Design Trade-offs for a Robust Dynamic Hybrid Hash Join

Shiva Jahangiri
University of California, Irvine
shivaj@uci.edu

Michael J. Carey
University of California, Irvine
mjcarey@ics.uci.edu

Johann-Christoph Freytag
Humboldt-Universität zu Berlin
freytag@informatik.hu-berlin.de

ABSTRACT

Hybrid Hash Join (HHJ) has proven to be one of the most efficient and widely-used join algorithms. While HHJ's performance depends largely on accurate statistics and information about the input relations, it may not always be practical or possible for a system to have such information available.

HHJ's design depends on many details to perform well. This paper is an experimental and analytical study of the trade-offs in designing a robust and dynamic HHJ operator. We revisit the design and optimization techniques suggested by previous studies through extensive experiments, comparing them with other algorithms designed by us or used in related studies.

We explore the impact of the number of partitions on HHJ's performance and propose a new lower bound for the number of partitions. We design and evaluate different partition insertion techniques to maximize memory utilization with the least CPU cost. Additionally, we consider a comprehensive set of algorithms for dynamically selecting a partition to spill and compare the results against previously published studies. We then present and evaluate two alternative growth policies for spilled partitions.

These algorithms have been implemented in the context of Apache AsterixDB and evaluated under different scenarios such as variable record sizes, different distributions of join attributes, and different storage types, including HDD, SSD, and Amazon Elastic Block Store (Amazon EBS).

PVLDB Reference Format:
Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. Design Trade-offs for a Robust Dynamic Hybrid Hash Join. PVLDB, 15(10): 2257 - 2269, 2022.
doi:10.14778/3547305.3547327

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/shivajah/VLDB_2022.

1 INTRODUCTION

As one of the most popular and expensive DBMS operators, the join operator can significantly impact the performance of a DBMS. HHJ [15, 40] has shown superior performance in computing the equijoin of two datasets among other kinds of join operators. In a nutshell, HHJ groups the records of each dataset into disjoint partitions. A hash table is created to hold one of the partitions in memory (memory-resident partition), while the rest will be written (spilled) to disk to be processed in the next rounds of HHJ. The number of partitions and the selection of the memory-resident partition are static decisions made at compile time for an HHJ operator. While previous studies [23, 40] have suggested various cost models and optimization techniques for enhancing such decisions, these studies have two shortcomings: (1) They assume a uniform distribution for join attribute values. (2) Their cost models rely on having accurate statistical information such as input sizes prior to query execution.

Unfortunately, collecting and accessing or predicting such information may not always be feasible. For example:

- Many data management systems process external data that resides outside their storage for which they have little or no information. (Examples include: Apache AsterixDB [3], Apache Spark [4], and Oracle external tables [7].)
- The accurate sizes of join inputs may not be known if they result from other operators instead of being base relations.
- Newly developed DBMSs may not have statistics available until they become more mature in other dimensions.

Not having sufficient statistics can be detrimental to the performance of operators whose designs depend on such information. [35] has proposed Dynamic HHJ to address the unbalanced distribution of join attribute values by dynamically destaging the partitions at the runtime of a join query.

Investigating the Dynamic HHJ algorithm reveals several design questions that must be explored carefully, as they may impact the system's overall performance:

- Number of partitions: How many partitions should the records be hashed into if the sizes of inputs are unknown or inaccurate?
- Partition Insertion: How can we find a "good" page (memory frame) within a partition for inserting a new record?
- Victim Selection Policy: How can we select a "good" partition to spill in the case of insufficient memory?
- Growth Policy: How many memory frames should a spilled partition be allowed to occupy?

With this motivation, this paper is an experimental survey of the trade-offs in designing a robust Dynamic HHJ algorithm. We answer the questions above through a comprehensive evaluation of different design aspects of the Dynamic HHJ algorithm and evaluate the alternative options through extensive experimental and model-based analyses. First, we propose a new lower bound for the number of partitions for Dynamic HHJ. We show that our proposed lower bound, while simple, can reduce the total amount of I/O by up to a factor of three in some investigated scenarios. Second, we study different partition insertion algorithms to efficiently find a frame with enough space in the target partition. We evaluate the effectiveness of these algorithms on partition compactness (fullness) and total I/O reduction. Additionally, we propose and evaluate two policies for allocating memory frames to spilled partitions. Finally,

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547327

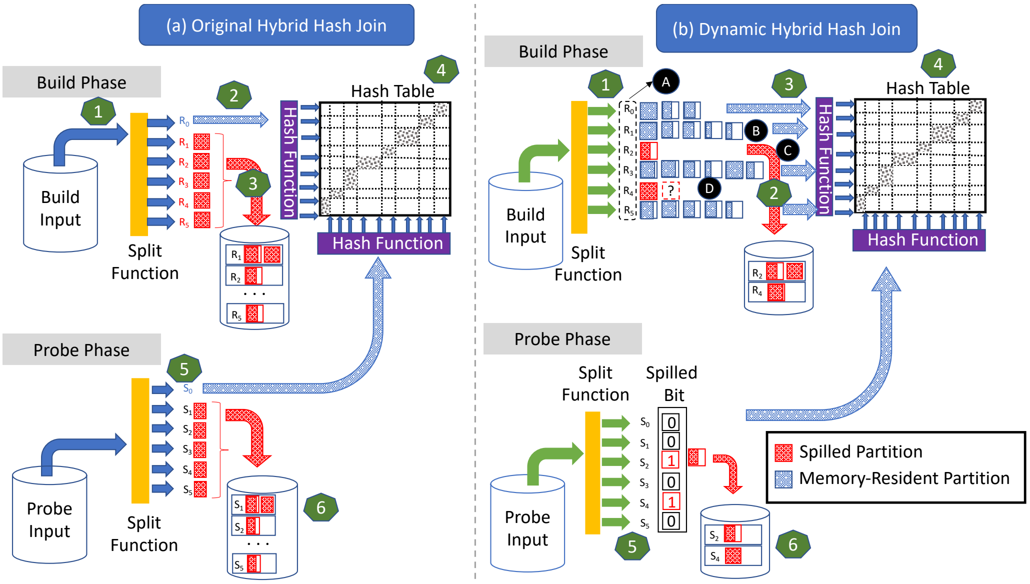


Figure 1: Workflow of (a)- Hybrid Hash Join (left) and (b)- Dynamic HHJ (right)

we propose and implement various dynamic destaging (victim selection) strategies and evaluate them under different scenarios such as different record size distributions, join attribute value distributions, and combinations thereof. The suggested optimization techniques and algorithm variants have been implemented in the Apache AsterixDB system and evaluated on different storage types, including HDD, SSD, and Amazon EBS.

The remainder of the paper is organized as follows: Section 2 provides background information on Apache AsterixDB and the workflow of the HHJ and Dynamic HHJ operators. Section 3 discusses previous work related to this study. In Section 4, we discuss the lower bound on the number of partitions to use in practice. Section 5 introduces and evaluates different partition insertion algorithms. In Section 6, two policies for the growth of spilled partitions are discussed and evaluated. Section 7 discusses and evaluates various destaging partition selection policies. In Section 8, we study the performance tradeoffs between single-core and multi-core execution of Dynamic HHJ. Section 9 summarizes the paper.

2 BACKGROUND

2.1 Hybrid Hash Join

Like other hash-based join algorithms, HHJ uses hashing to stage large inputs to reduce record comparisons during the join. HHJ has been shown to outperform other join types in computing equijoins of two datasets. It was designed as a hybrid version of the Grace Hash Join and Simple Hash Join algorithms [15, 40]. All three mentioned hash join algorithms consist of two phases, namely "build" and "probe". During the build phase, they partition the smaller input, which we refer to as "build input", into disjoint subsets. Similarly, the probe phase divides the larger input, which we refer to as "probe input", into the same number of partitions as the build input. While all three algorithms share a similar high-level design, they differ in their details, making each of them suitable for a specific scenario.

Grace Hash Join partitions the build and probe inputs consecutively, writing each partition back to disk in a separate file. This partitioning process continues for each partition until it fits into memory. A hash table is created to process the join once a partition is small enough to fit in memory. Grace Hash Join performs best when the smaller dataset is significantly larger than the main memory.

In Simple Hash Join, records are hashed into two partitions: a memory-resident and a disk (spilled) partition. A portion of memory is used for a hash table to hold the memory-resident partition's records. Simple Hash Join performs well when memory is large enough to hold most of the smaller dataset. In Grace Hash Join, the idea is to use memory to divide a large amount of data into smaller partitions that fit into memory, while Simple Hash Join focuses on the idea of keeping some portion of data in memory to reduce the total amount of I/O, considering that a large amount of memory is available. In the following, we discuss the details of the HHJ operator and compare its design with its parent algorithms.

