Memory Management in Complex Join Queries: A Re-evaluation Study

Shiva Jahangiri[*]
Santa Clara University
Santa Clara, USA
sjahangiri@scu.edu

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

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

ABSTRACT

Efficient multi-join query processing is crucial but remains a complex, ongoing challenge for high-performance data management systems (DBMSs). This paper studies the impact of different memory distribution techniques among join operators on different classes of multi-join query plans under different assumptions regarding memory availability and storage devices such as HDD and SSD on Amazon Web Services (AWS). We re-evaluate the results of one of the early impactful studies from the 1990s that was originally done using a simulator for the Gamma database system.

The main goal of our study is to scientifically re-evaluate and build upon previous studies whose results have become the basis for the design of past and modern database systems, and to provide a solid foundation for understanding basic "join physics", which is essential for eventually designing a resource-based scheduler for concurrent complex workloads.

CCS CONCEPTS

• **Information systems** → **Query planning**; **Query optimization**; *Join algorithms.*

KEYWORDS

Memory Management, Query Plan, Multi-Join, Hash Join

ACM Reference Format:
Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. 2024. Memory Management in Complex Join Queries: A Re-evaluation Study. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3698038.3698565

[*]This work was done while the author was at the University of California, Irvine.

1 INTRODUCTION

Multi-join queries, among the most important and common in DBMSs, have been a research focus for decades, particularly in terms of their processing and performance evaluation. A key challenge in processing these queries lies in selecting an appropriate query plan, determining the join order, and employing effective memory distribution techniques to optimize performance. Despite years of research, they continue to be poorly understood with respect to their dynamic behavior and memory usage due to the complexity and multidimensional nature of multi-join queries.

For a scientific approach, it is essential to reproduce the results of prior studies before proceeding with more complex cases. Accordingly, we decided to re-evaluate the results of a key study done by Schneider & DeWitt [36] [10] in 1990 first in which they studied the performance of multi-join queries for shared-nothing clusters. They used the Hybrid Hash Join operator (which is still extensively used today) as their join operator and used a Gamma [10] DBMS simulator based on HDD storage.

In this study, we evaluate various query plan shapes for multi-join queries and examine the impact of different memory allocation and intra-query parallelism techniques. We include the Left Deep Tree (LDT), Right Deep Tree (RDT), Static Right Deep Tree (Static-RDT), and a Bushy Tree (BT). Motivated by hardware advancements and the cloud environment, we re-evaluate previous results, which were based on simulation, using Apache AsterixDB on both HDD and SSD on AWS.

The contributions of this paper are: 1) Re-evaluating query plan studies from [36] and [10] with larger data sizes on real hardware, 2) including sample BTs in query plans, 3) studying the performance of these plans on SSDs, and 4) using a new evaluation metric called Gigabyte*Seconds to compare the monetary costs of different query plans executed in the cloud.

The remainder of this paper is organized as follows: Section 2 provides background information on stage-based query execution and Apache AsterixDB's design. Section 3 discusses the different dimensions considered in our experiments for evaluating various query plans, including query plan shapes, memory distribution approaches, and storage architecture. Section 4 presents the settings and results of the experiments conducted in this study. Section 5 reviews previous work related to this study, before Section 6 concludes the paper and provides directions for future research.

2 BACKGROUND

2.1 Stage-Based Query Execution

In a parallel DBMS, each query tree consists of operators and data flow connectors, forming an activity dependency graph with operators as nodes and data flow edges as links [4, 5, 15, 22, 41]. Hybrid Hash Join (HHJ) is a two-phase join where the start of the probe phase depends on the completion of the build phase, creating a blocking dependency.

An activity cluster, or pipelined stage, is a group of activities connected by data flows without blocking dependencies, allowing them to execute concurrently [5]. When an operator such as HHJ introduces a blocking dependency, a new activity cluster is created to separate stages of execution. These dependencies form a partial or total order of execution in the query. By executing in stages, the system improves predictability and resource management, allocating memory only to the operators of the active stage.

2.2 AsterixDB

Apache AsterixDB [3] 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. AsterixDB's architecture consists of a Cluster Controller (CC) and one or more Node Controllers (NC). The CC is responsible for receiving queries, parsing and optimizing them, and providing query plans as executable Directed Acyclic Graphs (DAGs), *a.k.a.* jobs, to the NCs. The NCs are the worker nodes that execute the job DAGs on their portions of data (data partitions) and return the results. Each NC can have one or more data partitions, and each job DAG will be executed on each related data partition in parallel. AsterixDB utilizes Log-Structured Merge (LSM) trees for storing and indexing the records in a single or multi-node cluster. All records are hash partitioned to data partitions of the cluster based on their primary key. AsterixDB supports various join algorithms, including Block Nested Loop Join, Hybrid Hash Join (HHJ), Broadcast Join, and Indexed Nested Loop Join. However, HHJ is the default and primary join type for processing equijoins due to its superior performance and wide usage in modern DBMSs.

We chose AsterixDB as our primary platform for implementing and evaluating our proposed techniques for several reasons. First, AsterixDB is an open-source platform that gives us the capability to implement and evaluate our techniques and share them with the community. More importantly, AsterixDB is a parallel big data management system for large semi-structured data with a declarative language. Finally, its similarity in structure and design to other parallel SQL and NoSQL database systems makes our results and techniques applicable to other systems as well.

