9-11-2023

A Memory Contention Responsive Hash Join Algorithm Design and Implementation on Apache AsterixDB

Giulliano Silva Zanotti Siviero

SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: September 11, 2023

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Giulliano Silva Zanotti Siviero

ENTITLED

A Memory Contention Responsive Hash Join Algorithm Design and Implementation on Apache AsterixDB

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

DocuSigned by:
Shiva Jahangiri
46FB5AED309540B...

Thesis Advisor: Dr. Shiva Jahangiri

DocuSigned by:
Behnam Dezfouli
6ED175D6B91A4FF...

Thesis Reader: Dr. Behnam Dezfouli

Department Chair: Dr. Silvia Figueira

A Memory Contention Responsive Hash Join Algorithm Design and Implementation on Apache AsterixDB

by

Giulliano Silva Zanotti Siviero

Submitted in partial fulfillment of the requirements
for the degree of
Master of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
September 11, 2023

A Memory Contention Responsive Hash Join Algorithm Design and Implementation on Apache AsterixDB

Giulliano Silva Zanotti Siviero

Department of Computer Science and Engineering
Santa Clara University
September 11, 2023

ABSTRACT

Efficient data management is crucial in complex computer systems, and Database Management Systems (DBMS) are indispensable for handling and processing large datasets. In DBMSs that concurrently execute multiple queries, adapting to varying workloads is desirable. Yet, predicting the fluctuating quantity and size of queries in such environments proves challenging. Over-allocating resources to a single query can impede the execution of future queries while under-allocating resources to a query expecting increased workload can lead to significant processing delays. Moreover, join operations place substantial demands on memory. This resource's availability fluctuates as queries enter and exit the DBMS. The development of join operators capable of dynamically adapting to memory fluctuations is a complex undertaking, with few recent authors proposing memory-adaptive algorithms. This scarcity of proposals suggests the inherent difficulty in designing, implementing, and analyzing such algorithms.

This thesis proposes a new memory adaptive Hash-Based join algorithm extended from designs presented by prior authors. This algorithm is implemented and experimented with in a real DBMS environment to evaluate its memory fluctuation responsiveness. A mathematical model for the increase in I/O caused by it is proposed and compared with actual results. The impacts of memory variation and frequence of memory updates reveal the importance of this thesis for further development of memory adaptive algorithms.

Table of Contents

List of plots

Chapter 1

Introduction

For a large portion of modern complex computer systems, data management is a crucial component and this task is often executed by specialized software called Database Management Systems (DBMS), capable of efficiently storing, managing, retrieving, and analyzing large amounts of data.

DBMSs capable of executing multiple queries concurrently must be able to adapt to different workload scenarios. In such systems, the number and size of queries can rapidly change, and predicting future workloads is quite difficult, if possible. Statically allocating too many resources to process a single query, assuming few others will arrive shortly, can prevent future queries from starting their execution. On the other hand, allocating too few resources for a query expecting a future heavy workload can considerably increase the time necessary to process it.

Queries that require joining relations are especially demanding on a specific resource: primary storage, referred to throughout this thesis simply as memory. This resource's availability for an operator may change as queries arrive or leave the DBMS. Join operators capable of dynamically adapting to memory fluctuations not only can leverage this resource when it becomes available but also can contribute to other operators releasing memory in case of scarcity.

Although desirable, designing and implementing a join operator capable of gracefully adapting to memory fluctuations is not a simple task and requires careful consideration. Only a few authors have proposed new memory adaptive algorithms [5, 14, 18] in recent decades. It is speculated by authors of [1] that the reason for this hiatus is the difficulty of designing, implementing and analyzing such algorithms.

The primary objective of this thesis is to reexamine algorithms that have been put forward in prior research and juxtapose their empirical simulations against real-world executions. Through experiments conducted within an authentic DBMS application, novel insights emerge that go beyond the scope of the original authors' considerations, further enhancing the proposed algorithms.

Before we present a proposed implementation of our Memory-Contention Responsive Hash Join (MCRHJ), we offer a comprehensive introduction to Hash-Based Join algorithms in Chapter 2, accompanied by a succinct description of the Big Data Management System (BDMS) we used as our testbed, the open source project Apache AsterixDB.

Chapter 3 reviews related papers on the topic, while the actual design of our implementation is presented in Chapter 4. Ultimately, experiments and conclusions are presented in Chapter 5, while potential future studies are proposed in Chapter 6.

Chapter 2

Background

This section presents the groundwork for the thesis by covering fundamental concepts related to DBMS operators. To begin, we will offer a comprehensive introduction to Hash-Based Join algorithms, emphasizing their unique features and referencing key papers for deeper insights. Additionally, since the proposed Join algorithm is implemented in Apache AsterixDB, an open-source DBMS, we will also explore its architecture, thereby concluding the background chapter.

2.1 DBMS operators

A DBMS employs a variety of operators to manipulate stored data effectively. These operators can be interconnected sequentially through dataflow connectors to achieve the desired output efficiently. To illustrate the concepts discussed in this chapter, we will use a bookstore database as a practical example

That bookstore's manager analyzes the book portfolio managed by a DBMS to gather business insights regarding its customer's reading profile. To do so, he elaborates an SQL query to extract information about the average number of pages in novel books published after the year 2000. In the bookstore's database, the *Books* relation is composed of the following attributes : Title, Author, Pages, Genre, and Year.

Table 2.1 provides a set of seven potential entries within this relation serving as an example throughout this chapter, while Listing 2.1 presents the final SQL query statement created by the manager.

```
SELECT AVG(Pages) FROM Books WHERE Year >= 2000 and Genre = "Novel"
```

Listing 2.1: Select and Aggregate

Books Relation					
#	**Title**	**Author**	**Pages**	**Genre**	**Year**
1	1984	George Orwell	328	Romance	1949
2	The Fountainhead	Ayn Rand	753	Romance	1943
3	Sapiens	Yuval Noah Harari	443	Non-fiction	2014
4	The little prince	Antoine de Saint-Exupéry	96	Fable	1943
5	Zero to One	Peter Thiel	224	Business	2014
6	Life after life	Jill McCorkle	352	Novel	2013
7	Life after life	Kate Atkinson	529	Novel	2013

Table 2.1: Books Relation

Note that the combination of *Author* and *Title* is the primary key of this relation since different authors can have written other books with the same name.

The Scan Operator

To obtain the desired result, unless an index is constructed for the *Year* and/or *Genre* attributes, the DBMS will need to iterate over the entire *Books* relation scanning each entry and passing it forward to the next operator. The cost of this operation can be anticipated as the size of the relation is known.

The Selection Operator

As the bookstore manager is not interested in novel books published before the year 2000, the *Select* operator (represented with the *Where* clause in the query) is used to filter the unwanted rows. Table 2.2 displays the resulting records from this operation that will serve as input for the next one.

Filtered Books Relation					
#	**Title**	**Author**	**Pages**	**Genre**	**Year**
6	Life after life	Jill McCorkle	352	Novel	2013
7	Life after life	Kate Atkinson	529	Novel	2013

Table 2.2: Novels Published after year 2000

The Project Operator

While the attributes *Genre* and *Year* have been employed for filtering out unwanted rows, *Pages* is the sole attribute essential for computing the desired output. Consequently, it's projected into a new relation, creating a single-attribute structure. The resulting projection of the attribute *Pages* in Table 2.2 is shown in Table 2.3; nevertheless, it is not

yet the desired output from the issued query. Therefore, it is forwarded to another operator with the capability of calculating the average.

Pages
352
529

Table 2.3: Number of pages in novel books published after year 2000

The Aggregate Operator

To calculate the average value of the *Pages* attribute based on the data in Table 2.3, an *Aggregate* operator must be applied. The result of this operation, which is a scalar, serves as the desired output for the query presented in Listing 2.1, and it is displayed in Table 2.4.

AVG Pages
440.5

Table 2.4: Average number of pages in novel books published after the year 2000.

In a real workload, queries can be much more complex than the sample example described above and therefore some optimizations must be applied. Initially, a query issued by a client will be compiled into a tree of logical operators called a logical query plan. This plan is not yet optimized and does not carry information about what algorithms will be employed in each operator. Before reaching its final form, this logical query plan will go through several rounds of optimization rules until no further improvement is possible. The outcome of such a process is an optimized physical query plan containing all the information about operators and their selected algorithms. This physical plan will then be used to execute the query using the query engine module.

It is important to note that the intermediate results will flow through the connectors (edges in the tree) from the bottom of the plan to the top until the final result is produced. The input and output of each operator is a table. Hence, operators can easily connect and pass their results to the next operator in the tree.

The logical query plan for this query would be as shown in Figure 2.1.

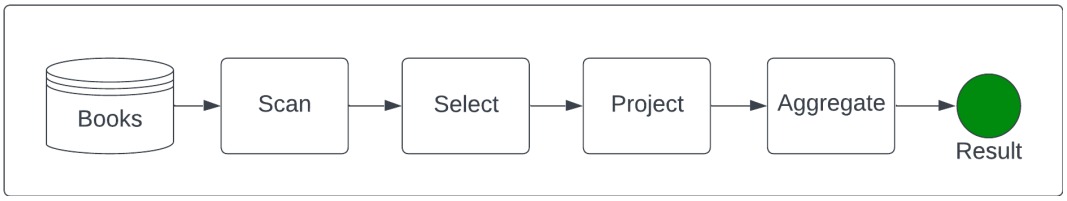


Figure 2.1: Logical Query Plan for calculating the average number of pages.

Join Operator

Expanding on the previous example, the bookstore's manager seeks to identify the most profitable book genre among the ones sold. To achieve this, the manager elaborates the SQL query presented in Listing 2.2 to issue to the DBMS and obtain this valuable information.

```
SELECT B.Genre, SUM(S.Price) FROM Books B
             JOIN Sales S ON B.Author = S.Author and B.Title = S.Title
             GROUP BY B.Genre
```

Listing 2.2: Join and Aggregate

Note that the *Book* relation does not hold any data about sales; instead, there is another relation in the database called *Sales* with one entry for each book unit sold. An example of this relation is shown in the table Table 2.5 below:

Sales Relation				
#	**Title**	**Author**	**Price**	**Date Sold**
1	1984	George Orwell	12.00	2022-01-01
2	The Fountainhead	Ayn Rand	21.00	2022-03-01
3	Life after life	Jill McCorkle	8.00	2022-04-01
4	1984	George Orwell	12.00	2022-01-01
5	Life after life	Kate Atkinson	20.00	2022-05-02
6	Zero to One	Peter Thiel	12.00	2022-06-01

Table 2.5: Sales Relation

To produce the desired result, the DBMS must join these relations to find each book's sales information. Note that the *Sales* relation and the *Books* relation share some common attributes. These attributes can be used to relate the entries in each relation and will be called throughout this thesis *join attributes*. In this example, the only *join attributes* are the Primary Key in the *Books* relation used as Foreign Key in the Sales relation.

A careful reader would notice that there are two different entries in *Books* with the same value for the *Title* attribute, but they can be distinguished by the *Author* attribute. Table 2.6 illustrates the result of a *Join* operation using the *join attributes Title* and *Author* in both relations.

Book_Sales Relation							
#	**Title**	**Author**	**Pages**	**Genre**	**Year**	**Price**	**Date Sold**
1	1984	George Orwell	328	Romance	1949	12.00	2022-01-01
2	The Fountainhead	Ayn Rand	753	Romance	1943	21.00	2022-03-01
3	Zero to One	Peter Thiel	224	Business	2014	12.00	2022-06-01
4	1984	George Orwell	328	Romance	1949	12.00	2022-05-02
5	Life after life	Jill McCorkle	352	Novel	2013	8.00	2022-04-01
6	Life after life	Kate Atkinson	529	Novel	2013	20.00	2022-05-02

Table 2.6: Books and Sales join result

Book records are only related to *Sales* records if both *join attributes Title* and *Author* match. Such operation is called *equijoin*; entries will only be combined and outputted if they have the exact same value for the *join attributes*. An *Aggregate* operator can be applied over the resulting relation to group the entries by *Genre* and compute the sum of the *Price* value for each group of entries.

The execution plan for this query is visually depicted in Figure 2.2.

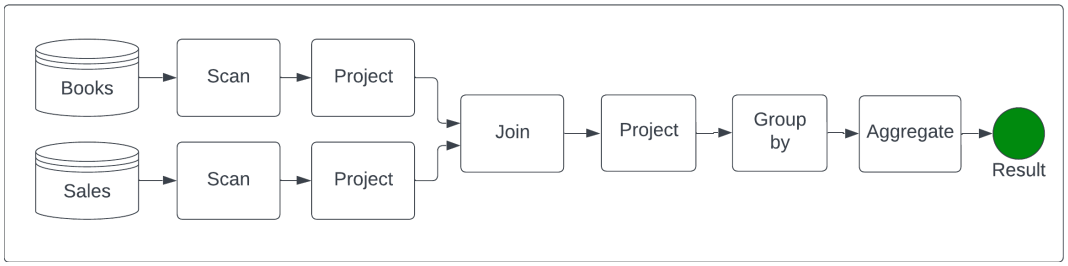


Figure 2.2: Logical Query Plan for equijoin and aggregate.

Several algorithms can be utilized to implement a Join operator; the simplest and most naive one, Nested Loop Join, comprises of comparing the *join attributes* of each entry in one relation to all entries in the other. Although correct, the number of comparison operations necessary in this algorithm is calculated by the product of entries in each relation; that number can be significantly large when joining large relations. A family of algorithms that can more efficiently implement this operator, the Hash-Based Join algorithms, is presented in the next section.

2.2 Introduction to Hash-Based Join Algorithms

The Hash-Based Join family comprises a group of algorithms that implement the Join operator using hash functions to relate tuples with equal *join attributes*. All algorithms described in this section that are capable of handling relations larger than the available memory use a second hash function to divide the input data into smaller partitions. Before

introducing algorithms that use the partitioning technique, we will present the simplest algorithm in the Hash-Based Join family, the In-memory Hash-Join.