Like Grace Hash Join, HHJ uses hash partitioning to group each input's records into "join-able" partitions to avoid unnecessary record comparisons. Like Simple Hash Join, HHJ uses a portion of memory to keep one of the partitions and its hash table in memory, while the rest write to disk. Keeping data in memory reduces the total amount of I/O, and utilizing a hash table lowers the number of record comparisons.

During the build phase of HHJ, the records of the smaller input are scanned and hash-partitioned based on the values of the join attributes (Figure 1-(a)-1). The hash function used for partitioning is called a "split function." The records mapped to the memory-resident partition remain in memory (Figure 1-(a)-2), while the rest of the partitions are written (frame by frame) to disk (Figure 1-(a)-3). Pointers to the memory-resident partition's records are inserted into a hash table at the end of the build phase (Figure 1-(a)-4).

After the build phase ends, the probe phase starts by scanning and hash-partitioning the records of the larger input. The same split function used during the build phase is used for this step. The records that map to the memory-resident partition are hashed using the same hash function used in the build phase to probe the hash table. All other records are written (frame by frame) to their partition's probe file on disk (Figure 1-(a)-5).

After all records of the probe input have been processed, the pairs of spilled partitions from the build phase and probe phase are processed as inputs to the next rounds of HHJ (Figure 1-(a)-6). The initial execution of build and probe inputs is considered round 1, and round *n* consists of joining a set of spilled partitions pairs from round*(n-1)* using HHJ recursively.

2.2 Dynamic Hybrid Hash Join

Dynamic HHJ was first introduced in [35], where the authors used dynamic destaging instead of the static predefined memory-resident partition method. As Figure 1-(b)-1 shows, the build phase starts by reading the records of the build input frame by frame into memory. In Dynamic HHJ, as opposed to HHJ, all partitions in the build phase have an equal chance to grow as long as enough memory frames are available. This flexibility in acquiring frames may cause some partitions to receive more frames than others if join attribute values are skewed. Every time that all of the memory frames are allocated, one of the partitions will be dynamically selected to be written to disk (Figure 1-(b)-2). This dynamic destaging is especially useful when the build input size or the distributions of join attribute values are unknown or inaccurate.

After partitioning the build dataset's records, pointers to the records of the memory-resident partitions are hashed and inserted into the hash table to be probed (Figure 1-(b)-3 and Figure 1-(b)-4). Once the build phase is over and the hash table is created, the probe phase starts by reading the probe dataset into memory one frame after another. All of the incoming records will be hashed using the same split function used during the build phase to find out if their corresponding partition from the build phase is a disk or an in-memory partition with the assistance of a bit vector (Figure 1-(b)-5). The records mapping to a disk-resident partition will be written to disk using an output frame. The records that belong to an in-memory partition will be hashed using the same hash function used during the build phase in order to find their potential matches by probing the hash table. As the last step (Figure 1-(b)-6), once the probe phase is over, the spilled partitions from the build phase along with their corresponding partitions from the probe phase will be processed in a similar way in the next round of the HHJ operator (Steps 1 through 6 in Figure 1-(b)).

2.3 AsterixDB

Apache AsterixDB [3, 8, 29] is an open-source, parallel, shared-nothing big data management system (BDMS) built to support the storage, indexing, modifying, analyzing, and querying of large volumes of semi-structured data. Figure 2 shows the logical architecture of AsterixDB. The Cluster Controller node is the entry point for the user requests and compiles and transforms the requests into executable jobs. The Node Controllers are the worker nodes that execute the jobs sent by Cluster Controller. One Node Controller also serves as the Meta Data Controller Node and provides access to AsterixDB's metadata. Each Node Controller manages one or more data partitions. An instance of each query will be executed in parallel on each data partition that the query needs to access. Throughout this paper and for simplicity, we use one Node Controller with one data partition, unless otherwise mentioned.

The unit of data that is transferred within AsterixDB, as well as between AsterixDB and disk, is called a "frame". A frame is a fixed-size and configurable set of contiguous bytes. AsterixDB uses Dynamic HHJ, whose design and optimization is the main topic of this paper. In AsterixDB, a single thread is used per data partition during the build and probe phase. AsterixDB supports different join algorithms such as Block Nested Loop Join, Dynamic HHJ, Broadcast Join, and Indexed Nested Loop Join. However, Dynamic HHJ is the default and primary join type in AsterixDB for processing equi-joins due to its superior and robust performance.

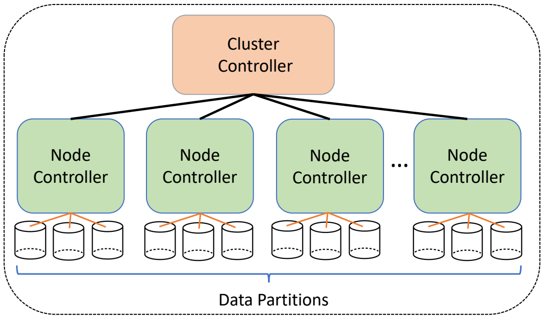


Figure 2: AsterixDB Architecture

We chose Apache AsterixDB as our primary platform for implementing and evaluating our proposed techniques for several reasons. First, it is an open-source platform that allows us to share our techniques and their evaluations with the community. More importantly, AsterixDB is a parallel big data management system for managing and processing large amounts of semi-structured data with a declarative language. Finally, its similarity in structure and design to other NoSQL and NewSQL database systems and query engines makes our results and techniques applicable to other systems as well.

3 RELATED WORK

HHJ was first proposed in [15]. It was shown to have superior performance compared to other types of joins using simple cost models, especially if a large amount of memory is available[40]. In [23], the authors provided a more detailed cost model to determine the optimal buffer allocation for various join types.

One of the key problems in configuring HHJ for execution is to choose the number of partitions into which to hash the records. In [40], the author provided an equation for calculating the number of partitions based on the memory and build input size. In [31], the authors derived an upper bound on the number of partitions and then merged smaller partitions to reduce the fragmentation in each partition, which is helpful when the join attribute values are skewed. Our paper introduces a lower bound for the number of partitions and shows how it can significantly reduce the total amount of I/O in some cases.

Another challenge for executing HHJ is to efficiently find a frame with sufficient space in the target partition for each incoming record.

This problem is similar to the Bin-Packing problem [16, 32]. The problem has also been widely studied in the operating system and the DBMS literature [34, 41] for managing free disk space. This paper will examine those algorithms and a few more for inserting records in partitions during HHJ. The difference between our work and disk-related studies is that in our work records will not reside in the partitions long term, and no deletion apart from partition spilling happens in this case.

The authors of [35] proposed a dynamic destaging scheme where the partition written to disk is selected dynamically during execution. In [19], Graefe et al. detailed the optimization techniques and the design of Dynamic HHJ variant in Microsoft SQL Server. Those two studies are closely related to our work; both choose the largest partition to be written to disk. Despite some reasoning, the authors discuss no other options, nor do they evaluate them. Our study defines 13 different possibilities and evaluates them under various record sizes and join attribute value distributions.

In a concurrent study, the authors in [10] have investigated how and when to use radix join instead of the non-partitioned hash join in a main memory DBMS. Regarding AsterixDB [3, 8, 29], the details of its default Dynamic HHJ can be found in [29].

4 NUMBER OF PARTITIONS

The first step in configuring the HHJ operator is to determine the number of the partitions for partitioning the input datasets.

There are two main constraints to be considered when choosing the number of partitions: (1) An HHJ operator needs at least two partitions to divide the input dataset into smaller subsets. (2) Each partition needs at least one output frame in order not to spill less than half-full frames to disk.

As such, the number of partitions for an HHJ should be chosen from the range of:

$$Number\ of\ Partitions = [2, \#\ of\ memory\ frames] \tag{1}$$

In [40], the author offers the following equation to calculate the number of partitions for an HHJ operator.

$$B = \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil \tag{2}$$

|R| represents the size of the build input in frames, F is a fudge factor, |M| represents the size of the memory in frames available to this join operator, and B is the number of disk-resident partitions. Based on this equation, the HHJ operator will use B+1 partitions (including a memory-resident partition) and finish in B+1 rounds.

While this equation calculates the number of partitions in a way that minimizes the total amount of I/O and rounds in HHJ, any inaccuracy in estimating its input parameter, |R|, can introduce fluctuations in the performance of HHJ as the amount of available memory varies. This is especially true when only a few partitions are created (large memory). In this case, data is distributed among just a few partitions, causing a high penalty for spilling a partition as a large amount of data will be written to disk. The purpose of this section of the paper is to provide a lower bound on the number of partitions to prevent excessive spilling due to inaccuracy of the provided information.

