IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 26, 2020, accepted June 14, 2020, date of publication June 18, 2020, date of current version June 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3003269

Distributed ATrie Group Join: Towards Zero Network Cost

PRAJWOL SANGAT, DAVID TANIAR, AND CHRISTOPHER MESSOM
Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia
Corresponding author: Prajwol Sangat (prajwol.sangat@monash.edu)

ABSTRACT The combination of powerful parallel frameworks and on-demand commodity hardware in distributed computing has made both analytics and decision support systems canonical to enterprises of all sizes. The unprecedented volumes of data stacked by companies present challenges to process analytical queries efficiently. This data is often organised as star schema, in which star join and group-by are ubiquitous and expensive operations. Although parallel frameworks such as Apache Spark facilitate join and group-by, the implementation can only process two tables at a time and fail to handle the excessive network communication, disk spills and multiple scans of data. In this paper, we present *Distributed ATrie Group Join (DATGJ)*, a fast distributed star join and group-by algorithm for column-stores. DATGJ uses divide and broadcast-based joining technique where the fact table columns are partitioned equally and *fast hash table (FHT)* for each dimension table are broadcasted. This technique helps it avoid cross communication between workers and disk spills. DATGJ performs a single scan of partitioned fact table columns and use FHT to join data. FHT uses Robin Hood hashing with the upper limit on number of probes and achieve significant speed up during join. DATGJ performs group-by and aggregation leveraging progressive materialisation and realising grouping attributes as a tree shaped deterministic finite automation known as *Aggregate Trie* or *ATrie*. We evaluated our algorithm using Star Schema Benchmark (SSBM) to show that it is 1.5X to 6X faster than the most prominent approaches while having zero data shuffle and consistently perform well with addition of resources and in memory-constrained scenarios.

INDEX TERMS Big data processing, column-stores, distributed processing, parallel group-by, parallel-join.

I. INTRODUCTION

Everyday companies are generating large volumes of data and their traditional relational database management systems (a.k.a. row-stores) fail to handle this data efficiently [1]. An alternative to row-stores are column-stores which store information about a logical entity as separate columns in multiple locations on the disk [2]. This novel layout improves the query performance on analytical workloads [2], [3]. The analytical queries focus on scan, predicate evaluation, join, grouping and aggregation operations. A study of customer queries in DB2 [4] has found that the group-by construct occurs in a large proportion of the analytical queries [5]. The queries in benchmarks such as Star Schema Benchmark (SSBM) [6] spend more than 50% CPU time in join, group-by and aggregation operations [7]. These queries often aggregate large portions of the data, which can lead to performance issues with big data sets. Therefore, efficient processing and optimization of queries involving join, group-by and aggregation operations using distributed computing is of paramount importance to improve and maintain the performance.

Large-scale data shuffling is inevitable in analytical queries such as distributed join between two large tables. This is still a less popular research topic or is left for data-centric generic distributed systems such as Apache Spark [8] for batch processing [9], [10]. While joins are the fundamental building block of any analytics pipeline, they are expensive to perform. In particular, the shuffling of data raises the concern of network communication cost in a distributed setting. Distributed transaction performance is mostly dominated by network latency rather than the throughput [10]. Although Spark facilitates joins and group-by using Resilient Distributed Datasets (RDDs) [11], Spark SQL [12] and Spark DataFrame [12] operations, it can process only two tables at a time, inducing multiple scans of data for star joins and requires one or two map-reduce iterations per join [13]. This means that the analytical queries will need $n-1$ or $2*(n-1)$ map-reduce iteration where n is the number of tables used by the query. In addition, it requires excessive disk access and network communication because of cross-communication

The associate editor coordinating the review of this manuscript and approving it for publication was Asad Waqar Malik.

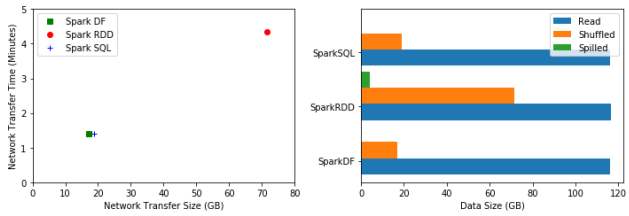


FIGURE 1. Average disk access and network transfers communication for SparkRDD, SparkDF and SparkSQL based joins for SSBM Queries [SF=200, Nodes=5, Number of Cores=35 (7 per node) and Total Memory=150GB (30GB per node)].

between the worker nodes as shown in Figure 1. In particular, unnecessary disk access is often the result of disk spill: the data is spilled into the disk when the memory buffer overflows. Furthermore, the excessive shuffling of records not only significantly increases the network communication cost, but also prevents further processing of the algorithm [14]. Therefore, naive Spark implementation fails to handle the issues such as multiple scans of data, excessive network communication and disk spill.

This paper extends the work of our earlier paper [15]. There, we presented ATrie Group Join (ATGJ) and compared its performance against parallel star join algorithms [3], [7], [16] in a multi-threaded environment. In this paper, we present Distributed ATrie Group Join (DATGJ), a fast-distributed star join and group-by algorithm for column-stores. DATGJ has only one map-reduce iteration regardless of the number of tables used in the query.

In the map phase, DATGJ builds a fast hash table (FHT) for each dimension table and broadcasts each one to separate worker nodes. FHT implementation uses open addressing [17], linear probing [18], Robin hood hashing [19], and a prime number of slots with an upper limit on the number of probes. These four methods are common in hash table implementation, although our new contribution and the primary source of speed-up is the setting of an upper limit on the number of probes.

In the reduce phase, DATGJ performs a single scan of the partitioned fact table columns. Each record is checked against corresponding FHT based on the foreign key/primary key relationship between the fact and dimension table, and the matching records are grouped and aggregated using *Aggregate Trie* or *ATrie*. The divide and broadcast-based joining technique helps DATGJ avoid cross-communication between worker nodes and the disk spills.

Our approach to optimise group join and aggregation was evaluated against the SSBM benchmark. The performance results show that our approach is 1.5X to 6X faster than the most popular current approaches while having zero data shuffle. Moreover, it consistently performs well with the addition of resources and in constrained memory scenarios. In summary, we make the following contributions in this paper:

1) We present a new optimisation technique for efficient search in the hash table. The key idea is to use Robin Hood hashing [19] with an upper limit imposed on the number of probes which is implemented in *Fast Hash Table (FHT)*.
2) We propose a novel approach to perform group-by and aggregation operation by realising grouping attributes as a tree-shaped deterministic finite automation known as *Aggregate Trie* or *ATrie*.
3) We propose a new star group join and aggregation algorithm for distributed column-stores known as *Distributed ATrie Group Join (DATGJ)*. DATGJ requires only one map-reduce iteration regardless of the number of tables used in the query. It uses hash-based broadcast technique, performs a single scan join and leverages progressive materialisation to solve the problem of grouping and aggregating data using ATrie.
4) We perform *extensive experiments* using the SSBM benchmark and compare the performance with some of the most prominent approaches. The results show that our strategy has zero data shuffle and zero disk spill, and avoids multiple scans of data while being competitive and better than the current approaches.
5) We propose *an analytical model* to understand and predict the query performance of DATGJ. The model accuracy has been verified by detailed experiments with different hardware parameters.

Rest of the paper is organised as follows: First, we describe the related works in Section II. Then, we discuss the benchmark used in the experiment in Section III. After that, we explain the optimisation technique for efficient search in the hash table in Section IV. Next, we propose a new star join and group-by algorithm in Section V and outline the proposed grouping method in Section V-C. After that, we report the results of experimental evaluation in Section VI and analytical evaluation in Section VII. Finally, we present the conclusions in Section VIII.

II. RELATED WORKS

In this section, we review the latest research on parallel star joins and distributed star joins. Parallel star joins use parallel computing and execute join tasks simultaneously using multiple processors, whereas distributed star joins use distributed computing to divide a single join task among multiple worker computers in order to complete the join operation. Although parallel star joins improve the performance, they are limited by the hardware as the join operation is performed using a single computer [15], [16]. On the other hand, distributed star joins involve multiple worker computers and the distributed computing not only improves scalability, fault tolerance, resource sharing but also helps perform computations efficiently [14], [20].

