

A Hybrid Approach to Group-By and Aggregation Query Execution

Gaurav Vaghasiya
Santa Clara University
Santa Clara, CA
gvaghasiya@scu.edu

Shiva Jahangiri
Santa Clara University
Santa Clara, CA
sjahangiri@scu.edu

Abstract—Group-By-Aggregate (GBA) queries are fundamental in data analysis, allowing users to organize data by specific attributes and apply aggregate functions, such as sum, average, and count, to generate insights from large datasets. Traditionally, DBMSs handle GBA queries through either sort-based or hash-based methods, each with unique strengths and limitations. Sort-based approaches, while effective in certain contexts, can become computationally expensive when few groups are involved due to extensive record comparisons. On the other hand, hash-based methods are efficient with larger group counts but are memory-intensive and may suffer from hash collisions, depending on data distribution and hashing functions.

This paper presents an empirical performance study focused on traditional sort- and hash-based GBA operators, introducing a robust hybrid approach that unifies their strengths to deliver consistent, high performance across diverse data conditions. The hybrid GBA method combines hash-based grouping and early aggregation for the local phase with sort-based aggregation for the global phase. It forwards partially aggregated results when memory limits are reached, reducing data volume and minimizing memory usage and disk I/O. By eliminating the need to choose between sort and hash GBA operators and removing dependency on precise statistics or query optimizer intervention, the hybrid method significantly enhances query execution reliability. Extensive testing on Wisconsin, TPC-H, and TPC-DS benchmark datasets demonstrates that the hybrid approach consistently matches the performance of the best-suited algorithm—whether sort or hash—in each scenario, across varying dataset sizes and complexities. This unified solution replaces separate GBA algorithms, providing a flexible, easily adaptable, and highly efficient method for processing GBA queries in big data management systems.

Index Terms—Group-By, Aggregate, Sort, Hash

I. INTRODUCTION

Group-by and aggregate operations are fundamental to data analysis, allowing for the categorization of data by specific attributes and the application of aggregate functions such as SUM, AVERAGE, and COUNT. These operators enable users to identify trends and patterns in large datasets and are crucial in database management systems (DBMS), supporting key applications such as reporting, business intelligence, and decision support systems. Given the widespread use of GBA queries across industries and their high computational cost, optimizing their execution is critical, especially as datasets grow in size and complexity.

Traditional methods for executing GBA queries typically fall into two categories: sort-based and hash-based approaches.

Sort-based GBA methods involve sorting data by grouping attributes and then scanning the sorted data to apply aggregate functions. Although effective in certain scenarios, this approach can become computationally expensive depending on the data type and value distribution. For instance, it is highly inefficient when the group-by field is of type string with common prefixes, where the sorting step often becomes a bottleneck. This inefficiency demands extensive CPU and I/O resources, resulting in prolonged query execution times.

In contrast, the hash-based GBA method leverages a hash table to partition data by grouping attributes, enabling early aggregation and often proving computationally and memory efficient, particularly when aggregates fit within memory, as it bypasses the need for record sorting. However, hash-based methods introduce challenges, such as substantial memory overhead for maintaining hash tables when handling a large number of groups, which can lead to extensive data spilling. Additional performance issues may arise from hash collisions, data skew, and the approach's reliance on data distribution and the quality of the hashing function, resulting in potentially unpredictable performance across different scenarios. Since particular scenarios and settings favor one method over the other, many query-processing and data management systems support both hash- and sort-based algorithms, relying on the query optimizer and runtime to choose between them. However, selecting the optimal algorithm requires precise statistics, which has proven to be a challenging task [8], [23].

Recognizing the limitations of both sort- and hash-based approaches, this paper introduces a hybrid method that leverages the strengths of both sort-based and hash-based techniques for executing group-by queries. The hybrid GBA method starts with hash-based aggregation within each data partition, performing early aggregation and forwarding partially aggregated results to the global phase when memory limits are reached. The global phase applies sort-based aggregation to the reduced data volume, optimizing memory usage and minimizing disk I/O by avoiding unnecessary spills and extensive sorting. The key contributions of this paper are threefold: 1) We conduct an empirical study of existing hash-based and sort-based Group-by-aggregation (GBA) methods to understand their underlying mechanics, strengths, and limitations. 2) Building on these insights, we propose a new hybrid GBA approach and provide a detailed description and implementation of this method,

demonstrating its performance benefits over traditional approaches. 3) We implement the hybrid approach on Apache AsterixDB, an open-source big data management system, and make our implementation available to the community for adoption or for reproducibility purposes.

Our experimental results demonstrate that the hybrid method achieves query execution times comparable to the best of the sort- and hash-based GBA operators. Replacing them with a single versatile operator eliminates the need for the optimizer to decide between them, reducing dependence on potentially inaccurate statistics. By overcoming the limitations of existing methods, being independent of statistics, and offering a more adaptable and efficient solution, the hybrid approach can replace the traditional sorting and hashing approaches.

The remainder of the paper is organized as follows: Section II provides background on the design of group-by operators and specifications of AsterixDB. In Section III, we explain the design and implementation details of traditional GBA approaches as implemented in AsterixDB, which are largely similar to those in other data management systems. In Section IV, we discuss the design and implementation of our combined approach for executing GBA queries and include the specifications, algorithm, and rationale behind its design. In Section V, we present our experimental results which demonstrate the performance improvements of our combined approach by comparing its results with traditional sort-based and hash-based methods. In Section VI we will discuss previous work related to this study. Section VII concludes the paper and provides suggestions for future work.

II. BACKGROUND

Group-By-Aggregate (GBA) queries group records that share a common value in one or more columns and apply aggregate functions to each group. These aggregate functions typically include COUNT, SUM, AVG, MAX, and MIN, which perform calculations on the grouped data. We refer to the columns used for grouping as “group fields” and we call the columns involved in aggregate calculations “aggregate fields”.

In current DBMSs, the group-by operator and aggregate function computations are typically handled using either sort-based or hash-based group-by designs. In this section, we describe in detail how these approaches function in the execution of GBA queries both in general and in Apache AsterixDB.