Figure 3 shows the result of a simulation study that explores the impact of the number of partitions on the total amount of I/O during the execution of an HHJ operator. Final result writing is excluded from this measurement. Both the build and probe inputs contain the same size of data for simplicity and the amount of memory is set to 10GB in all cases. In Figure 3-(a), a fixed number of partitions have been used for all rounds of HHJ. The black diamonds on each line show the number of partitions suggested by Eq. 2 given accurate parameter values. As Figure 3-(a) shows, if accurate input values such as input dataset sizes were provided, Eq. 2 can accurately calculate the minimum number of partitions that minimizes the total amount of I/O for HHJ. However, if there is no a priori information or if the provided information is inaccurate and the build input is larger than anticipated, Eq. 2 will suggest a smaller number of partitions than needed and cause extra I/O. As Figure 3-(a) shows, choosing a small number of partitions can lead to a large amount of unnecessary I/O and degrade the system's performance. We can, however, use Eq. 2 to calculate the number of partitions for the subsequent rounds of HHJ as the sizes of spilled partitions are known. Figure 3-(b) shows how using the spilled partition sizes to calculate the number of partitions for the next rounds of an HHJ can reduce the total amount of spilling of the HHJ operator.

We recommend using 20 as the minimum number of partitions instead of 2 when accurate a priori information is not available for the HHJ operator. As Figures 3-(a) and 3-(b) show, the amount of I/O drops dramatically before 20 partitions. By having a lower bound of 20, each spilled partition spills no more than 5% of the data, so the potential for significant "spilling error" is low.

As we saw so far, choosing too few partitions leads to a handful of large-sized partitions causing extra rounds of HHJ and a large amount of spilling to disk. On the other hand, while using a larger number of partitions can reduce the total amount of spilling, it can make the join's I/O pattern more random due to frequent writings of partitions containing just a few frames. Fragmentation within frames is another downside of having a very large number of partitions. In [31], the authors defined an upper bound for the number of partitions in order to reduce fragmentation and random writes due to too many single-frame partitions. However, to the best of our knowledge, no lower bound on the number of partitions has been suggested to improve the performance of the HHJ algorithm.

Additionally, we have studied the impact of frame size on the amount and pattern of I/Os happening during the execution of the HHJ operator. Figure 3-(c) shows the impact of the number of partitions on the amount of I/O when the frame size is set to 128KB. By comparing Figures of 3-(b) and 3-(c), we can see that changing the size of memory frames from 32KB to 128KB does not change the total amount of I/O occurred during the join execution. Figures 4-(a) and 4-(b) show the percentage of writes (excluding final result writing) that are conducted randomly when the memory frame size is 32KB and 128KB, respectively. As these figures show, using either 32KB or 128KB leads to a similar I/O pattern since for each spilling the first write is random and the rest of the data is written sequentially regardless of being a large frame or several small frames. Lastly, a lower bound of 20 partitions does not cause too many random I/Os since data will be written to only a few (at most 20) files on the disk. A modest filesystem cache can turn many of these random writes into sequential ones (Elevator Algorithm).

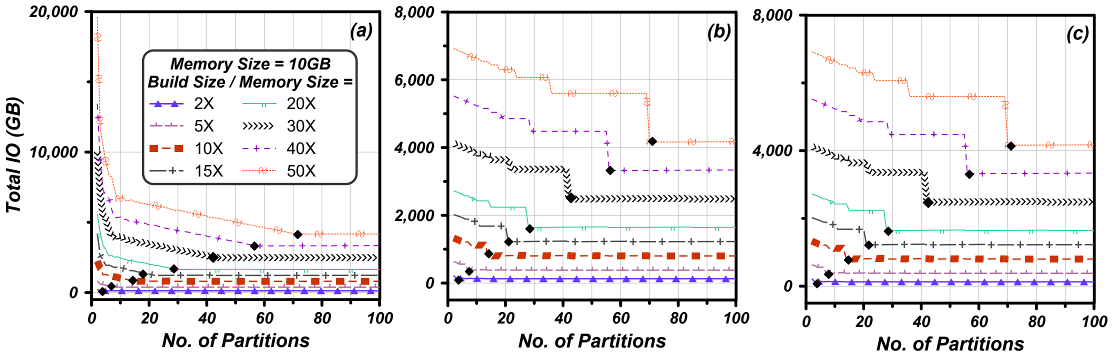


Figure 3: Impact of number of partitions on the total amount of I/O in Dynamic HHJ (excluding final result write). (a) Fixed number of partitions used in all rounds of Dynamic HHJ (Frame Size=32KB) - (b,c) Fixed number of partitions used in the first round, Eq. 2 used for rounds +2.((b). Frame Size = 32KB, (c). Frame Size = 128KB).

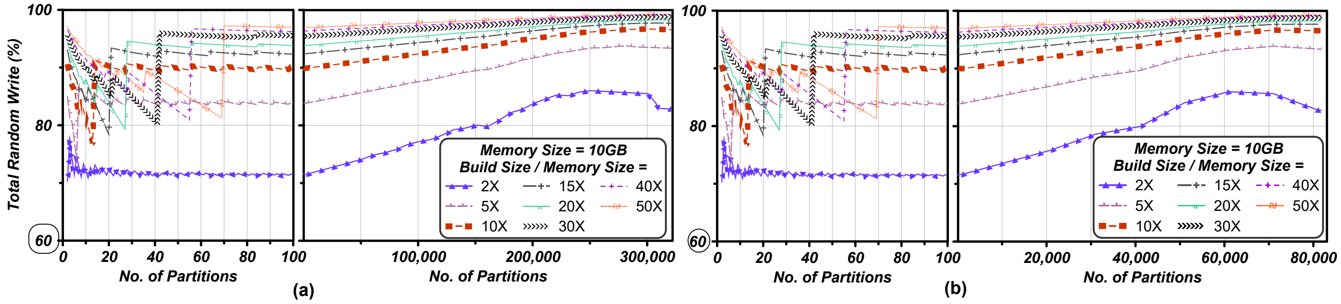


Figure 4: Ratio of random writes over total amount of writes reported in percentages (excluding writing the final results)- (a) Frame Size = 32KB, (b) Frame Size = 128KB

As Figure 4 shows, choosing a very large number of partitions can cause the majority of the writes to disk to be random.

5 PARTITION INSERTION

After choosing the number of partitions (P), the build phase starts by reading its input into memory one frame after another. The split function is applied to each incoming record's join attribute(s) to find their destination partition. Once the partition is known, we need to search for a frame with sufficient space within the destination partition to hold the record. If all of the records have the same or similar sizes, all of the previously allocated frames apart from the last frame will be similarly full. In this case, we only need to check if the last frame can hold the record or a new frame should be allocated. However, if records are variable in size, then each allocated frame may have a different amount of leftover space. Thus, for each incoming record, we need to search for a frame with enough space to hold it. Note that leftover space in frames can also happen when records are fixed-size (i.e. when the frame size is not divisible by the record size), but then all of the frames will have the same amount of free space. The search starts from the newest allocated frame and proceeds towards the oldest one. If this search is unsuccessful, a new frame will be allocated and appended to this partition's in-memory frames array if enough memory is available. However, if the available memory is not sufficient for a new frame allocation, one of the memory-resident partitions will be selected for spilling to release some memory space. This choice is called victim selection and will be discussed in Section 7.

Problem Definition. Our goal for partition insertion is to make each partition as non-fragmented as possible by choosing the destination frame for each incoming record in such a way that minimizes the free space in each frame to avoid unnecessary I/O. On the other hand, searching for a proper frame for each record could be CPU-time-consuming. Our goal is to find a destination frame efficiently while making the partition as compact as possible. Two influential factors should be considered for designing partition insertion algorithms. First, there will be no record deletions to cause fragmentation in this scenario; only a complete partition will be written to disk in case of insufficient memory. Second, records can come in many different sizes. This variation in record sizes adds to the complexity of partition insertion for two reasons. First, the space required for each record is different from other records. Second, the insertion of variable-sized records in fixed-size frames leaves a different amount of free space in each frame. Placing variable-sized objects in a fixed-size space is known as an "online object placement" or "online organization" problem. It is an example of the online bin-packing problem, a well-known NP-hard problem. Some object placement strategies have been studied and optimized for free space management on disk for permanent placement of objects [34]; however, they may not exhibit similar performance characteristics when used for memory space management.

In the following, we present and evaluate different algorithms for partition insertion. The algorithms considered here are presented in Table 1 and their detailed description can be found in an extended version of this paper [28].