In-memory Hash-Join Algorithm

As its name suggests, the In-memory Hash-Join algorithm is employed in cases where it is feasible to store a hash table that fully maps the smallest relation being joined into memory. We will extend the bookstore example to illustrate how the In-memory Hash-Join algorithm works.

In our example, the hash function employed to index the hash table is defined as follows: When provided with a string input, this function calculates the sum of the ASCII values of all characters and applies a modulo operation with a divisor of 256. The pseudocode for this function is outlined below:

```
hashFunction(string):
    sum = 0;
    for character in string:
        sum += getAscCode(character);
    return sum (mod 256)
```

Listing 2.3: Hash Function

Both *join attributes*, *Author* and *Title*, will be combined into a single string and used as the hash function input; the resulting hash table for the *Book* relation would be as shown in Table 2.7, note that pointers to the entries are represented as **RelationName_#**.

Hash Table	
Hash (key)	**Pointers(value)**
0	-
⋮	⋮
36	Books_1
⋮	⋮
132	Books_2
⋮	⋮
137	Books_4
⋮	⋮
61	Books_3
⋮	⋮
205	Books_6
⋮	⋮
222	Books_7
⋮	⋮
255	-

Table 2.7: Hash Table

The described steps are called the Build phase of the In-memory Hash-Join algorithm, and the relation used to build the hash table is referred to as the Build relation or simply as **R**. The next phase, the Probe phase, will test each

tuple from the second relation, called Probe relation or simply **S**, against this hash table.

During the Probe phase, the **S** relation will be scanned, and its *join attributes* will be hashed and looked up in the hash table; we refer to this comparison as *probe*. If a match occurs, the tuple from **R** relation and **S** relation with the same **join attributes** will be combined and outputted.

For example, the hash generated by applying the hash function (Listing 2.3) to the *join attributes* of the first entry in the *Sales* relation is 36. That value matches the entry in the hash table that points to *Book_1*. The Join result for this pair of entries is precisely the first line in Table 2.1. This somewhat simplified example considers no hash collisions but illustrates how the In-memory Hash-Join algorithm can significantly reduce the number of comparison operations. The pseudocode snippet displayed in Listing 2.4 details this algorithm:

```
#Input:     File R -> reference to file containing relation R
#           File S -> reference to file containing relation S
#           List<attributes> list of join attributes
#Return :   List of pairs of tuples
Function inMemoryHashJoin(file R, file S,list<attributes> joinAttributes,file output)
    #Build Phase
    let hashTable = new HashTable
    for tuple b in R
        bTuple = loadFromDisk(b,R)              #Load Tuple from Build relation file to memory
        hashTable.insert(bTuple,joinAttributes) #Insert Tuple into hash table

    #PROBE Phase
    for tuple p in S
        pTuple = loadFromDisk(p,S)
        let matches = hashTable.probe(pTuple)   #Load Tuple from Probe relation file to memory
        for m in matches
            output.write(<m,p>)
    return output
```

Listing 2.4: In-memory Hash Join

Although the above examples present important concepts, they still don´t represent a real scenario. Next, we will discuss algorithms capable of dealing with relations larger than the available memory.

Dealing With Large Relations

Real DBMSs routinely handle relations with entries numbering in the millions or more. A prime illustration of such extensive datasets is Amazon's book listings and sales records that, as of March 2023, had over 32 million books listed [3], and in the year 2020 alone, counted near one billion sales [2].

Joining entries from such extensive relations demands a substantial memory allocation. However, even when ample memory is accessible, the DBMS must take into account the possibility of concurrent query processing. Allocating all available memory to a single query could impede the execution of others.

In the subsequent subsections of this chapter, we will introduce various Hash-Based Join algorithms designed to tackle the challenge of insufficient memory to accommodate both the **R** relation and the hash table in memory. To simplify the representation of size, we will utilize the modulus notation. Thus, $|M|$, $|S|$, and $|R|$ denote the available memory size for an operator, the size of the largest relation, and the size of the smallest relation, respectively.

Simple Hash-Join Algorithm

The Hash-Based Join algorithm described here introduces a technique that makes possible joining relations that exceed the available memory capacity possible. Such a technique uses an additional hash function, called the Split function, to divide the large input relation into smaller-sized groups of tuples called partitions.

The algorithm initiates its operation by utilizing the Split function to divide the input relation **R** into two distinct groups called partitions. The primary objective is to keep the size of the first partition sufficiently small to ensure that a hash table capable of mapping all its tuples can be accommodated fully in memory, this specific partition is referred to as a memory-resident partition. Meanwhile, the second group, denoted as a spilled partition, is written to the disk and reserved as input for subsequent processing rounds. The hash table mapping the tuples from the memory-resident partition is constructed following the procedure outlined in section 2.2.

Similarly, the relation **S** undergoes partitioning using the same Split function, and tuples associated with the memory-resident partition are probed against the hash table. In contrast, tuples mapped to the spilled partition are written to the disk to be carried forward to the subsequent processing round.

The pseudocode snippet in Listing 2.5 outlines this algorithm more clearly; it is worth noting that for improved efficiency, the In-memory Hash-Join algorithm is applied when it is feasible to execute this operation in memory.

```
#Input:     File R -> reference to a file containing relation R
#           File S -> reference to a file containing relation S
#           List<attributes> list of join attributes
#           int M -> Bytes of Memory available for this operation
#           int F -> Fudge Factor

#Return :   List of pairs of tuples
Function simpleHashJoin(file R, file S, list<attributes> joinAttributes, int M, int F,file output
    )
    if (R.size() * F <= M.size())                       #Recursive Condition Break
        return inMemoryHashJoin(R,S)                     #Refer to Listing 2.4
    else
        #BUILD Phase
        passOver = new file
        splitFunction = selectSplitFunction(R,M,F)      #Select proper Split function
        for frame in R.readFrame()
            for tuple b in frame
                if(splitFunction(bTuple))
                    hashTable.add(bTuple)
                else
                    writeToDisk(bTuple,passOver)            #Write tuple to disk for next round
        #PROBE Phase
        for frame in S.readFrame()
            for tuple frame in S
                let matches = hashTable.probe(p)
                for m in matches
                    output.write(<m,p>)
        return output.concat(simpleHashJoin(passOver,S,M,F)    #Recursive Call
```

Listing 2.5: Simple Join

Note that the spilled partition from the **R** relation is fully written to the disk and fully reloaded during the next round. Consequently, if this input relation is much larger than the memory available, the number and volume of I/O

operations will be considerably large, decreasing the performance of this algorithm. On the other hand, the Simple Hash-Join algorithm performs particularly well when the size of relation **R** is only slightly larger than the size of **M**. Next, an algorithm designed to handle relations much larger than the available memory is presented.

Grace Hash-Join Algorithm

Another Hash-Based Join algorithm, the Grace Hash-Join [12], was introduced by Kitsuregawa, Tanaka, and Moto-Oka in 1983. Grace Hash-Join is designed for joining relations significantly larger than the available memory. This algorithm employs a Split function to partition the input data into multiple pairs of smaller partitions and then uses these pairs as input for the In-memory Hash-Join algorithm to those pairs. Although this was this algorithm's first version, we will refer to a variant described in [15] by Shapiro. Unlike the Simple Hash-Join, the **S** relation is partitioned before any probe operation. A proper Split function is selected in such a way that the **R** relation will be split into **n** partitions called R_i where

$$n = \sqrt{F \times |R|}$$

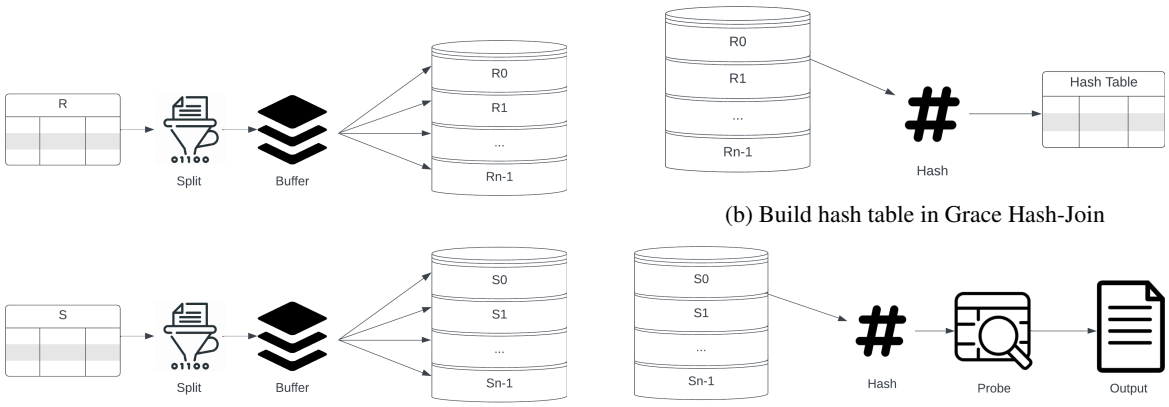
$$|R_i| * F \leq |M| \quad \forall \quad 0 \leq i < n$$

The memory available for the Join operation is divided into **n** buffers, one for each partition to store tuples assigned to them. The relation **R** is then scanned, and each of its tuples is mapped to a partition and moved to its designated output buffer. If there is not enough space for the mapped tuple in its designated buffer, tuples stored in that buffer are spilled to the disk, and the buffer is cleared so other incoming tuples mapped to this partition can be inserted.

This procedure is executed for **R** and then for **S**, generating **n** partitions for each relation. We will call these partitions S_i and R_i where $0 \leq i < n$. The steps described so far are referred to as the Partitioning phase, as no partitions can fully fit in memory the hash table is not built during this phase.

During the next rounds, one at a time, pairs of partitions R_i and S_i are reloaded into memory. A hash table is built with tuples from R_i as shown in Figure 2.3 (b), and tuples from S_i are probed against this hash table (Figure 2.3 (c)). Grace Hash-Join performs better than the Simple Hash-Join whenever the size of **R** is significantly larger than the size of **M** available for the Join operation. Regardless of the number of tuples in **S**, they are only read and written to the disk once in Grace Hash-Join, while in Simple Hash-Join, large relations can have their tuples read and write several times depending on the number of rounds.

Figure 2.3: Grace Hash-Join



(a) Partitioning phase of Grace Hash-Join

(b) Build hash table in Grace Hash-Join

(c) Probe in Grace Hash-Join

Hybrid Hash Join Algorithm

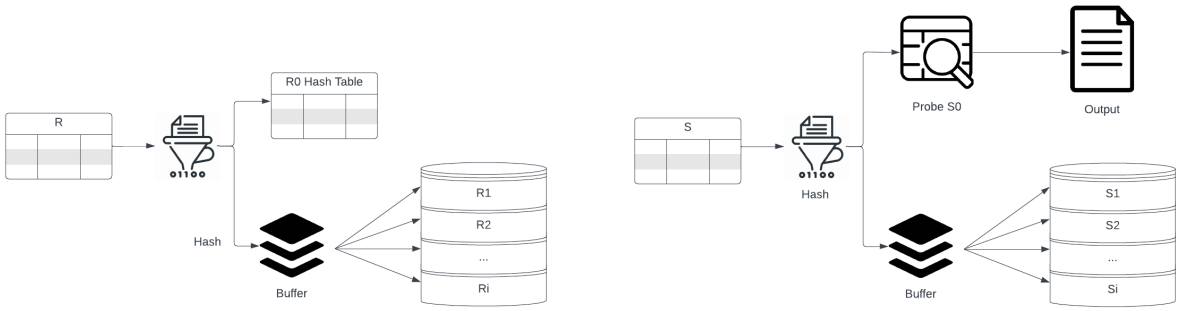
Hybrid Hash Join (HHJ) is a combination of both Simple Hash Join and Grace Hash Join; it was first introduced by DeWitt et al. in [6], this algorithm was designed to handle relations that are not too large or too small compared to the available memory.

This Hash-Based Join algorithm has a Partitioning phase similar to Grace Hash-Join. However, instead of writing all partitions to the disk, one partition of **R** is kept memory-resident with its hash table, and all other partitions have at least one frame as an output buffer. Figure 2.4a illustrates this design choice that reduces the need to write an entire partition to the disk and reload it in the next round.

During the Partitioning phase, a block of memory is reserved for storing a hash table for the first memory-resident partition R_0 while the rest of the available memory is equally divided into buffers for each partition R_i where $0 < i < n$. By the end of the first round of the Partitioning phase, also called the Build phase, a hash table is built mapping all tuples in partition R_0. The other partitions are completely written to the disk to be later used as input for the next rounds, finalizing the Build phase, and then the Probe phase starts. The HHJ algorithm is designed based on the premise that all partitions R_i will have similar sizes. However, the actual size of each partition depends on the chosen Split function's quality and data skewness.

After partitioning the **R** relation and building the hash table, the algorithm will partition the relation **S**. Tuples mapped to the first partition, which is the memory-resident, will be probed against the hash table as shown in Figure 2.4b. **S** relation's tuples mapped to other partitions are moved to the disk.

Figure 2.4: Hybrid Hash Join



(a) Partitioning phase of HHJ for **R**

(b) Partitioning phase of HHJ for **S**

While this algorithm performs well when the **R** relation is evenly distributed among partitions, such uniform distribution is rarely encountered in practice. Consequently, there is no guarantee that the selected partition designated to reside in memory will fit within the available memory space. In the following section, we will introduce an algorithm designed to address the challenges posed by unevenly distributed partitions.

Dynamic Hybrid Hash Join Algorithm

Most often, the Join operations receive input relations, which are outputs of previous operators; hence, maintaining statistics about these input relations, such as their accurate sizes and attribute value distributions, is not always feasible. Additionally, *join attribute* values may be skewed; therefore, finding the perfect split function to produce equal-sized partitions may not be trivial or even possible. The consequence of having uneven partitions is that it is hard to predict the exact size of the first partition. Having a miss calculation or a bad Split function can cause this algorithm to turn into a Grace Hash-Join algorithm when all of its partitions get spilled by the end of the Partitioning phase. To solve this issue, another Hash-Based algorithm was proposed by Nakayama, Kitsuregawa, and Takagi in their paper "Hash-Partitioned Join Method Using Dynamic Destaging Strategy" [13]. This thesis will reference their Dynamic Destaging HHJ as Dynamic Hybrid Hash Join (DHHJ).