A. Apache AsterixDB

Apache AsterixDB [1], [2], [19] 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 entry point for user requests is the Cluster Controller (CC), which compiles requests into executable jobs, while Node Controllers (NCs) serve as worker nodes to execute these jobs. One NC also acts as the Metadata Controller, managing AsterixDB’s metadata. Each NC oversees one or

more data partitions, executing query instances in parallel across necessary partitions.

AsterixDB allows optional hints and parameters for users to control resource allocation, such as memory and CPU usage. Memory budget parameters let users set memory limits for memory-intensive operators (e.g., joins, sorts). Additionally, users can provide hints to select physical operators; for instance, the hash-based group-by operator can replace the default sort-based option if specified [4] as in the following example:

```
USE TinySocial;
SELECT uid AS user, COUNT(cm) AS count
FROM ChirpMessages cm
/*+ hash */ GROUP BY
cm.user.screenName AS uid;
```

AsterixDB supports querying both open and closed datasets, providing flexibility for semi-structured data [4]. Closed datasets enforce a strict schema, storing records in binary-encoded ADM format with declared fields in the metadata node, optimizing query performance and storage efficiency through predictable data structures [3] [24]. In contrast, open datasets allow additional fields beyond the predefined schema, storing information for undeclared fields within each record, which benefits applications needing schema flexibility for new attributes over time.

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 SQL and NoSQL database systems and query engines (stage-based query execution, operators’ algorithms, iterator model, and more) makes our results and techniques applicable to other systems as well.

III. TRADITIONAL GBA APPROACHES

This section describes the design and implementation details of traditional GBA approaches in AsterixDB to shed light on their performance differences, as examined in Section V.

A. Sort-based Group-by and Aggregation

In AsterixDB, sort-based grouping serves as a fundamental mechanism to efficiently handle GBA queries. AsterixDB uses an external merge sort algorithm to sort the data based on the specified group fields. The operation of sort-based grouping in Apache AsterixDB is structured into two phases as shown in Figure 1, Sort and Merge.

During the sort phase, records are loaded into memory page by page until the memory is full or all data has been read, then sorted by group field values. AsterixDB uses normalized keys for sorting, as comparing normalized binary representations is faster than comparing actual values. It also employs a

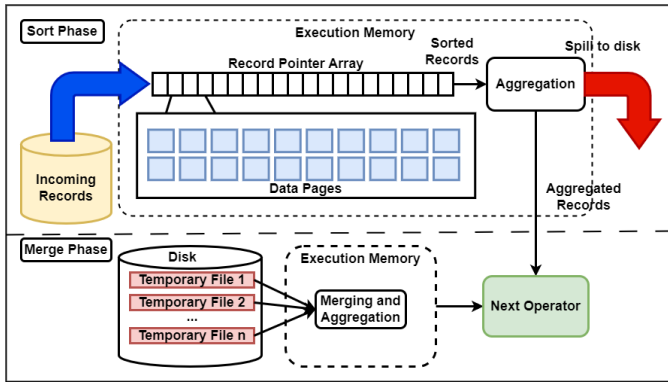


Fig. 1. Workflow of Sort-Based Group-By for GBA queries

record pointer array, containing the normalized key and the memory location of records, which eliminates the need to move full records in memory [12]. However, when using open types, the actual records are sorted rather than the normalized representation.

After sorting, the records are sent to the aggregation operation, which processes each record sequentially by updating aggregate fields like sums and counts. Since the records are sorted, managing aggregation states is simplified by tracking only the current and previous group values. If memory is sufficient, the aggregated records are passed to the next operator; otherwise, AsterixDB writes them to disk to free up memory and maintain performance. This process continues until all records in the data partition are processed.

The Merge phase consolidates fragmented aggregates from multiple sorted run files into a final output. The group-by operator assigns an input frame to each temporary file, reads records, and maintains aggregate values based on group field values. If there are more files than input frames, merging occurs recursively in multiple rounds. The final result is either returned as the query output or used for further processing. This sort-based approach is both simple and effective, handling large datasets efficiently.

In AsterixDB, the execution plan for GBA queries involves two group-by operators with an implicit aggregation step, separated by a hash-partition exchange operator, as shown in Figure 2. The hash-partition exchange distributes data across partitions based on hash values computed from grouping attributes in GBA case, balancing the load across resources. At each NC, it hashes the data and transfers it to the appropriate partition, enabling efficient parallel processing. This plan applies to both sort-based and hash-based GBA methods, with the main difference being the implementation of the group-by operators. The process begins with “local aggregation” where each data partition aggregates results from its available data.

After local aggregation, the Hash-Partition Exchange operator redistributes records across nodes and partitions using group-by fields as hash keys, ensuring identical fields are placed together. The redistributed records are then processed by a second group-by operator for final parallel aggregation on

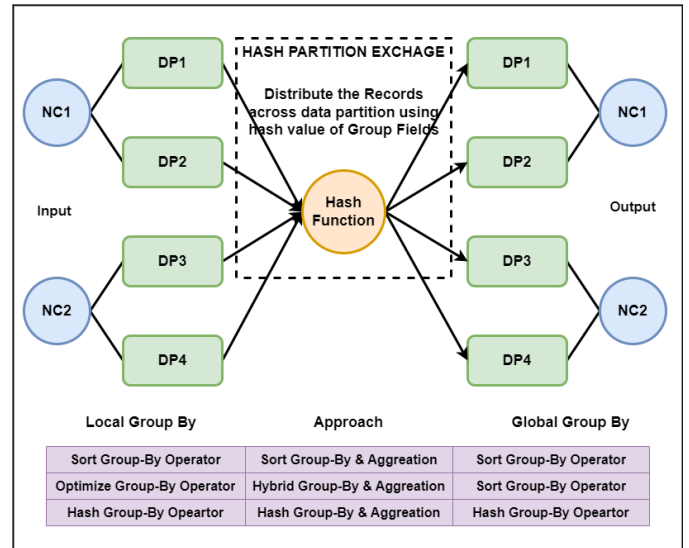


Fig. 2. Execution plan for GBA in AsterixDB

each NC. We refer to this phase as “global aggregation”. The results are then sent to the CC and delivered to the user.