A. PARALLEL STAR JOINS

We outline the related works that have been proposed for two different storage architectures: 1) Row-Oriented and 2) Column-Oriented for parallel star joins.

1) ROW-ORIENTED STORAGE

O'Neil *et al.* [21] proposed a bitmap star join using bitmap indices which suggested that an index lookup in the dimension table could be faster than hash join, whereas, Markl *et al.* [22] proposed hierarchical physical clustering as an alternative to the use of indices and aimed at limiting the number of I/O access to the fact table.

Weininger [23] proposed index union and semi-join reduction plans with bitmap filters for efficient execution of star schema joins. Aguilar-Saborit *et al.* [6] revisited the star join techniques to analyse the most up-to-date strategy for ad-hoc star join query processing. They proposed a hybrid solution that improved the features of bitmap star join [21] and hierarchical physical clustering [22], and showed near-optimal results in multiple use-cases. Galindo-Legaria *et al.* [24] proposed novel execution strategies for star join queries such as index intersections, dimension cross-product with fact table lookup, and semi-join reduction using bitmap filters. They showed that the optimisation strategies improved the star join performance. Fang *et al.* [25] vertically or horizontally vectorised the probing phase using SIMD instruction and sped up the probe by prefetching. They showed that the vertical vectorised integrated probe is faster than the scalar version.

All these works have been motivational approaches towards developing star join algorithms for column-stores. ATGJ is inspired by [21], [24], [26], although it has been designed to address queries with join, group and aggregation operations for column-stores.

2) COLUMN-ORIENTED STORAGE

Abadi *et al.* [3] extended the work on improving the performance for star joins [21], [23] by taking advantage of the column-oriented layout and rewriting the predicates to avoid the hash lookups. However, the algorithm has performance bottleneck of a multi-pass scan for column processing and increases memory consumption with the increasing number of tables in the join query. Also, it incurs a significant memory overhead cost because it creates multiple intermediate position lists and follows a task-parallel approach resulting in sub-optimal use of resources. ATGJ [15], on the other hand, performed a single scan of the fact columns and used a mixture of data and task parallelism for optimal use of computing resources.

Yuan *et al.* [27] comprehensively evaluated the performance of graphical processing unit (GPU) query execution, conducting a detailed analysis and comparison of GPU and CPU. They conclude that GPUs significantly outperform CPU only when processing certain kinds of queries when data are available in the pinned memory and the performance of analytical queries does not increase correspondingly with the rapid advancement of GPU hardware. However, Guoliang and Guilan *et al.* [28] proposed a massively parallel and highly scalable star join algorithm based on GPU. To facilitate and improve the execution of hash joins in GPUs, they used a bloom filter instead of hash lookup and integrated late materialisation such that the fact table is accessed only once. This algorithm is inspired from [3] and modified to work on GPU which is outside of the scope of this research.

Sangat *et al.* [16] proposed a progressive parallel join algorithm for column- stores. They proposed a new data structure known as a Multi-Attribute Array Table (MAAT) based on the concept of Concise Array Table (CAT) [29]. MAAT is a variation of CAT that consists of *an indexed array* storing multiple attributes. The key advantage offered by an indexed array is the elimination of nodes and pointers that are used in Standard-Chain Hash Table (SCHT) [30]. This configuration permits the good use of CPU cache and hardware data prefetch while simultaneously saving memory space. MAAT holds the intermediate attributes required for the join query processing. It eliminates the problem of re-scanning of fact table columns performed by [3]. However, this joining technique is not optimised for group-by and aggregation operation whereas ATGJ [15] focuses on improving group-by and aggregation operation using ATrie.

Chavan *et al.* [7] optimised aggregation operations over joins by pushing group-by expressions down to the scan of dimension tables. Their solution replaces traditional join and group-by operators with fast in-lined scan operators. This algorithm is efficient only if the In-memory Aggregator (IMA) does not become too large [7]. Also, with the increasing number of dimensions and grouping attributes, this algorithm creates additional *key vectors* and *temporary tables* to process the group join, which significantly increases the execution time. ATGJ [15], on the other hand, uses ATrie to facilitate grouping and processing the data in tight loops. It avoids the creation of additional data structures with the increase of group-by attributes as well as perform efficiently even when ATrie becomes bushy unlike [7] where the performance degrades if IMA becomes too large. Finally, the design of [7] is such that it is not suitable to port to a distributed environment whereas ATGJ [15] can be ported to a distributed environment with moderate modifications which is discussed in Section V and the proposed algorithm in this paper.

B. DISTRIBUTED STAR JOINS

We outline the related works that have been proposed for two different storage architectures: 1) Row-Oriented and 2) Column-Oriented for distributed star joins.

1) ROW-ORIENTED STORAGE

Datta *et al.* [31] proposed a parallel star join algorithm based on the vertical partitioning of data in a distributed environment. Aguilar-Saborit *et al.* [26] proposed a star hash join based on the use of bloom filters in cluster architectures to reduce both I/O and data traffic communication. Also, several other researchers such as [32], [33] have proposed the use of bloom filters [34] for map-side joins. The use of a bloom filter is based on an allowable error and require a high number of hash functions to be executed against every tuple from the fact table columns to decide whether or not the tuple should be filtered out [34]. This process becomes

computationally expensive with a large amount of data. Purdilă and Pentiuc *et al.* [13] proposed a fast and efficient star-join query execution algorithm built on top of the map-reduce framework using dynamic filters against dimension tables, which reduced I/O operations and computational complexity. Ramdane *et al.* [35] combined a data-driven and a workload-driven model to create a new scheme for distributed big data warehouses using Hadoop. They performed a one-stage star join operation and skipped the loading of unnecessary HDFS blocks. All of these algorithms present high network communication and several sequential jobs that produce challenging bottlenecks in distributed systems.

Many other algorithms such as [14], [36] applied predicate on the dimensions, broadcast the results to all nodes, and applied joins locally which minimised the disk spills and network communication. However, these algorithms are designed for row-oriented data, not column-oriented data.

2) COLUMN-ORIENTED STORAGE

Zhu *et al.* [37] proposed a star join method for column-oriented data stored in Hadoop Distributed File System (HDFS). This join used the *HdBmp Index* which can filter out most of the unnecessary tuples in tables, thereby greatly reducing the network overload. Zhou *et al.* [38] proposed two cache-conscious algorithms in the map-reduce environment that avoids fact table data movement. The fact table is partitioned into several column groups for cache optimization. The algorithms proposed by Zhou *et al.* [38] are based on Abadi *et al.* [3] and therefore, the problems discussed in Section II-A2 remain. Besides, Zhou *et al.* [38] deal with join operations only and not group-by and aggregation operations.

Therefore, we propose a Distributed ATrie Join (DATGJ) that is designed for column-oriented data and performs faster than its competing algorithms. We implement the column-oriented version of Brito *et al.* [14] and show that DATGJ outperforms these Spark based algorithms.

III. STAR SCHEMA BENCHMARK (SSBM)

The Star Schema Benchmark (SSBM) [39] is widely used in various data warehousing research studies [3], [7], [25], [27]. It consists of a single fact table `LINEORDER` table and four dimension tables `CUSTOMER`, `SUPPLIER`, `PART` and `DATE` table, which are organised as a star schema, as shown in Figure 2.

The SSBM consists of thirteen queries divided into four flights:

Flight 1 consists of three queries that have a restriction on one dimension attribute, as well as the `DISCOUNT` and `QUANTITY` columns of the `LINEORDER` table.

Flight 2 consists of three queries that have a restriction on two dimension attributes and calculate the revenue for particular product classes in particular regions, grouped by product class and year.

Flight 3 consists of four queries that have a restriction on three dimension attributes and calculate the revenue in a particular region over a time period, grouped by customer nation, supplier nation, and year.