3 DESIGN SPACE

Our work studies the performance of multi-join queries in a three-dimensional design space. The first design dimension is the query plan shape, which includes LDT, RDT, and BT. As the next design dimension, we consider various memory management techniques for distributing memory between the join operators of a query, including equal and bottom-up memory management techniques. As the last design dimension, we consider three different storage alternatives, including HDD and SSD, evaluating the performance of multi-join queries executing with different values for the first two dimensions on these storage alternatives.

In the next sections we explain these design dimensions and their possible variations in greater depth.

3.1 Dimension 1: Query Shapes

Multi-join queries can be executed using three main query shapes: Left-Deep Trees (LDT), Right-Deep Trees (RDT), and Bushy Trees (BT). Fig. **??** shows examples of LDT, RDT, and BT, with each enclosed dashed area representing an activity cluster and stage. In LDT, the output of each probe phase feeds into the next join's build phase, allowing at most two joins to be active simultaneously. This creates a sequential query plan with memory shared between consecutive joins, with execution order defined by intra-operator control dependencies.

RDTs offer the highest parallelism among query plans. All build phases execute concurrently, sharing memory across joins. For *n* joins, *n* hash tables are created. Once all builds are complete, the probe phases start simultaneously, with records flowing through a pipeline from one join to the next.

BTs combine aspects of both LDTs and RDTs. They allow joins to run in parallel or sequentially, with inputs that can be non-base datasets. BTs benefit from independent parallelism, enabling concurrent execution without blocking dependencies; however, the flexibility of BTs complicates scheduling and resource estimation. We use techniques from [28] to generate sample BTs.

3.2 Dimension 2: Memory Management

Memory significantly influences the choice of query plans for multi-join queries. A DBMS aims to select a query plan and memory distribution that reduces execution time.

In a LDT, memory is always distributed between two adjacent joins in the query plan. Memory distribution in BTs must consider overlapping join executions. A precise approach controls and orders independent activity clusters,

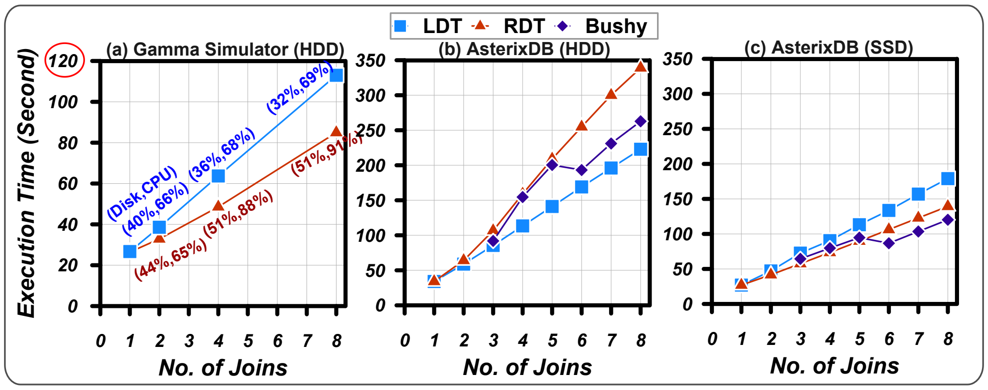

Figure 1: Experiment 1 - Execution Time

while a simpler method divides memory equally among join operators. We chose the latter for simplicity in this work. Next, we introduce the memory distribution strategies for the RDT query shape.

3.2.1 Equal Memory Distribution. In the Equal Memory Distribution strategy for an RDT, each join receives an equal share of memory. If some joins need less memory, the excess can be used by others. A DBMS with accurate knowledge of build input sizes and join selectivities can statically assign memory to enable this sharing among join operators.

3.2.2 Bottom-Up Memory Distribution. In the bottom-up Memory Distribution strategy, the DBMS assigns each join operator its ideal memory from the bottom of the query plan to prevent spilling. Known as "Static-RDT" by [9], this approach avoids data spilling. In case of insufficient memory for the whole query to fit in memory, the query plan is "broken" by materializing the last fitting join's output, which is then used as the probe input for the next join.

3.3 Dimension 3: Storage Architecture

In the third dimension, we study the performance of multi-join query plans across different storage types, mainly using HDDs and SSDs. HDDs rely on mechanical arms, making them inefficient for random I/Os, while SSDs, made of non-volatile flash memory, handle random I/Os more efficiently. For both HDD and SSD, base tables and spill data are stored on the same device in separate files.

4 EXPERIMENTAL ANALYSIS

This section compares the performance of different query plan shapes under various memory availability, query complexities, and join and scan selectivities using HDD and SSD storage alternatives. Our codebase as well as information for replicating the experiments and generating data can be found in [37].

4.1 Datasets and Benchmark

We used an updated Wisconsin Benchmark and the JSON data generator [18] to evaluate multi-join query plans. This benchmark's tunability and selectivity make it ideal for our tests. We replicated experiment conditions (queries and benchmark) from [36], adjusting them for modern storage capabilities with larger record sizes. Records are 1073B, memory frames are 32KB, and experiments vary in dataset sizes and selectivities. Queries run on single CPU cores of NC nodes in AWS US-West-2, using d2.xlarge (4 vCPUs, 2 CPU cores, 2 threads per core, 2.4 GHz Intel Xeon E52676v3 Processor, 3 x 2048 GB storage, moderate network speed, 30.5GB RAM) for HDD and i3.xlarge (4 vCPUs, 2 CPU cores, 2 threads per core, 2.3 GHz Intel Xeon E5 2686 v4 Processor, 1 x 950 GB NVMe SSD storage, 206250 100% random read IOPS and 70000 100% random write IOPS, 1.25 GbPS baseline and 10 GbPS burst network bandwidth, 30.5GB RAM) for the SSD setting. In our future work, we plan to explore additional settings, including other node architectures such as AWS EBS and AWS EBS-Hybrid, where spill data is stored on a local SSD while base relations reside on networked SSD storage. We also intend to utilize multiple CPU cores in a single NC configuration and experiment with clusters of varying numbers of NCs.