B. Hash-based Group-by and Aggregation

In AsterixDB, hash-based group-by uses a hash function to process each incoming record for execution of GBA queries. This function determines the bucket or partition where each record’s data will be aggregated based on specific fields which in our case are “group fields”. The hash function creates a hash value for group fields, and records are aggregated accordingly in their respective partitions.

The process is divided into two phases to efficiently manage memory and handle large datasets. In the first phase, the operator calculates the number of partitions based on the input data and available memory. Each record’s hash value is computed, and if an aggregate already exists in memory, the new data is added to it; otherwise, a new aggregate is created. This phase enables fast in-memory aggregation, minimizing disk access. If the aggregated data exceeds memory capacity, the operator spills partitions to disk to free up space and continue processing.

After processing all records, phase 2 begins. The operator sends the in-memory partitions to the next operator as it will not need any spill partitions to generate final results. Then it goes to phase 1 again and processes all spilled partitions as shown in Figure 3.

The hash table uses a data structure design that includes a data partition table and a hash table. The data partition table stores the actual records being aggregated, while the hash table holds pointers to these records, facilitating quick location and retrieval of data across data pages. This separation enhances CPU cache efficiency, as the smaller hash table size allows for frequent access. A detailed explanation of the architecture and design of the hash tables of AsterixDB can be found in [19] and [18].

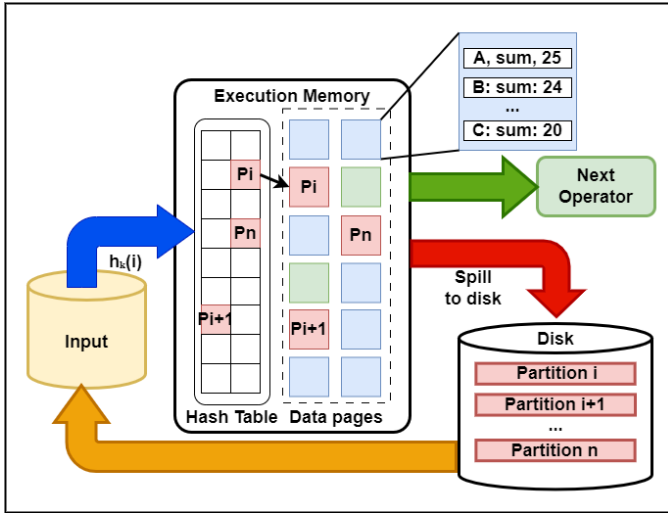


Fig. 3. Workflow of Hash-Based Group-By for GBA queries

IV. DESIGN AND IMPLEMENTATION OF HYBRID GROUP-BY AND AGGREGATION

Although sort-based grouping demonstrates a capability to handle complex data aggregation tasks and ensures high-performance querying, it lacks performance when group-by fields are similar. For example, if there are only a small number of groups, then sorting can become highly time-consuming. It may also lead to unnecessary disk spilling, especially when dealing with large datasets. On the other hand, the hash-based approach, which performs early aggregation and maintains only distinct groups in memory, avoids disk spilling as long as the memory budget is sufficient. However, the hash-based algorithm struggles significantly if the number of distinct groups is too large to fit in memory the sort-based algorithm would be superior in this case.

Another observation is that in local GBA, it is not necessary to compute the final local aggregate values for the entire set of records in each partition. Instead, the local GBA can ship partially computed aggregates to the final destination (the global aggregators) instead of spilling to disk, as the global aggregator would likely spill to disk as well.

In designing our hybrid approach for processing GBA queries in Apache AsterixDB, we aim to address the inherent inefficiencies present in current methodologies. There are cases where sort-based methods outperform hash-based ones and vice versa, requiring accurate statistics to determine the optimal approach based on the conditions. However, maintaining precise statistics has proven to be a challenge. Our primary motivation is to design a hybrid GBA method that leverages the strengths of both sort- and hash-based approaches, performing as well as or better than either method under different conditions, without relying on precise statistics. We employ hash-based grouping and aggregation for the local group-by phase, followed by sort-based grouping and aggregation in the global phase.

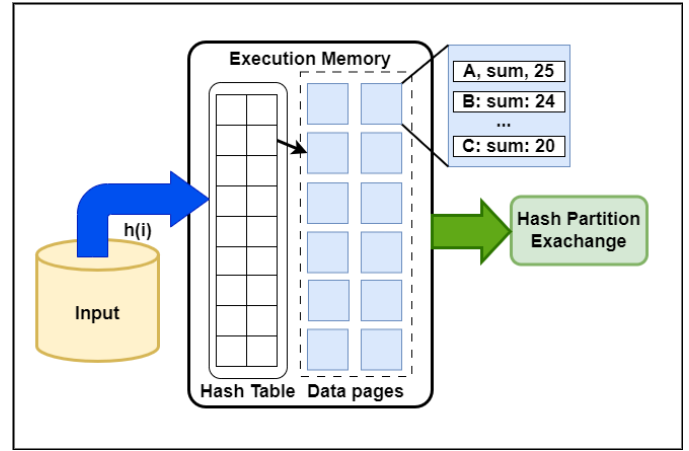


Fig. 4. Local Phase of Hybrid GBA Operator

The hybrid approach begins with hash-based grouping and aggregation during the local group-by phase, where entries are aggregated in memory. When the hash table reaches its memory limit, the operator spills the aggregated records to the Hash-Partition Exchange, creating space for continued processing. After local grouping and aggregation are completed, the system transitions to a sort-based approach for the global GBA phase. The accumulated entries are sorted and written to disk, with the sort-based GBA computing the final aggregate results.

This hybrid method optimizes local aggregation with a hash-based approach, avoiding the need to sort the dataset early. Instead of spilling to disk when memory is full, partially computed aggregates are sent directly to the global group-by operator, reducing disk load and minimizing data movement and disk I/O, which are common bottlenecks. By combining the strengths of both algorithms, this approach significantly reduces disk spills, confining them to a single instance during global aggregation, thereby improving overall GBA execution efficiency.