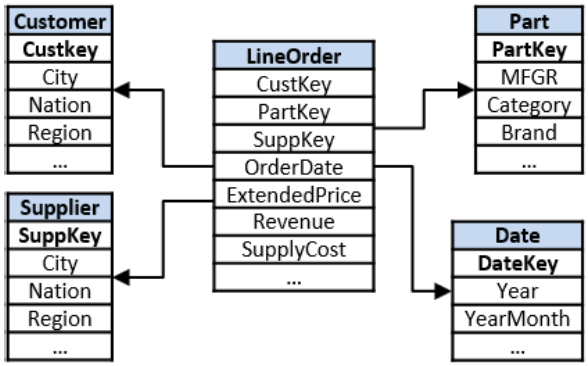


FIGURE 2. Schema of SSBM.

Flight 4 consists of three queries that have a restriction on three dimension attributes and calculate profit (`REVENUE - SUPPLYCOST`) grouped by year, nation, and category for query 1; and for queries 2 and 3, region and category.

Table 1 summarises the major characteristics of the SSBM queries. Let us consider the Query 3.1 (shown below) that finds the revenue volume for the line order transactions by customer nation, supplier nation and year within a region 'Asia' in 1992 and 1997. It will be used as a running example to reflect the 4 phases in our algorithm.

```
SELECT c.nation, s.nation, d.year,
sum(lo.revenue) AS revenue
FROM customer AS c, lineorder AS lo,
supplier AS s, [Date] AS d
WHERE lo.custkey = c.custkey
AND lo.suppkey = s.suppkey
AND lo.orderdate = d.orderdate
AND c.region = 'ASIA'
AND s.region = 'ASIA'
AND d.year BETWEEN~1992~and 1997
GROUP BY  c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

IV. FAST HASH TABLE (FHT)

Hash tables provide an efficient way to maintain a set of keys or map keys to values. The theoretical run-time to search, insert, and delete an item in the hash table is amortized $\mathcal{O}(1)$. By 'amortized' we mean that, on average, an operation (e.g. an insertion) takes $\mathcal{O}(1)$, but occasionally it may take more time. We can not improve on the theory of hash tables, but we can improve on the practice. We have improved the hash table for the fastest lookup, while having fast inserts and deletes. The key idea is to use Robin Hood hashing [19] with an upper limit imposed on the number of probes. If an element has to be more than X positions away from its ideal position, we increase the table because, with a bigger table, every element can be close to its desired position. X can be relatively small which allows optimisations for the inner loop

TABLE 1. Summary of major operations and Filter Factor (FF) analysis of SSBM queries. L represents the `LINEORDER` fact table and D, S, C and P represent the `DATE`, `SUPPLIER`, `CUSTOMER` and `PART` dimension tables.

Query	Operation	FF Customer	FF Supplier	FF Part	FF Date	FF LineOrder	Selectivity (SF = 1)
1.1	L ⋈ D	-	-	-	1/7	0.47*3/11	0.019
1.2	L ⋈ D	-	-	-	1/84	0.2*3/11	0.000065
1.3	L ⋈ D	-	-	-	1/364	0.1*3/11	0.000075
2.1	L ⋈ P ⋈ S ⋈ D	-	1/5	1/25	-	-	0.008
2.2	L ⋈ P ⋈ S ⋈ D	-	1/5	1/25	-	-	0.0016
2.3	L ⋈ P ⋈ S ⋈ D	-	1/5	1/1000	-	-	0.0002
3.1	L ⋈ C ⋈ S ⋈ D	1/5	1/5	-	6/7	-	0.034
3.2	L ⋈ C ⋈ S ⋈ D	1/25	1/25	-	6/7	-	0.0014
3.3	L ⋈ C ⋈ S ⋈ D	1/125	1/125	-	6/7	-	0.000055
3.4	L ⋈ C ⋈ S ⋈ D	1/125	1/125	-	1/84	-	0.00000076
4.1	L ⋈ C ⋈ P ⋈ S ⋈ D	1/5	1/5	2/5	-	-	0.016
4.2	L ⋈ C ⋈ P ⋈ S ⋈ D	1/5	1/5	2/5	2/7	-	0.0046
4.3	L ⋈ C ⋈ P ⋈ S ⋈ D	1/5	1/125	1/25	2/7	-	0.000091

of a hash table lookup. The FHT implementation involves open addressing [17], linear probing [18], Robin hood hashing [19], and a prime number of slots with an upper limit set for the number of probes. These four methods are common in hash table implementation; however, our new contribution and the primary source of speed-up is based on setting an upper limit for the number of probes.

A. AN UPPER LIMIT ON THE NUMBER OF PROBES

We try to limit the number of slots the table would consider before increasing the underlying array. Initially, the number of probes is set to a low number, such as five. This works well for small tables, but if there are random inserts into a large table, it is easy to reach five probes and increase the table even though it is mostly empty.

During random inserts, using $log_2(n)$ as the limit, where n is the number of slots in the table, we are able to reallocate only when the table is approximately 65% full. However, when inserting sequential values, we have a 100% fill factor before reallocation.

1) WHY USE UPPER LIMITS?

Let us say we rehash the table so that it has 1000 slots. The hash table will then increase to 1009 slots (i.e. the closest prime number). $log_2(1009) = 10$, so the probe count limit is set to 10. Therefore, the key idea is to allocate an array of 1019 slots instead of 1009 slots. Now, if two elements hash to index 1008, we can go over the end and insert at index 1009. This avoids any checking of bounds because the probe count limit ensures that we will never go beyond index 1018. If we have eleven elements that go into the last slot, the table will increase and all those elements will hash to different slots.

Algorithm 1 is basically a linear search and is better than simple linear probing in two ways:

- **No bounds checking:** Empty slots have -1 in their `distanceFromDesired` value so the empty case is the same case as finding a different element.
- **Better performance:** This algorithm performs at most $log_2(n)$ iterations. Normally, the worst case time complexity for search in a hash table is $\mathcal{O}(n)$. However, in our case, it is $\mathcal{O}(log_2(n))$. This is significant because, with linear probing, it is highly likely that we will hit the worst case since linear probing usually groups elements together.

Algorithm 1 Search Fast Hash Table