4.2 Experiment 1 - Unlimited Memory

In this subsection, we examine how query complexity affects the execution time of a join query without spilling to disk, following the "Unlimited Memory" experiment in [36]. We used 1GB datasets with 1,000,000 records for each build and probe, ensuring each join's output is also 1GB. Although fixing intermediate result sizes is unrealistic, it simplifies comparing different query shapes. Additionally, such set-up keeps our re-evaluation fair to the original results of [36] by following their settings as faithfully as possible. Our future work will involve more complex queries using advanced

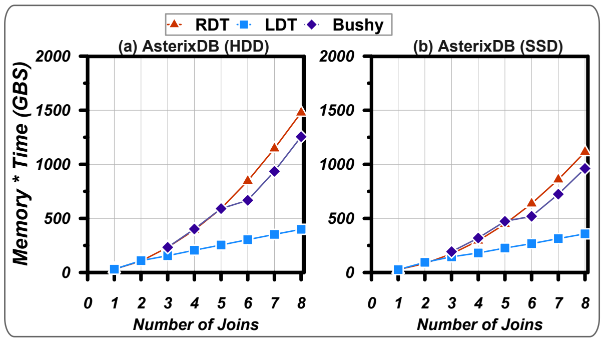

Figure 2: Experiment 1 - Resource Cost - AsterixDB

benchmarks such as JOB, TPC-H, and TPC-DS as well as queries with higher complexity.

In Experiment 1, we increased query complexity by varying the number of joins from 1 to 8. Fig. 1-a shows results from the Gamma simulator [36], while Fig. 1-b and 1-c present AsterixDB results using HDD and SSD.

The Gamma simulator's results were based on simulating HDD storage devices from the 1990s. As Fig. 1-a shows, in the Gamma simulator, RDT generally had a lower execution time than LDT. In this figure from [36], disk utilization in RDT is only slightly increasing, while its CPU becomes almost fully utilized as the number of joins in the query increases. However, we would have expected high disk utilization to be the bottleneck instead of the CPU since concurrently reading all the build datasets can cause high disk arm contention in HDD. From the reported device utilization and comparing the reported execution times of RDT with LDT, we believe that the Gamma simulator was not properly simulating disk arm movement and its impact on disk performance.

BT is another parallel query plan with shorter pipelines than RDT. These shorter pipelines and the independent parallelism make some build and probe phases of different joins overlap. Thus, BTs take the middle path between RDT and LDT. The jumps in the execution time of the BT in Figure 1 is due to the change of the query shape when adding more joins to the query. We are using the algorithm suggested in [28] for generating BTs, which keeps the pipelines' lengths to less than four joins.

In SSD storage (Fig. 1-b), BTs excel due to their parallel execution, improving CPU utilization. This set-up makes them superior to RDT on arm-less storage devices. The Gamma simulator results for HDDs resemble the AsterixDB results for SSDs, indicating that [36]'s simulator didn't model disk arm movement accurately. Thus, RDT's parallel I/O is better for SSDs when ample memory is available, while for HDDs, LDT has a better performance due to its sequential execution pattern and reduced I/O.

With the prevalence of cloud service providers for data management, it is valuable to compare different query plans based on their (monetary) cost, considering both resource usage and execution duration. Hence, we use a metric that calculates the resource (memory) cost as the product of memory usage and execution time (Gigabytes * seconds, GBS). Fig. 2 shows that the LDT plan's low memory requirement makes it the most cost-effective option for a cloud setting across various storage choices.

4.3 Experiment 2 - Limited Memory

For the second experiment, we study the impact of the amount of available memory on the execution times of various query shapes and memory distribution strategies for an eight-join query. This experiment was designed similarly to the "Limited Memory - High Resource Contention" experiment of [36]. We evaluate the performance of each query plan as a function of memory availability, thus the x-axis represents the ratio of available memory over the amount of memory required to keep all eight joins in memory. All inputs con-

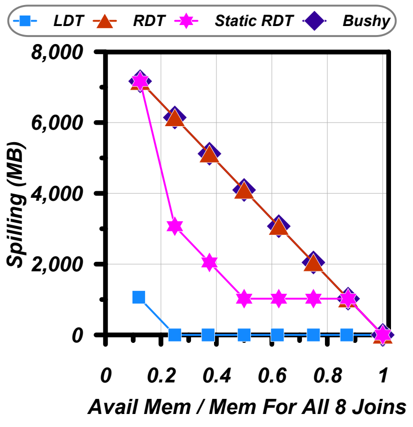

Figure 3: Experiment 2 - Data Spilling - AsterixDB

sist of 1 GB of data, and the size of the intermediate results remains constant and equal to 1 GB throughout the query plan's joins. Each record consists of 1073B.

As depicted in Fig. 3, basic RDT incurs the most I/O, dividing memory among all joins, leading to more spilling. Static-RDT, in most cases, has less I/O than RDT as it spills only intermediate results at breakpoints. The initial high spillage in Static-RDT occurs because all of the joins, along with all necessary intermediate results, spill to disk. LDT, on the other hand, shows the least I/O, allocating memory between just two consecutive joins at a time. In terms of parallelism, RDT and BT rank high, while Static-RDT's parallelism varies with the available memory, as joins within each segment run concurrently.