Table 1: Partition Insertion Algorithm Descriptions

Algorithm	Start Point	Search Direction	Stopping Criteria
Append(8)[1]	Newest frame	Towards the oldest frame	8 frames checked or finding a frame with enough space
First-Fit	Newest frame	Towards the oldest frame	Finding a frame with enough space
Best-Fit	Newest frame	Towards the oldest frame	All frames should be checked
Next-Fit	Last insertion frame	Guided[2]	Finding a frame with enough space
First-Fit(10%)[1]	Newest frame	Towards the oldest frame	10% × $Total\ Frames$ checked or finding a frame with enough space
Random(10%)[1]	Random	Random	10% × $Total\ Frames$ checked or finding a frame with enough space

[1] [28] explains how the parameter values are chosen for these algorithms.

[2] Search will be towards the newest frames if the current record is larger than the previously inserted record; otherwise, it will be towards the oldest frames.

5.1 Dataset and Experiment Design

We use an updated and modified version of the Wisconsin Benchmark [14] data to evaluate the partition insertion algorithms. Its attributes and datasets' high tunability and selectivity make the Wisconsin Benchmark's dataset a good synthetic benchmark dataset for evaluating and benchmarking join queries.

We use variable-length records, one of the modifications added to the Wisconsin Benchmark data in [27], to introduce two groups of small-sized and large-sized records with a specific ratio between these two groups. We use what we call the 1-Large Record Coexist, 3-Large Record Coexist, and All Small Records datasets in this study, each of which is 1 GB in size. Each memory frame is 32KB in size. The names of 1-Large Record Coexist and 3-Large Record Coexist come from the number of large records that can fit in one frame. Variable-length records are used for small and large records to represent a more realistic scenario. We consider two specific ranges for large records (1-Large and 3-Large record coexist) to study the impact of semi-large and extra-large record sizes fitting in one frame to cover the two ends of the spectrum of large record sizes. Table 2 contains the details of the datasets used.

Each experiment is conducted using an AsterixDB cluster consisting of a Cluster Controller and a Node Controller with one data partition executing on two different nodes of the same AWS type. Each query runs in isolation and utilizes one CPU core. All instances are chosen from US-West-2 availability zone of Amazon AWS and have 4 vCPUs and 30.5GB of RAM. The d2.xlarge instance type was used for the HDD experiments, while i3.xlarge and r4.xlarge were used for the SSD and EBS experiments, respectively.

Table 2: Dataset Specifications

Dataset	Small Records	Large Records
1-Large Record Coexist	700 B - 1500 B	18 KB - 20 KB
3-Large Records Coexist	700 B - 1500 B	8 KB - 10 KB
All Small Records	700 B - 1500 B	None

5.2 Partition Insertion Algorithms' Evaluation

This section evaluates the performance of the described partition insertion algorithms for fixed- and variable-sized records.

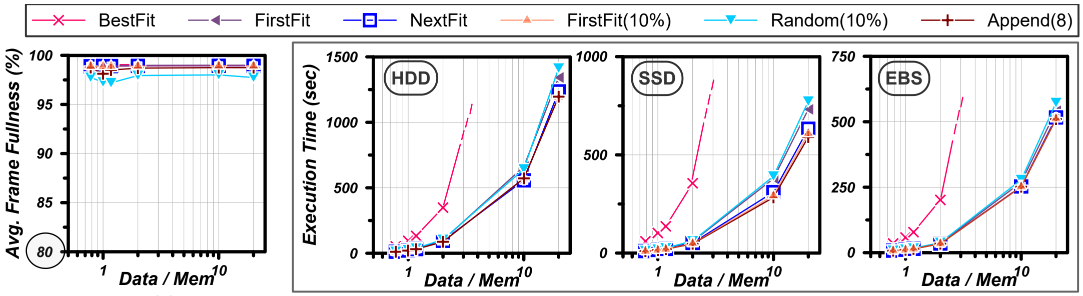
5.2.1 *Small Records Experiment.*

In our first experiment, both the build and probe datasets are 1GB in size and follow the All Small Records dataset configuration. In this experiment, we are interested in comparing the partition insertion algorithms with respect to the average frame fullness (compactness) and the query execution time to evaluate the efficiency of each algorithm in reaching this degree of frame fullness. The query execution time is the time that it took for a query to execute, excluding the time for query compilation and result returning. Since queries were running in an isolated setting with no other queries running concurrently, the execution time includes zero wait time. In these experiments, we consider different ratios of record sizes over the memory frame size. Since memory frames and records can come in many different sizes, the *ratio* of their sizes is the important factor here. Similarly, we consider various ratios between the data and memory sizes to study the performance trends of the various algorithms. Figure 5(a) shows the average frame fullness as a function of the ratio of the build dataset size to the amount of available memory. The Y-axis starts from 80% for a better visualization. As this figure shows, all algorithms deliver a high and similar average frame fullness when the records are small. This is because small records can easily fit in most frames and increase the average frame fullness by minimizing the leftover space in each frame.

Next, we analyze the performance of the different partition insertion algorithms in reaching their reported frame compactness. Figure 5(b) exhibits the execution time of the partition insertion algorithms for three storage types of HDD, SSD, and Amazon EBS. We use different storage types to study the impact of the difference in frame compactness of different partition insertion algorithms (which can lead to differences in the amount of disk I/O) on the execution time for each storage type.

The similarity in the size of the records makes the frames, especially the older ones, similarly full. Additionally, suppose a previous record could not find a frame by checking all of the partition's frames due to similarity in record sizes. In that case, it is likely that the next record will not fit in those frames either.

As Figure 5(b) shows, the CPU cost due to extensive searching in Best-Fit significantly degrades its performance in all three storage types. Random(10%) is the second-worst algorithm with a slightly higher execution time than the others. Although Random(10%) benefits from the additional stopping criteria, the high time-overhead of the Random function and the high frequency of calling it degrades its performance. First-Fit is the third-worst algorithm in our experiments. First-Fit has a higher execution time than the algorithms with a guided search method (Next-Fit) or additional stopping criteria. This is due to the extensive search of First-Fit. However, the performance of First-Fit is much better than Best-Fit, another extensive search algorithm, as First-Fit stops if it finds a suitable frame. This "first find" strategy has a high impact, especially in this experiment, as all of the records are small and have a good chance to fit in even a relatively full frame.



(a) (b)

Figure 5: Partition Insertion - Small Record Sizes (a) Average frame fullness (b) Execution time on different storage types

Next-Fit and First-Fit(10%) perform similarly here with relatively low execution times. Next-Fit's different starting point and its guided search improve its performance. The early termination due to stopping criteria in First-Fit(10%) makes it one of the best-performing algorithms here. Append(8), however, seems to be the best algorithm in this experiment. As Figures 5(a) and 5(b) show, Append(8) reaches a similar average frame fullness as the other alternatives with the least amount of search effort. (For each record, at most 8 frames are checked.)

5.2.2 ***Variable Size Records.*** This section evaluates the performance of different partition insertion algorithms with input datasets containing records of various sizes.

We use the 3-Large Record Coexist dataset for this experiment. The large records versus small records ratio varies between 10%, 50%, and 90%. As Figure 6-(a) shows, increasing the percentage of large records lowers the average frame fullness in all algorithms and minimizes their differences in frame compactness. Inserting large records in a frame may leave a large leftover space that can only be filled with small records. If the small records are limited in number (higher percentage of large records), these leftover spaces remain unfilled and decrease the average fullness. Additionally, the difference between the average frame fullness of the various algorithms diminishes if most of the records are large since only a few frames may have enough space for large records.

As Figure 6-(b) shows, Best-Fit again has the highest execution time since for each record insertion it searches all of the in-memory frames of the partition. Furthermore, a higher number of records leads to more searching and thus to a higher execution time for Best-Fit. This rationale is true for the Random algorithm, too, since the random function will be called for 10% of the frames per record insertion. In all of these experiments, Append(8) has the lowest execution time; doing the least amount of work, it still achieves a similar frame fullness to the more intelligent and search-intensive algorithms. While the algorithms other than Append(8) and Best-Fit perform similarly, the algorithms with a stopping criteria perform slightly better. Storage-wise, the overall execution time is higher for HDD than for SSD and Amazon EBS due to its longer time for I/O operations. The impact of the difference in the amount of I/O on the execution times of the different algorithms is greater in HDD due to the efficiency of SSD in handling I/O and the high network latency in Amazon EBS.

Append(8) was seen to have the lowest execution time for both small and variable sized records, so we will use Append(8) as the partition insertion algorithm for the rest of this study. We also used 1-Large Record Coexist for another variation of this experiment whose results can be found in [28]. Append(8) also did well there.