```
Data: FindKey key
Result: EntryPointer ep
index <− hashPolicy.indexForHash(hashObject(key))
ep <− entries + index
distance <− 0
while true do
    if ep.distanceFromDesired < distance then
        return end
    else if comparesEqual(key, ep.value) then
        return ep
    distance++
    ep++
end
```

2) MEMORY OVERHEAD

The memory overhead of the search operation is one byte per item. One byte is padded out to the alignment of the data type that is inserted. For instance, if we insert `int`, the one byte will obtain three bytes of padding. Hence, we have four bytes of overhead per item. If we insert `pointers`, there will be seven bytes of padding so that we have eight bytes of overhead per item. We can change the memory layout to solve this, but it would incur two cache misses for each lookup instead of one cache miss. Therefore, the memory overhead is one byte per item plus padding.

B. EVALUATION

We experimented to identify the differences in the performance and memory consumption of Standard-Chain Hash Table (SCHT), Concise Hash Table (CHT) [29], Concise Array Table (CAT) [29], Multi-attribute Array Table (MAAT) [16] and Fast Hash Table (FHT). A total of

50 million records were inserted, and the same amount of data were searched and deleted. We measured both the 100% successful searches and 100% unsuccessful searches. Each dataset consisted of $< key, value >$ where *key* is the hash key, and *value* is its associated value. We recorded the memory usages using `Pympler` that measures, monitors and analyses memory behaviour and returns the size of an object in bytes. The numbers reported are the averages of ten iterations.

1) SEARCH PERFORMANCE

Case 1: 100% Successful - In this test, all the search keys are guaranteed to be found in the table.

Case 2: 100% Unsuccessful - In this test, none of the search keys is found in the table.

Figure 3 (b) shows that FHT performs better than the other data structures in both cases. All the data structures have different performances depending on the current load factor. For example, when a table is 25% full, the search will be more faster than when it is 50% full because there are more hash collisions when the table has a high fill factor. For Case 2, the load factor is of paramount importance. The higher the fill factor, the more elements there are to search before concluding that an item is not in the table. Therefore, better performance can be achieved by limiting the probe count. With the maximum load factor set to $log_2(n)$, the hash can be mapped to a slot just by looking at the lower bits. The only significant difference is that FHT requires one byte extra storage (plus padding) per slot; therefore, it uses slightly more memory than CAT and MAAT as shown in Figure 3 (a).

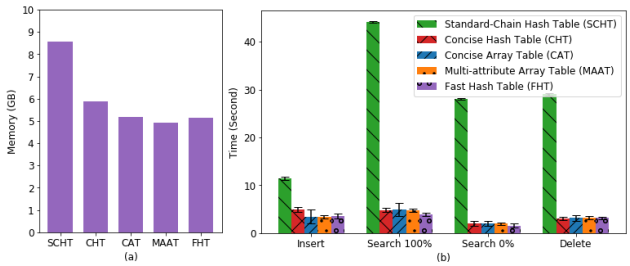


FIGURE 3. (a) Memory usages comparison of various data structures (b) Performance comparison of various data structures to `insert` a new key-value pair and `search` or `delete` the value associated with a key (Search 100% = 100% Successful and Search 0% = 100% Unsuccessful).

2) INSERT PERFORMANCE

Figure 3 (b) shows that FHT has a comparable performance with CAT and MAT. FHT is slightly slower compared to CAT and MAAT because they do not move elements around when inserting. FHT uses Robin Hood hashing that requires moving elements around when inserting so that every node is as close as possible to its ideal position. It is a trade-off where insertion becomes more expensive, but the search becomes faster.

3) DELETE PERFORMANCE

Figure 3 (b) shows that CHT, CAT, MAAT and FHT all have similar performance. However, one only significant difference between FHT and CHT is that when CHT deletes an element, it leaves behind a tombstone. That tombstone will be removed if we insert a new element in that slot. A tombstone is a requirement of the quadratic probing that CHT does on search: When an element is deleted, it is very difficult to find another element to take its slot. In Robin Hood hashing with linear probing it is trivial to find an element that should go into the recent empty slot: just move the next element one forward if it is not in its ideal slot. In quadratic probing, it might have an element that is four slots over. When that one gets moved, we need to find a node to insert into the newly vacated slot. Instead, it inserts a tombstone and then the table knows to ignore tombstones on search which will be replaced on the next insert i.e. the table will be slightly slower once it has tombstones in the table. Therefore, CHT has a fast delete at the cost of slow search after a delete.

V. DISTRIBUTED ATrie GROUP JOIN (DATGJ)

The Distributed ATrie Group Join (DATGJ) has four different phases: a) Broadcast Phase, b) Single Scan Hash Join, c) Group-By using ATrie and d) Merge ATries.

A. BROADCAST PHASE

In this phase, predicates are applied to the appropriate dimension tables to create mappings in the respective filtered dimension tables (FDims). All FDims are collected as the hash table (FHDims). The primary key of the dimension tables acts as the key in the hash table and the grouping attributes act as the value. These hash tables are broadcast to all workers that help to efficiently prune out non-qualifying rows.

The size of the HDims are smaller than the dimension tables, which makes it a suitable candidate for broadcasting. The broadcasting of FHDims saves significant network communication cost by avoiding the re-transmission of FHDims when many join tasks execute in parallel on each worker [11], [12]. This phase helps to reduce the network communication cost of tasks, which is one of the important features of the algorithm. An example of the execution of this phase in one worker node on some sample data is shown in Figure 4.

This phase can be easily adapted to other schema such as the snowflake schema. Predicates can be applied to the appropriate dimension tables to create the respective filtered dimension tables with the primary key of the main dimension table and the foreign key of the child look-up tables. Using the foreign key of the look-up tables, we can obtain the associated value, which is the grouping attribute in the query. The main idea is to generate a hash table with *dkey*: the primary key from the respective dimension table and *value*: the grouping attribute in the query from the same dimension table.

In Figure 5, the dimensions are already partitioned and stored in each worker node which is represented as Dim_{ij} where $i = 1 \ldots M$ workers and $j = 1 \ldots N$ dimensions. The predicate filters PF_i where $i = 1 \ldots N$ is applied to appropriate dimension tables to create $FDim_{ij}$ where $i = 1 \ldots M$ workers

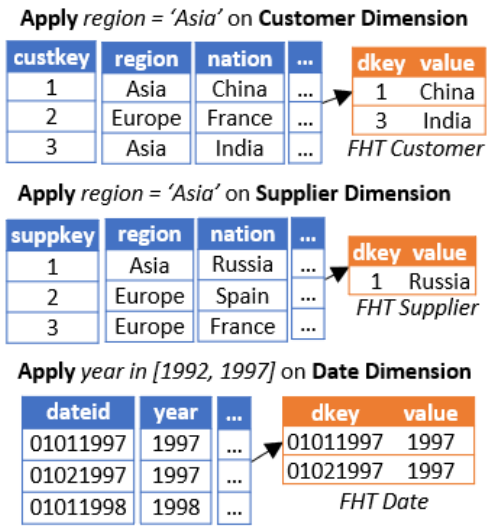


FIGURE 4. Predicate filtering and hash table creation sample demonstration on Query 3.1 from SSBM in one worker node. *dkey* in the hash table is the primary key from the respective dimension table and *value* is the grouping attribute in the query from the same dimension table.

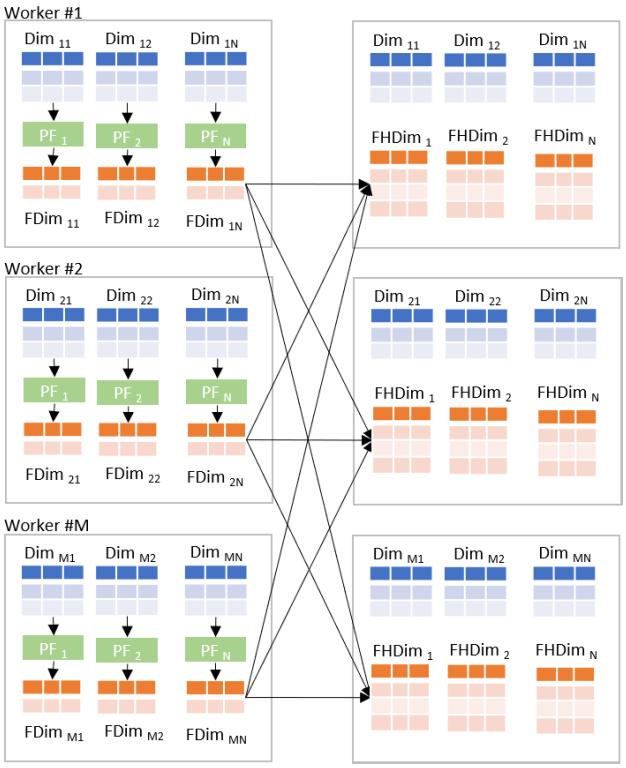


FIGURE 5. Applying the predicate filter and broadcasting the hash table.

and $j = 1 \ldots N$ dimensions. All $FDim_{ij}$ are collected to create $FHDim_i$ where $i = 1 \ldots N$ that are broadcast to all workers.

B. SINGLE SCAN HASH JOIN

The size of the fact table is significantly larger than the dimension tables. The data is partitioned using a random-equal partitioning technique where each worker works on the equal amount of data. The single scan is performed on the fact table columns and the broadcast FHTs are used to perform the join. Each task works on its allocated partition of the fact table data to retrieve foreign key and probe it into the corresponding FHTs to create a group aggregation object (GAO) (refer Definition 1).

From the load balancing perspective, the load of each processor in terms of the number of records processed is the same; i.e., in each processor there will be an equal fragment of the fact table and the entire hash table for the corresponding dimension tables; hence, there is no load imbalance problem. However, the load balancing problem theoretically might still occur even when fact table is partitioned equally. This problem arises from the imbalance of result production such as the cost of join with hash table and the cost of group-by operation using ATrie. Some processors that produce more results than others might require more time to complete join processing. However, this problem is significantly minor compared to the situation when the fact table is not being partitioned equally [40]. In addition, if data parallel probing approach results in an unbalanced workload, it can be handled using techniques such as Morsel-driven parallelism [41] or Index Vector Partitioning (IVP) [42].

