time was 6,908 seconds and the average result count was 737,406. Note that we used the Reddit and Twitter datasets as the outer branches of two three-stage-similarity-joins since the global token order was generated from the inner branch that fetched ten records from the Amazon Review dataset. An example record that this query found was "So Comfy", "So comfy....", and "So comfy" from the Amazon Review, Reddit, and Twitter datasets respectively.

5.6.5 Cluster Scalability Tests

We used both speed-up and scale-out metrics to evaluate similarity-query processing in a parallel environment.

```
select value count(*) from (
  select element {"tid":tw.id, "sid":second.sid,
                  "fid":second.fid}
  from Twitter tw, (
    select element {"sid":re.id, "fid":first.oid,
                    "title": re.title}
    from Reddit re,
      (select element {"oid":o.id, "summary":o.summary}
       from ProductID as pr, AmazonReview as o
       where pr.product_id /* +indexnl */ = o.product_id
       and pr.id = "B00103DCIZ"
      ) as first
    where similarity_jaccard(re.title, first.summary)
          >= 0.8
    ) as second
  where similarity_jaccard(tw.text, second.title)
        >= 0.8
) as third;
```

Figure 5.33: An example SQL++ multi-way three-similarity-join query.

5.6.5.1 Speed-up

First, for our speed-up experiment, we used five cluster sizes (1, 2, 4, 8, and 16 nodes), with each cluster size being given the entire (100%) dataset to spread out across its partitions. Figures 5.34 and Figure 5.35 show the speed-up and average execution time of the previous Jaccard selection and join queries with the threshold set to be 0.8. For each type of query, we measured the average execution time of 100 indexed and 100 non-indexed queries. The speed-up of the selection queries was proportional to the number of nodes. However, both of the join queries showed non-linear behaviors here that we did not observe earlier in [58], where the maximum number of nodes was 8. With 16 nodes, the graph here showed a clear distinction among these queries. (Another difference here is that we increased the number of queries from 10 to 100.)

One observation is that the speed-up of the three-stage-similarity-join query in Figures 5.34-5.35 appears to be sub-linear. This is mainly due to the communication cost among all nodes, as the three-stage-similarity-join involves a number of tuple exchanges (as was shown in Figure 5.14). In particular, before each hash join, every tuple from both join branches is hash-partitioned to a potentially different node based on the join key's hash value. Except

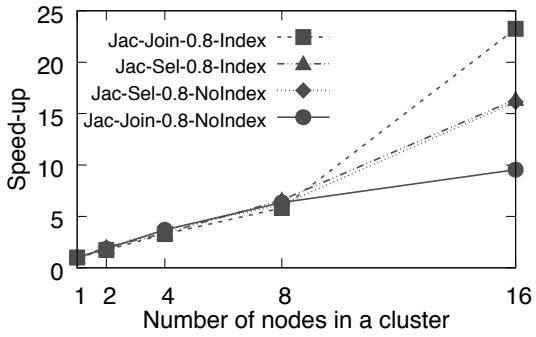


Figure 5.34: Speed-up on Jaccard on Amazon Review dataset.

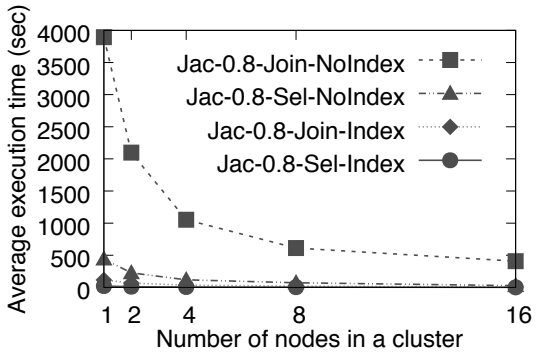


Figure 5.35: Times for Jaccard speed-up on Amazon Review dataset.

for the joins in stage 3, where extra fields from two branches of the original similarity join are being fetched, all hash joins are conducted on a token or a prefix (based on the global token order). That is, each field in the original query generates more join keys that are joined in stage 2 since each field value is tokenized. The tokens from both branches in these joins need to be hash-partitioned. Also, in stage 1, after the sorting on the token frequency on each node is done, the partial results from each node need to be merged on one node to create the global token order. This merge operation is conducted in a serial fashion; thus, it became a bottleneck for the global sort operation. Based on these characteristics, the three-stage-similarity-join can be regarded as a communication-bound process and the "sub-linear" behavior of the three-stage-similarity-join stems from the fact that the speed-up of a heavily communication-bound process is about $\frac{k}{2}$, where k is the number of nodes, as explained in Appendix A.1. If there is only a small number of nodes in a cluster, the speed-up is less than $\frac{k}{2}$. As we increase the number of nodes, we see the eventual linear nature of the speed-up in

the graphs, which indeed goes as approximately $\frac{k}{2}$. Although $\frac{k}{2}$ is a linear speed-up trend, still, the ratio is less than k.