This algorithm aims to optimize memory usage, reducing the number of I/O operations and increasing sequential I/O by reducing random I/O. This algorithm chooses no partition as a memory- or disk-resident partition a priori. Instead, all partitions are given equal opportunity to grow. In case of memory scarcity, one of the memory-resident partitions will be chosen as a victim to be spilled to the disk. Authors of [10] chose to prioritize spilling partitions in the following order:

- Already spilled partition.
- Memory-resident partition using most memory.

This order increases the possibility of keeping other partitions as memory-resident by releasing the maximum number of frames possible and writing data to disk sequentially in a large batch. DHHJ moves the frames returned by a partition after spilling to a list of unused frames called memory pool. Memory frames in this list can be allocated to other partition's buffer as it grows.

Although DHHJ can better use the memory assigned for that operation, it is designed to operate with a static amount of memory, being incapable of receiving or releasing memory to other concurrent operators. Next, we will introduce the concept of memory adaptive algorithms.

Memory Adaptive Hash-Based Join Algorithms

Memory adaptable capabilities for Hash-Based Join algorithms were first studied by Zeller and Gray [18], where they introduced a memory adaptive Hash-Based Join algorithm. Although pioneer, their algorithm will not be further examined in this thesis, as it is limited to adapting to memory fluctuations during the Build phase. Instead, we will focus on algorithms capable of adapting to memory fluctuations during both the Build and the Probe phases.

Memory Adaptive Hash-Based Join algorithms must handle two contention scenarios: **memory contraction** and **memory expansion**. **Memory contraction** is when the resource manager requests the operator to release some memory to benefit other concurrent operators. In contrast, memory expansion is when memory becomes available for the operator.

How the operator interacts with the resource manager can determine how and when these memory adaptions will occur. Two important Memory Adaptive Hash-Based Join algorithms and memory fluctuation scenarios are better explained in chapter 3 and chapter 4. Also in Chapter 3, the idea of a software component that manages the memory allocated to different operators is presented.

2.3 Apache AsterixDB

Apache AsterixDB is an Open Source Big Data Management System, designed to manage large amounts of semi-structured data in a distributed setting. An instance of AsterixDB has one Cluster Controller (CC) and one or more Node Controller (NC)s. Each NC and can manage one or more *data partition*. The CC is responsible for query compilation, query distribution among NCs, and managing the functionality of the entire cluster. Figure 2.5 illustrates this BDMS architecture.

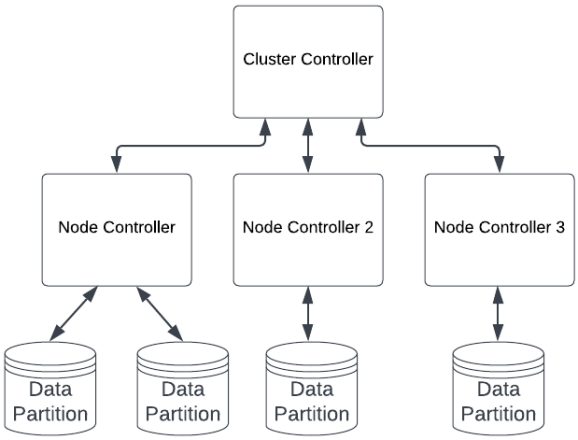


Figure 2.5: AsterixDB architecture

Throughout this thesis, only one NC and one *data partition* will be used to isolate variables, and we will only consider changes in *memory budget* for a single Join query.

Iterator Desing Pattern

Asterix DB's operators are implemented similarly to the Iterator Design Pattern [7]; the application of this pattern for DBMSs was first introduced by Graefe in [8]. Adopting this pattern facilitates the integration of new operators into an already built DBMS. A class implementing the Iterator Pattern only exposes three methods: Open(), Next(), and Close(). All the concrete implementations are hidden, making it a decoupled pattern. This pattern works great for processing large data sets, for example, in a large byte stream or in a database relation with multiple data pages. This pattern provides a structure in which after the operator's initialization (Open()), data is processed one unit at a time by (Next()), only starting the next data unit's processing after the current one is finished, finally, the operator is finished invoking the Close() method. The diagram below Figure 2.6 depicts the Open, Next, and Close method calls for an operation implemented over the Iterator Pattern processing multiple pages.

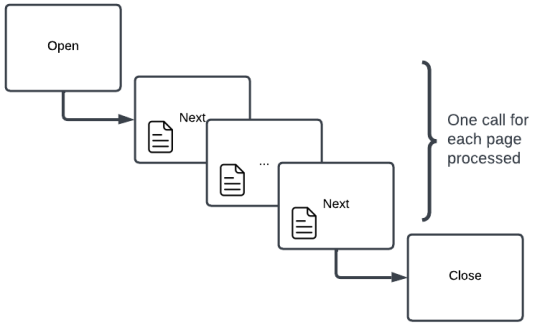


Figure 2.6: Iterator Paradigm

Chapter 3

Related Work

This chapter reviews important papers related to Memory Adaptive Hash-Based Join algorithms. The presented algorithms and experiments serve as a foundation for our implementation design and analysis. All described algorithms in this chapter were empirically evaluated in simulated environments. Although these environments are valid for research, they have limitations that we address in the following chapter.

3.1 Partially Preemptible Hybrid Hash Join Algorithm

First introduced by Pang, Carey, and Livny in [14], the pphj presents relevant ideas to memory adaptive Hash-Based Join algorithms. The PPHJ algorithm uses excess frames not as output buffers for partitions, like DHHJ. Instead, excess frames are used as I/O buffers. In PPHJ, spilled partitions can have at most one frame, and any other frame that is not allocated to a memory-resident partition is used as an I/O buffer called the spooling area. This spooling area is managed by the Least Recently Used policy, and frames buffered in the spooling area are flushed to disk in blocks of several frames to reduce disk seeks. In PPHJ, partitions are spilled or reloaded based on their index in descending order during the Build and the Probe phase. This way, partition 2 is preferred to be spilled or reloaded before partition 1. The authors also introduced a restoring mechanism that made PPHJ capable of adapting to memory expansion during the Probe phase. This mechanism reloads spilled partitions from **R** during the Probe phase into memory whenever possible. The results of their memory fluctuation experiments led them to conclude that rapid memory variations could harm the PPHJ performance when the restoring mechanism is in use.

Review

Introducing a restoration mechanism during the Probe phase and the experiments conducted are relevant to memory adaptive Hash-Based Join algorithms. Experimentation with different memory fluctuation ratios showed this is an important factor to consider since these variations may happen quickly or slower depending on the application and environment. There is possibly no ”one size fits all” solution regarding adopting restoration or not.

Another interesting mechanism presented as a variation of PPHJ avoids spilling frames from partitions with higher chances of being reloaded in the near future. Instead of spilling frames from the spooling area using LRU policy, frames of partitions with higher indexes are spilled first. The reason is that partitions with lower indexes will be memory-resident during the Probe phase.

3.2 Memory-Contention Responsive Hash Join Algorithm

Memory-Contention Responsive Hash Join (MCRHJ) [5] is an iteration of Dynamic Hybrid Hash Join (DHHJ) [13] capable of gracefully adapting to new memory scenarios. Unlike the PPHJ algorithm presented in the previous section, MCRHJ uses excess memory frames as output buffers for partitions. Although partitions can grow, the authors defined a limit of frames to every build partition's buffer, called C_{tgt}. A variation of MCRHJ also sets a limit to the size of the output buffers for the Probe partitions called C_{eff}. These bounds directly impact the response time of the proposed algorithm as smaller values of C_{tgt} increase the number of I/O operations, and larger values showed no benefit empirically. Based on the original DHHJ, the author described three possible scenarios during a memory contraction; the first is when free memory frames are available in the memory pool. Reclaiming those frames is virtually instantaneous since no I/O operation is required. The second possible scenario is when no frames are available in the memory pool and a spilled partition has a buffer larger than one frame. This partition is preferred to be spilled, returning all but one frame to the memory pool and making them available for release. The third possibility is having to spill a memory-resident partition. In that case, the largest partition is preferred to be spilled for the same reasons mentioned in section 2.2.

Four variants of MCRHJ are presented in the paper; the first one, Fixed Cluster (FixN), uses a fixed-sized buffer of N frames for each partition. This variation is not memory adaptive and serves as the basis for comparison.

The second variation is applied to both MCRHJ and FixN; authors called "Balance large cluster size with restoration" (:bal). This variation affects only the Probe phase, as they set a limit to the size of the partition's output buffers to C_{eff} and apply the restoration mechanism as described in the previous section. Table 3.1 summarizes the MCRHJ variations.

	Fix**N**	Fix**N**:bal	MCRHJ	MCRHJ:bal
Build Buffers	N	N	1 to C_{tgt}	1 to C_{tgt}
Probe Output Buffers	1 to C_{tgt}	1 to C_{eff}	1 to C_{tgt}	1 to C_{eff}
Restoration	No	Yes	No	Yes

Table 3.1: MCRHJ Variations

Experiments

The authors simulated the response time of joining relations R and S of size 5 and 50MB, respectively. To account for memory variation, the following scenario was introduced:

Let **M** be the system's memory; the memory allocation for the join operator varies uniformly from 80-100% of M during 80% of time and from 0-100% of **M** for the other 20%. A high concurrency situation was defined as $|M| = 1MB$ and a low concurrency situation as $|M| = 8MB$. Experiments evaluated the response time for all variations of MCRHJ and also PPHJ with and without restoration.

Another experiment to evaluate the responsiveness of the memory adaptive algorithms to memory fluctuation was conducted. In this experiment, the memory contention scenario changed based on an exponential time distribution for different mean values, from 0.1s to 10s.

Davison and Graefe concluded in [5] that the proposed algorithm MCRHJ:bal outperformed the other variants, and Partially Preemptive Hash Join [14] in response time for different memory contention scenarios and responsiveness to memory fluctuation.

Review

The proposed memory adaptive algorithm empirically demonstrated gains in efficiency, even though they have no theoretical performance bounds to the best of our knowledge. Theoretical and empirical studies of memory adaptive algorithms are scarce, even though the need for such algorithms is increasing. We speculate that the reason for that is the difficulty of designing, implementing, and analyzing such algorithms. Davison and Graefe's work still holds relevance even decades after it was first published, serving as a reference for other papers related to memory adaption [1]. Despite the significant contribution, simulations are not as accurate as actual implementations. Some factors are hard to consider in a simulation, such as the effects of the file system's cache on the algorithm. Other aspects, for example, the maintenance of the hash table and a more careful examination of the choices of C_{eff} and C_{tgt}, were not well described in their paper, possibly for simplification.

The original MCRHJ description also does not state clearly when tuples from the **S** relation reserved to the next rounds are probed if a restoration happens. The next chapter elaborates on the causes of this situation.

3.3 Resource Broker

The above-described algorithms are not responsible for defining their own memory budget; instead, they must communicate with another module capable of distributing this resource. The Resource Broker is a software component capable of distributing the available memory among concurrent operators. The here-described Resource Broker more closely resembles the one proposed in [4]. This idea was first introduced in [4]. The Resource Broker can allocate

different resources such as IO bandwidth, memory, network bandwidth, or computing time for operators. Whenever a new query arrives at the DBMS, the resource manager will evaluate if it is possible to execute this query, given the system's available resources. The Resource Broker will move it to the admission queue if the system has enough resources to execute that query. A query can enter the admission queue at any time; the Resource Broker will admit that query to be executed when the minimum amount of necessary resources are available.

Once a query is admitted, its operators must frequently bid for resources. These bids are based on how much the resource will affect the query's execution time. For example, if a query has a memory-intensive operator, this operator can leverage extra memory; therefore, the bid will be higher. The Resource Broker will allocate resources based on the value of the bids to maximize its profit, allocating more resources to the higher bidders. This is a simple but effective approach. It is hard to predict future workloads and how they will impact resource usage; using a broker system can simplify and optimize resource allocation.

Although this process seems similar to an auction, it does not require the operators to wait for the broker to collect bids from all operators and distribute the resources. This broker algorithm operates with a short-term reserve of resources that it can allocate based on the value of each bid. Also, the Resource Broker can "buy" back resources from operators with low bid values. This framework is illustrated in the Figure3.1.

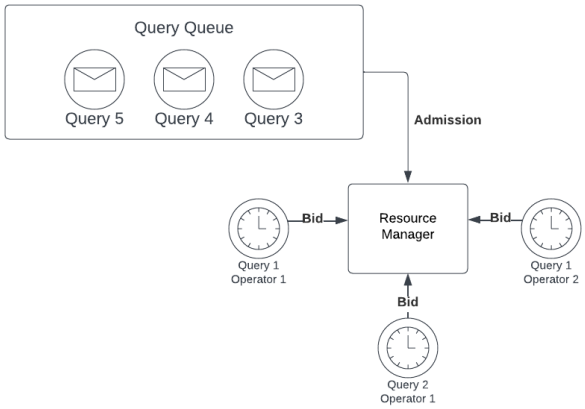


Figure 3.1: Resource Broker Framework

The currency used in this system is ROC (return over consumption), which is given by the following equation:

$$ROC = \frac{benefit}{cost} = \frac{T(M_{min}) - T(M)}{M \times T(M) - M_{min} \times T(M_{min})}$$

M is the memory allocated to the operator, T(M) is the operator's response time estimative, and M_{min} is the minimum memory for that operation. The ROC equation generates a hook-shaped curve. The Resource Broker can allocate resources to the operator, and the operator aims to maximize the ROC. The operator always bids for the highest ROC it can achieve. This thesis does not aim to explain in detail this resource manager framework. Still, an overview

of how it works is necessary to understand the decisions of frequency of changes in the memory budget covered in Chapter 4. Also, Apache AsterixDB does not implement a Resource Broker, at least until the date of this writing. A simulation of this software component is implemented for experimentation, Chapter 5 has a complete description of this simulated component.