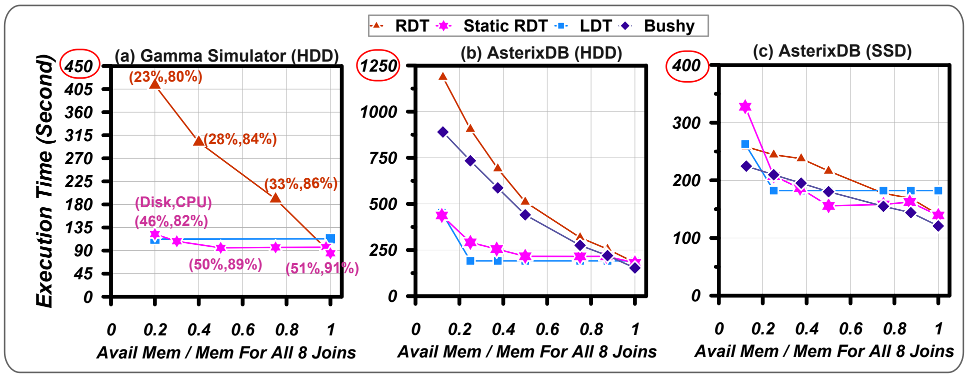

Figure 4: Experiment 2 - Execution Time

Fig. 4-a shows the results of the Gamma simulator as reported in [36], and Fig. 4-b and 4-b show the results of similar queries executed using Apache AsterixDB on HDD and SSD. As Fig. 4-b shows, LDT has the fastest HDD execution time since it performs the least amount of I/O and its sequential execution pattern is disk arm-friendly. After LDT, the Static-RDT has the lowest execution times due to its smaller amount of I/O and sequential execution pattern. Static-RDT's parallelism increases with more memory.

RDT shows the poorest HDD performance when memory is scarce, due to extensive spilling and frequent random disk access (All of its joins build concurrently and have to split up the available memory.) BT, second only to RDT in poor performance, suffers from high I/O and parallel execution causing random disk access.

As Fig. 4-c shows, parallel query plans such as RDT and BT perform better in SSD than HDD due to the lack of the disk arm issue in SSD and its capability to handle random disk I/Os and large volumes of I/Os efficiently. RDT, BT, and Static-RDT outperform LDT when the available memory is very large. LDT is still one of the best-performing query plans due to its small spilling to disk, especially when memory is very scarce. Static-RDT performs well, especially with more memory, due to its semi-parallel execution pattern and relatively little spilling to disk.

Our AsterixDB results show that LDT is one of the best query plans, especially with very limited memory. LDT outperforms other plans on HDD due to minimal data spilling and a disk-friendly sequential execution pattern. On SSD, LDT remains one of the best-performing query plans due to its low I/O and consistent average CPU utilization (42%-48%) from overlapping disk and CPU operations. Variations of parallel query plans, including RDT, Static-RDT, and BT, perform better as the memory increases since the amount of their spilling to disk drops then significantly.

Comparing the AsterixDB results with the Gamma simulator shows that RDT's excessive data spilling makes it

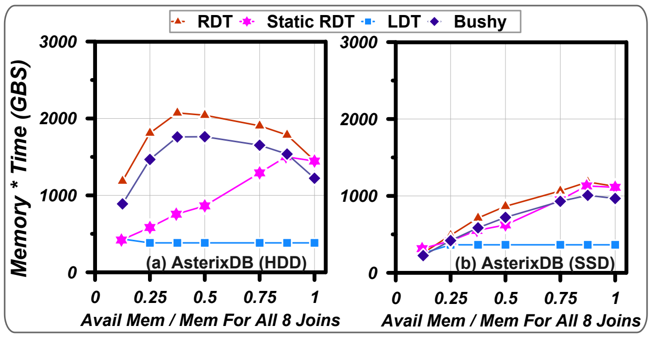

Figure 5: Experiment 2 - Resource Cost - AsterixDB

the worst-performing plan when memory is less than 80% of the required amount. This trend matches AsterixDB results for HDD. Fig. 5 shows that, similar to Experiment 1, the low memory usage of the LDT plan makes it the most cost-effective option.

4.4 Experiment 3 - Non-Restrictive Selections

Next, we re-evaluate the "Large Building Relations - Full Declustering" experiment from [36] to study the performance of various multi-join query plans with low-restrictive select conditions that minimally reduce the base dataset sizes.

Echoing Schneider and DeWitt's approach, our four-join query used relations with cardinalities and selectivities as follows: 10^6 records at 50%, 10^6 records at 50%, 10^6 records at 20%, 5×10^5 records at 10%, and 2×10^5 records at 25%. For a direct comparison to their results, we also set join selectivities to yield intermediate join results of $5{\times}10^4$, $5{\times}10^4$, 10^5, and 10^5 tuples each.