Figure 5.36 shows the per-stage-execution-time of the three-stage-similarity-join query on each cluster setting. We can see that the most time was spent on stage 2. Most of the communication cost in stage 2 is due to the fact that we need to tokenize the given field from the dataset and conduct a hash join to match each token against each token in the global token order chosen in stage 1. In fact, we observed that on the 16-node cluster, it took 255 seconds to exchange the tokens and the primary keys among all the nodes in one of the hash-joins of stage 2. Since the query took 423 seconds in total, 60% of the query execution time was spent just on the hash-partition exchange involved in stage 2.

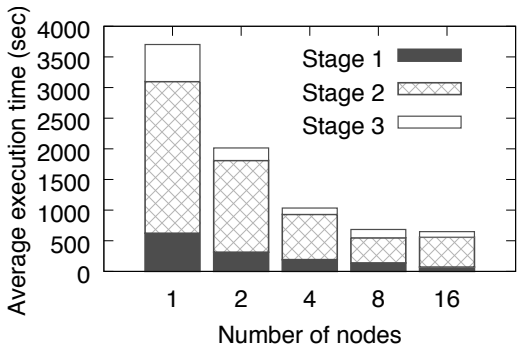


Figure 5.36: Per-stage execution time of the three-stage-similarity-join query on Amazon Review dataset.

In contrast, the speed-up of the indexed-nested-Jaccard-join query in Figure 5.34 is seen to be super-linear. To explain this behavior, we can decompose the indexed-nested-Jaccard-join query into three steps. In step 1, AsterixDB extracts 10 random tuples from the outer branch and broadcasts them to all nodes. It then extracts the given field, tokenizes the field value, and conducts keyword-index searches using the tokens. The keyword-index search yields candidate primary keys. In step 2, these primary keys are sorted and fed into the primary-index lookup. AsterixDB extracts the field from the record to again verify the Jaccard condition and other predicates. In step 3, AsterixDB employs a surrogate hash-join at the top level to merge any other fields that are not the secondary key or the primary

key fields since this is an inverted-index-join. After this join, the count of primary keys will be gathered and returned to the user. Since the surrogate-hash-join is for primary keys on the same dataset, there is little communication required since records are partitioned on the primary key. Figure 5.37 shows the execution time of this join query per stage. Most of the time was spent in step 2, as shown in the figure. Note that the speed-up of each operation in each step is linear except for the sorting operation. That is, when we reduce the size of the dataset partition on each node, the amount of work is reduced linearly. For example, if it takes 1 second to conduct 1,000 primary key lookups on a 1-node cluster, it takes about 0.5 seconds to conduct 500 primary key lookups on each node of a 2-node cluster. Unlike other operations in the query plan, The speed-up of the sorting operation is $k \cdot log_{\frac{N}{k}} N$, where N is the number of tuples and k is the number of nodes, as explained in Appendix A.2. This ratio is super-linear because of its second term, which explains the super-linear behavior of the query. For instance, on a 16-node cluster, the speed-up was 20.02 (>16) when N was 1 million.

One more observation was that the speed-up on the 8-node cluster was lower than expected in general because there was skewness of the data distribution among the nodes in the cluster. In our experiment, when loading the data, each record received a randomly generated UUID value as its primary key. Therefore, in each cluster setting, the actual data distribution was different. Note that our search was conducted on a secondary key field, not the primary key. We observed that on the 8-node cluster, it took about 27 seconds to conduct step 2, and there was a 17-second difference between the time when the first node finished and the time when the last node finished this step. This showed the skewness of the data distribution.

5.6.5.2 Scale-out

To explore scale-out, we again used five clusters of different sizes, namely 1, 2, 4, 8, and 16 nodes. In this case, however, when we doubled the number of nodes in the cluster, we also

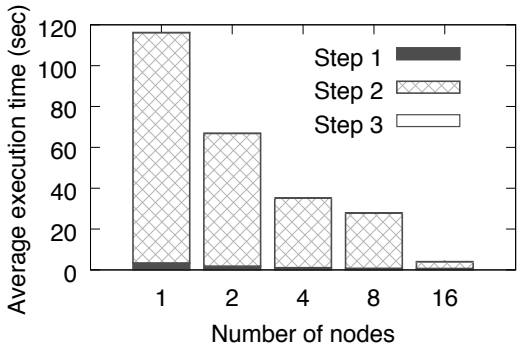


Figure 5.37: Detailed execution time of index-nested-loop-Jaccard-join queries on Amazon Review dataset.

doubled the data size to yield the same amount of data per node. The 1-node cluster had just 6.25% of the original total data set size, the 2-node cluster had 12.5% of the data, the 4-node cluster had 25% of the data, and the 8-node cluster had 50% of the data. The 16-node cluster had the entire original dataset. Ideally, for linear scale-out, the response-time graph would show a flat line per query. In fact, the response times for each cluster size were similar, as shown in Figure 5.38, except in the case of the ad hoc Jaccard-similarity join without an index. As we described in the speed-up section, this was due to the fact that the three-stage-similarity-join is a communication-bound join method, so its communication cost increases as the number of the nodes increases in a cluster based on the fact that the volume of data on each node remains the same.