The motivation behind designing the hybrid GBA approach was to create a unified method that effectively matches the performance of the best choice between hash-based and sort-based techniques in varied scenarios, enhancing robustness and adaptability without relying on precise statistics or requiring optimizer intervention. Its simple yet effective design allows for easy integration across big data platforms and database management systems, offering practical benefits such as reduced data spilling and lower computational costs compared to traditional methods.

V. EXPERIMENTS AND RESULTS

In this section, we provide a detailed analysis of experiments conducted to evaluate the performance of traditional GBA approaches and the effectiveness of our proposed hybrid approach for GBA queries in Apache AsterixDB. These experiments rigorously assess performance across dimensions such as data size, aggregation type, and data types.

A. Experiments Setting and Dataset

Our experimental framework is built upon a modified version of the Wisconsin benchmark dataset [17], specifically tailored to evaluate the performance of GBA queries. The key modifications to the default Wisconsin benchmark dataset include the addition of fields to define the number of groups, variable length records, and strings with different sizes and distributions, allowing us to simulate more complex scenarios.

In addition to Wisconsin datasets, we have also used TPC-H and TPC-DS datasets to benchmark our results with a more realistic but less controllable dataset structure.

The experiments are conducted on an AsterixDB cluster setup that includes one CC and two NC nodes, with each NC managing two data partitions. All CC and NCs run on a separate Intel NUC machine equipped with 8-core Intel i7 processors, 64GB of RAM, and a SSD with 2TB of storage. For each NC, we limit the JVM memory of AsterixDB to 8GB per node, allocating 64MB per data partition for GBA operations. This fixed memory constraint allows us to evaluate the efficiency and adaptability of the hybrid GBA approach under the limited memory conditions commonly encountered in big data processing environments. Each query is executed in eleven iterations, reporting the average execution time of the last ten runs to exclude cold cache starts from the results. To minimize the impact of data caching, a full scan query on a dataset that is five times larger than the available memory is run after the execution of each query.

B. Experiment 1: Impact of Aggregation Types

This experiment evaluates the scalability and performance of our hybrid group-by approach against traditional sort-based and hash-based methods across dataset sizes ranging from 2GB to 64GB, aiming to understand how increasing data volumes impact GBA query execution times across the three GBA approaches.

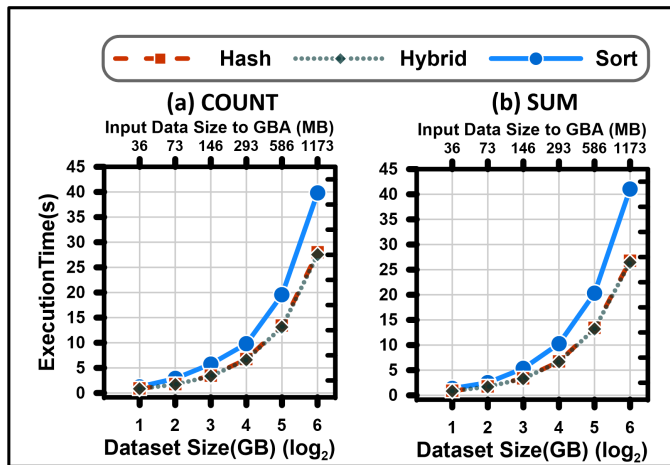


Fig. 5. Expr 1: Impact of Data Size with different aggregation types

For this experiment, we used a modified version of the Wisconsin Benchmark dataset. The Wisconsin datasets were loaded with an open type configuration, using a UUID primary

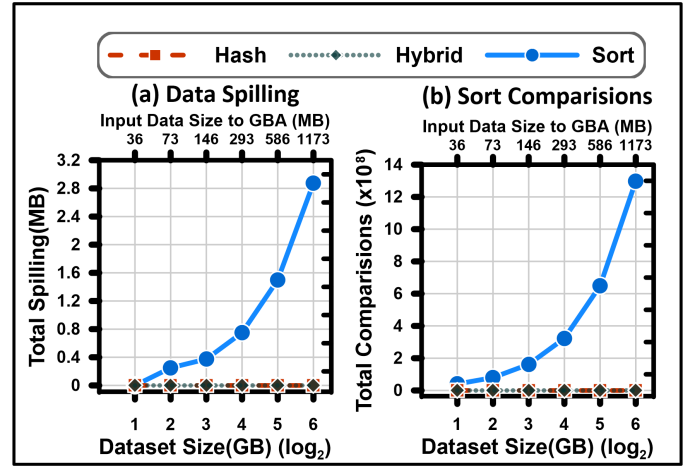


Fig. 6. Expr 1: Spilling and Sort Comparisons for 'COUNT'

key randomly generated by AsterixDB [4]. Figure 5 in which the x-axis represents dataset sizes in gigabytes on a logarithmic scale to effectively showcase the extensive range and size of data in megabytes going as input to GBA, and the y-axis tracks the execution time. Also, Each graph plots the performance trajectories for the sort-based, hash-based, and hybrid group-by approaches. The queries utilized for interacting with the Wisconsin dataset are structured as follows.

```
USE wisconsin_dataverse;
SELECT attribute, COUNT(attribute)
FROM wisconsin_dataset GROUP BY
attribute;
```

Figure 5-(a), illustrates group-by operations over integer values ranging from 0 to 9, with the aggregation function set to COUNT. This setup yields a limited grouping scenario, producing only 10 output groups. Figure 6 represents data spilling and sort comparisons done by each approach with 'COUNT' aggregation type in the case of integer values mentioned above. The y-axis represents data spilling in megabytes (MB) for (a) and for (b) its number of comparisons in multiple of 10⁸. Given the configuration of 64MB memory allocation for each GBA operator (in total 256MB of memory for 2 NCs each with 2 data partitions), both the hybrid and hash-based methods perform entirely in memory leading to near zero spilling and comparisons compared sort-based and, achieving comparable execution times. In contrast, the sort-based approach incurs additional computational overhead from sorting shown in Figure 6, resulting in higher execution time. While the amount of data being spilled by the sort-based approach is small due to the small number of groups, the comparison operations contribute to less efficient performance in this scenario.