The corresponding Gamma simulator results in Fig. 6-a show that RDT had the worst performance with very limited memory due to spilling a large amount of data to disk, while LDT was the best-performing query plan shape due to its

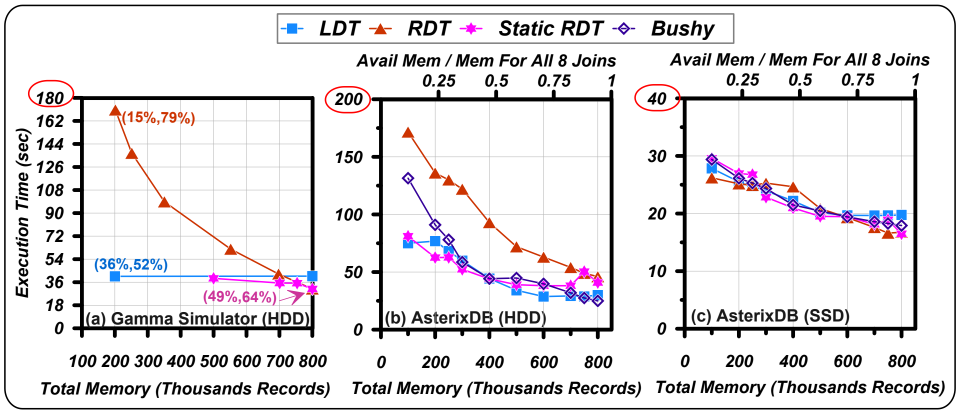

Figure 6: Experiment 3 - Execution Time

minimal spilling to disk. Static-RDT performed similarly to LDT when the available memory is significant. As Fig. 6-b shows, RDT performs the worst on HDD in AsterixDB due to high data spilling and disk contention from concurrent builds. BT performs better by using smaller intermediate results and less parallelism. LDT and Static-RDT are the best performers on HDD.

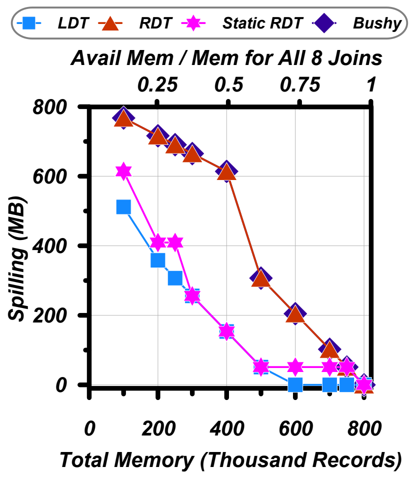

Figure 7: Experiment 3 - Data Spilling - AsterixDB

LDT excels in low-memory scenarios by sharing memory between two joins, which helps avoid disk contention. Static-RDT improves over RDT by cutting parallelism and reducing data spilling. On HDD, AsterixDB results match those from Gamma, where RDT lags behind LDT due to excessive spilling. However, on SSD, RDT performs better with reduced I/O costs, though CPU usage is inconsistent (Fig. 6-c). Nevertheless, LDT remains one of the best performers, efficiently handling memory and CPU with its sequential execution, while BT and Static-RDT also achieve strong results with semi-parallel execution.

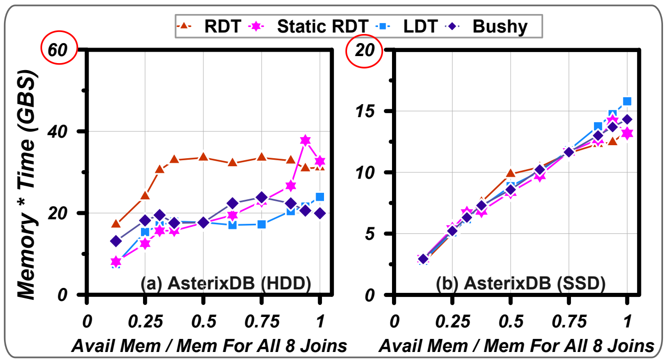

Figure 8: Experiment 3 - Resource Cost - AsterixDB

Large input datasets with minimal selection predicates and small join outputs are more in favor of LDT over more parallel plans such as RDT that require more memory and spill more data to disk when memory is limited. Fig. 7 shows the amount of spilling for different query plan shapes, while Fig. 8 presents the resource cost based on memory and execution time. LDT remains one of the lowest-cost plans for HDD due to efficient memory use and low execution time, with similar costs to other plans on SSD.

4.5 Experiment 4 - Non-Restrictive Joins

Next, we study the performance of different query plan shapes with non-restrictive join conditions, where each join can produce numerous output records per input record. This setup favors plan shapes such as RDT that use base datasets as inputs for their build phases. Similar to the "High Join Selectivity" experiment of [36], the base datasets have original sizes of 10^6, 10^6, 10^6, 5×10^5, and 2×10^5 records with scan selectivities of 50%, 50%, 20%, 10%, and 25%, respectively. These join selectivities cause the joins to produce 5×10^4, 2×10^5,4×10^5, and 5×10^5 records as their outputs.