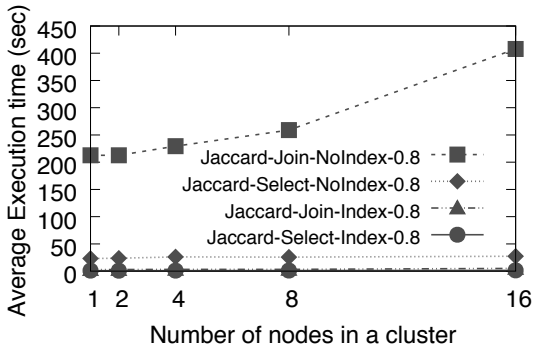


Figure 5.38: Scale-out for Jaccard on Amazon Review dataset

5.6.6 Comparison with Other Systems

In addition to evaluating the performance characteristics of AsterixDB's algorithms, we evaluated the performance of similarity queries on two other systems that also support certain types of similarity queries. One is basic Apache Hadoop and the other is Couchbase. We selected these two systems because we had previously implemented the three-stage-similarity-join as a map/reduce job in Apache Hadoop and because Couchbase supports scalable edit distance queries.

5.6.6.1 Apache Hadoop

The three-stage-similarity join [94] was originally implemented by hand using Apache Hadoop. Based on the original code [11], we replicated the three-stage-join experiment on Apache Hadoop Map/Reduce 1.2.1, the most recent version compatible with that three-stage-similarity-join code. To make the execution environment similar to that of AsterixDB, we designated one node to host master daemons to run the Hadoop jobs and to control the Hadoop Distributed File System (HDFS). Including this node, eight nodes were utilized to run map and reduce tasks. Each node had two directories since there were two AsterixDB partitions. We set the HDFS block size to 128MB and allocated 1GB of virtual memory to each HDFS daemon. Since AsterixDB used 2 GB as buffer space, we allocated 2GB of virtual memory to each map/reduce task. We ran two map tasks and two reduce tasks on each node so that the degree of parallelism was also the same for both systems. The replication factor was set to 1, and Hadoop's speculative task execution feature was disabled. Figure 5.39 shows the execution times for the same three-stage-similarity-join query that used Jaccard with a threshold of 0.8 for several different cases (explained below).

One difference between the Apache Hadoop implementation and AsterixDB is their global token order generation in stage 1. When calculating the global token order of an R and S

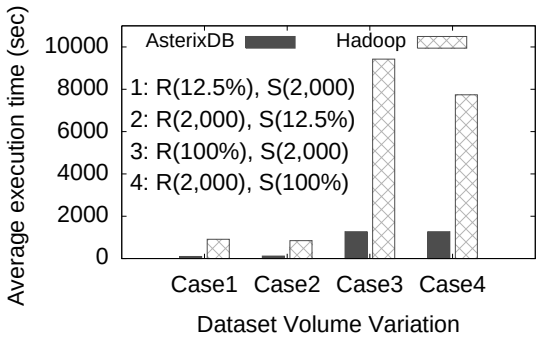


Figure 5.39: Three-stage-similarity-join queries on AsterixDB and Hadoop Map/Reduce.

join, Hadoop uses R to build the global token order. AsterixDB instead uses S to build the global token order. Thus, we experimented with four variations for the dataset size in Figure 5.39. In the first case, the left branch (R) used 12.5% of the tuple of the Amazon Review dataset while the right branch (S) used $2,000$ tuples of the same dataset. We then switched the left and the right branches in the second case. By checking these two cases in Figure 5.39, we can see that the execution time was smaller when the size of the dataset used to generate the global token order was smaller. We also see that the execution time for AsterixDB was about ten times faster than that of Hadoop. This was because the execution of AsterixDB is pipelined whenever possible. In contrast, the result of each map-reduce task is written to disk and then read again in Hadoop.

5.6.6.2 Couchbase

As described in Section 5.1.1, Couchbase is unique in providing support for edit distance search queries on NoSQL data with its new full-text search service. It does so via a separate full-text API (not its N1QL query language). A full-text index must be built before sending full-text queries involving edit distance. Given the provided support, only an indexed-edit-distance query comparison between AsterixDB and Couchbase is appropriate. In our experiments, AsterixDB had two physical partitions on two separate hard disks on each node to increase its degree of parallelism. Since Couchbase can only support multiple physical

partitions using a redundant array of independent disks (RAID), we also ended up using only one partition on each node of the AsterixDB cluster for this comparison experiment. For Couchbase, we used version 5.0, and we set up the full-text service on the same nodes and allocated 2GB of the memory to the full-text service on each node. After loading the Amazon Review dataset into both systems, we sent ten random indexed-edit-distance queries to AserixDB and Couchbase and measured their average execution times. The results are shown in Figure 5.40.