In Figure 6, each Task $t = 1 \ldots K$ work on the independent partition of data from the fact table columns and perform join with all broadcast fast hash tables $FHDim_i$ where $i = 1 \ldots N$. Each record on the fact table columns is scanned only once during the join process unlike other algorithms such as Invisible Join [3] which reduces the disk access time in the algorithm.

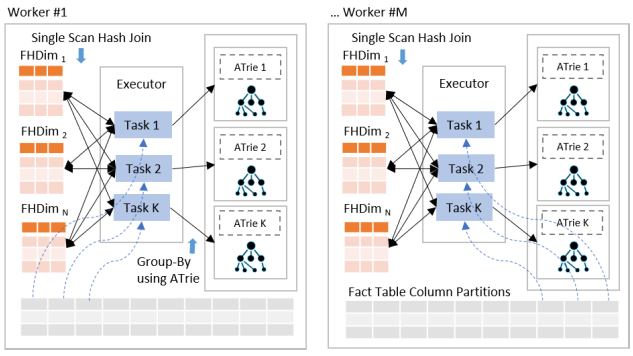


FIGURE 6. Broadcast join using a fast hash table and group-by using ATrie.

C. GROUP-BY USING ATrie

Before we explain the grouping stage, we discuss our novel grouping technique below. We focus on the terminologies used, a formal definition of the aggregate trie and the core operational concepts.

1) TERMINOLOGIES

An *Aggregate Trie (a.k.a. ATrie)* is a collection of grouping attributes or value called nodes. *Node* is the main component of the ATrie. It stores the actual data along with links to other

nodes. The topmost node in the ATrie is called the *root node*. The root node is non-empty and does not have a parent. Each node in a tree can have zero or more *child nodes*. A node that has a child is a *parent node*. An *internal node* is any node in ATrie that has both *parent* and *child nodes*. Similarly, the bottom-most node in the ATrie that does not have a child node is called the *leaf node*. The *height* of the ATrie is the height of the root node.

2) FORMAL DEFINITION

As the foundation of this work, we define the *group aggregation object* and the *aggregate trie* as follows:

Definition 1: Group Aggregation Object (GAO): Group Aggregation Object (GAO) is a list data structure that represents a set of grouping attributes. It includes an aggregation attribute at the end. Formally, the semantics of the GAO is:

$$GAO = \{[x_1, x_2, \ldots, x_{n-1}, x_n] \mid x_n \in \mathbb{Z} \wedge (x_1, x_2, \ldots, x_{n-1}) \in \textit{group attributes}\} \quad (1)$$

Let us consider a record where the customer is from Nepal, the supplier is from China, the item was ordered in 2019 and the revenue collected is 10421. The GAO for this record is GAO = [''Nepal'', ''China'', ''2019'', 10421].

Definition 2: Aggregate Trie: The aggregate trie (a.k.a ATrie) is a deterministic tree of GAOs with height h = `sizeOfGAO()`. The root node in ATrie describes level h, while nodes at level 1 are the leaf nodes and hold the aggregated value. Each node of the ATrie includes a hash table that has a variable size depending on the distinct group of attributes. All the descendants of a node have a common attribute associated with that node, and the root node is associated with the empty node. Furthermore, ATrie has following three important properties:

- *Deterministic Property:* Each distinct GAO has only one path within the ATrie. Due to these deterministic paths, only a single key comparison at each level is required and there is no dynamic reorganisation of attributes for any operations.
- *Data Compression:* The ATrie can represent GAO in a compact form. When many GAOs share the same grouping attribute, these shared grouping attributes can be represented by a shared part of the ATrie, allowing the representation to use less space than it would take to list out all the distinct GAO separately. For example, any GAO can be represented as paths in the ATrie by forming a vertex for every grouping attribute and making the parent of one of these vertices represent the attribute with one fewer element.
- *Progressive Materialisation:* ATGJ maneuver the idea of progressive materialisation [16] by using the ATrie as a means of performing materialisation and aggregation on the fly when scanning the fact columns and inserting GAOs into the ATrie. Progressive Materialization adopts the notion of late materialization to push the tuple construction as late as possible but carries attribute values required in the query processing throughout the query plan [16].

3) PHYSICAL DATA STRUCTURE

The basic form of implementing the ATrie is with the use of the hash table, where each node contains a hash table with child node(s), one for each unique value of grouping attributes in GAO. Therefore, we use FHT for the implementation. Note that using a FHT for children would not allow lexicographic sorting because FHT would not preserve the order of keys. Nevertheless, sorting the attributes is not the focus of this paper.

4) ATrie OPERATIONS

Figure 7 shows an example of the step-wise insertion of GAOs into the ATrie which will be used to describe the operations relating to the ATrie. For simplicity, assume that K = the maximum number of distinct attributes at all levels of the ATrie.

Reading or Searching the ATrie: To read or search the ATrie, follow the path designated by addresses advancing to the indicated height of ATrie each time we move to a new grouping attribute. At each height, we search for a new grouping attribute. If the new grouping attribute exists, we move to that address. If we come to a height that contains no address then we have reached the leaf node that holds the aggregated value. The worst-case time complexity for this operation is $\mathcal{O}(h * K)$.

Let us read or search attributes in a GAO = [''Nepal'', ''China'', ''2019'', 11000] in a complete ATrie (refer Figure 7 (d)). At height = 4 (root node), we check for the existence of customer nation = ''Nepal''. Since it exists, we move to height = 3 and check for the existence of supplier nation = ''China'' in the hash table that was pointed by customer nation = ''Nepal''. We find the match, therefore, we move to height = 2 and check for the existence of order year = 2019 in the hash table pointed by supplier nation = ''China''. At height = 1, there is no pointer to the hash table, therefore, we read the aggregated value revenue = 11000.

Insert a GAO into the ATrie: To insert a GAO into the ATrie, we first read the ATrie. If the grouping attribute present in GAO is found, it is not inserted, otherwise, it is inserted into the ATrie. This setup enables shorter access time, makes it easier to add nodes or update the values, and offers greater convenience in handling a varying number of grouping attributes. The main disadvantage is storage space inefficiency, which is not problematic when the storage is large. The worst-case time complexity for this operation is $\mathcal{O}(h + h * K) \approx \mathcal{O}(h * K)$.

Figure 7 shows an example of the step-wise insertion of GAOs into the ATrie. We discuss two examples of insertion shown in Figure 7 (b) and (c). Let us insert a GAO = [''Nepal'', ''India'', ''2018'', 4196] (Figure 7 (b)). At height = 4 (root node), we check for the existence of customer nation = ''Nepal''. Since it exists, we move to

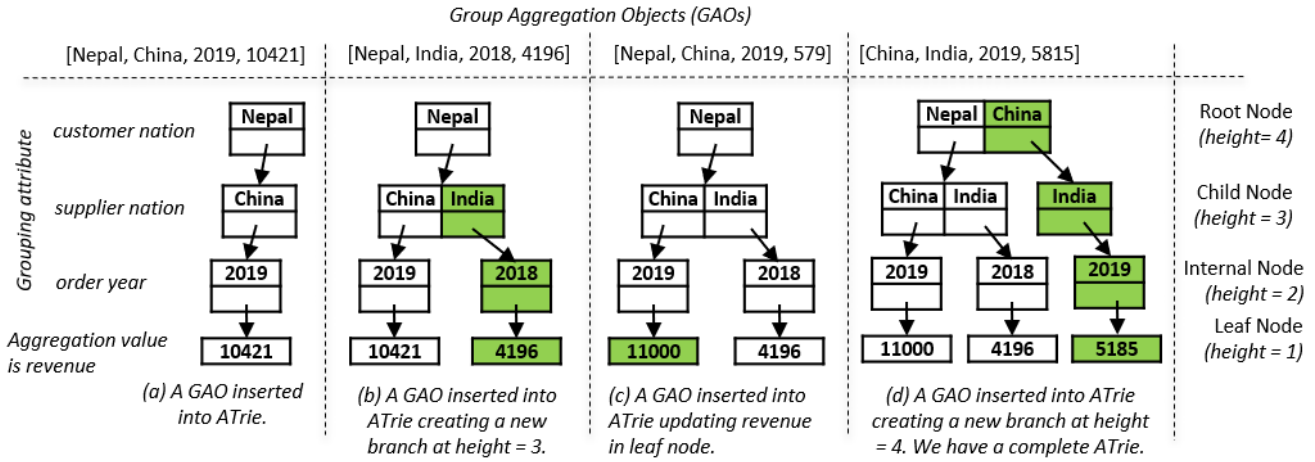


FIGURE 7. A step-wise insertion of GAOs in the ATrie. The new insertion of the group attribute or update of aggregate value has been highlighted after each insertion of a GAO.

height = 3 and check for the existence of supplier nation = "India" in the hash table that was pointed by customer nation = "Nepal". We do not find the match, therefore, we create a new entry in the hash table as supplier nation = "India". As this is a new attribute that was added, rest of the attributes will not exist. We move to height = 2 and create a new entry in the hash table as order year = 2018. At height = 1, we have reached the leaf node and we insert the aggregation value revenue = 4196.

Let us insert another GAO = ["Nepal", "China", "2019", 579] (Figure 7 (c)). At height = 4 (root node), we check for the existence of customer nation = "Nepal". Since it exists, we move to height = 3 and check for the existence of supplier nation = "India". In this example, all the grouping attributes already exist as a result of the first insertion procedure (Figure 7 (a)). Therefore, we update the aggregation value in the leaf node revenue = 10421 + 579 = 11000. The pseudocode for the insertion of data into an ATrie is given in Algorithm 2.

Merging ATries: To merge two ATries, we read the right ATrie, create a GAO and insert into the left ATrie. The worst-case time complexity for this operation is $\mathcal{O}((h * K)^2)$.

Figure 8 shows an example of the step-wise merging of GAOs into the Left ATrie. Firstly, we read the right ATrie (Figure 8 (b)) to create two GAOs: ["Nepal", "Russia", "2019", 15437] and ["Nepal", "India", "2018", 804]. Then, we insert these GAOs one-by-one into left ATrie as shown in Figure 8 (d) and Figure 8 (e) respectively. The pseudocode for merging two ATries is shown in Algorithm 3.

Deleting an attribute from the ATrie: To delete an attribute from the ATrie, we first read the ATrie. Through reading, we establish that the attribute to be deleted is present in the ATrie. The attribute to be deleted is passed to the ATrie as a GAO. As we read our way up through the corresponding K heights of the ATrie, we not only examine the attribute that is in our entry, but also ensure that there are no other attributes

Algorithm 2 Inserting Into an ATrie