Chapter 4

Memory-Contention Responsive Hash Join

This chapter will present an implementation of memory adaptive Hash-Based Join algorithm that implements some features of the MCRHJ algorithm presented in [5] and introduces new features related to interactions with a resource manager component. As the proposed algorithm uses as a foundation a version of DHHJ already built in Apache AsterixDB, this current implementation will be presented in detail to put the basis of understanding for the memory-adaptive implementation of it. Additionally, we describe various memory contention scenarios that can possibly occur during the execution of the join algorithm and motivate the implementation of key features in the proposed algorithm. Before proceeding with the current implementation details, Table 4.1 presents some essential definitions and terminologies used throughout this chapter.

Definitions	
Relation **R**	The smallest relation being joined, the input to Build Phase
Relation **S**	The largest relation being joined, the input to Probe Phase
Split Function	A hash function used for partitioning data
Join Attributes	Relation attributes that are used for joining two relations together
Frame	A fixed-sized and configurable amount of contiguous bytes.
Buffer Manager	The module responsible for allocating and deallocating frames to Partitions' buffers
$\lvert \quad \rvert$	The modular notation represents size, for example $\lvert R \rvert$ represents the size of relation **R**

Table 4.1: Terminologies and Definitions

4.1 Optimized Dynamic Hybrid Hash Join in AsterixDB: Current Design

Optimized Hybrid Hash Join (OHHJ) [11] is used as the default join algorithm for equi-join queries in AsterixDB which its implementation is inspired by DHHJ design. Similar to other Hash-Based Join algorithms presented in Chapter 2, the Optimized Hybrid Hash Join algorithm is a two-phase blocking join operator as the second phase, the Probe phase, does not start unless the first phase, the Build phase, is finished.

Figure 4.1 illustrates the outline of the two phases of OHHJ implemented over the Iterator design pattern described in subsection 2.3. We will detail both phases of OHHJ in the following subsections.

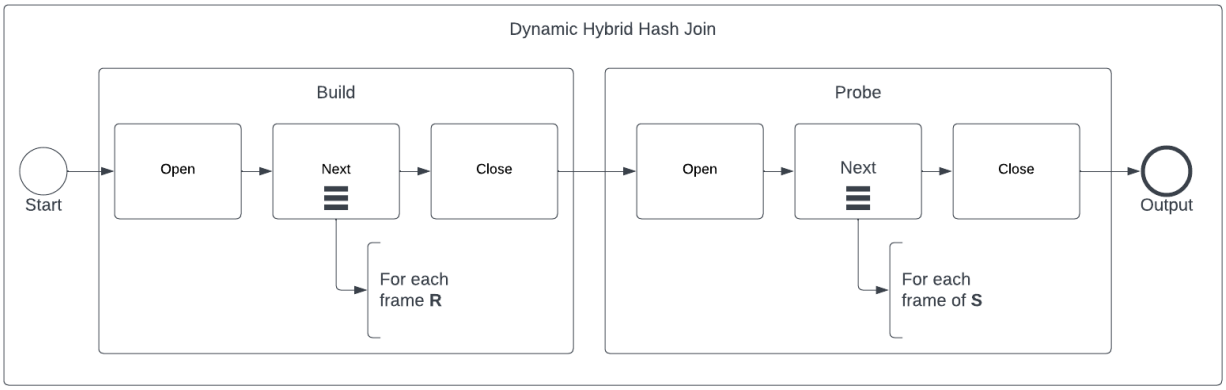


Figure 4.1: Current Optimized Hybrid Hash Join Design

4.1.1 Build Phase

During this phase, the relation **R** is divided into a number of partitions **B** to be later probed. The value of **B** is determined based on the size of the input relation **R** and the memory available for this operation; such value is crucial for selecting a Split function that will ideally distributes tuples evenly among partitions.

Once the value of **B** is determined, the **R** relation is scanned frame by frame, with each tuple within a frame being assigned to a partition based on its hash value as calculated by the Split function. Throughout this process, if a partition's buffer becomes insufficient to accommodate a newly mapped tuple, one or more partitions may be written to the disk to free up memory. Finally, when all frames are completely processed, a hash table will be created, and records from the memory-resident partitions will be inserted there to be probed. Next, we will describe further the design and implementation details of the Build phase of OHHJ.

Open() Function

A great part of the initialization of the OHHJ algorithm is done during the Open() function in the Build phase. In Apache AsterixDB, the amount of memory $|M|$ assigned to a Join operator is defined by the user and passed to the operator during its initialization.

$|M|$, also referred to as the memory budget, is used to define **B** according to the equation introduced in [15] and mentioned below. **B** is calculated in such a way that allows this join operation to be concluded in at most two rounds.

$$B = \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil \quad \text{(Equation 4.1)}$$

Another study by Jahangiri, Carey, and Freytag [9] empirically concluded that the *minimum* number of *Partitions* to be used in a DHHJ algorithm should be 20 to avoid huge I/O penalties for spilling large partitions due to lack of accuracy of statistics about the inputs and their attributes' distributions. AsterixDB uses this lower bound on its number of partitions **B** in its current implementation of OHHJ. Therefore, we can re-write the equation:

$$B = Max\left(20, \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil\right) \quad \text{(Equation 4.2)}$$

After calculating the number of partitions, memory for allocating one frame to each partition will be calculated and set aside to avoid excessive frame stealing between partitions.

The last step of the Open() function is to choose a proper Split function. Having the number of partitions **B** defined, buffers initialized, and a Split function selected, the algorithm can start processing the frames from **R** relation.

Next() Function

Relation R will be read frame by frame into the memory to be processed by the Next() function in the Build phase. Tuples contained in each frame will be processed and hashed using the Split function to find their destination partition. Figure 4.3 presents a diagram of this step.

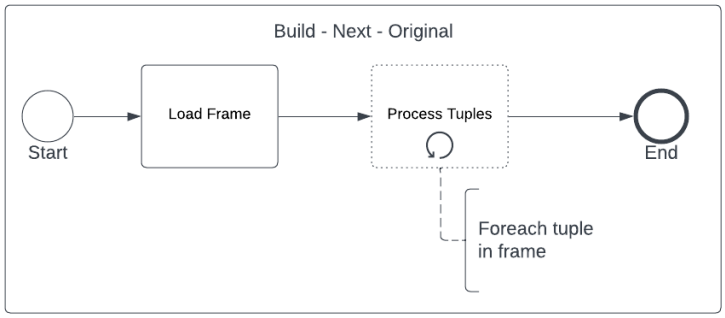


Figure 4.2: Current Build Phase, function Next diagram

The algorithm will then try to insert this tuple into the designated partition's buffer. The tuple insertion will be successful under three situations:

1. The last frame in the destination partition has enough space to accommodate the tuple.

2. There is an unused frame in the memory pool; in that case, a new memory frame will be appended to the destination partition, and insertion is retried.

3. The memory budget is not fully utilized; in that case, a new memory frame will be appended to the destination partition, and insertion is retried.

In case of an unsuccessful insertion, one memory-resident partition will be selected as the victim partition to be written to the disk. The frames released by evicting the victim partition will be returned to the memory pool so other partitions can borrow them. The process of flushing tuples from a memory buffer into a temporary file in the disk is called spilling. Any partition with at least one tuple written to the disk is called a spilled partition.

Figure 4.3 and the pseudocode snippet in Listing 4.1 provide further details on how tuples are processed during the Build phase.

```
#Input:     Tuple t -> Tuple to be processed
#           BufferManager bufferManager -> Module responsible for managing frames
#           List<Partitions> partitions -> List of partitions
Function processTupleBuildPhase(tuple t,BufferManager bufferManager,List<Partitions> partitions)
    pId = SplitFunction(t);
    partitionToInsert = partitions.getById(pId)
    while(partitionToInsert.insertTuple(t) == false)
        if(bufferManager.hasFrame())
            bufferManager.assignFrameToPartition(partitionToInsert)
        else
            partitionToSpill = selectPartitionToSpill(partitions)
            spillPartition(partitionToSpill,bufferManager)

Function selectPartitionToSpill(List<Partitions> partitions)
    return partitions.orderBy(Spilled First).thenBy(Larger Buffer First).first()

Function spillPartition(Partition partition,BufferManager bufferManager)
    releasedFrames = partition.spill()
    bufferManager.retrieveFrames(releasedFrames)
```

Listing 4.1: Pseudocode for Process Tuple Build Phase in current implementation.

Under the special case of having tuples larger than a regular-sized frames, those large tuples will be directly spilled to disk and their partition will be marked as a spilled partition. A suggestion for further study regarding this situation is made in Chapter 5.

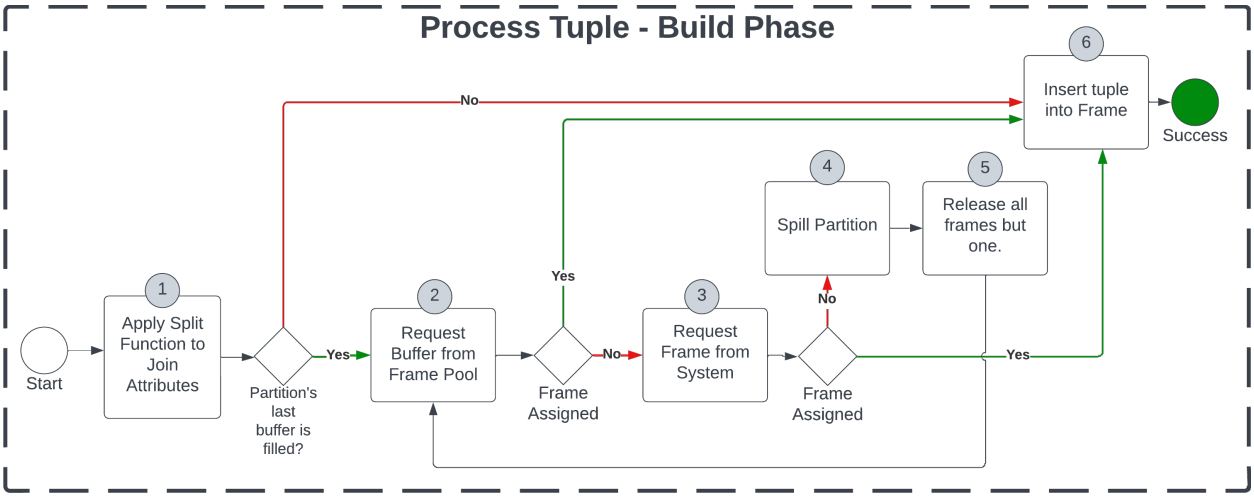


Figure 4.3: Tuple processing during the Build Phase in Current Implementation of AsterixDB

Close() Function

After partitioning the relation **R**, a hash table will be created to hold the memory-resident records from the Build phase in order to be probed. As opposed to HHJ described in [15], which would use a predefined partition as its memory-resident partition, OHHJ follows the DHHJ approach in dynamically trying to keep as many partitions as possible in memory during the Build phase.

After all records are processed, the output frames of the spilled partitions will be flushed to disk, and their memory frames are returned to the buffer manager. Next, by estimating the size of the hash table considering the memory-resident records, OHHJ evaluates how a hash table can be created given the available memory. Under this evaluation, three cases may happen:

1. The available memory is not sufficient for creating the hash table for the current memory-resident partitions. Thus, one or more partitions will be spilled to make room for the hash table.

2. The amount of available memory is more than the size of the hash table. In this case, the algorithm will try to reload one or more partitions into memory if the available memory is enough to hold them and the expansion in the hash table size.

3. The available memory is enough for creating the hash table, and there is no room for reloading any spilled partition into memory.

It is important to note that the size of entries in the hash table will be the same for each record regardless of their size. Thus, the overhead of the hash table might be significant if the memory-resident records are numerous and small. On the contrary, this overhead can become minimal if the memory-resident records are not as many but large in size.

Listing 4.2 and Figure 4.4 illustrate the steps of the Close() function during the Build phase of OHHJ in AsterixDB.

```
#Input:     BufferManager bufferManager -> Module responsible for managing frames
#           List<Partitions> -> List of Partitions built prior to closing
#           int memoryBudget -> Memory Budget Allocated to this Join Operation
#Return:    HashTable mapping all memory-resident tuples

Function closeBuild(BufferManager bufferManager,List<Partitions> partitions, int memoryBudget)
    spilledPartitions = partitions.getAllSpilled()  #Get list of Spilled Partitions
    for Partition p in spilledPartitions
        spillPartition(p,bufferManager)             #Refer to Listings 4.1

    expectedHashTableSize = calculateHashTableSize(partitions)
    freeMemory = calculateFreeMemory(partitions,memoryBudget)

    while(freeMemory < expectedHashTableSize)      #While can't fit Hash Table in memory
        partitionToSpill = selectPartitionToSpill(partitions)
        spillPartition(partitionToSpill,bufferManager)
    reloadPartitionsIfPossible(spilledPartitions,freeMemory)
    return buildHashTable(partitions.getAllMemoryResident())


#Input: List<Partitions> spilledPartitions: List of spilled partitions
#       int freeMemory: Memory available for using
Function reloadPartitionsIfPossible(List<Partitions> spilledPartitions,int freeMemory)
    for Partition p in spilledPartitions
        hashTableSizeIncrease = calculateHashTableIncrease(p)
        if(freeMemory > (p.size + hashTableSizeIncrease))
            p.restore()
            freeMemory -= (p.size + hashTableSizeIncrease)

#Input: List<Partitions> partitions: List of memory-resident partitions
Function buildHashTable(List<Partitions> partitions)
    hashTable = new HashTable()
    for Partition p in partitions
        for tuple t in p
            hashTable.add(t)
    return hashTable;
```

Listing 4.2: Pseudocode for Close() Function of the Build Phase in current implementation.