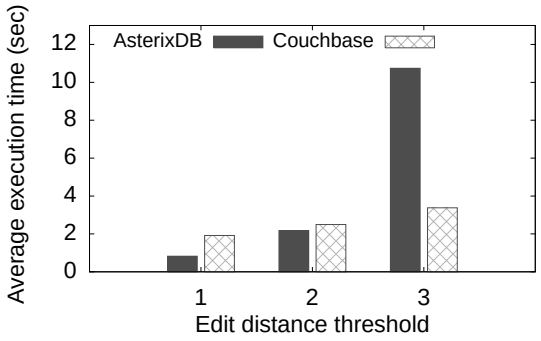


Figure 5.40: Edit-distance queries on AsterixDB and Couchbase.

When the edit distance threshold was 1 or 2, AsterixDB performed better than Couchbase. When the threshold was 3, however, AsterixDB became about five times slower than Couchbase. A careful investigation revealed that the main reason is that the inverted-index search in AsterixDB generated many candidates as the threshold increased. These candidates needed to be verified via a primary-index search and applying the edit distance function on the fetched field. That is, an inverted-index search alone in AsterixDB cannot generate the final result, as described earlier. In contrast, the full-text index in Couchbase alone can generate the final answer without having to check the original data because their index contains all the data needed to generate the final answer. That is, a tentative result from an edit distance query is then verified within the full-text index to generate the final result. Note that this design implies that there may be inconsistencies between the full-text index and the actual data in a bucket until the synchronization between a bucket and the full-text index is done. In contrast, AsterixDB always generates an answer consistent with the most

current data.

5.7 Conclusions

In this chapter, we have described the support for similarity queries in Apache AsterixDB, a parallel open source Big Data management system. We described the entire life cycle of a similarity query in the system, including the query language, indexing, execution plans, and plan rewriting to optimize query execution. Our similarity search solution leverages the existing infrastructure of AsterixDB, including its operators, query engine, and rule-based optimizer. We presented an experimental study based on several large, real-world data sets on a parallel computing cluster to evaluate the proposed techniques and showed their efficacy and performance for supporting similarity queries on large data sets using parallel computing. Also, we presented a performance comparison with two other systems.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented how we enhanced the support for both search and analytics in AsterixDB to deal with large amounts of data.

In Chapter 3, we described a budget-driven approach to memory management in AsterixDB since proper memory management is a crucial part of search and analytics. We 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 a budget. Each memory-intensive operator's implementation requires careful attention regarding memory usage since memory-intensive operators need to support 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 original implementation of AsterixDB's memory-intensive operators and the memory details that they had overlooked. We then described how we modified these operators to truly run within a budget. We also discussed issues related to the global memory management in AsterixDB. Lastly, we presented exper-

iments to empirically explore the effect of not carefully accounting for the size of the data structures used in memory-intensive operators. We used both synthetic and real datasets and showed that the current AsterixDB implementations of memory-intensive operators are well-controlled and scalable.

In Chapter 4, we described the index-only query plan implementation in AsterixDB. We discussed a few challenges that needed to be addressed to implement an index-only plan in our context. We then described the required conditions for use of index-only plans and we explained how to create a correct index-only plan. We also showed how to transform a scan-based plan into an index-only plan during the optimization process. We presented an experimental study on a real dataset, using both spatial and temporal indexes, and showed that the average execution performance of index-only plan queries was several orders of magnitude faster than scan-based and index-based queries.

In Chapter 5, we described a performance evaluation of similarity queries in AsterixDB as well as two other systems. We first described the entire life cycle of a similarity query in AsterixDB, including the query language, indexing, execution plans, and plan rewriting to optimize query execution. Our similarity search solution leverages the existing infrastructure of AsterixDB, including its operators, query engine, and rule-based optimizer. We then presented an experimental study based on several large, real-world datasets on a parallel computing cluster to evaluate the proposed techniques and showed their efficacy and performance for supporting similarity queries on large data sets using parallel computing. We presented a performance comparison with two related systems.

6.2 Future Work

In Chapter 3, we discussed how AsterixDB manages its memory. AsterixDB assigns a budget to each memory-intensive operator and makes the operator conform to the budget to ensure the stability of the system. Currently, almost every budget has a simplistic default or is manually set by a user; AsterixDB does not tune the budget by itself during runtime. A few other systems such as DB2 [3] have a feature called "self tuning" to automatically set the values for their memory configuration parameters. The AsterixDB system's performance can be potentially improved if the system had the ability to adjust the parameters based on information that it collected during runtime.