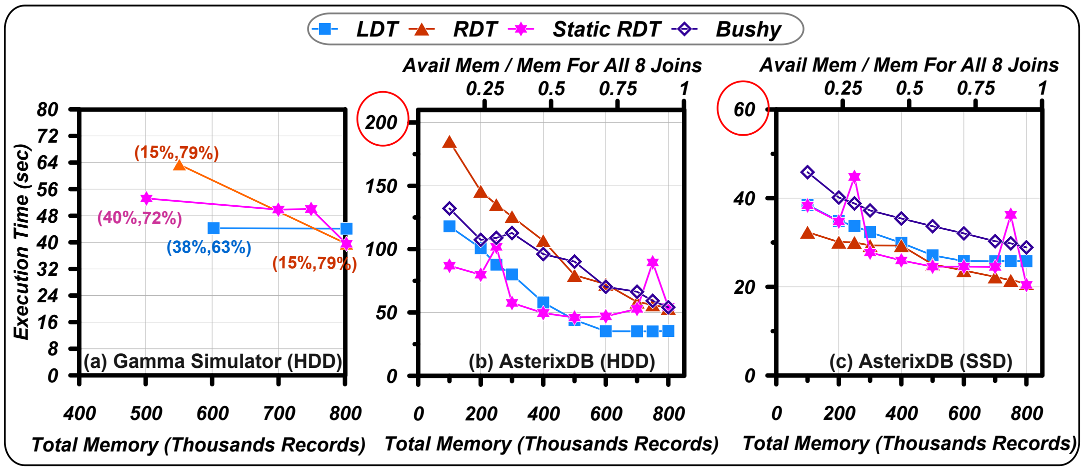

Figure 9: Experiment 4 - Execution Time

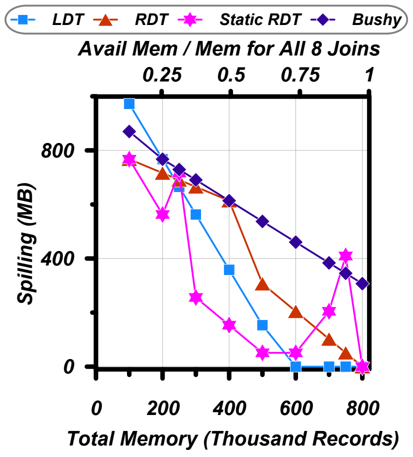

Figure 10: Experiment 4 - Data Spilling - AsterixDB

In Fig. 9-a, RDT has the highest execution time due to extensive spilling to disk when memory is limited in the original Gamma simulator. However, its parallel nature allowed RDT to outperform LDT and Static-RDT when a large amount of memory memory was available. Unfortunately, the [36]'s figure does not show data points for highly scarce memory conditions.

As Fig. 9-b shows, high disk arm contention from parallel access makes RDT one of the worst-performing query plans in AsterixDB on HDD. The performance of RDT improves as more memory becomes available. Despite using large non-base relations as build inputs, LDT performs well by dividing memory between only two consecutive joins and following a sequential disk access pattern. As Fig. 9-b and 9-c exhibit, there are two spikes in the performance of Static-RDT where the increment of memory has shifted the breaking point to a higher point in the tree with a larger intermediate result size. This highlights the importance of considering intermediate result sizes when setting breaking points in Static-RDT. Similar observations were expected in the Gamma simulator results but were not reflected or discussed in [36].

As shown in Fig. 9-c, RDT is the top performer due to its small build inputs, which result in low memory usage, and its parallel execution, benefiting from SSD's random-access efficiency. LDT also performs well despite its sequential pattern, as each join spills less data and the pipeline between join phases improves CPU utilization.

As Fig. 10 shows, RDT's spilling decreases with more memory, but it requires more overall memory since it divides memory among all joins. LDT, though expected to spill more due to large intermediate results, spills less because memory is shared between only two joins at a time. The amount of spilling in Static-RDT is dependent on the location of the breaking points in the query tree.

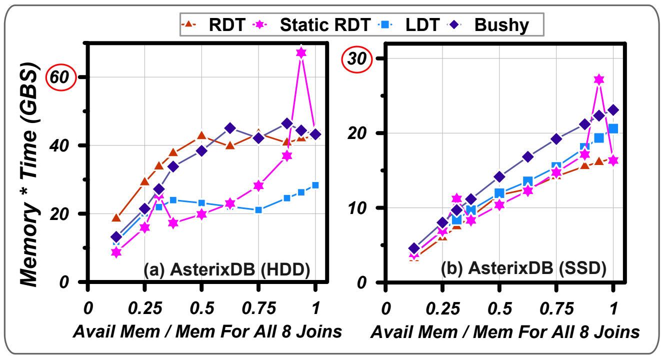

Figure 11: Experiment 4 - Resource Cost - AsterixDB

As in previous experiments, LDT is the lowest-cost query plan on HDD due to its low execution time and memory usage (Fig. 11), while RDT is one of the most cost-efficient plans on SSD.

5 RELATED WORK

Resource management and multi-join query scheduling have been studied for over three decades. In the 1990s, various query plan shapes and resource allocation techniques were prevalent. Schneider and DeWitt (1990) explored trade-offs between query shapes for multi-join queries using the Gamma system, focusing on Left Deep Tree (LDT) and Right Deep Tree (RDT) plans [10, 36]. They proposed Static-RDT, showing that RDT performs best with high parallelism if most build inputs remain in memory. Philip Yu et al. proposed Segmented-RDT, a BT composed of smaller RDT subtrees [6]. This approach showed performance benefits through simulations, indicating Segmented-RDT can outperform other query plan shapes, including RDT. The ZigZag Tree, a competitor to Static-RDT, reduces I/O by avoiding intermediate result materialization [44]. Wilschut et al. used PRISMA/DB to examine processor assignment and scheduling strategies for multi-join queries on an 80-processor system [39]. They evaluated various query plans and found sequential plans better for systems with fewer processors, while parallel plans suited systems with many processors. The authors of [2] addressed memory allocation for concurrent operations in query execution plans, proposing a post-optimization phase to identify concurrent operations and find near-optimal memory allocations. This technique improved execution times, especially for multi-join queries involving LDT, RDT, and BTs. The authors of [14] developed PipeSched, a fast resource scheduling algorithm for physical operator pipelines. This group of studies are the most related works to our study.

Another group of studies focused on dynamic memory management techniques and their algorithm designs. A significant part of this research involved designing memory-adaptive operators, with studies on adaptive and dynamic sort [12, 26, 33] and join [7, 17, 34, 38, 42] operators using simulators due to limited DBMS resources. Additionally, dynamic workload management was studied alongside query memory management [1, 8, 16, 23, 24, 43].