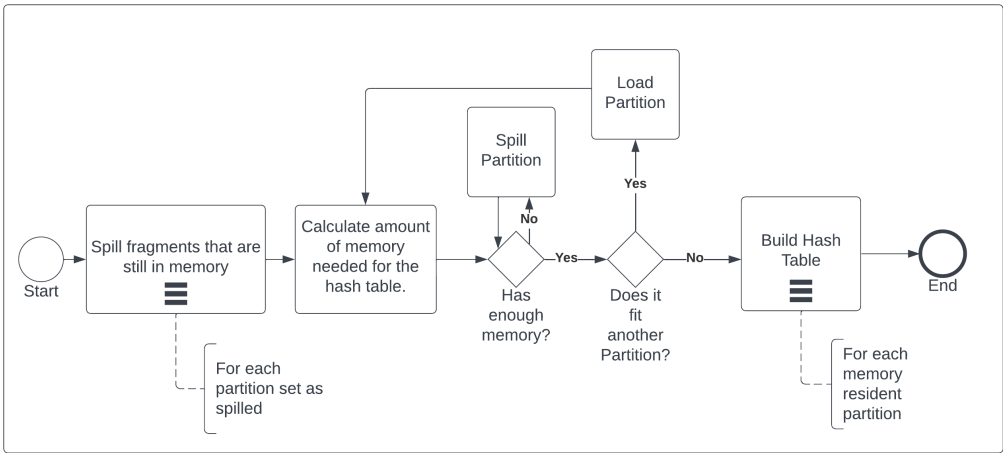


Figure 4.4: Close() Function of the Build Phase in Current Implementation of AsterixDB

4.1.2 Probe Phase

During this phase, tuples from the **S** relation mapped to memory-resident partitions are probed against the hash table while tuples mapped to spilled partitions are carried on to the next rounds of Join.

Open() Function

Before starting the Probe phase, it is necessary to configure the output buffers used to store tuples from the **S** relation. A study conducted by Jahangiri, Carey, and Freytag [9] concluded that if the system's file cache is active, there is no significant benefit of having a buffer larger than one frame. Therefore we configure the output buffer related to spilled partitions to a fixed size of one frame. We can relate this No Grow Policy to the MCRHJ algorithm described in [5] as the equivalent of setting $C_{eff} = 1$.

Next() Function

After properly configuring the buffers for the Probe phase, one at a time, each frame from relation **S** will be processed. If all tuples from **R** are memory-resident, there is no need for partitioning, and the tuples in the frame being processed can be probed against the hash table.

If any partition is spilled, partitioning is necessary, and only tuples mapped to memory-resident partitions will be probed against the hash table. This process is better described in the Listing 4.3.

```
#Input:     Tuple R -> reference to a file containing relation R
#           Buffers buff -> List of buffers
Function processTupleProbePhase(tuple t,BufferManager bufferManager,
                                    List<Partitions> partitions,
                                    List<Buffers> outputBuffers,
                                    file output)
    pId = SplitFunction(t);
    let partitionMapped = partitions.getById(pId);
    if(partitionMapped is MemoryResident)
        let matches = hashTable.probe(t)
        for m in matches
            output.write(<m,p>); #Write pair in the output file
            return
    else
    outputBuffer = outputBuffers.getBuffer(pId)
    while(!outputBuffer.insertTuple(t))
        outputBuffer.flushToDisk()
```

Listing 4.3: Pseudocode for Process Tuple Build Phase in current implementation.

The diagram in Figure 4.5 illustrates the sequence of steps.

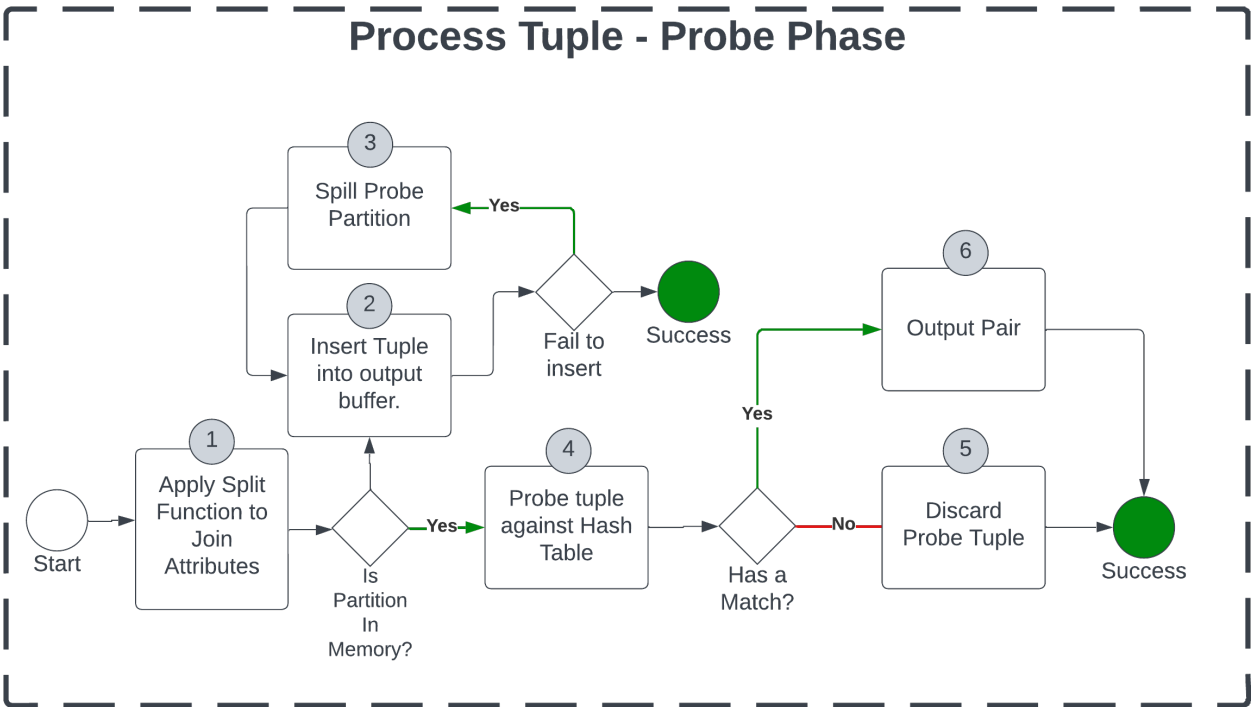


Figure 4.5: Tuple processing process during the Probe phase in Current Implementation of AsterixDB

Close() Function

By the end of the Probe phase, all tuples from relation **S** mapped to memory-resident partitions are guaranteed to have been probed. **S** relation's tuples mapped to spilled partitions are all written into a temporary file on disk. Each spilled partition will generate a pair of temporary files, R_i and S_i. The selection of which relation, either R_i or S_i, plays the role of the input for the Build phase and which one serves as the input for the Probe phase is determined dynamically at runtime. This process, known as role-reversal, involves choosing the smallest relation for the Build phase and the largest one for the Probe phase. Case the partition selected as input for the Build phase completely fits in memory along with a hash table, the second round of join will happen in memory. Otherwise, the DHHJ algorithm will be executed recursively.

4.2 Memory Fluctuation Scenarios

To adapt to new assigned memory budgets, the proposed algorithm will have to interact multiple times with a resource manager. This software component cable of defining the operator's memory budget dynamically will be referred to hereafter as Resource Broker in reference to the module described in [4]. As it is not the scope of this project to calculate the ROC or define an algorithm for memory distribution among operators, our proposed algorithm will only ask the Resource Broker for an updated memory budget. Regarding interactions with the Resource Broker, two approaches are explored; one is a frame-based approach, where these interactions will happen every **X** frames, **X** being a predetermined frame interval. The second approach is event-based, where the operator and the Resource Broker interact during specific events, such as when a partition is about to be spilled.

The frame approach increases the memory adaption responsiveness; a small **X** number will increase the frequency of interactions between the operator and the Resource Broker. Higher frequencies may cause a larger number of I/O operations, and smaller frequencies will cause the operator to be less responsive to memory fluctuations, taking longer to release memory frames when the need is over or to acquire memory frames when in need.

The event approach waits for key points in the algorithm to interact with the Resource Broker. For example, before spilling a partition, it is a good time to check if the Resource Broker can allocate more memory to this operator; this may avoid an I/O operation.

Additionally, both approaches may be combined, increasing the number of interactions between the operator and the Resource Broker. Before the proposed implementation is presented, the possible memory contention scenarios are described.

Memory Contraction During Build Phase

A memory contraction happens when the Resource Broker defines a smaller memory budget than the actual one. There are three possibilities for this scenario:

- The operator's actual memory budget is not fully utilized.
- The operator's actual memory budget is fully utilized. However, some unused frames are in the memory pool.
- The operator's actual memory budget is fully utilized, and there are no unused frames in the memory pool.

Memory Expansion During Build Phase

In case of having more memory being available to the join operator during the Build phase, we would just simply increase its memory budget and let the added memory budget be used in the time of need. It is important to note that memory expansion during the Build phase is only event-based and the join operator has to request it when it is about

to spill a partition. We do not allow a join operator to hold on to extra memory during the Build phase if there is no need for it.

Memory Contraction During Probe Phase

This scenario is similar to the memory contraction in the Build phase. However, when a partition is spilled, its tuples are mapped in the hash table. Not maintaining the hash table will not cause the algorithm to fail since tuples from **S** relation are not probed if the partition is spiled. However, not removing invalid entries from the hash table will cause some memory waste since some entries are pointing to spilled tuples. More about the consequences of this memory allocation related to the hash table are discussed in section 4.3.3.

Memory Expansion During Probe Phase

If the new memory budget assigned by the Resource Broker is larger than the current memory budget, there are two possibilities:

- It is possible to reload a spilled partition into memory and increase the hash table.
- It is not possible to reload a spilled partition into memory and increase the size of the hash table.

4.3 Proposed Design for Memory Contention Responsive Hash Join

This subsection presents a detailed design of the proposed Memory-Contention Responsive Hash Join algorithm. Similar to section 4.1, each step in the Iterator Design Pattern is presented so the reader can easily compare both implementations. Possible situations regarding memory allocation and their consequences are assessed and discussed.

4.3.1 Build Phase

Unlike the OHHJ algorithm, our proposed one can adapt to different memory availability scenarios by benefiting from extra memory or releasing some memory when the memory contention is high. The join algorithm will proactively contact the Resource Broker and request more memory when it runs out of memory. However, the join algorithm should frequently contact the Resource Broker and check if there is a request for it to release some memory. The frequency of interactions with the Resource Broker will impact the algorithm's responsiveness regarding the memory release. Next, we discuss how we implemented the memory adaptive features for our algorithm's Build phase.

Open() Function

The current implemented OHHJ already receives a memory budget during its initialization, so no changes are necessary regarding memory initialization. We have added a frame counter (F_i) initialized as zero that is used to trigger

interactions between our operator and the Resource Broker when the number of frames processed is a multiple of the selected interval. Next, we discuss changes in the Next() function for the proposed algorithm.

Next() Function

The Next() function implements two possible approaches for when to interact with the Resource Broker. The Frame approach takes place every **X** frames whereas the event-based approach happens proactively. Next, we describe these approaches implementations.

Frame Aproach

Before processing each frame, the operator will check if the frame counter F_i is a multiple of **X**. If this is the case it, the operator will contact the Resource Broker to check if there is an updated memory budget. The frame will be processed normally if F_i is not a multiple of **X** or if there is no change in the memory budget.
We will soon discuss the details of memory adaptation during the Build phase under both frame- and event-based approaches in subsubsection Update Memory During Build Phase.

Event Approach

In this approach, the join operator will only contact the Resource Broker at times of need and request a new memory budget to avoid partition spilling. However, the result of this request might be memory expansion, memory contraction, or no memory change. If the request results in a memory expansion, a partition spilling can be avoided or delayed.

On the other hand, if memory contraction occurs, the operator may have to spill more than one partition. Figure 4.6 illustrates the proposed sub-process. Note that it is similar to the one presented in subsubsection 4.1.1 except that instead of spilling a partition in block 4, the sub-process **Update Memory During Build Phase** presented in subsection 4.3.1 is called.

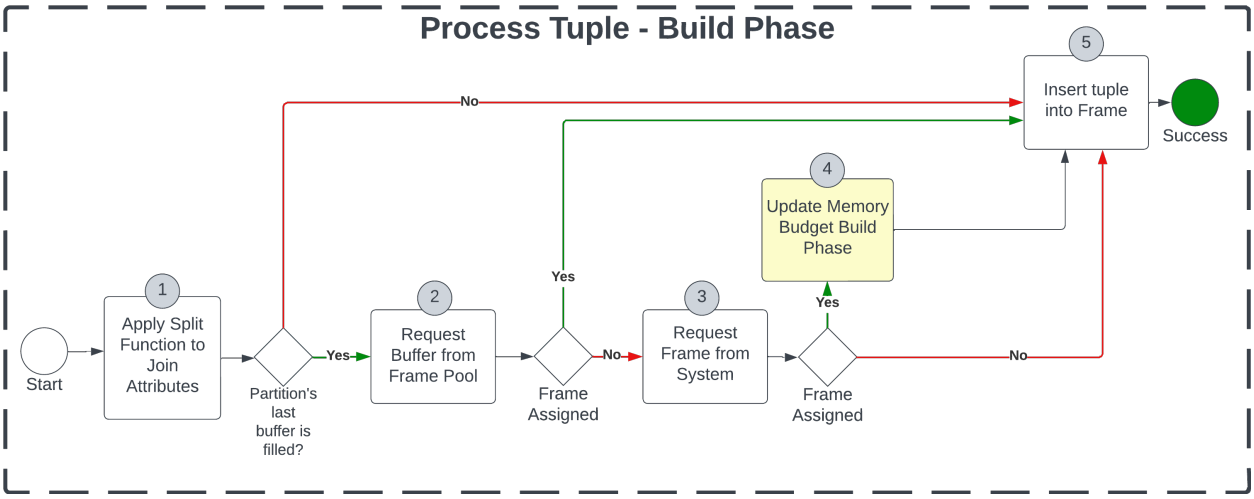


Figure 4.6: Next() Function During the Build Phase - Event-based Approach

Memory Budget Updating During Build Phase

This sub-process is the crucial difference between the Build phase of the currently implemented OHHJ algorithm and the proposed implementation of MCRHJ. This function is responsible for checking with the Resource Broker and adapting to any requests for memory change. The pseudocode snippet in Listing 4.4 and the Figure 4.7 detail how this function is implemented.

```
Function updateMemoryBudgetBuildPhase(List<Partitions> partitions, BufferManager bufferManager)
    newBudget = ResourceBroker.getNewBudget()
    while(newBudget != bufferManager.memoryBudget)
         while(newBudget < bufferManager.memoryAllocated)
            partitionToSpill = selectPartitionToSpill ( partitions )
            spillPartition ( partitionToSpill , bufferManager )
        release frames from memory pool
        currentBudget = newBudget
```

Listing 4.4: Pseudocode for Memory Budge Update Durign Build Phase.