Figure 5 extends the comparison by implementing the 'SUM' function instead of 'COUNT' on the same integer data field in Figure 5-(b). Even with the 'SUM' aggregation type number of comparisons and spilling was the same as 'COUNT' from our observation, as aggregation type does not affect the overall data size in memory or sort comparisons.

This comparison helps illustrate that while the type of aggregation function changes, the underlying efficiency of each group-by approach remains consistent, with the hybrid method maintaining a performance edge. The initial experiment confirms that the hybrid group-by approach provides an equivalent execution time to hash-based approaches as the number of groups fits into the memory. Furthermore, the results indicate that the choice of aggregation function—whether count or sum does not significantly impact the relative performance among all GBA methodologies. This underscores the versatility of the hybrid approach, making it a robust solution across various aggregation operations and dataset sizes.

C. Experiment 2: Impact of Data Types

In this series of experiments, we use queries similar to those used in Experiment 1 but focus on group-by operations involving different types of fields.

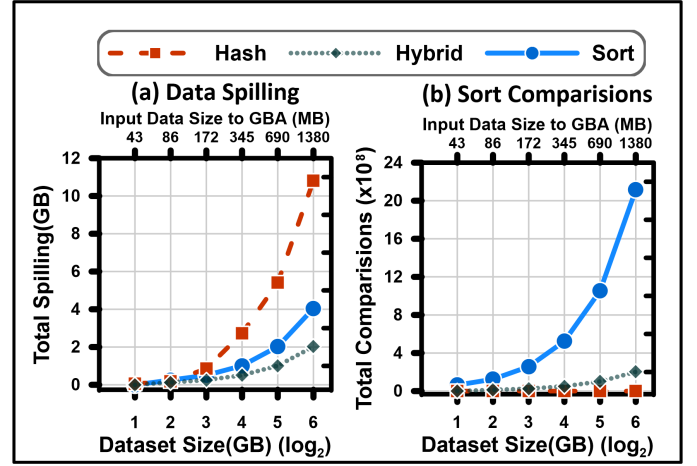


Fig. 9. Expr 2: Spilling and Sort Comparisons for 'Unique Integers'

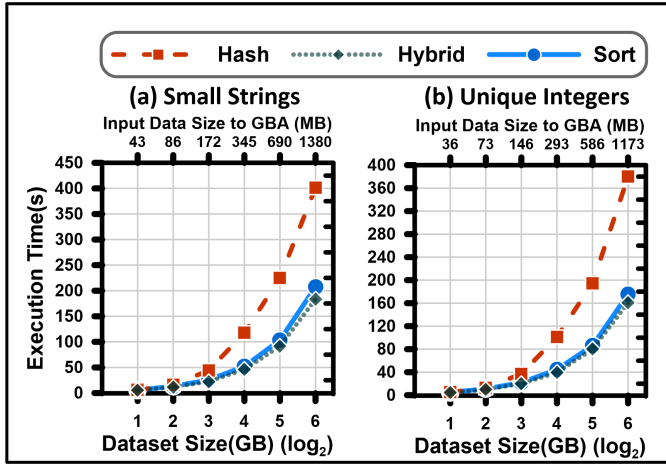


Fig. 7. Expr 2: Impact of Data Types String vs Integer

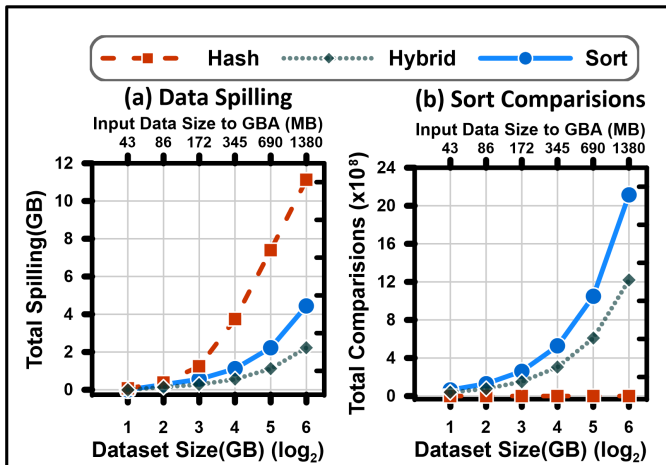


Fig. 8. Expr 2: Spilling and Sort Comparisons for 'Small Strings'

1) *String vs Integer Comparison:* In this experiment, we explore performance differences across string and integer data types. In Figure 7-(a), group by is performed on random

strings, each 10 characters long. Similarly, in Figure 7-(b) group-by will be performed on randomly generated unique integer values, mimicking the high cardinality of groups typically challenging for group-by operations. The aggregation function used in both of them is 'COUNT'. Given that the string values are unique, the number of groups generated matches the number of records maintaining comparability with integer data by ensuring a high number of unique group keys. These comparisons are intended to illustrate how performance varies when transitioning from integers to strings. As we can see integers have less execution times compared to strings. This happens due to the higher memory requirement of strings compared to integers leading to fewer runs and slightly lower spilling. Specifically, each integer value consumes 4 fewer bytes of memory than string values during execution of GBA for this case, leading to faster consumption of allocated group memory and more frequent spilling as shown in Figures 9 and 8. Additionally, hash-based GBA spills more data to disk compared to sort-based due to the hash table overhead taking about $1/3^{\text{rd}}$ of the given memory. This high overhead is due to the high number of groups stemming from unique group field values. Figures 8 and 9 illustrate data spilling and sort comparisons for each approach when handling smaller strings and unique integers respectively, as previously presented in Figure 7. In Figures 8 and 9, the y-axis represents data spilling in gigabytes (GB) and the total number of comparisons for sort operations in (a) and (b), respectively. As Figure 8-(a) shows, hash-based GBA has up to three times more data spillings compared to sort-based GBA with a dataset size of 64GB. Meanwhile, the execution time of sort-based GBA increases steadily with dataset size as do spilling and comparisons. The hybrid approach offers a slight advantage in execution time, reducing disk spilling by half compared to the sort-based method, as discussed earlier in Section IV. Moreover, the number of sorting comparisons is almost half for the hybrid approach compared to the sort-based approach as shown in Figure 8-(b). Similar behavior can also be seen in all three

approaches for spilling and comparison in the case of unique integers in Figure 9.