6 SPILLED PARTITIONS' GROWTH POLICIES

In the case of insufficient memory, some of the partitions must be written to disk to open up space for additional incoming records. We will consider several victim selection policies - policies which select a memory-resident partition to spill - under two variations of how the memory allocation to spilled partitions is managed:

(1) **No Grow-No Steal (NG-NS):** There are two main rules for this policy:
   - **No Grow**: A spilled partition can only have one frame to be used as its output buffer once it has spilled.
   - **No Steal**: Only unspilled partitions are selected as victims in case of insufficient memory. A spilled partition writes its output buffer to disk only if the next record hashed to that partition requires more space.

(2) **Grow-Steal (G-S):** This growth policy consists of two main rules as well:
   - **Grow**: Spilled partitions may grow as large as the available memory lets them.
   - **Steal**: Spilled partitions have a higher priority to be chosen as a victim partition in cases of insufficient memory.

While more growth policies could be considered for future work (e.g., spill only the full frames of a victim partition), we chose to study these two growth policies as the two ends of the spectrum.

6.1 Analytical I/O Study for NG-NS and G-S

In this section, we look at the I/O differences between the two growth policies for spilled partitions from an analytical point of view. It is important to realize that both policies perform almost the same amount of I/O; however, they differ from one another in their use of random versus sequential I/O. In the following equations, **R** represents the size of the build input in frames, **M** is the number of available memory frames, **P** is the number of partitions, and **x** is the number of spilled partitions.

Let us assume that records are similar in size and that there is no skew in join attribute values. Using this assumption, all partitions are similar in size, in the number of frames, and in the number of records. The following equation calculates the amount of temporary results (build phase only) written to disk in a random and sequential fashion under the NG-NS growth policy:

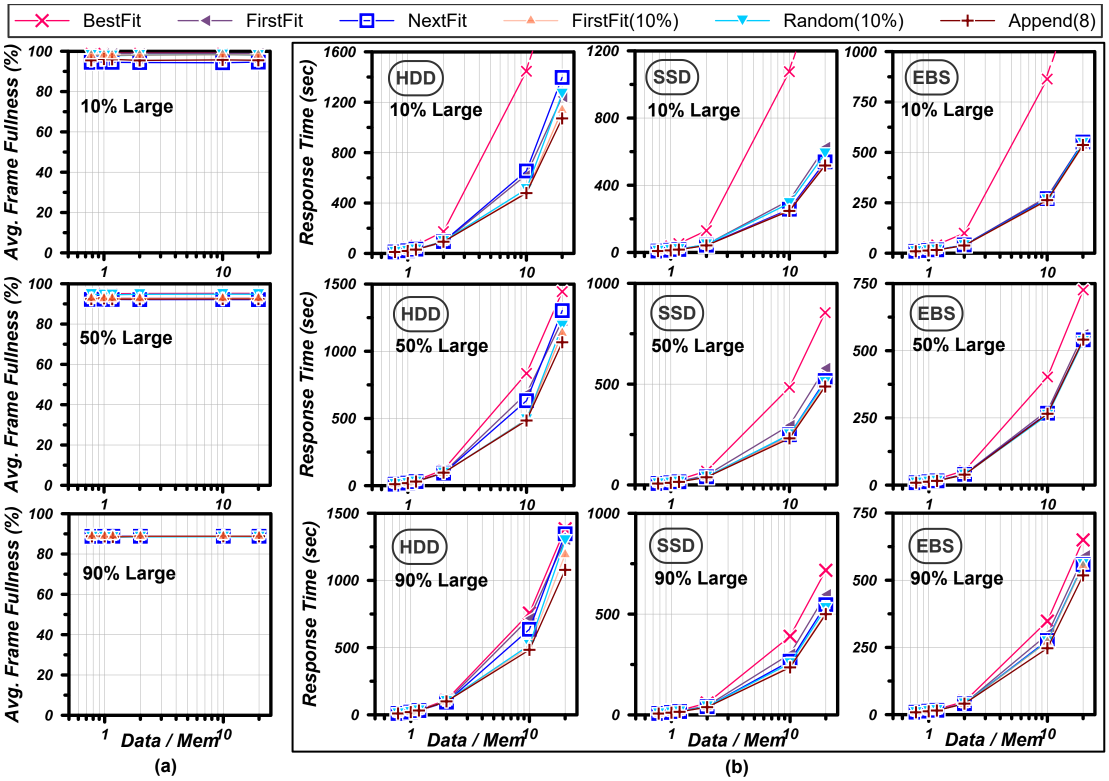


Figure 6: Partition Insertion - 3-Large Record Coexist (a) Average frame fullness (b) Execution time on different storage types

$$\sum_{i=1}^{x}\left(\frac{R}{P}-\frac{M-i+1}{P-i+1}\text{ Random I/O}\right)+\left(\frac{M-i+1}{P-i+1}\text{ Seq. I/O}\right) \qquad (3)$$

The following equation calculates the number of I/Os for G-S:

$$\sum_{i=1}^{x}\left(\lim_{P\to\infty}\frac{1}{1-\frac{1}{P}}\left(\frac{M-i+1}{P-i+1}\right)\text{ Seq. I/O}\right)+\left(\frac{M-i+1}{P-i+1}\text{ Seq. I/O}\right) \qquad (4)$$

The first term in Eq. 4 shows that in G-S, each spilled partition writes the rest of its data to disk sequentially. This I/O behavior is different from NG-NS (Eq. 3), in which the rest of a partition's data is written to disk frame by frame once it first spills. More detailed explanations of these two cost formulas can be found in [28].

6.2 Experimental Analysis of Growth Policies

Based on the cost functions we developed in the previous subsection, we showed that the NG-NS policy leads to more random writes due to using one output buffer allocation per spilled partition. On the other hand, G-S allows the spilled partitions to acquire more than one frame, so its I/O pattern becomes more sequential. Turning random writes into sequential ones can improve performance, especially in systems utilizing HDD. This section compares these two algorithms empirically to verify our expectations from the cost analysis. We used a single join query for which the build and probe datasets contain identical data generated based on the All Small Record dataset configuration. In this experiment, the available memory for the join is a fixed value of 1024MB, while the size of the build and probe inputs varies from 1.2GB, 2GB, 10GB, 20GB, to 100GB. A hard disk is used as the storage device in this experiment.

This experiment compares the two growth policies for spilled partitions under two variations of writing to disk: direct or through the filesystem cache. Some database management systems disable the filesystem cache and manage the buffer cache memory themselves. We use the IO_DIRECT library [5] for directly writing data to disk and bypassing the filesystem cache in Linux systems. Figures 7-d and 7-h show that G-S and NG-NS do the same amount of writing regardless of using or bypassing the filesystem cache. However, as Figures 7-c and 7-g show, G-S does up to 120x more sequential writes than NG-NS, while NG-NS does up to 120x more random writes than G-S (Figures 7-e and 7-f). This difference in the I/O patterns of the G-S and NG-NS while writing the same amount of data to disk aligns with our results from the previous subsection. Increasing the input sizes causes more spilling to disk, making the difference between these two policies even more significant.

Next, we study the performance of these growth policies with and without filesystem cache being present. Figure 7-e shows the execution time of G-S and NG-NS policies when data is written directly to disk (disabled filesystem cache). In this case, NG-NS takes a longer time than G-S to finish due to performing more random writes. The impact of random writes of NG-NS on its performance becomes more significant as the size of the data relative to memory increases; this is because more data is written randomly and the storage device is an HDD. However, Figure 7-a shows that using a filesystem cache minimizes the difference in execution times of these two policies. This is because the filesystem cache collects write requests and orders them based on their target file location on disk (Elevator Algorithm) before sending them to disk; as a result, many of the random writes turn into sequential ones in NG-NS.