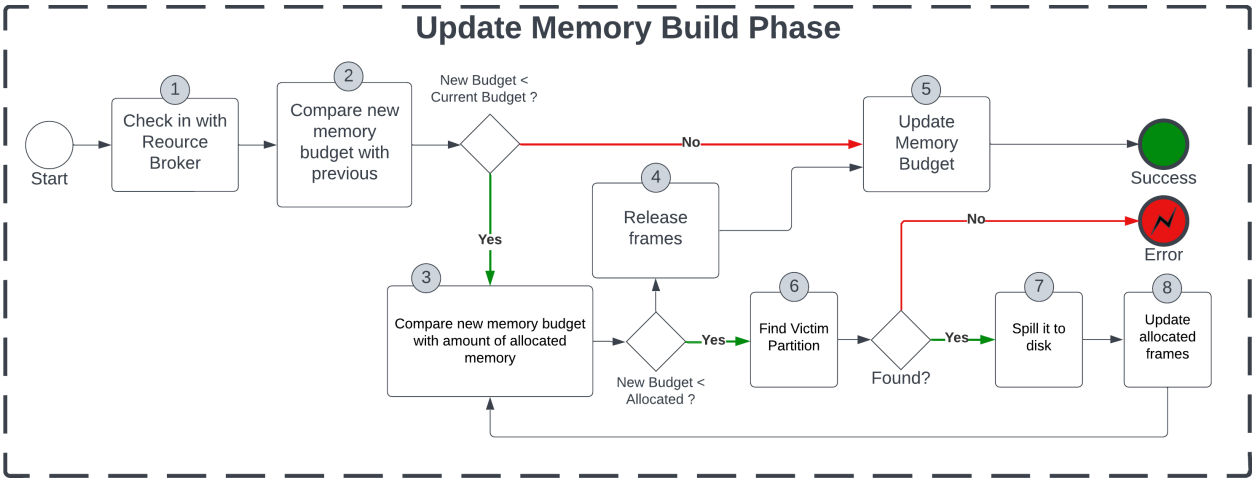


Figure 4.7: Update Memory Budget Build Phase

Note that while updating the memory budget, no actual bidding takes place as the operator does not inform the Resource Broker how much benefit it can get from receiving or holding memory. In Chapter 5, we further discuss how we implemented a substitute for the Resource Broker during our experiments.

Additionally, note that the operator does not release more frames than necessary to adapt to the new memory budget. If, after spilling partitions, the number of frames released surpasses the number of frames requested by the Resource Broker, this surplus is kept in the memory pool.

The proposed algorithm only increases its memory budget during the Build phase if this increase will immediately prevent a partition from spilling. This implementation prevents the algorithm from holding more frames than it actually needs.

Close() Function

The closing process is exactly the same as in the original OHHJ algorithm, where output buffers related to spilled partitions are flushed to disk to release some frames, and then the operator tries to reload spilled partitions that can fit in memory with the hash table. There is an extra implementation when the event approach is used, and this difference is described next.

Event Approach

Before closing the Build Phase, the algorithm requests an updated memory budget from the Resource Broker; this new value will determine how many partitions will fit in memory along with the hash table.

4.3.2 Probe Phase

For the Probe phase, unlike the Build phase, there are no distinct events that justify the implementation of an event-based approach for memory adaptation. Therefore, all steps described in this section refer to a frame-based approach. During the Probe phase, a memory expansion may add enough frames to reload a partition from **R** relation into memory. Such an event may create an inconsistent state where, by the end of the Probe phase, tuples mapped to memory-resident partitions are not guaranteed to be probed. Before proceeding with the algorithm description, this inconsistent state is studied in the next subsection.

4.3.3 The Inconsistent State

Every Hash-Based Join algorithm with a partitioning phase defines two states for an **R** relation's partition: disk or memory-resident. In the DHHJ algorithm, by the end of the Probe phase, all tuples from the **S** relation mapped to a memory-resident partition are guaranteed to have been probed. Therefore, there are no tuples from the **S** relation related to that partition that are passed forwared to another round of Join.

On the other hand, tuples from the **S** relation mapped to a spilled partition are guaranteed not to be probed will be processed in the next rounds of the join.

The same rationale does not apply for a MCRHJ algorithm as a partition can change its state from memory-resident to spilled during the execution. A tuple from the **S** relation, mapped to a spilled partition, undergoes no probing when it is processed. However, if this partition is subsequently reloaded and completes the Probe phase as a memory-resident partition, the mentioned tuple from the **S** relation remains unprobed. Therefore, it is not guaranteed that by the end of the Probe phase, all tuples from the **S** relation mapped to a memory-resident partition are probed. From this point forward, we will use the term inconsistent partitions to describe partitions reloaded at some point during the Probe phase. Likewise, we will refer to tuples from the **S** relation that were mapped to this partition while it was spilled as inconsistent tuples.

Open() Function

Since the memory budget was recently updated at the beginning of the Probe phase, there is no reason to check in with the resource manager. We will just reset the frame counter F_i

Next() Function

The Next() function is similar to the original implementation of OHHJ described in section 4.1.2, except that for every **X** frames processed, the operator will ask the resource manager for an updated memory budget. The sub-process to update the memory budget during the Probe phase is described in the following subsection.

Memory Budget Updating Process During Probe Phase

This sub-process updates the memory budget during the probe phase, relading spilled Build partitions into memory when its possible. Also, the hash table must be maintained as **R** partitions are reloaded into memory (memory expansion) or spilled into the disk (memory contraction). The following pseudocode snippet details this sub-process, and Figure 4.8 illustrates it. Note that it is similar to the one presented in Figure 4.7 except for the four blocks highlighted in yellow.

```
Function updateMemoryBudgetProbePhase(List<Partitions> partitions, BufferManager bufferManager)
    updateMemoryBudgetBuildPhase(partitions,bufferManager)
    partitionsReloaded = 0;
    while(partitions.canReloadAnother)
        partition.reload()
        partition.markAsInconsistent()
        partitionsReloaded++
    if(partitionsReloaded > 0)
        hashTable.rebuild()
```

Listing 4.5: Pseudocode for Memory Budge Update Durign Probe Phase.

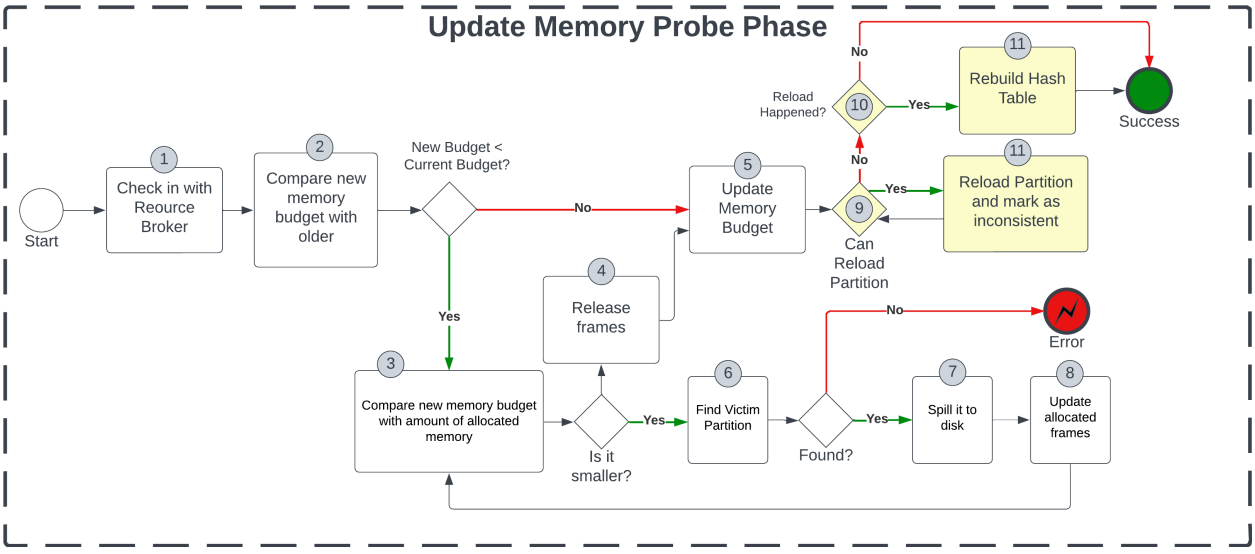


Figure 4.8: Memory Budget Updating Process - Probe Phase

In this sub-process, the hash table is dropped and rebuilt only when one or more partitions are reloaded due to memory expansion. We leave the hash table unchanged when memory contraction makes one or more partitions spill to disk (no hash table maintenance). Although maintaining the hash table after a contraction could release extra frames, it is a complicated process given the complex structure of the hash table built in Apache AsterixDB and may cause extra processing overhead. In addition, as memory is allocated in frames, rebuilding a hash table may decrease its size in bytes but not reduce the number of frames allocated. The strategy of rebuilding the hash table after a memory contraction is more beneficial when a record in relation **R** is not significantly larger than an entry in the hash table. Next subsection elaborates on the closing function of the proposed algorithm's Probe phase.

Close() Function

The Close() function for the proposed algorithm is similar to the currently implemented in Apache AsterixDB, described in subsection 4.1.2. However, the proposed Close() function for the Probe phase probes all inconsistent tuples mapped to partitions that are memory-resident. Although we could spill any inconsistent partition R_i and process the inconsistent tuples from S_i in the next round, we prefer to process S_i tuples in the current round since their R_i data is already in memory and can be processed without causing unnecessary I/O. Having the entire algorithm described, the next subsection will present a mathematical model to determine the increase in I/O volume caused by memory adaption operations.

4.4 I/O Calculation Model

Due to its memory-adaptive nature, this algorithm will impact the number and volume of I/O operations. The parameters determining this impact are the number of memory updates **N** and the memory variance σ^2. The following equation determines the total increase of I/O volume related to **R** partitions, V_p, during the Probe phase.

$$V_p = \sigma^2 \times N \qquad \text{(Equation 4.3)}$$

The variance is the average difference between any number in a certain distribution and the mean. and also the average difference between any two numbers generated. For every memory budget update, on average, the value will drop or increase by σ^2, possibly causing the spilling or reloading of a partition. It is not guaranteed that every run of this algorithm will produce this volume of I/O, but after a great number of runs, the average volume of I/O will converge to the output of the proposed equation. The increase in the number of I/O operations, N_{io}, can be derived from this volume equation by dividing it by the size of a $|R_i|$ partition.

$$N_{io} = \frac{V_p}{R_i} \qquad \text{(Equation 4.4)}$$

Chapter 5

Experiments

This chapter presents experiments we used to evaluate the proposed MCRHJ algorithm's performance regarding I/O volume and execution time. As mentioned earlier, the main goal for our work is to make DHHJ adapt efficiently to different possible workload changes to provide Resource Broker with the capability to rearrange the memory distribution among operators to fulfill its performance goal better. Since the focus of this thesis is on the design of MCRHJ, we have simulated concurrency variation scenarios and will consider the design of a real Resource Broker as our future work.

5.1 Resource Broker Simulation

This section provides a description of the concurrency scenarios and memory distribution employed in our simulated Resource Broker. The objective here is to establish scenarios and distributions that closely resemble those detailed in [5].

Concurrency Scenarios

In [5], the size of relation **R** was 5MB, with available memory ranging from 1MB in the highest contention scenario to 8MB in the lowest contention scenario. We opted for significantly larger data and memory sizes to capture the technological advancements while maintaining a similar ratio. In our experiments, **R** has grown to 1GB, while the maximum memory size ($|M_{max}|$) varies from 100MB to 900MB.

Memory Variation Distribution

Our proposed algorithm receives a new memory budget $|M_{budget}|$ every time it interacts with our simulated Resource Broker. The $|M_{budget}|$ is calculated based on the values of $|M_{max}|$ and the number of partitions **B** and follows the below-described distribution.

- During 80% of the time, the value of $|M_{budget}|$ will be randomly chosen from the range of $[\ 0.8 \times |M_{max}|,\ |M_{max}|\ ]$.

We will refer to this value as $|M_{budget\_major}|$

- During 20% of the time, the value of $|M_{budget}|$ will be randomly chosen from the range of [**B**, $|M_{max}|$]. We will refer to this value as $|M_{budget\_minor}|$

The distribution outlined above closely resembles the one utilized in simulations described in [5]. However, in our experiments, we made a slight adjustment to the range of $|M_{budget\_minor}|$, which now falls within [B, $|M_{max}|$] instead of [0, $|M_{max}|$]. This modification ensures the algorithm's functionality by guaranteeing at least one output frame for each partition. We use the same mentioned distribution for choosing memory budget values during the lifetime of the join operator, including the initial memory budget used in the calculation of the number of partitions.

The following equation calculates the value for memory variance, which can be plugged in Equation 4.3 to calculate the total amount of increased I/O compared to non-adaptive DHHJ.

$$\sigma^2 = \frac{0.8 \sum_{i=1}^{N} \left(|M_{budget\_major}| - 0.9|M|\right)^2 + 0.2 \sum_{j=1}^{N} \left(|M_{budget\_minor}| - \frac{|M|-B}{2}\right)^2}{N} \quad \text{(Equation 5.1)}$$

Next, we describe how the frequency of memory updates are simulated in our work.