Another group of works impacting query memory indirectly focuses on join ordering and plan enumeration. Leis et al. optimized join orders and algorithms with advanced techniques, improving parallel databases [27]. Recent research uses deep and machine learning to optimize DBMSs, including cardinality estimation [21, 31] and join correlations [40].

6 CONCLUSION AND FUTURE WORK

In this study, we re-evaluated Schneider and DeWitt's [36] seminal analysis of multi-join queries on shared-nothing clusters using HDD. We revisited their results from the Gamma database simulator using Apache AsterixDB with both HDD and SSD, and also analyzed the performance of a BT plan.

RDT has long been considered efficient due to its parallel execution, but *our studies show that it excels in SSD-based systems mainly for queries with few joins and when memory can accommodate over 80% of the build datasets.* Static-RDT may spill less data to disk than RDT when there is enough memory for each build dataset. If multiple build phases fit into memory, *Static-RDT requires careful placement of break points in the tree, avoiding joins that produce large outputs. LDT was seen to perform best overall, spilling less data and maintaining steady CPU utilization* (42%-48%) from overlapping disk and CPU operations. In contrast, *RDT shows greater variability in CPU usage*, with lower utilization during the I/O-bound build phase (28%-35%) and higher during the probe phase (75%-88%), where CPU usage fluctuates depending on whether disk spilling occurs. Static-RDT's CPU utilization approaches that of LDT with limited memory (due to more sequential operations) and resembles RDT when memory is abundant. *BTs, on the other hand, achieve more consistent CPU utilization than both RDT and Static-RDT by overlapping subtrees, but they require careful input selection during the build phase.* CPU utilization was generally lower on HDDs, as disk I/O became the primary bottleneck. *Our results emphasize the need to consider storage architecture and periodically re-evaluate past studies due to hardware advancements.*

To enable a meaningful comparison, we designed our experiments similarly to [36]. While simple, these experiments are crucial for understanding "join physics." Future work will explore realistic queries from benchmarks such as JOB, TPC-H, and TPC-DS, testing various configurations, storage setups such as AWS EBS, and cluster sizes to analyze scalable query performance. Additionally, comparing memory management techniques between Volcano-style execution and modern approaches such as vectorized and code-generation methods is essential for today's DBMSs.

ACKNOWLEDGEMENT

This research was supported in part by NSF award IIS-1954962 and by the Donald Bren Foundation (via a Bren Chair).

REFERENCES

[1] Ashraf Aboulnaga and Shivnath Babu. 2013. Workload management 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, 929–932. https://doi.org/10.1145/2463676.2467801

[2] Josep Aguilar-Saborit, Mohammad Jalali, Dave Sharpe, and Victor Muntés-Mulero. 2008. Exploiting pipeline interruptions for efficient memory allocation. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, James G. Shanahan, Sihem Amer-Yahia, Ioana Manolescu, Yi Zhang, David A. Evans, Aleksander Kolcz,

Key-Sun Choi, and Abdur Chowdhury (Eds.). ACM, 639–648. https://doi.org/10.1145/1458082.1458169

[3] 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.

[4] Shivnath Babu and Herodotos Herodotou. 2013. Massively Parallel Databases and MapReduce Systems. *Found. Trends Databases* 5, 1 (2013), 1–104. https://doi.org/10.1561/1900000036

[5] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 1151–1162. https://doi.org/10.1109/ICDE.2011.5767921

[6] Ming-Syan Chen, Ming-Ling Lo, Philip S. Yu, and Honesty C. Young. 1992. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, Li-Yan Yuan (Ed.). Morgan Kaufmann, 15–26. http://www.vldb.org/conf/1992/P015.PDF

[7] Diane L. Davison and Goetz Graefe. 1994. Memory-Contention Responsive Hash Joins. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 379–390. http://www.vldb.org/conf/1994/P379.PDF

[8] Diane L. Davison and Goetz Graefe. 1995. Dynamic Resource Brokering for Multi-User Query Execution. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 281–292. https://doi.org/10.1145/223784.223845

[9] 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.

[10] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. 1986. GAMMA - A High Performance Dataflow Database Machine. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 228–237. http://www.vldb.org/conf/1986/P228.PDF

[11] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.* 2, 1 (1990), 44–62. https://doi.org/10.1109/69.50905

[12] Thanh Do and Goetz Graefe. 2023. Robust and Efficient Sorting with Offset-value Coding. *ACM Trans. Database Syst.* 48, 1 (2023), 2:1–2:23. https://doi.org/10.1145/3570956

[13] Marius Eich, Pit Fender, and Guido Moerkotte. 2018. Efficient generation of query plans containing group-by, join, and groupjoin. *VLDB J.* 27, 5 (2018), 617–641. https://doi.org/10.1007/S00778-017-0476-3

[14] Minos N. Garofalakis and Yannis E. Ioannidis. 2014. Multi-Resource Parallel Query Scheduling and Optimization. *CoRR* abs/1403.7729 (2014). arXiv:1403.7729 http://arxiv.org/abs/1403.7729

[15] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170. https://doi.org/10.1145/152610.152611

[16] Goetz Graefe. 1996. Iterators, Schedulers, and Distributed-memory Parallelism. *Softw. Pract. Exp.* 26, 4 (1996), 427–452. https://doi.org/10.1002/(SICI)1097-024X(199604)26:4427::AID-SPE20\protect\protect\leavevmode@ifvmode\kern+.2222em\relax3.0.CO;2-H

[17] 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. http://www.vldb.org/conf/1998/p086.pdf

[18] Shiva Jahangiri. 2020. *"Wisconsin Benchmark Data Generator"*. http://doi.org/10.5281/zenodo.4316003

[19] 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.