Next, we compare the performance of all three approaches using different types of strings as group-by values.

2) *Exploring Variations in String Values*: In this experiment, we evaluate the performance implications of group-by operations on a common ‘similar string’ attribute across all records in two separate tests: one with strings of 10 characters and another with 100 characters containing the same value across all records. This setup seeks to evaluate the computational overhead involved in sorting operations typical of the sort-based GBA method, which is expected to be inefficient when grouping values are highly skewed. Although less realistic, this experiment highlights the maximum performance difference between these approaches by representing the worst-case scenario for sort-based methods and one of the best-case scenarios for hash-based approaches. As shown in Figures 10-(a) and -(b), the sort-based GBA operator has a longer execution time due to the significantly higher number of comparisons required during sorting compared to other approaches (Figure 11). In contrast, both hybrid and hash-based GBA group the identical attribute values into a single group, leading to much faster execution. Since the aggregated record will fit into memory for both larger and smaller string cases and have zero spilling, the hash-based and hybrid methods maintain consistent timings. However, the performance of the sort-based approach degrades as string length increases due to the higher cost of sort comparisons, widening the performance gap, as illustrated in Figure 12.

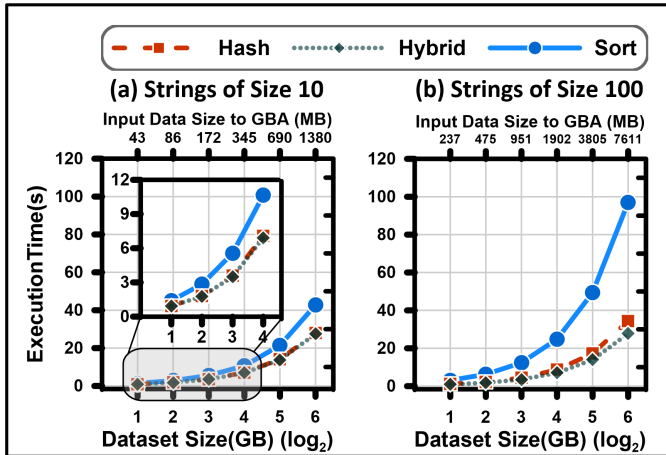


Fig. 10. Expr 2: Impact of Having Identical Group-by Values

The results of this experiment demonstrate that the hybrid GBA approach not only adapts efficiently to various data types but also offers significant performance advantages over traditional methods, especially in cases involving complex data types or high uniformity in group keys.

D. Experiment 3: Impact of Normalized Keys

In Experiment 3, we delve into the impact of normalized keys, enabled only in closed data types, on the performance

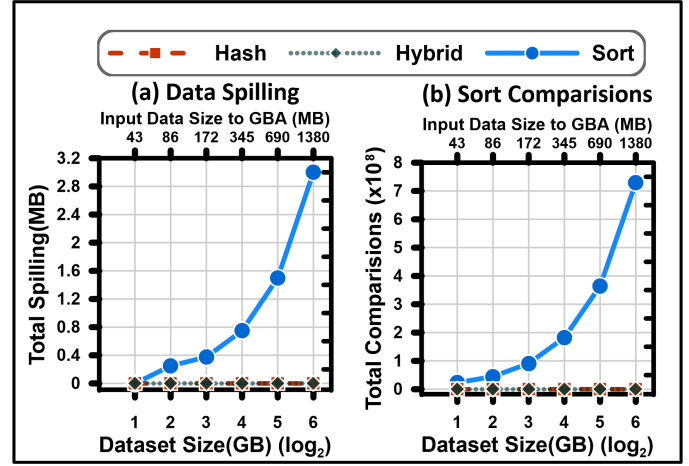


Fig. 11. Expr 2: Spilling and Sort Comparisons for Similar Strings - Size 10

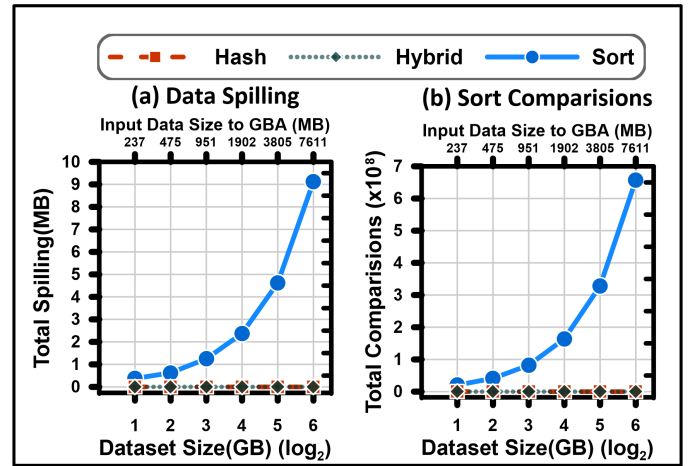


Fig. 12. Expr 2: Spilling and Sort Comparisons for Similar Strings - Size 100

of the three GBA approaches. This experiment contrasts with previous ones, where open data types were primarily used. In this experiment, we assess the differences in execution times when employing closed data types, where the schema is predefined and stored in metadata, facilitating optimized data access and processing.

For this experiment, the setup involved two scenarios. In the first scenario, the group-by was performed on a string attribute of the same size and identical values (Figure 13-(a)), while in the second scenario, the group-by field contained small random string values of size 10 (Figure 13-(b)). This setup allows us to observe the performance implications of group-by queries under closed data type conditions.