Regarding query admission control, we showed that AsterixDB puts a query into the execution queue if the available CPU and memory resources are not enough but the cluster's maximum resources can host the query. Since this queue works in a first-in-first-out fashion, each query may need to wait a long time in the queue. Now consider a high priority query that a user just issued, and suppose the user wants to see the result right away. Since AsterixDB does not differentiate priority among queries, every query currently needs to wait until resources become available. In the future, we can add a priority to each query and let the query admission control consider this priority. Moreover, we can then also explore another topic regarding query workload management. That is, is it possible to make AsterixDB dynamically adjust its memory allocation up or down for the current query plan? If this became possible, a high-priority query could be admitted sooner and/or get more resources during execution so that it may finish its execution earlier than expected.

In Chapter 4, we presented the index-only query plan implementation in AsterixDB. We showed that the performance of an index-only plan was orders of magnitude faster than that of a scan-based execution plan and a non-index-only plan. Although the performance improvement of the current implementation of the index-only plan is significant, there are

currently some limitations. First of all, AsterixDB does not yet support an index-only plan on an inverted index since multiple secondary keys may exist per primary key. Because these multiple ⟨secondary key, primary key⟩ pairs cannot be fetched from an inverted index at the same time, AsterixDB cannot guarantee that these pairs come from the same version of a record. That is, a record could be updated during an inverted-index search on a given primary key. This issue can be solved if the system can present a consistent view of multiple secondary keys. One possible solution is employing a multi-version concurrency control scheme. Another possibility would be to add a timestamp value to the primary key, based on the creation time of a record, to secondary keys in the inverted index so that the system can decide whether multiple secondary keys belong to the same record version.

In Chapter 5, we evaluated the performance of similarity queries. We explained the AQL+ framework and how we addressed the divergence issue between AQL+ and AQL languages. Recall that the AQL+ framework transforms a scan-based similarity join plan into a three-stage-similarity join plan during the optimization process. In a three-stage-similarity-join plan, stage 1 generates a global token order and stage 2 conducts several hash joins using the token order generated in stage 1 and generates the actual primary key pairs whose similarity satisfies the predicate. Stage 3 re-fetches any extra fields that need to be returned in a query. In summary, stage 1 calculates statistics and stage 2 utilizes these statistics. In the future, we could seek to design a general join framework that consists of two phases. In phase 1, it would collect some statistics about the input datasets. Then, in phase 2, the framework would use the generated statistics to conduct the actual join and generate the result. A three-stage-similarity join could then also be implemented using such a join framework since similarity join stages 1 and 2 correspond to phases 1 and 2 respectively. The benefit of such a join framework would be that would then have a general framework that makes each join method more modular and easier to maintain.

Bibliography

[1] Apache solr. http://lucene.apache.org/solr/.

[2] DB2 - objects that are subject to locks. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_11.0.0/perf/src/tpc/db2z_objectoflock.html.

[3] Db2 - self-tuning memory overview. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.perf.doc/doc/c0024366.html.

[4] DOMO - data never sleeps 6.0. https://www.domo.com/learn/data-never-sleeps-6.

[5] Elastic search. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html.

[6] Facebook - second quarter 2018 operational and other financial highlights. https://investor.fb.com/investor-news/press-release-details/2018/Facebook-Reports-Second-Quarter-2018-Results/default.aspx.

[7] MySQL - locks set by different sql statements in innodb. https://dev.mysql.com/doc/refman/8.0/en/innodb-locks-set.html.

[8] Oracle - data concurrency and consistency. https://docs.oracle.com/cd/E25054_01/server.1111/e25789/consist.htm.

[9] PostgreSQL - index-only scans. https://www.postgresql.org/docs/10/static/indexes-index-only-scans.html.

[10] PostgreSQL - visibility map. https://www.postgresql.org/docs/9.5/static/storage-vm.html.

[11] Source code - Apache Hadoop Map/Reduce version of the three-stage-similarity join. http://asterix.ics.uci.edu/fuzzyjoin/.

[12] Twitter - second quarter 2018 - selected company metrics and financials. https://investor.twitterinc.com/results.cfm.

[13] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[14] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *PVLDB*, 7(10):841–852, June 2014.

[15] Apache AsterixDB, http://asterixdb.apache.org.

[16] Apache Spark, `https://spark.apache.org/docs/latest/tuning.html#memory-management-overview`.

[17] Basic Memory Structures, `https://docs.oracle.com/cd/B28359_01/server.111/b28318/memory.htm#CNCPT1221`.

[18] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 2007 WWW Conference*, 2007.

[19] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *Proceedings of the 2009 ICDE Conference*, 2009.

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

[21] S. P. Borgatti, A. Mehra, D. J. Brass, and G. Labianca. Network analysis in the social sciences. *Science*, 2009.