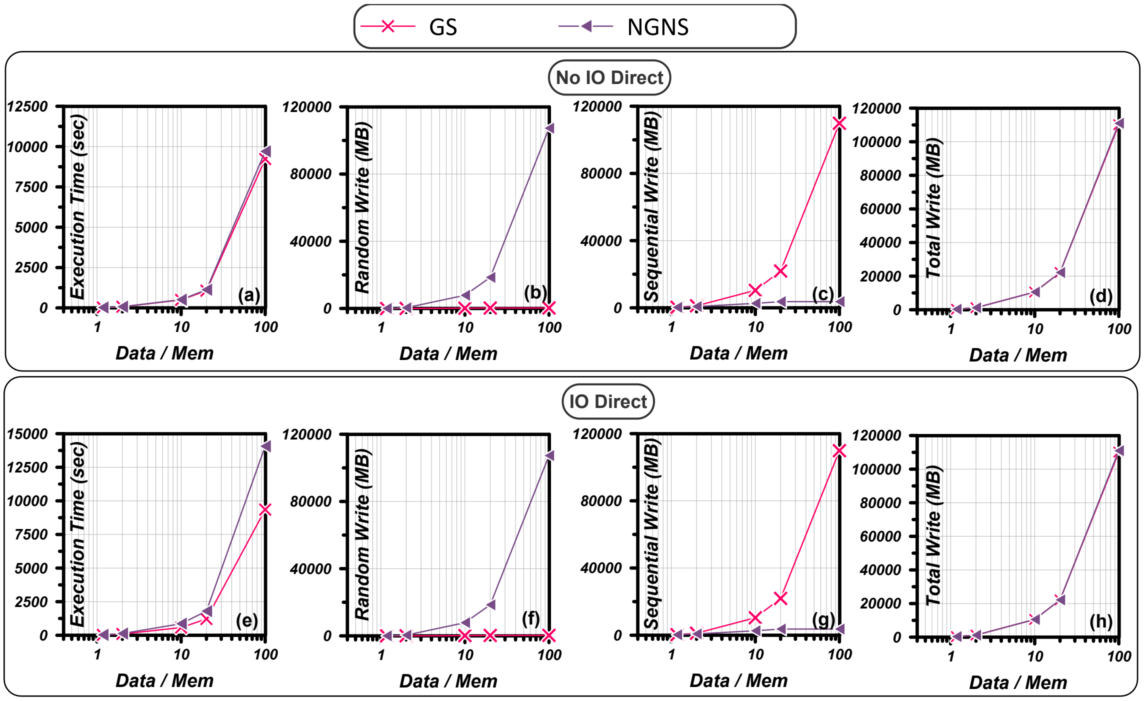


Figure 7: Spilled Partition Growth Policies. (a,b,c,d) - Statistics of GS and NG-NS policies with filesystem cache in use. (e,f,g,h) - Statistics of GS and NG-NS policies with filesystem cache disabled.

Based on our results, choosing the preferred growth policy depends on whether the DBMS performs its own caching or uses the filesystem cache. In AsterixDB, we decided to use NG-NS for two reasons: (1) The filesystem cache is used. (2) NG-NS does not fully utilize its given memory. In future work, we intend to use this leftover memory for other operators of the same query or other queries under a more global memory management policy.

7 VICTIM SELECTION POLICIES

One or more memory-resident partitions must be written to disk to regain enough space for the incoming records if the available memory is insufficient. In-memory partitions may have different sizes if records have variable sizes or if their distribution between partitions is unbalanced due to skew in join attribute values. In the case of variable-sized partitions, we must decide which partition(s) should spill to disk, considering that we do not know how much data is left to be processed. The partition selected for spilling is called a victim partition, and the policy based on which victim partitions are selected is called the victim selection policy.

In the original HHJ algorithm [15, 40], one partition is selected upfront (before query execution) as the in-memory partition, while the rest of the partitions are disk partitions. To ensure that the chosen partition can indeed remain in memory, we must know the sizes of the inputs and the distribution of join attribute values.

As mentioned earlier, the authors of [35] and [19] instead use dynamic destaging to choose victim partitions at runtime. They always select the largest memory-resident partition as the victim partition and limit the spilled partitions to acquiring a maximum of one frame, following the NG-NS growth policy. Neither of these studies considers other victim selection policies or spilled-partition growth policies. Additionally, they do not provide any experiments to show the superiority of their approach.

In the following, we consider 13 possible policies for selecting the next victim partition among non-spilled partitions. These victim selection policies are designed for the NG-NS growth policy.

The design space for these policies is based on data size and frame fragmentation considerations. Largest Size, Largest Records, Median Size, Median Records, Smallest Size, Smallest Records, Record Size Ratio, Half Empty, and Low High are designed with respect to the data size. The Largest Records and Largest Size are expected to perform well when a large portion of the build dataset is left to be processed. In contrast, the Smallest Records and Smallest Size are expected to perform well when a small portion of the build input remains to be processed. Record Size Ratio considers the number of records in choosing a large partition as victim and is expected to perform well when a large portion of the build input remains to be processed. Half Empty, Low High, Median Size, Random, and Median Records are designed to take a middle ground and are expected to have an average but stable performance for various cases. Least Fragmentation considers the frames' fragmentation to choose a victim and is expected to have an average amount of spilling since it may choose a partition of any size to spill. Smallest Size Self Victim and Largest Size Self Victim are policies which take both the frame fragmentation and data size into consideration and are expected to have an average amount of spilling since the victim partition can be of any size when it is the inserting partition itself.

The overall input dataset sizes are unknown to the DBMS during these experiments. The following list describes the considered victim selection policies:

Largest Size: Choose the partition with the largest size in memory as a victim to maximize sequential writes and to defer the next spill(s) as long as possible.

Largest Records: Choose the partition with the maximum number of records to spill.

Largest Size Self Victim: Choose the partition into which the record is hashed if it has at least one frame. Otherwise, choose the largest partition to spill.

Median Size: Choose the partition with the median size among all of the memory-resident partitions as the victim partition.

Median Records: Choose the partition with the median number of records to spill.

Smallest Size: Choose the smallest partition with at least one memory frame as the victim partition to avoid overspilling.

Smallest Records: Choose the memory-resident partition with the minimum number of records (>=1) for spilling.

Smallest Size Self Victim: Choose the partition into which the record is hashed to spill if it has any frames. Otherwise, the smallest-size partition will be selected as the victim.

Random: Choose randomly any of the memory-resident partitions as the victim partition.

Half Empty: This victim selection policy starts optimistically by guessing that the remainder of the build input is small and spills the smallest partition. However, it acts pessimistically and spills the largest partition if more than half of the partitions have spilled.

Least Fragmentation: Choose those partitions that have the least amount of fragmentation in their frames, thus trying to reduce I/O.

Low High: Alternate between spilling the smallest and the largest partition.

Record Size Ratio: Choose a partition that holds the smallest number of records among partitions whose size is equal to or exceeds 80% of the largest partition size (low ratio of the number of records to the partition size); this expedites record processing by storing more records in the memory.

7.1 Victim Selection Policy Experiments

This section studies the impacts of join attribute value skew and record size variation on the victim selection policies.

7.1.1 *Impact of Join skew.*

In our first experiment, we study the impact of join attribute value skew on the 13 different victim selection algorithms. In Figure 8-a, both the build and probe datasets use the All Small Record configuration, and the join attribute values are unique integers (Non Skewed join attribute value case).

In Figure 8-b, the join attribute values of the build dataset are integers drawn from a Normal Distribution to make them skewed, while the probe dataset uses unique integers as its join attribute (Skewed join attribute value case). Both relations are 1GB in size and contain 985, 000 records. The authors of [12, 38] used a Normal Distribution in which 99% of the join attribute values are coming from 5% of the possible values, justifying this as similar to the skew found in real-world data. To achieve this data skew, we use a Normal Distribution on an integer attribute with the mean of 492500 (equal to half of the cardinality), a standard deviation of 8208, and a range of possible values varying from 1 to the dataset cardinality.

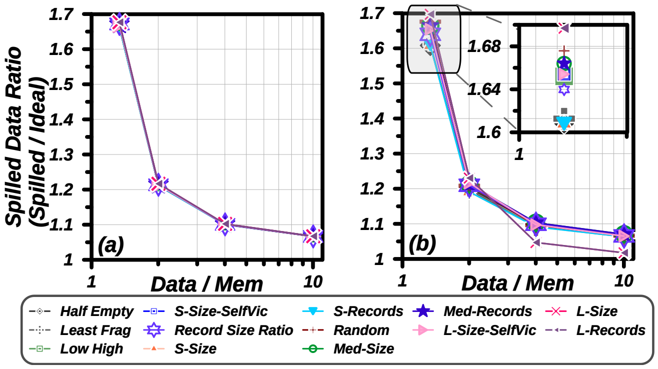


Figure 8: Impact of Join Attribute Value Skew in Victim Selection Policies. (a) - No skew. (b) - Skewed.

The metric used in Figure 8 is the ratio of the amount of spilled data over the ideal amount of spilling. The ideal amount of spilling is the minimum amount of data that must be spilled to disk during the build phase. We determine this ideal amount by using a simple simulation program. This simulator minimizes the data spilling by maximizing the memory used in each round of HHJ by the in-memory partition. Eq. 2 with accurate a priori information and with a fudge factor of 1.4 is used in this simulator to ensure that the amount of spilling is minimal.

As Figure 8-a shows, all of the algorithms have a similar performance if records are similar in size and the join attribute values are uniformly distributed. Figure 8-b shows that skew in the join attribute values can cause different spilling behavior for some victim selection policies. In Figure 8-b, the Largest-Size and Largest-Record policies overspill when data is slightly larger than the available memory. However, as the data size increases, spilling the larger partitions releases more frames, saving other partitions from spilling.