Frequency of Updates

The MCRHJ algorithm proposed by Davison and Graefe in [5] did not specify any interaction with other software components responsible for memory allocation. The same authors discussed the concept of a Resource Broker in [4] one year later. Therefore, the idea of a frequency of memory budget updates was not part of the original MCRHJ algorithm. In our implementation, we have incorporated a parameter that allows us to regulate the frequency of interactions with our simulated Resource Broker. This frequency can be adjusted by specifying the number of frames processed before each interaction. To explore a range of interaction rates, we conducted experiments with several frame intervals, spanning from 300 to 2,000 frames, encompassing higher and lower interaction frequencies. We deliberately chose to avoid employing smaller frame intervals to prevent the risk of system thrashing. This decision is based on the fact that the number **N** of interactions, as used as a parameter in Equation 4.3, is determined by dividing the size of relation **S** by the chosen frame interval. Reducing this interval to values below 300 would result in a rapid increase in the number of interactions.

5.2 Data

In this subsection, we provide details about the settings for the data used during our experiments. We used relations of **R** and **S**, each containing 1GB of data. Each record in the relation **R** had exactly one match in relation **S** and vice

versa; relations had records with the same size, and the *join attributes* used during the operations were not indexed by the DBMS. Both relations were generated using Wisconsin JSON data generator [16].

5.3 Expected Results

Having $|R|$, $|S|$, and the variance (σ^2) of the memory budget distribution (Equation 5.1), we can use the Equation 4.3 to simulate the expected results of runs for different values of $|M_{max}|$ and **N** number of updates. A Python code for this simulation is available in [17]. The chart depicted in Figure 5.1 illustrates the anticipated rise in I/O volume attributed to the spilling and restoring of **R** partitions. It's noteworthy that the lines corresponding to higher frequencies of updates (smaller frame intervals) exhibit significantly steeper slopes. The memory distribution previously described (as per Equation 5.1) introduces a variance (σ^2) that grows as $|M_{max}|$ becomes larger. This variance, when coupled with a higher frequency of updates, results in a rapid escalation of I/O volume.

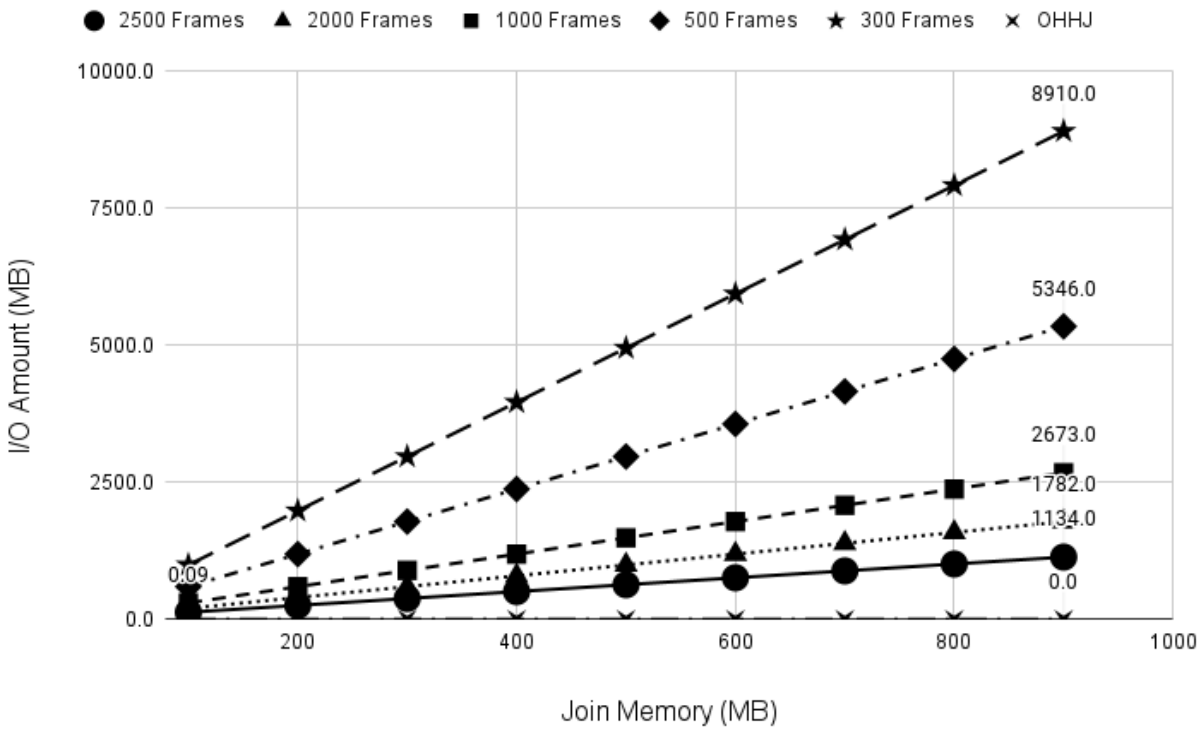


Figure 5.1: Expected Increase in I/O Volume.

Next, we describe the system environment and settings used to run our experiments.

5.4 System Configuration

The system's configuration used to run this experiment and the Apache AsterixDB configuration are presented in Table 5.1. For our experiments, we did not bypass the file system's cache; we will leave this approach as a suggestion for further study.

Parameter	Configuration
Processor	11th Gen Intel® Core™ i7-1165G7 × 8
Operating System	Ubuntu 22.10
RAM	2x32GB DDR4 3200MHz (CT32G4SFD832A)
Storage	Samsung SSD 980 PRO 2TB
Java Runtime Version	17
Apache AsterixDB Version	0.9.8
Number of NCs	1
Number of IO Devices	1
Frame Size	32KB

Table 5.1: System Configuration

5.4.1 Methodology

Relations **R** and **S** described in section 5.2 will be joined using a nonindexed attribute. An experiment suite will run 11 times for each combination of frequency and maximum memory $|M_{max}|$ experimented, and the average results of the last ten runs are reported. Between each run, the file system's cache is cleared to eliminate cache effects.

Chapter 6

Results

In the course of our experiments, we assessed various facets of MCRHJ algorithm, with particular emphasis on examining the impact of update frequency on both the quantity and size of I/O operations. In the next sections, we will evaluate the results obtained from our experiments and define a cost calculation for the number of I/O operations.

6.1 Memory Fluctuation Responsiveness

It is desirable for a robust memory contention-responsive algorithm to remain stable in its execution during memory fluctuations. Our experiments measure this responsiveness by considering the number of memory-resident partitions during the Probe phase. We decided not to evaluate the number of partitions kept memory-resident during the Build phase because a single small value of M_{budget_minor} can cause all partitions to be set spilled early on this phase, and as there is no restoration, this will provide no insightful information regarding responsiveness. However, during the Probe phase, the algorithm shows its responsiveness capability by spilling and reloading **R** partitions. The chart Figure 6.1 presents one line for each frequency of updates used during the experiments. The left side of the chart depicts scenarios with higher concurrency, whereas the right side represents scenarios with lower concurrency. Results of the experiments during the Probe phase also show great stability for each line. Additionally, we can observe the increased responsiveness of the algorithm with more frequent consultations with the Resource Broker.

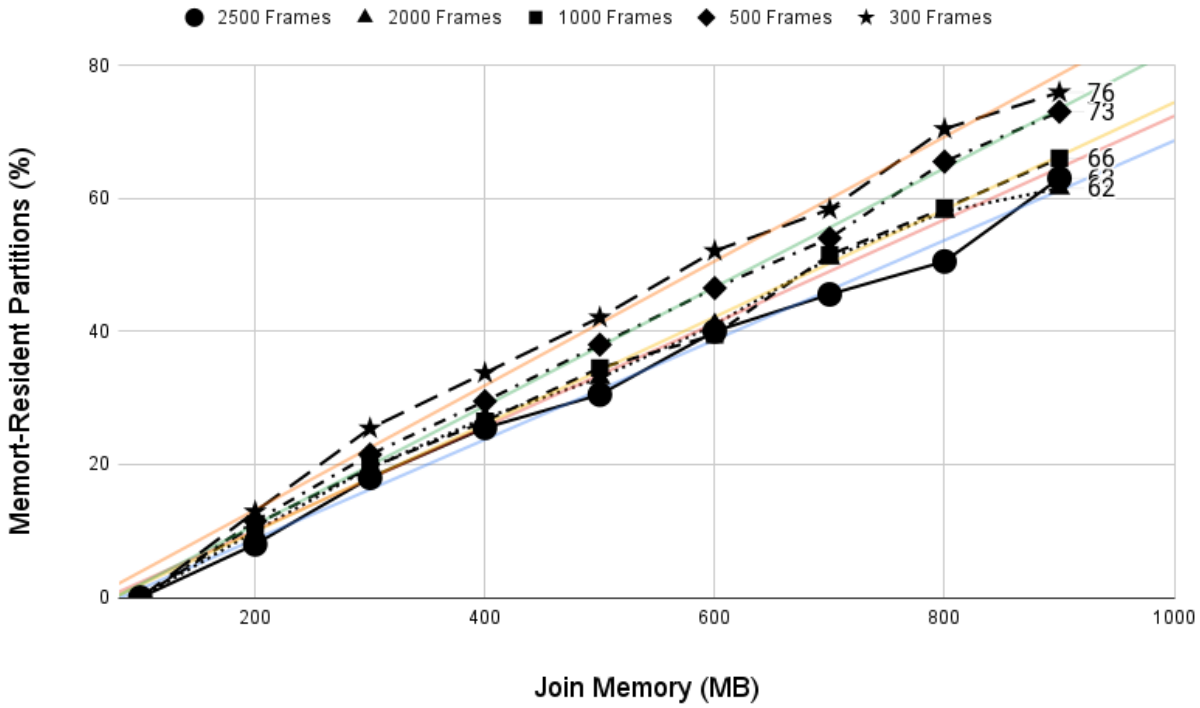


Figure 6.1: Memory-resident partitions during the Probe phase

Keeping more memory-resident partitions during the Probe phase is beneficial for this algorithm as more tuples from **S** relation can be probed in the current round, reducing I/O operations. Nonetheless, it's worth noting that the increase in I/O operations resulting from memory adaptation could potentially offset these benefits. Next, we will present the effects of memory adaption over I/O volume.

6.2 I/O Operations

While memory adaptive operators can indeed equip the DBMS with the ability to respond to workload variations, it's important to acknowledge that frequent adjustments to the memory budgets of operators may result in an excessive volume of I/O operations. This is a consequence of additional data spillings and reloadings caused by memory fluctuation. Our experiments have uncovered that the expense incurred in enhancing responsiveness may outweigh the advantages of increased memory availability. Figure 6.2 presents the increase in the volume of writing operations caused by spilling **R** partitions.

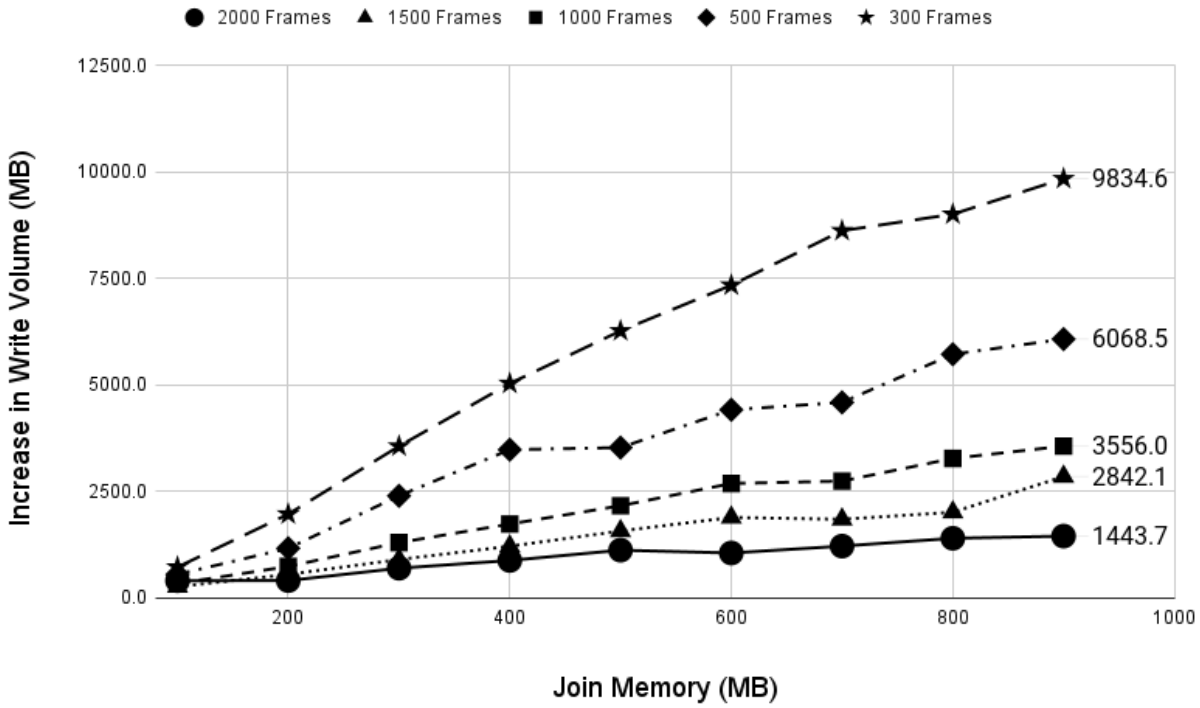


Figure 6.2: Increase in writing volume caused by memory contraction events.

The results approximate the expected values described in Figure 5.1. The small number of partitions calculated in Equation 4.2 cause the size of **R** partitions to be considerably large. We believe that smaller partitions would cause a more fine-grained memory adaption approximating the experiment results to the simulation results. The same rationale applies to reading volume due to memory expansion events. Figure 6.3 presents the increase in reading volume related to the restoration of **R** partitions during the Probe phase. Once again, the algorithm's behavior is close to the expected, but smaller partitions may cause a more fine-grained memory adaption. Moreover, in a somewhat counterintuitive manner, a larger memory allocation led to an increase in the volume of I/O operations. This counterintuitive effect arises from the fact that memory variance escalates with the size of the $|M_{max}|$. Consequently, the volume of I/O operations also experiences an increase, as indicated by Equation Equation 4.3. This effect is a reflection of the specific characteristics of the memory distribution employed in our experiments. It's worth noting that different distributions may exhibit distinct behaviors.