[22] V. R. Borkar, Y. Bu, E. P. C. Jr., N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages. In *Proc. SCC*, pages 422–433, Kohala, HI, USA, 2015.

[23] V. R. Borkar, M. J. Carey, R. Grover, et al. Hyracks: A flexible and extensible foundation for data-intensive computing. Proc. 27th ICDE, pages 1151–1162, 2011.

[24] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment*, 2012.

[25] Y. Bu, V. R. Borkar, G. H. Xu, and M. J. Carey. A bloat-aware design for big data applications. Int'l Symp. on Memory Management, ISMM '13, Seattle, WA, USA - June 20 - 20, 2013, pages 119–130, 2013.

[26] D. Chamberlin. *SQL++ for SQL Users: A Tutorial*. September 2018. (Available via Amazon.com.).

[27] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system R. *Commun. ACM*, 24(10):632–646, 1981.

[28] S. Chaudhuri, V. Ganti, and R. Kaushik. Data debugger: An operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, 2006.

[29] P. Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. SSBM, 2012.

[30] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An e cient access method for similarity search in metric spaces. In *Proceedings of the 1997 VLDB Conference*, 1997.

[31] D. L. Davison and G. Graefe. Memory-contention responsive hash joins. In *Proceedings of the 20th International Conference on Very Large Data Bases*, Proc. 20th Int'l Conf. on VLDB, pages 379–390, San Francisco, CA, USA, 1994.

[32] DB2 - Memory sets overview, `https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.perf.doc/doc/c0059501.html`.

[33] DB2 memory allocation,`https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.config.doc/doc/r0000259.html`.

[34] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *Proceedings of the 2014 SIGMOD Conference*, 2014.

[35] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *Proceedings of the 2014 ICDE Conference*, 2014.

[36] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*. ACM, 1984.

[37] C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in mapreduce. *Proceedings of the VLDB Endowment*, 2014.

[38] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[39] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *Proceedings of the VLDB Endowment*, 2012.

[40] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[41] G. Graefe. Sorting and indexing with partitioned b-trees. CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings, 2003.

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

[43] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proc. the 3rd Int'l Workshop on Data Management on New Hardware*, DaMoN '07, pages 6:1–6:9, Beijing, China, 2007.

[44] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[45] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 2001 VLDB Conference*, pages 491–500, 2001.

[46] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *Proceedings of the 2003 WWW Conference*, 2003.

[47] J. Gray, editor. *The Wisconsin Benchmark: Past, Present, and Future.* Morgan Kaufmann, 1993.

[48] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, Dec. 1997.

[49] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. Proc. 1987 ACM SIGMOD, pages 395–398, San Francisco, California, USA, 1987.

[50] T. Härder. A scan-driven sort facility for a relational database system. Proc. 3rd Int'l Conf. on VLDB, pages 236–244, Tokyo, Japan, 1977.

[51] G. Held, M. Stonebraker, and E. Wong. INGRES: A relational data base system. In *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA*, pages 409–416, 1975.

[52] How MySQL Uses Memory, `https://dev.mysql.com/doc/refman/8.0/en/memory-use.html`.

[53] How to Configure Resource Management for Impala, `https://www.cloudera.com/documentation/enterprise/5-9-x/topics/impala_howto_rm.html`.

[54] Lucene - IndexWriterConfig class documentation, `https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/index/IndexWriterConfig.html#setRAMBufferSizeMB-double-`.

[55] JavaCC,`https://javacc.org`.

[56] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *Proceedings of the VLDB Endowment*, 2014.

[57] P. Jokinen and E. Ukkonen. Two algorithms for approxmate string matching in static texts. Proc. 16th Int'l Symp. on MFCS, pages 240–248, Kazimierz Dolny, Poland, 1991.

[58] T. Kim, W. Li, A. Behm, I. Cetindil, R. Vernica, V. Borkar, M. J. Carey, and C. Li. Supporting similarity queries in apache asterixdb. In *EDBT*, 2018.

[59] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[60] D. E. Knuth. The art of computer programming, volume. 3: Sorting and searching". 1973.

[61] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the 2008 ICDE Conference*, 2008.

[62] C. Li, B. Wang, and X. Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In *Proceedings of the 2012 VLDB Conference*, 2007.

[63] B. Lindsay. Hash joins in db2 udb the inside story, 1999. `https://pdfs.semanticscholar.org/presentation/38c4/bcd4fe05787c1c8af74981a4e182ff3411c2.pdf`.

[64] Apache Lucene, https://lucene.apache.org.

[65] W. Mann and N. Augsten. Pel: Position-enhanced length filter for set similarity joins. In *Grundlagen von Datenbanken*, 2014.

[66] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 2016.

[67] J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 2015 SIGKDD Conference*, 2015.

[68] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 2012.

[69] Y. Minghe, L. Guoliang, D. Dong, and F. Jianhua. String similarity search and join: a survey. *Frontiers of Computer Science*, 2016.