The Smallest-Size and Smallest-Record policies, which spill less data initially, will spill more when the ratio of data to memory is higher. All other policies show a spilling behavior that lies between these two categories of policies. However, the overall difference between most of the policies is almost insignificant.

7.1.2 *Impact of Variable-Sized Records.*

Next, we study the impact of variable-sized records on the performance of the victim selection policies. We used a set of 1GB relations based on the 1-Large Record Coexist and 3-Large Record Coexist dataset configurations. As Figures 9 and 10 show, most of the policies perform similarly as the ratio of data over memory is increased in both experiments. The Largest-Size and Largest-Record policies spill less data and fewer partitions to disk than the other victim selection policies in most of the data points. This is because the number of frames that larger partitions free can save more partitions from spilling.

In both Figures 9 and 10, increasing the population of large records leads to a larger difference between victim selection policies. The variations in the size of the records and the high impact of large records on the partitions' sizes, compared to fixed sized records in Figure 8-a, make it possible to see differences between these victim selection policies. In both 1-Large Record Coexist and 3-Large Record Coexist cases (Figures 9 and 10), the Largest-Size, Largest-Records, and in some cases Largest-Size-Smallest-Record policies spill the least amount of data and the fewest number of

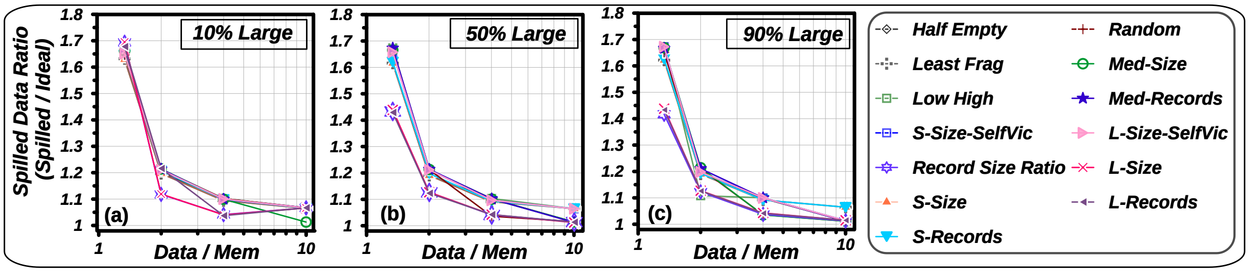


Figure 9: Impact of Variable Record Size (1-Large Record Coexist) in Victim Selection Policies. (a,b,c) - Spilled Data Ratio when 10%, 50%, and 90% of the records are large, respectively.

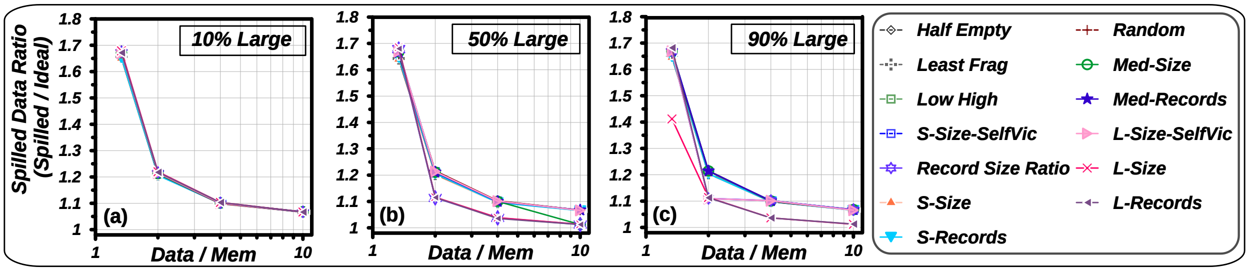


Figure 10: Impact of Variable Record Size (3-Large Records Coexist) in Victim Selection Policies. (a,b,c) - Spilled Data Ratio when 10%, 50%, and 90% of the records are large, respectively.

partitions in most of the data points by spilling the largest partitions first. This difference between policies in the 3-Large Record Coexist experiment is less obvious since the large records are 1/3 of the size of the large records in 1-Large Record Coexist dataset. In Figure 9-a most policies perform similarly as there are fewer large records thus, less opportunity for these policies to perform differently.

7.1.3 *Impact of Join Skew & Variable-Sized Records*.

In this experiment, we study the impact of the combination of join attribute value skew and variable-sized records on the proposed victim selection policies. The Normal distribution discussed in Section 7.1.1 is used for making the build dataset skewed. The record sizes are chosen from the same distribution used for 1-Large Record Coexist (Figure 11) and 3-Large Record Coexist (Figure 12) cases. The probe inputs have the same cardinality and record size distribution as the build input, while their join attributes are unique integers.

Similar to the previous experiment, Largest-Size and Largest-Record are two well-performing policies when larger records have a lower population. The Median Size and Median Records policies perform well by taking a middle route if data is skewed and most of the records are large. The skew in data makes some partitions get more records; partitions with more records will have larger sizes if records are mostly large-sized, and thus the Largest-Size and Largest-Record algorithms can overspill. In the case of very limited memory for the 1-Large Record Coexist case (the first data point in Figure 11-a, 11-b, and 11-c), Smallest-Records and Smallest-Size are two of the best performing policies. Since most of the data is located in a few partitions, there are many small partitions with only a few frames. As such, Smallest-Records and Smallest-Size can avoid overspilling by spilling these small partitions when data is just slightly larger than memory.

In the 3-Large Records Coexist case, the victim selection policies' performance is similar to the 1-Large Record Coexist case with the difference that algorithms such as Median Records also perform well in this case due to the smaller sizes of large records. Largest-Size and Largest-Records tend to write larger numbers of frames sequentially, while others such as Smallest-Size and Smallest-Records write a smaller number of frames in a more random manner. As our experiments for G-S and NG-NS showed, this difference in their I/O patterns may not impact performance as much as otherwise expected if filesystem caching is enabled.

7.2 Results for Victim Selection Policy

Based on our experiments in the previous subsection, the Largest-Size and Largest-Record policies result in less I/O in most cases than the other alternative policies. Our results confirm the conjecture of [19, 35] that the Largest-Size policy (as well as the Largest-Record policy, based on our results) is a good selection policy for the following two reasons: (1) Larger partitions release many frames; thus, they save other partitions from spilling to disk. (2) Writing larger partitions leads to more sequential and less random writes.

However, our results also show that the difference in the amount of spilled data makes only a slight difference in performance. The gained benefits for having a more sequential pattern by spilling larger partitions are diminished if filesystem caching is enabled.

8 SINGLE CORE VS. MULTI-CORE

So far all of our experiments have used one thread for executing join queries in one data partition. In this section, we study the impact of the number of partitions and threads (and hence the cores) on the performance of Dynamic HHJ for various storage types.

For these experiments, both the build and probe datasets contain 1GB of data following the All Small Records dataset design. We used one Node Controller with 1,2, and 4 data partitions to utilize 1,2, and 4 CPU cores respectively. All joins use Append(8) as the

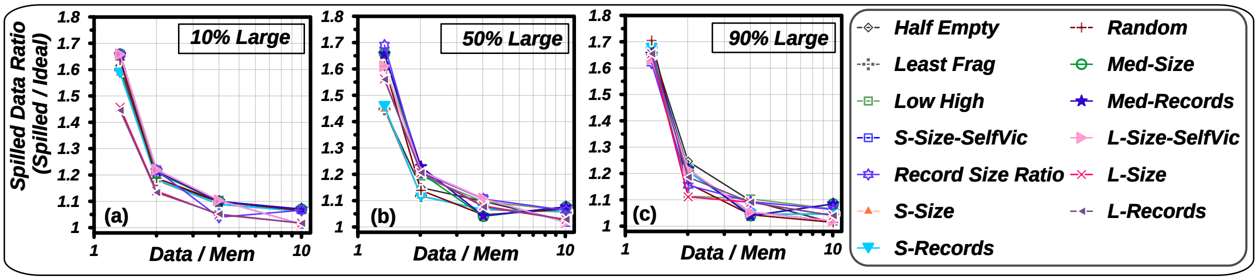


Figure 11: Impact of Skew & Variable Record Sizes (1-Large Record Coexist) in Victim Selection Policies.