Comparing the results of this experiment for similar strings (Figure 13-(a)) with those of Experiment 2-2 (Figure 10-(a)) reveals a general improvement in execution times for all three group-by approaches when using closed data types instead of open data types. This improvement can be attributed to the predefined schema in closed datasets, which enhances data retrieval. Specifically, for the sort-based approach, the

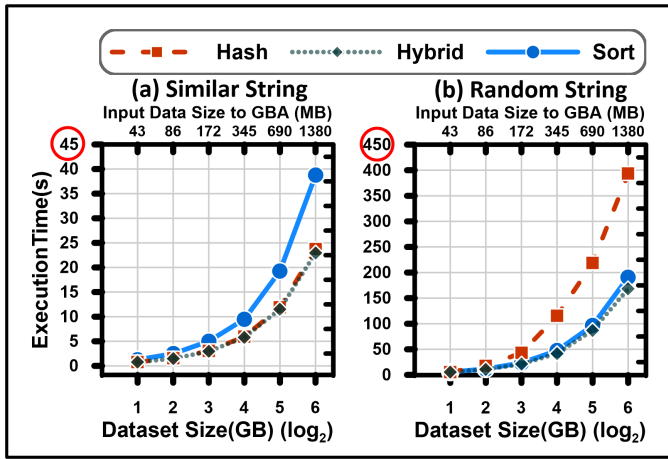


Fig. 13. Expr 3: Impact of Normalized Keys (Closed Dataset Type)

use of normalized keys for sorting further contributed to a more efficient sorting process, offering more improvement in performance as observed in the first graph compared to other approaches. In the case of random strings similar to open data types hash-based will have more spilling compared to sort and hybrid approaches and spilling keeps increasing with dataset size, which leads hash-group by performing worst of all. While sort-based still does more spilling than hybrid group which is later on noticeable with higher dataset sizes.

For similar strings, the hash and hybrid approaches experience no spilling due to identical values, while the sort-based approach incurs numerous comparisons, as each record must be fully compared when normalized keys are identical, leading to higher execution times.

The performance improvement observed with closed data types was nearly equivalent across all approaches, indicating that the benefits of a well-defined schema apply uniformly across different group-by methodologies. This consistency in performance enhancement highlights the advantages of using closed data types when possible.

E. Experiment 4: Benchmarking with TPC-H and TPC-DS

So far, we have evaluated the performance of various GBA operators under different scenarios using a modified version of Wisconsin Benchmark's dataset. In this experiment, we use TPC-H and TPC-DS benchmarks to evaluate the performance of the three GBA approaches under a more realistic dataset and query setting.

For both TPC-H and TPC-DS, we have used 100GB size of data with the same configuration of 2 NC and each having two data partitions as we have in previous experiments. For both benchmarks, we have evaluated all three approaches by selecting randomly from given substitution parameters mentioned by queries running it for eleven iterations and plotting the average of execution times of the last ten iterations on the y-axis while the x-axis shows query IDs. For TPC-H we have used Query 4, Query 9, Query 13, and Query 18.

For TPC-DS we have used Query 3, Query 15, Query 37, and Query 52.

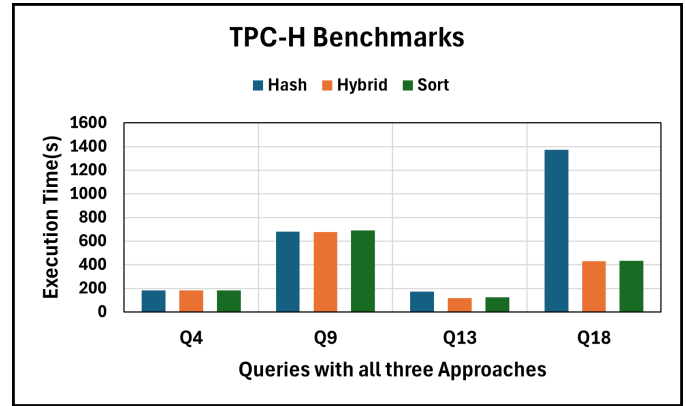


Fig. 14. Expr 4: Execution TPC-H benchmark queries

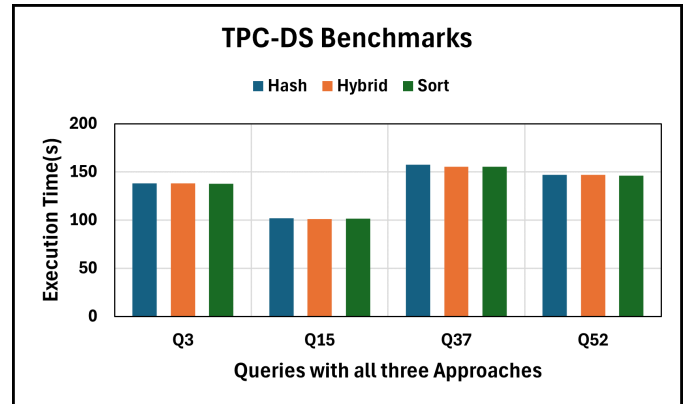


Fig. 15. Expr 4: Execution TPC-DS benchmark queries

As shown in Figures 14 and 15, in the case of all queries, our hybrid GBA performs better or at least closer to one of the traditional methods.

TPC-DS shows more variation in the execution times of different GBA approaches, largely due to the complexity of its queries and the involvement of other operators. For instance, in Query 18 and Query 13, the group-by operator appears in both the subquery and the outer query, interacting with other operators, which adds to the complexity. By contrast, the consistent results in TPC-H queries suggest that these queries may not be specifically designed to assess GBA approaches in depth.

While it is difficult to fully interpret the execution time similarities or differences in complex queries and datasets—given the lack of control over attribute values—the benchmarks show that the hybrid GBA approach consistently outperforms or at least matches the best of the hash and sort GBA methods. This holds true even in realistic datasets and workloads where multiple operators are involved in the query execution plan.

VI. RELATED WORK

Sorting and hashing have long been two primary methods widely used by the database community to implement and optimize the performance of various operators. In early relational database systems, only sort-based algorithms were employed. However, over time, hash-based algorithms gained popularity and are now often considered superior for certain use cases. Sorting-based algorithms remained of interest to researchers due to their sorted output, which can benefit subsequent operations requiring sorted input, a concept known as “interesting order”. However, with the introduction of hash-based algorithms, some hash-based methods outperformed sort-based ones. For example, Sort-Merge was outperformed by Hybrid Hash Join [6], [11], [15].

Sorting and hashing have been extensively used for designing operators such as joins [11], [13], [20], [22], [25], duplicate removal [7], [9], and grouping and aggregation [9], [13]. Their performance has been a subject of study for many researchers [6], [14], [21]. Traditionally, duplicate removal was performed as a post-step after sorting. However, the authors of [16] demonstrated in System R [5] that duplicates can be eliminated during the sorting process, rather than afterward.