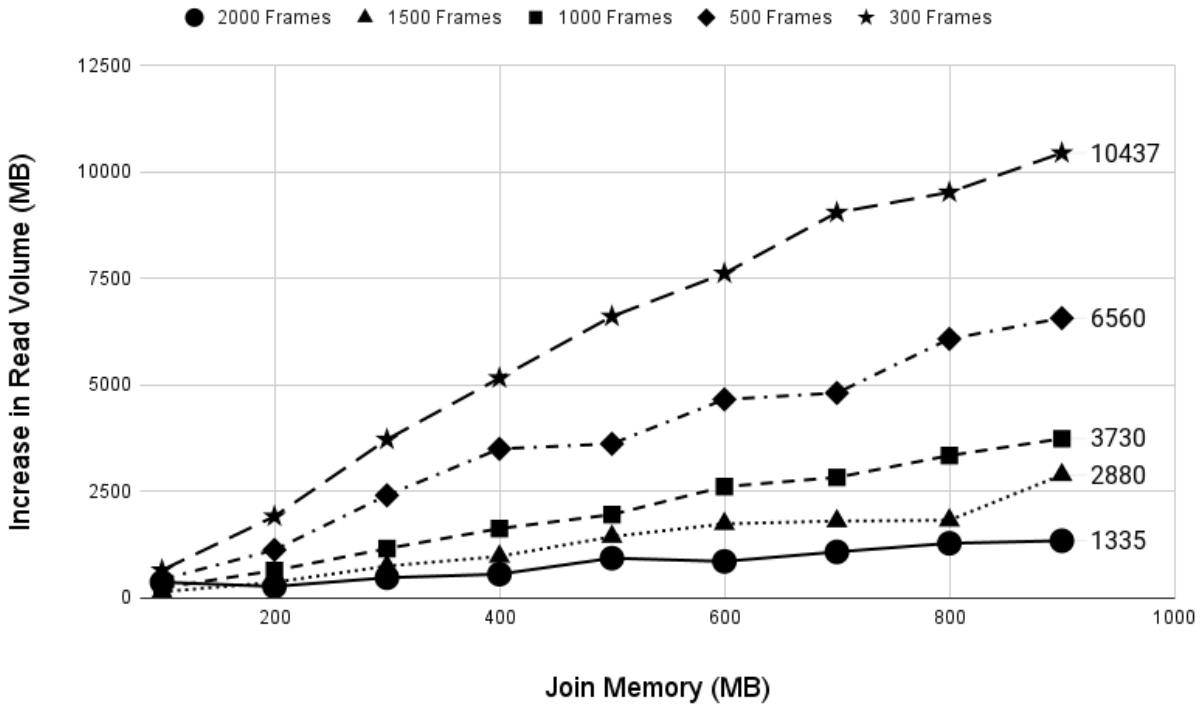


Figure 6.3: Increase in reading volume caused by memory contraction events.

Next, we will evaluate the impact of memory adaption over the execution time.

6.3 Execution Time

A memory-adaptive algorithm can enhance the overall system's performance by introducing the potential to reclaim memory that has been previously allocated to an operator. This reallocation allows the DBMS to increase the number of operators being processed concurrently, thereby improving system's query throughput. However, using the proposed MCRHJ algorithm may cause the join operator to lose performance compared to a static memory algorithm with the same maximum memory. Throughout our experiments, we measured the time spanning from the initialization (Open() function) of the Build phase to the conclusion of the Probe phase (Close() function). Figure 6.4 illustrates this execution time across various frequencies of memory updates and different maximum memory values.

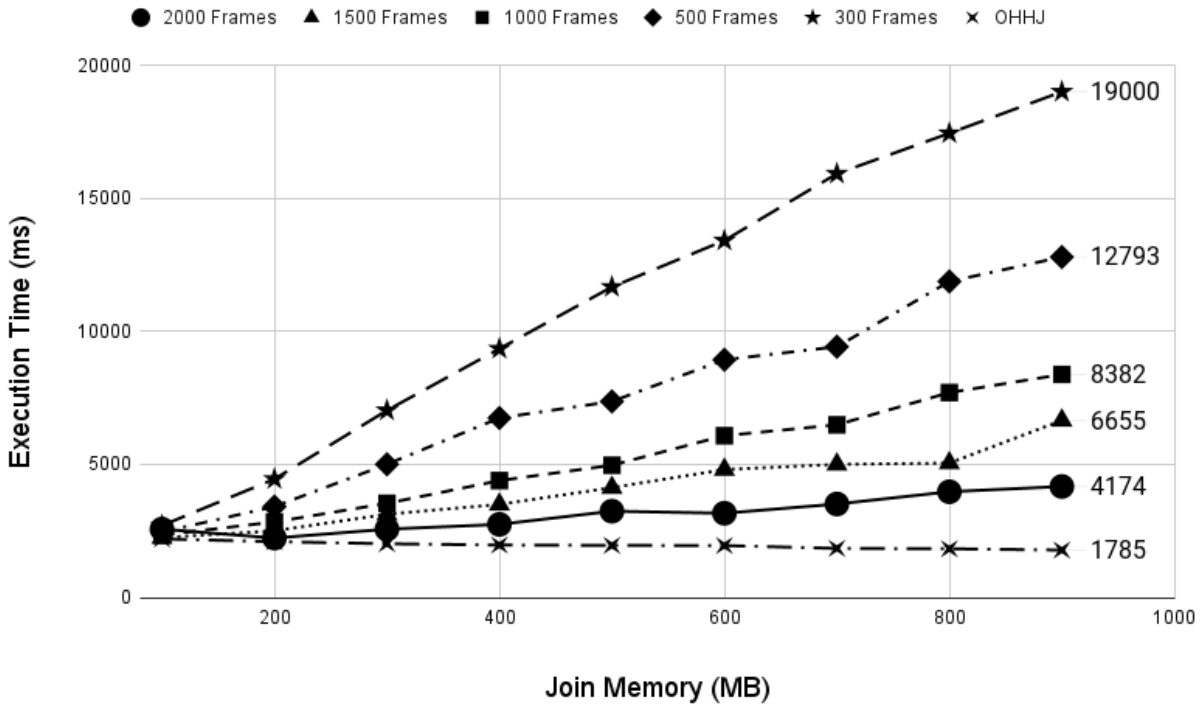


Figure 6.4: Increase in reading volume caused by memory contraction events.

The execution time curves are similar in shape as the I/O increase curves presented in Figure 6.2 and Figure 6.3 . This is also expected since I/O is a costly operation that highly impacts the execution time of this algorithm. Furthermore, we present the response time for OHHJ as a base for comparison. The frequency of updates can increase the responsiveness to memory fluctuations, but the consequences of these adaptations must be considered before adopting this algorithm, especially in environments where the memory variance is higher. Next, we present our conclusions and suggestions for further studies.

6.4 Conclusion

The proposed MCRHJ algorithm introduced important concepts that extend the original design presented in [5]. The central contribution of this thesis lies in understanding how memory variance and the frequency of memory budget updates impact the I/O volume and, consequently, the execution time of an operator. Equation 4.3 could serve as a valuable foundation for future research into memory adaptive algorithms, extending beyond the confines of the join operator. Despite the increase in complexity, the proposed algorithm may be beneficial to DBMSs that have to deal with variations in concurrency. Additionally, implementing a resource manager aware of the memory variance can prevent this algorithm from becoming harmful to the system by adjusting the frequency of updates accordingly.

6.5 Further Studies

This algorithm can be further developed to use its memory budget in a more efficient manner. During the Build phase, an increase in the memory budget is not being leveraged since we only allow an increase if a partition is about to be spilled. Implementing the spooling area described in PPHJ may be useful in reducing the overall amount of I/O.

Another topic of further study is reducing and managing the hash table overhead when records being joined have significantly different sizes. This overhead can directly impact the calculation of the number of partitions. Also, in Chapter 4, we have mentioned that records larger than a certain limit are spilled automatically, not even moved to the buffers during the Build phase. As semi-structured data allows records to be very large, automatically spilling these large records can harm the algorithm's performance. Another topic for further study is an implementation that can better deal with such large records.

Lastly, the Resource Broker is a vast research topic since its design and memory distribution logic require careful consideration. Otherwise, it may degrade the DBMS's performance due to thrashing. Stage-based memory sharing between operators can be used to ensure no deadlock happens due to memory being shared between too many operators.

Bibliography

[1] Michael A. Bender et al. "Cache-Adaptive Algorithms". In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '14. Portland, Oregon: Society for Industrial and Applied Mathematics, 2014, pp. 958–971. ISBN: 9781611973389.

[2] Dimitrije Curcic. *Amazon Printed Books Sold*. https://wordsrated.com/print-book-sales-statistics/. Accessed: 2023-06-29.

[3] Dimitrije Curcic. *Amazon Publishing Statistics*. https://wordsrated.com/amazon-publishing-statistics/. Accessed: 2023-06-29.

[4] Diane L. Davison and Goetz Graefe. "Dynamic Resource Brokering for Multi-User Query Execution". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: Association for Computing Machinery, 1995, pp. 281–292. ISBN: 0897917316. DOI: 10.1145/223784.223845. URL: https://doi.org/10.1145/223784.223845.

[5] Diane L. Davison and Goetz Graefe. "Memory-Contention Responsive Hash Joins". In: *Very Large Data Bases Conference*. 1994.

[6] D.J. DeWitt et al. *Implementation Techniques for Main Memory Database Systems*. Tech. rep. UCB/ERL M84/5. EECS Department, University of California, Berkeley, Jan. 1984. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/246.html.

[7] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[8] G. Graefe. "Volcano— An Extensible and Parallel Query Evaluation System". In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135. ISSN: 1041-4347. DOI: 10.1109/69.273032. URL: https://doi.org/10.1109/69.273032.

[9] Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. *Design Trade-offs for a Robust Dynamic Hybrid Hash Join (Extended Version)*. 2021. arXiv: 2112.02480 [cs.DB].

[10] Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. "Design Trade-offs for a Robust Dynamic Hybrid Hash Join (Extended Version)". In: *CoRR* abs/2112.02480 (2021). arXiv: 2112.02480. URL: https://arxiv.org/abs/2112.02480.

[11] Taewoo Kim et al. "Robust and efficient memory management in Apache AsterixDB". In: *Software: Practice and Experience* 50.7 (2020), pp. 1114–1151. DOI: https://doi.org/10.1002/spe.2799. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2799. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2799.

[12] M. Kitsuregawa, H. Tanaka, and T Moto-Oka. "Application of hash to data base machine and its architecture." In: *New Generation Compute* 1 (1983), pp. 62–74. DOI: https://doi.org/10.1007/BF03037022.

[13] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. "Hash-Partitioned Join Method Using Dynamic Destaging Strategy". In: *Proceedings of the 14th International Conference on Very Large Data Bases*. VLDB '88. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 468–478. ISBN: 0934613753.

[14] Hwee Hwa Pang, Michael J. Carey, and Miron Livny. "Partially Preemptible Hash Joins". In: *SIGMOD Rec.* 22.2 (June 1993), pp. 59–68. ISSN: 0163-5808. DOI: 10.1145/170036.170051. URL: https://doi.org/10.1145/170036.170051.

[15] Leonard D. Shapiro. "Join processing in database systems with large main memories". In: *ACM Trans. Database Syst.* 11 (1986), pp. 239–264.

[16] Jahangiri Shiva. *"shivajah/JSON-Wisconsin-Data-Generator"=¿"First Release"*. Version v1.0.1, Dec. 2020. DOI: 10.5281/zenodo.4316003,. URL: https://github.com/shivajah/JSON-Wisconsin-Data-Generator.

[17] Giulliano Siviero. *IO Volume Increase Simulator for MCRHJ*. https://github.com/gSiviero/IOIncreaseMCHJ.

[18] Hansjörg Zeller and Jim Gray. "An Adaptive Hash Join Algorithm for Multiuser Environments". In: *Very Large Data Bases Conference*. 1990.

Acronyms

BDMS Big Data Management System. 1, 14

CC Cluster Controller. 14

DBMS Database Management Systems. iii, iv, 1, 3, 4, 6, 9, 15, 38, 44

DHHJ Dynamic Hybrid Hash Join. 13, 14, 16, 17, 21, 23, 28, 33, 36, 37

HHJ Hybrid Hash Join. v, 12, 13, 25

HJ Hash Join. 12

MCRHJ Memory-Contention Responsive Hash Join. iv, 1, 17, 18, 21, 27, 30, 32, 33, 36, 37, 40, 43, 44

NC Node Controller. 14, 15, 39

OHHJ Optimized Hybrid Hash Join. 22, 25, 30, 32–34, 44

PPHJ Partially Preemptible Hybrid Hash Join. iv, 16–18, 45

Glossary

data partition A semistructured dataset that can be either stored in AsterixDb or in an external file.. 14, 15

memory budget Amount of memory measured in Frames, designated to a specific operation.. 15

Apache AsterixDB A semistructured dataset that can be either stored in AsterixDb or in an external file.. 1, 3, 14, 20–22, 35, 39

Acronyms

BDMS Big Data Management System. 1, 14

CC Cluster Controller. 14

DBMS Database Management Systems. iii, iv, 1, 3, 4, 6, 9, 15, 38, 44

DHHJ Dynamic Hybrid Hash Join. 13, 14, 16, 17, 21, 23, 28, 33, 36, 37

HHJ Hybrid Hash Join. v, 12, 13, 25

HJ Hash Join. 12

MCRHJ Memory-Contention Responsive Hash Join. iv, 1, 17, 18, 21, 27, 30, 32, 33, 36, 37, 40, 43, 44

NC Node Controller. 14, 15, 39

OHHJ Optimized Hybrid Hash Join. 22, 25, 30, 32–34, 44

PPHJ Partially Preemptible Hybrid Hash Join. iv, 16–18, 45