```
Data: ATrie root, GAO gao
Result: complete ATrie
node <− root
height <− sizeofGAO()
value <− gao.pop(-1)
for attribute ∈ gao do
    if attribute NOT IN node.children then
        node.children.Add(attribute)
    end
    node.height = −−height
    node = node.children[attribute]
end
node.height = −−height
node.value + = value
```

at that level. If we find that there are other entries, this means that the edges are being shared by multiple attributes and we must not delete these shared paths. Note that the deletion of an attribute from the ATrie is not required for DATGJ.

5) GROUP-BY PROCESSING

We create a task local ATrie and the GAO is inserted into these ATries. Insertion proceeds by walking the ATrie according to the attributes in GAO, then appending the new node for the attribute that is not present in the ATrie. We start with the empty node (root node). Then, we insert each GAO into ATrie and build up the required branches as we move through the internal nodes in the ATrie. Leaf node holds the aggregated value. The output of this phase is a complete local ATrie with grouping attributes on its edges to guide the grouping process and aggregate values on the leaf node.

In Figure 6, each Task $t = 1 \ldots K$ work on a task local ATrie, $ATrie1, ATrie2, \ldots, ATrieK$. The GAOs created during

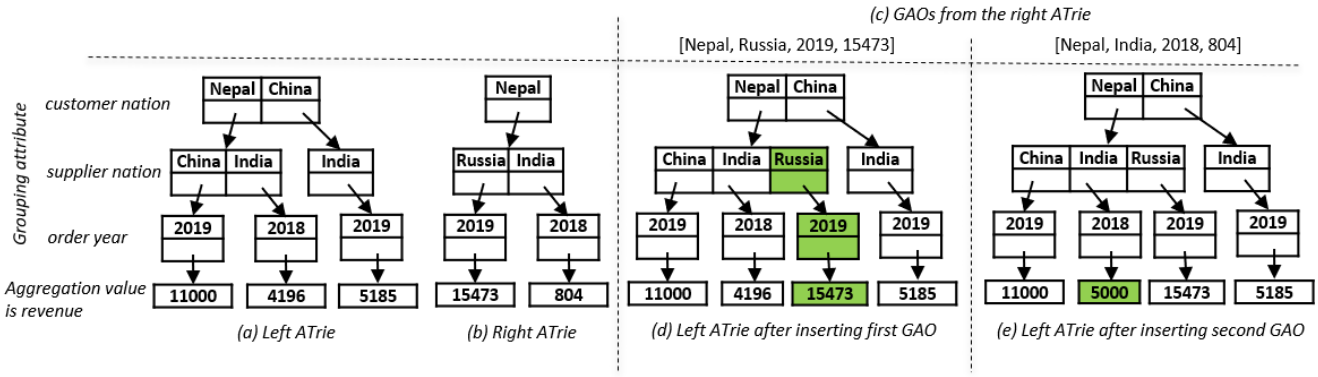


FIGURE 8. A step-wise merging of two ATries. The new insertion of the group attribute or update of aggregate value has been highlighted after each insertion of a GAO.

Algorithm 3 MergeATries

Data: ATrie atrie1, ATrie atrie2
Result: Atrie atrie1