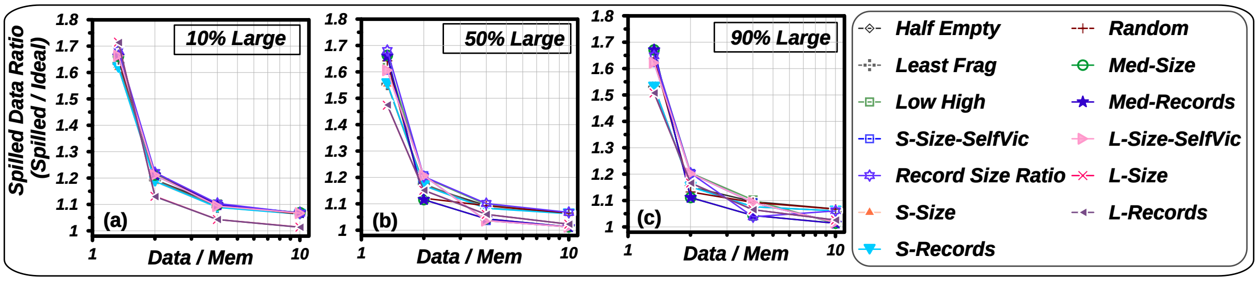


Figure 12: Impact of Skew & Variable Record Sizes (3-Large Records Coexist) in Victim Selection Policies.

partition insertion algorithm and NG-NS and Largest Size as their Growth and Victim Selection policies, respectively. Figure 13 shows

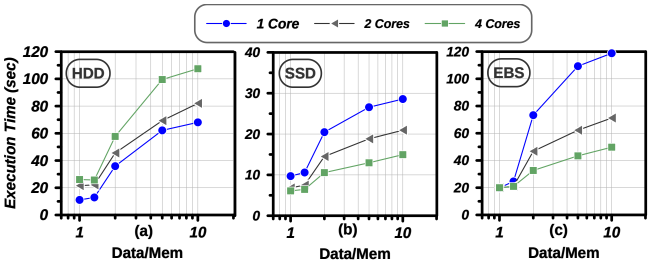


Figure 13: Impact of number of cores on the performance of Dynamic Hybrid Hash Join (a)- HDD, (b) - SSD, (c) - EBS

the execution time of a single join executed on HDD, SSD, and Amazon EBS. As this figure shows, increasing the number of cores (threads) causes disk contention and degrades the performance for HDD. On the other hand, SSD and Amazon EBS benefit from more worker threads due to the efficiency of SSD storage device in handling random I/Os. The overall execution time of Amazon EBS is higher than that of SSD due to the network latency associated with over-the-network storage.

9 CONCLUSION AND FUTURE DIRECTIONS

Our experimental study has investigated different policies to design a robust Dynamic HHJ operator when no accurate a priori information about the input datasets is available.

Although previous studies have suggested an upper bound for the number of partitions, no lower bound for this parameter has been proposed to the best of our knowledge. Not having a reasonable lower bound can lead to having too few partitions, causing detrimental overspilling. Based on a simulation study, we recommend using 20 as a minimum number of partitions so that each spilled partition writes only 5% or less of the build input to disk.

Furthermore, we have explored different partition insertion algorithms for incoming records to find a frame with enough space among a partition's in-memory frames. Append(8) showed the best performance among the partition insertion algorithms.

Next, we considered two potential post-spilling growth policies for spilled partitions, Grow-Steal and No Grow-No Steal. Our cost model showed that Grow-Steal should perform better than No Grow-No Steal due to doing more sequential I/O. However, our experiments showed that a modest file system cache can mitigate this difference by turning most random I/Os into sequential ones.

Additionally, we designed and evaluated 13 different victim selection policies. Our results confirmed the conjecture in previous work that the Largest Size policy is one of the best policies in most cases. However, this difference was not large enough to significantly impact overall system performance.

As a future direction, we would like to compare the performance of Dynamic HHJ with the radix join algorithm suggested in [10].

ACKNOWLEDGMENTS

This work has been supported by NSF awards IIS-1838248, CNS-1925610, and IIS-1954962 along with industrial support from Google and support from the Donald Bren Foundation (via a Bren Chair).

REFERENCES

[1] 1993. TPC Benchmark^{TM} A: Standard Specification. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, Jim Gray (Ed.). Morgan Kaufmann.
[2] 2021. *"Amazon Elastic Block Store"*. https://aws.amazon.com/ebs/
[3] 2021. *"Apache AsterixDB"*. https://asterixdb.apache.org
[4] 2021. *"Apache Spark"*. https://spark.apache.org
[5] 2021. *"IO_Direct Java Library"*. https://github.com/smacke/jaydio

[6] 2021. *"Mersenne Twister Fast"*. https://cs.gmu.edu/~sean/research/mersenne/MersenneTwisterFast.java

[7] 2021. *"Oracle"*. http://www.oracle.com/

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

[9] Carrie Ballinger. 1993. TPC-D: Benchmarking for Decision Support. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, Jim Gray (Ed.). Morgan Kaufmann.

[10] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 168–180. https://doi.org/10.1145/3448016.3452831

[11] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. 1983. Benchmarking Database Systems A Systematic Approach. In *9th International Conference on Very Large Data Bases, October 31 - November 2, 1983, Florence, Italy, Proceedings*, Mario Schkolnick and Costantino Thanos (Eds.). Morgan Kaufmann, 8–19.

[12] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Raghunath Nambiar and Meikel Poess (Eds.), Vol. 10661. Springer, 103–119.

[13] Alain Crolotte and Ahmad Ghazal. 2011. Introducing Skew into the TPC-H Benchmark. In *Topics in Performance Evaluation, Measurement and Characterization - Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.), Vol. 7144. Springer, 137–145.

[14] David J. DeWitt. 1991. The Wisconsin Benchmark: Past, Present, and Future. In *The Benchmark Handbook for Database and Transaction Systems (1st Edition)*, Jim Gray (Ed.). Morgan Kaufmann, 119–165.

[15] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 1–8.

[16] György Dósa and Jirí Sgall. 2014. Optimal Analysis of Best Fit Bin Packing. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I (Lecture Notes in Computer Science)*, Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.), Vol. 8572. Springer, 429–441.

[17] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1197–1208.

[18] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

[19] Goetz Graefe, Ross Bunker, and Shaun Cooper. 1998. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 86–97.

[20] Jim Gray (Ed.). 1993. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*.

[21] Jim Gray. 1993. Database and Transaction Processing Performance Handbook. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*.

[22] Jim Gray. 2005. A "Measure of Transaction Processing" 20 Years Later. *IEEE Data Eng. Bull.* 28, 2 (2005), 3–4.

[23] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. 1997. Seeking the Truth About ad hoc Join Costs. *VLDB J.* 6, 3 (1997), 241–256.

[24] Rui Han, Lizy Kurian John, and Jianfeng Zhan. 2018. Benchmarking Big Data Systems: A Review. *IEEE Trans. Serv. Comput.* 11, 3 (2018), 580–597.

[25] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1035–1050.

[26] Shiva Jahangiri. 2020. Re-evaluating the Performance Trade-offs for Hash-Based Multi-Join Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference, June 14-19, 2020.* ACM, online conference [Portland, OR, USA], 2845–2847.

[27] Shiva Jahangiri. 2021. Wisconsin Benchmark Data Generator: To JSON and Beyond. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2887–2889.

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

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

[30] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Cardinalities with Deep Sketches. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1937–1940.

[31] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. 1989. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, Peter M. G. Apers and Gio Wiederhold (Eds.). Morgan Kaufmann, 257–266.

[32] Frank M. Liang. 1980. A Lower Bound for On-Line Bin Packing. *Inf. Process. Lett.* 10, 2 (1980), 76–79.

[33] Patrick Martin, Per-Åke Larson, and Vinay Deshpande. 1994. Parallel Hash-Based Join Algorithms for a Shared-Everything. *IEEE Trans. Knowl. Data Eng.* 6, 5 (1994), 750–763.

[34] Mark L. McAuliffe, Michael J. Carey, and Marvin H. Solomon. 1996. Towards Effective and Efficient Free Space Management. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 389–400.

[35] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. 1988. Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*, François Bancilhon and David J. DeWitt (Eds.). Morgan Kaufmann, 468–478.

[36] HweeHwa Pang, Michael J. Carey, and Miron Livny. 1993. Partially Preemptive Hash Joins. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, Peter Buneman and Sushil Jajodia (Eds.). ACM Press, 59–68.

[37] Francois Raab. 1993. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, Jim Gray (Ed.). Morgan Kaufmann.

[38] Donovan A. Schneider and David J. DeWitt. 1989. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). ACM Press, 110–121.

[39] Omri Serlin. 1993. The History of DebitCredit and the TPC. In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, Jim Gray (Ed.).

[40] Leonard D. Shapiro. 1986. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst.* 11, 3 (1986), 239–264.

[41] Anton M. van Wezenbeek and Willem Jan Withagen. 1993. A survey of memory management. *Microprocess. Microprogramming* 36, 3 (1993), 141–162.