[70] MySQL memory allocation,`https://dev.mysql.com/doc/refman/8.0/en/memory-use.html`.

[71] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR*, abs/1405.3631, 2014.

[72] Oracle memory allocation,`https://docs.oracle.com/cd/B28359_01/server.111/b28318/memory.htm#CNCPT1221`.

[73] H. Pang, M. J. Carey, and M. Livny. Memory-adaptive external sorting. Proc. the 19th Int'l Conf. on VLDB, pages 618–629, San Francisco, CA, USA, 1993.

[74] H. H. Pang, M. J. Carey, and M. Livny. Partially preemptible hash joins. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Proc. 1993 ACM SIGMOD, pages 59–68, Washington, D.C., USA, 1993.

[75] P. Pirzadeh, M. J. Carey, and T. Westmann. BigFUN: A performance study of big data management system functionality. In *Proceedings of the IEEE International Conference on Big Data, Santa Clara, CA, October 29 - November 1, 2015.*, 2015.

[76] Postgres memory allocation,`https://www.postgresql.org/docs/9.4/static/runtime-config-resource.html`.

[77] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceeding of the 2011 SIGMOD Conference*, 2011.

[78] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 2000.

[79] Reddit comment dataset,`https://files.pushshift.io/reddit/comments/`.

[80] Reddit Dataset, https://reddit.com/r/datasets/comments/3mg812/.

[81] Resource Consumption, `https://www.postgresql.org/docs/9.4/static/runtime-config-resource.html`.

[82] L. A. Ribeiro and T. Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 2011.

[83] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the 2004 SIGMOD Conference*, 2004.

[84] Y. seok Kim. *Transactional and Spatial Query Processing in the Big Data Era*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2016.

[85] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, 11(3):239–264, 1986.

[86] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *Proceedings of the 2010 ICDE Conference*, 2010.

[87] Y. N. Silva, S. S. Pearson, J. Chon, and R. Roberts. Similarity joins: Their implementation and interactions with other database operators. *Information Systems*, 2015.

[88] Y. N. Silva and J. M. Reed. Exploiting mapreduce-based similarity joins. In *Proceedings of the 2012 SIGMOD Conference*, 2012.

[89] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao. Dima: A distributed in-memory similarity-based query processing system. *Proceedings of the VLDB Endowment*, 2017.

[90] Tez Configuration, https://tez.apache.org/releases/0.8.2/tez-api-javadocs/configs/TezConfiguration.html.

[91] TPC-H Benchmark,http://www.tpc.org/tpch/.

[92] Tuning Hive, https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_hive_tuning.html.

[93] Twitter Streaming API, https://developer.twitter.com.

[94] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 SIGMOD Conference*, 2010.

[95] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012 SIGMOD Conference*, 2012.

[96] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *KDE, IEEE Transactions on*, 2013.

[97] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. In *Proceedings of the 2013 ACM SIGKDD Conference*, 2013.

[98] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *Proceedings of the 2008 VLDB Conference*, 2008.

[99] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 2008 WWW Conference*, 2008.

[100] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. Proc. 16th Int'l Conf. on VLDB, pages 186–197, San Francisco, CA, USA, 1990.

[101] W. Zhang and P.-Å. Larson. A memory-adaptive sort (masort) for database systems. Proc. 1996 CASCON, pages 41–, Toronto, Ontario, Canada, 1996.

[102] W. Zhang and P.-Å. Larson. Dynamic memory adjustment for external mergesort. Proc. 23rd Int'l Conf. on VLDB, pages 376–385, San Francisco, CA, USA, 1997.

Appendix A

Analyses Of Parallel Jobs

In this Appendix, we present detailed analyses of the speed-ups of an idealized communication-bound parallel job and of idealized parallel sort job to explain the super-linear and sub-linear speed-up behaviors of the two queries seen in Section 5.6.5.1.

A.1 Communication-bound Parallel Job

If a perfectly parallel data analysis job is totally communication-bound, how much speed-up can be expected? Table A.1 shows the fraction of the original data that must be transferred from each node in a single data-exchange operation, where k is the number of nodes. In this simple analysis, we assume that a data-exchange function evenly hashes the data among all the nodes and all communications are conducted in a fully parallel fashion, that is, there is no network congestion. If N is the number of tuples of original data, the communication cost per node is $\frac{N}{k} \cdot \frac{(k-1)}{k}$. The first term of the formula is derived from the fact that each node in a k-node parallel cluster contains $\frac{1}{k}$ of the original data N. When a data-exchange operation executes, $\frac{k-1}{k}$ of the data on each node needs to be transferred to other nodes,

and only $\frac{1}{k}$ of the data remains on the same node. Combining these facts yields $\frac{N}{k} \cdot \frac{(k-1)}{k}$. For example, consider a four-node cluster (k=4). Each node contains $\frac{1}{4}$ of the total data N. From each node, $\frac{3}{4}$ of the data on that node needs be transferred to other nodes. Therefore, overall, each of the four nodes transfers $\frac{3}{16}$ of N.