```
atrie2 gets merged to atrie1 for Key k in atrie2.children.Keys do
    attributes <− k attributes is a global GAO
    tempAtrie <− atrie2.childern[k]
    if tempAtrie.children is NOT NULL then
        Insert (atrie1, attributes)
        attibutes.Clear()
        break
    end
    MergeATries(atrie1, tempATrie)
end
return atrie1
```

Single Scan Hash Join are inserted into these ATries to group attributes on the fly.

D. MERGE ATries

Local ATries are collected back to the master before merging them. Once collected, merging of these ATries can be done in serial or parallel depending on the number of ATries. For example, let us say we have three ATries A1, A2 and A3. Merging these ATries will require two serial mergings of (A1, A2) and then (A1, A3). If we have four ATries A1, A2, A3 and A4, we can merge (A1, A2) and (A3, A4) in parallel and then merge (A1, A3). However, we have found that the number of nodes in ATrie is fewer than the number of records in a fact table or the dimension tables. Therefore, the cost of the *Merge ATries* phase is significantly less than the *Single Scan Hash Join* and *Group-by using ATrie* phases.

VI. EXPERIMENTAL EVALUATION

In this section, we give a brief description of the environment used and present a detailed analysis of the results. We conducted all our experiments on the standalone NeCTAR[1] cluster with one master and five worker nodes running Ubuntu 18.04 LTS. Each node in the cluster is equipped with 8-core Intel Haswell (no TSX) CPUs clocked at 2.99 GHz and 32 GB of RAM. The algorithms are implemented in python with Apache Spark 2.4.0. The Apache Spark Standalone Cluster is shown in Figure 9.

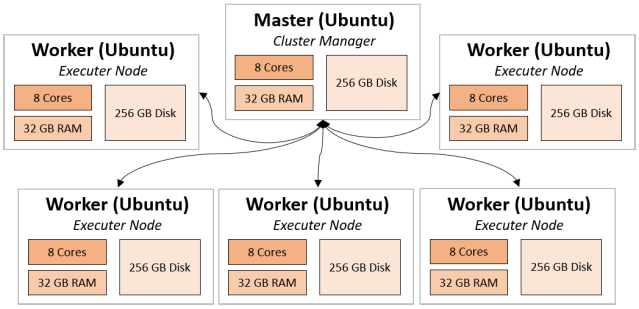


FIGURE 9. Apache Spark Standalone Cluster with one master and five worker nodes.

A. ALGORITHMS TESTED

The following distributed group-by join algorithms are evaluated in this section.

- SparkRDD (Naive): A direct Spark implementation of a sequence of joins and group by.
- Spark Bloom Filtered Cascade Join (SBFCJ) [14]: A join that processes star join queries using bloom filters, and is resilient when there is scant memory.
- Spark Broadcast Join (SBJ) [14]: A join that reduces excessive data spill and network communication and delivers better results when memory resources are abundant.
- Distributed ATrie Group Join (DATGJ): A fast distributed star join and group-by algorithm that is presented in this paper.

[1] https://www.nectar.org.au/about-nectar

B. BENCHMARK DATASET

We used the Star Schema Benchmark (SSBM) [39] for the experiment. Sanchez [43] reviewed SSBM and concluded that SSBM is a better benchmark than TPC-H that offers much simpler schema and query execution set. Query Flight 1 in SSBM does not contain queries with the group-by. Therefore, we have excluded Query Flight 1. This benchmark provides a base "Scale Factor (SF)" to scale the size of the data. Similar to [3], [7], [14], [16], [27], we use scale factors of 50, 100, 150 and 200 for the experiment. The details of the number of tuples in the fact table (i.e. `LINEORDER` table) and its disk size can be found in Table 2.

TABLE 2. Data characteristics used in the experiment showing for each scale factor (SF) the number of tuples in the fact table (#Tuples) and its disk size.

SF	#Tuples	Size(GB)
50	300 Million	30 GB
100	600 Million	60 GB
150	900 Million	90 GB
200	1.2 Billion	120 GB

C. EXPERIMENTAL RESULTS

The numbers reported here are the average of five iterations empirically determined to guarantee the mean confidence interval of $\pm$ 100s.

1) RUNTIME EFFICIENCY

We consider the elapsed time of the four aforementioned algorithms. The test was performed using 5 nodes (35 cores) cluster on the SSBM dataset SF = 200.

Figure 10 shows the results of total elapsed time broken down by the query flight; Figure 11 shows the average result for all the queries. The definition of metrics in Figure 11 can be found in Apache Spark documentation.[2] For all group-by queries in SSBM, DATGJ is 100% faster than all the competing algorithms. For all queries evaluated, on average, DATGJ is 1.5X faster than SBJ, 2X faster than SBFCJ and 6X faster than SparkRDD (Naive) algorithm.

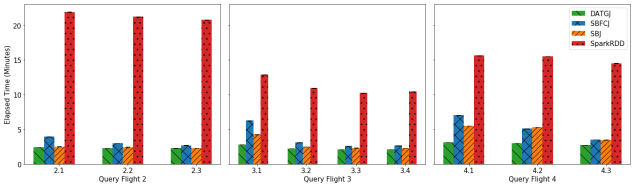


FIGURE 10. Elapsed time of all algorithms by SSBM query flights (# Worker Nodes = 5 and SF = 200).

The performance of DATGJ can be attributed to the fact that rather than constructing rows to be grouped by processing the fact data through successive series of join, DATGJ coalesces joins and applies all joins to the fact table in a single

[2]https://spark.apache.org/docs/latest/monitoring.html

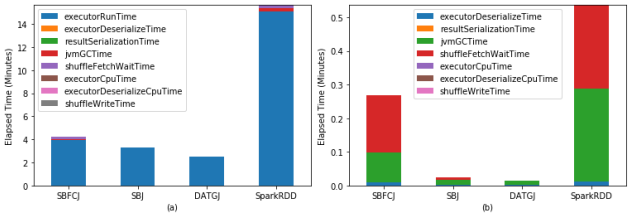


FIGURE 11. Average elapsed time of all algorithms (# Worker Nodes = 5 and SF = 200).

operation. After the probing phase in the single scan hash join stage, original join-keys are replaced with actual attribute values from the dimension table that are used both to perform group-by efficiently and aggregate rows using the ATrie and progressive materialisation.

In Figure 11 (a), it is interesting to note that *executorRunTime* accounts for a significant portion of elapsed time in all the algorithms, while other metrics are insignificant in SBJ and DATGJ. Upon further inspection, after removing *executorRunTime* (refer Figure 11 (b)), we observe the significant time taken for *jvmGCTime* and *shuffleFetchWaitTime* in SBFCJ and SparkRDD which is relatively insignificant in SBJ and DATGJ while DATGJ completely avoids the *shuffleWriteTime*.

2) NETWORK COMMUNICATION

Communication costs are determined by measuring the number of received tuples at each worker node, and the size of data shuffled. The actual and shuffled sizes of data in GB and the number of tuples for all the algorithms are shown in Table 3. *executorRunTime* includes time to fetch shuffle data [44]. Therefore, we are unable to include time to fetch shuffle data in Table 3.

TABLE 3. Actual and shuffled sizes of data in GB and # Tuples for all the algorithms. (# Worker Nodes = 5 and SF = 200).

	Data Size (GB)		# Tuples	
Algorithm	Total	Shuffled	Total	Shuffled
SparkRDD	116	68.94	1.2 Billion	389 Million
SBFCJ	116	2.09	1.2 Billion	60 Million
SBJ	116	0.13	1.2 Billion	1.2 Million
DATGJ	116	0.00	1.2 Billion	0

Excessive shuffling of records for join and group-by operations significantly increases the cost of network communication and blocks the further processing of the algorithm [14]. Table 3 demonstrates one of the main advantages of DATGJ: although SBFCJ and SBJ had significantly low data shuffle compared to SparkRDD, DATGJ show no data shuffle at all. DATGJ follows the divide and broadcast-based data partitioning method [40]. The fact table is divided into multiple disjoint partitions using random-equal partitioning technique, where each partition is allocated to a worker node, and the FHDims are broadcast all worker nodes. Each worker has one partition of fact table and a complete FHDim of the required

dimension tables. Therefore, DATGJ completely avoids the shuffling of data during the group join processing.

3) VARYING DATASET SIZE

We investigate the effect of dataset volume on the performance of all the algorithms. In general, the algorithms must be resilient and scale well with the dataset size.

Figure 12 (a) shows the linear increase of the elapsed time for SparkRDD whereas sub-linear (slow) increase of elapsed time for SBJ, SBFCJ and DATGJ while DATGJ has the least elapsed time of all the competing algorithms. We observe that SBJ and DATGJ have a similar performance when SF = 50. However, the dataset is small and does not reflect the applications in which the distributed approaches excel.

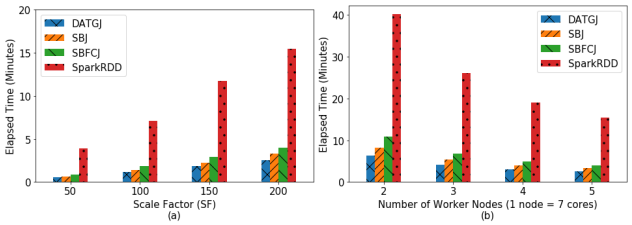


FIGURE 12. (a) Impact of Scale Factor (SF) on the performance of the algorithms (# Worker Nodes = 5). (b) Impact of the number of worker nodes on the scalability of the algorithms (SF = 200).

We can see the improved elapsed time between SBJ and DATGJ as data set size increases from SF 50 to 200. This is mainly due to the nature of attribute's insertion in ATrie (i.e. insert if not present) which enables a shorter access time, greater ease of addition of node or updating the value and greater convenience when handling a varying group of attributes. In addition, the deterministic property of ATrie avoids the dynamic reorganisation of attributes for insertion operations in ATrie, and the constant complexity in terms of the fill factor of ATrie improves elapsed time even when the dataset size increases.