There has been much discussion regarding the similarities between sort- and hash-based query processing algorithms. Some research suggests that their costs are similar, differing by small percentages rather than by orders of magnitude [10], [14]. Some even argue that hashing is equivalent to sorting by hash value [21]. However, studies have shown that special cases exist where one method outperforms the other, underscoring the importance of supporting both hash- and sort-based algorithms in query-processing systems [14]. Selecting the optimal algorithm requires accurate statistics, which has proven challenging. As a result, some research has focused on designing algorithms that are less dependent on statistics, while still delivering performance that is competitive with either approach.

One of the most significant contributions in this area comes from the work by Do et al. [9], which introduces two techniques for group-by and aggregate queries: “early aggregation” and “wide merging”. Early aggregation improves the input phase of external merge sort using in-memory indexing, similar to hash-based methods for unsorted input. Wide merging enhances the final merge step, making sort-based aggregation competitive with hash-based approaches across various input and output sizes. These innovations ensure that sort-based methods retain their advantage, especially when interesting orderings are needed, providing a robust alternative for query processing in complex data environments. This work is the closest study to our work and has greatly inspired us in our algorithm design choices.

VII. CONCLUSION AND FUTURE WORK

In this paper, we empirically examined the performance of two traditional group-by-aggregate (GBA) methods, hash- and sort-based approaches using Apache AsterixDB. Based on observed behavior, we proposed and evaluated a hybrid approach

that leverages the strengths of both methods. Our experiments showed that the sort-based approach performs well with large datasets and highly distinct group-by values, particularly for string data types. However, the high number of comparisons during sorting remains a primary bottleneck, especially for the string values with similar prefixes. Even normalized keys become ineffective under such cases. Conversely, the hash-based GBA is effective when the number of distinct grouping values is low; however, a large number of groups can lead to hash table overhead and excessive data spilling. Our hybrid approach combines the advantages of both methods, achieving optimal performance across various scenarios.

In conclusion, the evaluation of the hybrid approach for GBA queries in Apache AsterixDB shows significant efficiency gains. It achieves execution times comparable to the best sort- and hash-based operators simplifies optimization by unifying them into a single operator, and reduces reliance on inaccurate statistics. This study’s findings reinforce the potential of hybrid query processing methods to adapt to varying data characteristics and query demands, providing a scalable solution in big data environments.

The integration of sort-based and hash-based operations within a single framework, especially in handling all kinds of data types. Also, By minimizing disk I/O operations—a common bottleneck in large-scale data processing—the hybrid approach offers a compelling alternative to conventional methods and provides an alternative approach without knowing the details of data.

Future work includes adding support for additional aggregation functions, such as ‘AVERAGE’, and optimizing memory utilization for the hash table. We plan to expand experiments to cover various storage types (e.g., HDD, AWS EBS), hardware configurations, and real datasets. While current experiments focused on a limited cluster configuration to isolate algorithm behavior, future research will evaluate scalability in larger, more complex distributed environments.

ACKNOWLEDGEMENT

Our special thanks to Michael J. Carey and Wail Y. Alkowiileet for their invaluable feedback and insightful discussions, which greatly improved this work. We also deeply appreciate the Apache AsterixDB team for their continuous support and recommendations throughout the project.

REFERENCES

- [1] Apache AsterixDB. <https://asterixdb.apache.org/>. 2021.
- [2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelang, 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. Asterixdb: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.
- [3] Sattam Alsubaiee, Alexander Behm, Vinayak R. Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. *Proc. VLDB Endow.*, 7(10):841–852, 2014.
- [4] Apache AsterixDB. *AsterixDB SQL++ User Manual*, 2023.

- [5] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.
- [6] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, 2013.
- [7] Dina Bitton and David J. DeWitt. Duplicate record elimination in large data files. *ACM Trans. Database Syst.*, 8(2):255–265, 1983.
- [8] Nicolas Bruno and Surajit Chaudhuri. Interactive physical design tuning. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vasilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 1161–1164. IEEE Computer Society, 2010.
- [9] Thanh Do, Goetz Graefe, and Jeffrey F. Naughton. Efficient sorting, duplicate removal, grouping, and aggregation. *ACM Trans. Database Syst.*, 47(4):16:1–16:35, 2022.
- [10] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.
- [11] Goetz Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 406–417. IEEE Computer Society, 1994.
- [12] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3):10, 2006.
- [13] Goetz Graefe. New algorithms for join and grouping operations. *Comput. Sci. Res. Dev.*, 27(1):3–27, 2012.
- [14] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. Sort versus hash revisited. *IEEE Trans. Knowl. Data Eng.*, 6(6):934–944, 1994.
- [15] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.
- [16] Theo Härder. A scan-driven sort facility for a relational database system. In *Proceedings of the Third International Conference on Very Large Data Bases, October 6-8, 1977, Tokyo, Japan*, pages 236–244. IEEE Computer Society, 1977.
- [17] Shiva Jahangiri. Wisconsin benchmark data generator: To JSON and beyond. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2887–2889. ACM, 2021.
- [18] Taewoo Kim. *Enhancing Apache AsterixDB for Efficient Big Data Search and Analytics*. PhD thesis, University of California, Irvine, USA, 2018.
- [19] 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. Robust and efficient memory management in apache asterixdb. *Softw. Pract. Exp.*, 50(7):1114–1151, 2020.
- [20] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, pages 257–266. Morgan Kaufmann, 1989.
- [21] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1123–1136. ACM, 2015.
- [22] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. Hash-partitioned join method using dynamic destaging strategy. In François Bancilhon and David J. DeWitt, editors, *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings*, pages 468–478. Morgan Kaufmann, 1988.
- [23] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. Efficient use of the query optimizer for automated database design. In 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, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1093–1104. ACM, 2007.
- [24] Pouria Pirzadeh, Michael J. Carey, and Till Westmann. Bigfun: A performance study of big data management system functionality. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 507–514. IEEE Computer Society, 2015.
- [25] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.