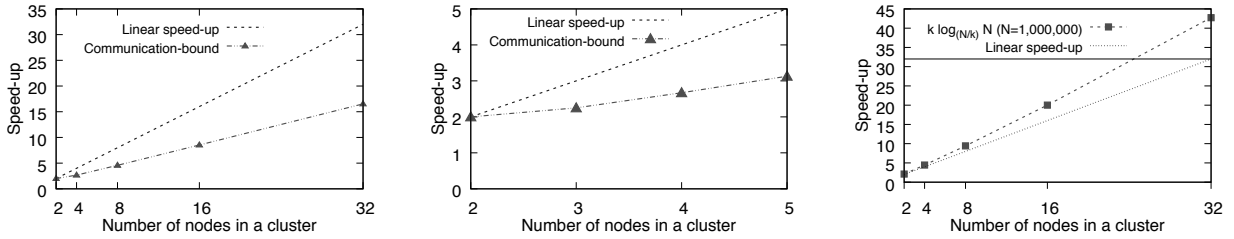
Table A.1: Communication cost per node on a hash exchange operation

Number of nodes	**Communication cost per node in terms of the original data**	**Speed-up**
2	$1 \ / \ 2^2$	2
4	$3 \ / \ 4^2$	2.67
8	$7 \ / \ 8^2$	4.57
16	$15 \ / \ 16^2$	8.53
32	$31 \ / \ 32^2$	16.52
64	$63 \ / \ 64^2$	32.51
128	$127 \ / \ 128^2$	64.5
...		
k	$(k-1)/k^2$	$\approx k/2$

The last column of Table A.1 shows the speed-up of such a communication-bound parallel job. We omit the 1-node cluster case here since there is no communication on one node. We denote the speed-up of the 2-node cluster as 2 as the base in order to have the speed-up metric be normalized based on a number of nodes that starts at 1 (as usual for speed-up). To get the speed-up of the k-node cluster, we first divide the communication cost per node of the 2-node cluster by the communication cost of the k-node cluster. We then multiply the base speed-up of the 2-node cluster, which is 2, by this calculated ratio to get the speed-up of the k-node cluster. For instance, the speed-up of 4-node cluster can be calculated as $\frac{1}{4}/\frac{3}{16} \cdot 2$, which is 2.67.

Figures A.1(a) and A.1(b) show the comparison between the speed-up of our computation-bound parallel job and ideal linear speed-up. Figure A.1(a) shows the number of nodes up to 64. We can see there that the trend of the speed-up of a communication-bound parallel job does not saturate. In fact, as we increase the number of nodes, the speed-up becomes $\frac{k}{2}$. In fact, after 12 nodes, the speed-up is always close to $\frac{k}{2}$. When we zoom in to a smaller number of nodes, in Figure A.1(b), we can see that the speed-up is less than $\frac{k}{2}$, though the

speed-up increases gradually as the number of nodes increases.



(a) speed-up of a parallel communication-bound job (up to 64 nodes).

(b) speed-up of a parallel communication-bound job (up to 5 nodes).

(c) speed-up of a parallel sort job

Figure A.1: speed-up of communication-bound parallel job and parallel sort job.

A.2 Parallel Sort Job

A sort operation takes $O(N \cdot logN)$ time where N is the amount of data. If there are k nodes in a parallel cluster, a parallel sort job takes $O(\frac{N}{k} \cdot log\ \frac{N}{k})$ time on each node since the amount of the data on each node is $\frac{N}{k}$. Thus, we can compute the speed-up of a parallel sort job on the k-node cluster as being $k \cdot log_{\frac{N}{k}}\ N$ by simplifying the formula $N \cdot logN\ /\ \frac{N}{k} \cdot log\ \frac{N}{k}$. As shown in Table A.2 and illustrated in Figure A.1(c), this speed-up is super-linear since the second term ($log_{\frac{N}{k}}\ N$) in the formula is always greater than 1 when k is greater than 1. For instance, on a 16-node cluster, the speed-up is 20.02 (>16) when N is 1,000,000.

Table A.2: Per node cost of a parallel sort (1 million tuples).

Number of nodes	**Sorting cost on a node**	**Speed-up**
1	$1,000,000 \cdot log\ 1,000,000$	1
2	$500,000 \cdot log\ 500,000$	2.11
4	$250,000 \cdot log\ 250,000$	4.45
8	$125,000 \cdot log\ 125,000$	9.42
16	$62,500 \cdot log\ 62,500$	20.02
...		
k	$\frac{N}{k} \cdot log\ \frac{N}{k}$	$k \cdot log_{\frac{N}{k}}\ N$