[20] Richard P. King. 1990. Disk Arm Movement in Anticipation of Future Requests. *ACM Trans. Comput. Syst.* 8, 3 (1990), 214–229. https://doi.org/10.1145/99926.99930

[21] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[22] Donald Kossmann. 2000. The State of the art in distributed query processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469. https://doi.org/10.1145/371578.371598

[23] Stefan Krompass, Umeshwar Dayal, Harumi A. Kuno, and Alfons Kemper. 2007. Dynamic Workload Management for Very Large Data Warehouses: Juggling Feathers and Bowling Balls. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, 1105–1115. http://www.vldb.org/conf/2007/papers/industrial/p1105-krompass.pdf

[24] Stefan Krompass, Harumi A. Kuno, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. 2010. Adaptive query scheduling for mixed database workloads with multiple objectives. In *Proceedings of the Third International Workshop on Testing Database Systems, DBTest 2010, Indianapolis, Indiana, USA, June 7, 2010*, Shivnath Babu and G. N. Paulley (Eds.). ACM. https://doi.org/10.1145/1838126.1838127

[25] Lukas Landgraf, Wolfgang Lehner, Florian Wolf, and Alexander Boehm. 2022. Memory Efficient Scheduling of Query Pipeline Execution. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p82-landgraf.pdf

[26] Per-Åke Larson and Goetz Graefe. 1998. Memory Management During Run Generation in External Sorting. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, Laura M. Haas and Ashutosh Tiwary (Eds.). ACM Press, 472–483. https://doi.org/10.1145/276304.276346

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[28] Bin Liu and Elke A. Rundensteiner. 2005. Revisiting Pipelined Parallelism in Multi-Join Query Processing. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway,*

August 30 - September 2, 2005, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 829–840. http://www.vldb.org/archives/website/2005/program/paper/thu/p829-liu.pdf

[29] William C. Lynch. 1972. Do disk arms move? *SIGMETRICS Perform. Evaluation Rev.* 1, 4 (1972), 3–16. https://doi.org/10.1145/1041603.1041604

[30] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2018. Many-query join: efficient shared execution of relational joins on modern hardware. *VLDB J.* 27, 5 (2018), 669–692. https://doi.org/10.1007/S00778-017-0475-4

[31] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[32] Oracle. 2023. *"Join Groups in Oracle"*. https://docs.oracle.com/en/database/oracle/oracle-database/23/inmem/optimizing-queries-with-join-groups.html#GUID-3E5491C4-B345-4A8E-8B1B-8DC150C8A797

[33] HweeHwa Pang, Michael J. Carey, and Miron Livny. 1993. Memory-Adaptive External Sorting. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, Rakesh Agrawal, Seán Baker, and David A. Bell (Eds.). Morgan Kaufmann, 618–629. http://www.vldb.org/conf/1993/P618.PDF

[34] 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. https://doi.org/10.1145/170035.170051

[35] Doron Rotem. 1992. Analysis of Disk Arm Movement for Large Sequential Reads. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, Moshe Y. Vardi and Paris C. Kanellakis (Eds.). ACM Press, 47–54. https://doi.org/10.1145/137097.137108

[36] Donovan A. Schneider and David J. DeWitt. 1990. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek (Eds.). Morgan Kaufmann, 469–480. http://www.vldb.org/conf/1990/P469.PDF

[37] Jahangiri Shiva. 2024. Memory Management in Complex Join Queries: A Re-evaluation Study. https://github.com/shivajah/Memory-Management-in-Complex-Join-Queries-A-Re-evaluation-Study. https://github.com/shivajah/Memory-Management-in-Complex-Join-Queries-A-Re-evaluation-Study GitHub repository.

[38] Giulliano Silva Zanotti Siviero and Shiva Jahangiri. 2023. Towards a Memory-Adaptive Hybrid Hash Join Design. In *IEEE International Conference on Big Data, BigData 2023, Sorrento, Italy, December 15-18, 2023*, Jingrui He, Themis Palpanas, Xiaohua Hu, Alfredo Cuzzocrea, Dejing Dou, Dominik Slezak, Wei Wang, Aleksandra Gruca, Jerry Chun-Wei Lin, and Rakesh Agrawal (Eds.). IEEE, 6283–6285. https://doi.org/10.1109/BIGDATA59044.2023.10386098

[39] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. 1995. Parallel Evaluation of Multi-Join Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 115–126. https://doi.org/10.1145/223784.223803

[40] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proc. ACM Manag. Data* 1, 1 (2023), 41:1–41:27. https://doi.org/10.1145/3588721

[41] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[42] Hansjörg Zeller and Jim Gray. 1990. An Adaptive Hash Join Algorithm for Multiuser Environments. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek (Eds.). Morgan Kaufmann, 186–197. http://www.vldb.org/conf/1990/P186.PDF

[43] Mingyi Zhang, Patrick Martin, Wendy Powley, and Jianjun Chen. 2018. Workload Management in Database Management Systems: A Taxonomy. *IEEE Trans. Knowl. Data Eng.* 30, 7 (2018), 1386–1402. https://doi.org/10.1109/TKDE.2017.2767044

[44] Mikal Ziane, Mohamed Zaït, and Pascale Borla-Salamet. 1993. Parallel Query Processing with Zigzag Trees. *VLDB J.* 2, 3 (1993), 277–301. http://www.vldb.org/journal/VLDBJ2/P277.pdf