4) VARYING NUMBER OF NODES

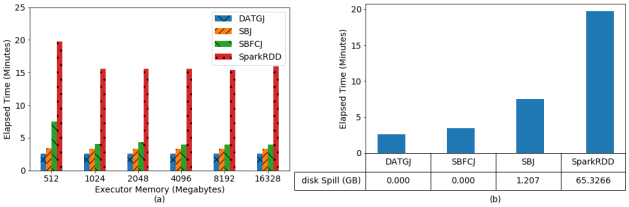
We investigate the effect of a varying number of nodes to evaluate the scalability of our DATGJ implementation by varying the number of processing cores from 14 cores (2 nodes) up to 35 cores (5 nodes).

Figure 12 (b) shows the execution time for DATGJ compared to competing algorithms for a varying number of nodes. The number of records in dimension tables is significantly less compared to the fact table. When predicate filters are applied, only a small percentage of these dimension table attributes are selected for grouping [40]. DATGJ uses ATrie to group attributes: a hash table (FHT) to track the edges and navigate through ATrie to update the aggregation value. Hashing is relatively faster in comparison to other DATGJ operations. Increasing the number of nodes involves creating more ATries in parallel which would require more steps during Merging ATries. However, Merging ATries take significantly less time than other stages in the algorithm [16]. Therefore, DATGJ still has a competitive advantage over other algorithms with additional resources.

5) CONSTRAINED MEMORY

While DATGJ has outperformed other solutions, scenarios with low memory per executor might compromise its performance. Next, we study how the memory available to each executor impacts all the algorithms for # Worker Nodes = 5 and SF = 200.

Figure 13 (a) shows that while SparkRDD and SBFCJ are affected by the constrained memory (i.e. 512 MB), the performance of SBJ and DATGJ remains unaffected. If enough memory is provided, the performance of all the algorithms remains consistent (1024 MB and above for SF = 200).



	DATGJ	SBFCJ	SBJ	SparkRDD
disk Spill (GB)	0.000	0.000	1.207	65.3266

FIGURE 13. (a) Performance of Algorithms under different memory conditions (# Worker Nodes = 5 and SF = 200). (b) Disk spill for 512 MB memory.

In the 512 MB scenario, SBFCJ and SparkRDD algorithms cause disk spills as shown in Figure 13 (b). Spilling occurs when the data-storage memory is insufficient. Generally, there are three occasions when data spilling occurs:

1) Hash Table Broadcast: Broadcast methods usually demand more memory to allocate dimension tables [14]. If the memory is insufficient, hash tables are spilled into the disk [40].
2) Data Shuffling: When the data is shuffled, the records are stored first in memory, and when memory hits some pre-defined throttle, this memory buffer is then flushed into disk [12].
3) Internal Data Structure: The internal data structures used in the algorithm might require a big memory chunk. When the memory is insufficient, data might be spilled to the disk and the current memory is cleaned for a new round of insertions [40], [45].

As discussed in Section VI-C2, DATGJ completely avoids the shuffling of data. Therefore, we do not have disk spill due to data shuffling.

Hash Table Broadcast - Except for SparkRDD, all other competing algorithms broadcast the hash tables. The broadcast size of the hash tables for some of the queries such as 4.1 and 4.2 are above the threshold of 512 MB as shown in Table 4. Therefore, the assumption is that we will incur some disk spill. However, in Spark, the driver program creates a local directory to store the data to be broadcasted and launches a `HttpBroadcast` or `TorrentBroadcast` with access to the directory. The data is actually written into the directory when the broadcast is called. At the same time, the data is also written into driver's `blockManager` with

a StorageLevel `MEMORY_AND_DISK_SER`.[3] Therefore, we do not encounter disk spill intrinsically.

Internal Data Structure - The disk spill could still occur because of the internal data structures such as ATrie. However, Table 4 shows that for SF = 200, the maximum size of ATrie is 100 MB for Query 3.2 whereas the minimum size is 0.354 MB for Query 3.4. ATrie features data compression by sharing the same grouping attributes, allowing it to use less space than it would take to list all the distinct results separately. The size of ATrie partially depends on the selectivity of the query (refer Table 1, Query 3.2 has high selectivity than 3.4) and the data type of the grouping attribute. Therefore, the observation is more likely to change depending on the query selectivity and the data type.

TABLE 4. Total size of the broadcasted hash tables and ATrie size in MB in each worker for SF = 200 and executor memory = 512 MB.

Query	Total Broadcast Size (MB)	ATrie Size / Worker (MB)
2.1	74.047	73.135
2.2	39.757	15.766
2.3	32.048	3.438
3.1	484.884	42.791
3.2	101.507	100.546
3.3	21.544	4.513
3.4	20.721	0.354
4.1	904.772	10.015
4.2	904.101	26.947
4.3	504.890	9.965

VII. ANALYTICAL EVALUATION

In this section, we introduce our modelling methodology and describe the cost model used to predict the cost of the distributed group join. We also present our model evaluation and statistical analysis in order to demonstrate the difference between the model and the experiment.

A. MODEL METHODOLOGY

To construct the cost model, the algorithm has been divided into logical steps, and each step is described by a formula based on the parameters that determine the execution time for that step. The cost model includes the following components: System Parameters and Data Parameters, Query Parameters, Time Unit and Communication Cost.

System Parameters and Data Parameters includes the number of records used to describe in-memory processing and the number of processors used to process the query. The number of processors determines the amount of information processed by each processor.

Query Parameters define the selectivity ratios. The selectivity ratio is the number of records in the query output divided by the total number of rows in the table.

Time Unit and Communication Cost are the parameters related to the technical characteristics of the system, such as time to read to/from main memory, time to hash, probe or filter a record, time to aggregate the value, cost associated with the initiation for a message transfer and the time for actual message transfer.

[3]https://spark.apache.org/docs/latest/rdd-programming-guide.html

In addition to general approach mentioned above, to construct a cost model for the algorithms in column-stores, it is necessary to take into account the specific features unique to column-stores such as forming a set of rows from individual columns.

B. COST MODELS

The parameters used to create the cost model are listed in Table 5. The symbols used in the formula: $\lceil\rceil$ is a ceiling function, $\lfloor\rfloor$ is a floor function and $\vee$ means maximum.

TABLE 5. The cost model parameters and notations.

Symbol	Description
	System and data parameters
D_i	Size of i-th dimension table column, i = 1 . . . n
$\mid D_i \mid$	Cardinality of i-th dimension table column
F_i	Size of the i-th fact table column chunk
$\mid F_i \mid$	Cardinality of the i-th fact table column chunk
H	Size of hash table
A	Size of aggregate trie
N	Number of worker nodes
P	Page size
n_d	Number of dimension tables
n_g	Number of attributes involved in the grouping
n_p	Number of processors in each worker node
	Query Parameters
σ_{d_i}	Selectivity ratio of the i-th dimension table column
σ_{f_i}	Selectivity ratio of the i-th fact table column
	Time Unit Cost
t_w	Time to write the record to the main memory
t_r	Time to read a record in the main memory
t_h	Time to hash a record
t_p	Time to probe a record
t_a	Time to aggregate
t_f	Time to filter a record
	Communication Cost
m_p	Message protocol cost per page
m_l	Message latency for one page

During the *Broadcast* phase, we read the dimension table columns from main memory and create the hash table in each worker node. Therefore, we encounter two different costs: *Scan Cost* and *Hash Cost* obtained with the following equations.

$$SC = \vee_{i=1}^{n_d} \frac{\mid D_i \mid}{N} \times (t_r + t_f) \tag{2}$$

$$Scan\ Cost = \vee_{j=1}^{N} SC_j \tag{3}$$

$$HC = \vee_{i=1}^{n_d} \frac{\mid D_i \mid}{N} \times \sigma_{d_i} \times (t_h + t_w) \tag{4}$$

$$Hash\ Cost = \vee_{j=1}^{N} HC_j \tag{5}$$

The hash tables are broadcast to all workers. Therefore, we encounter two different costs: *Hash Table Transfer Cost* and *Hash Table Receive Cost* obtained with the following equations.

$$Hash\ Table\ Transfer\ Cost = \sum_{i=1}^{n_d} \frac{\mid H_i \mid}{P} \times (m_p + m_l) \tag{6}$$

$$Hash\ Table\ Receive\ Cost = \sum_{i=1}^{n_d} \frac{\mid H_i \mid}{P} \times m_p \tag{7}$$

During the *Single Scan Hash Join* phase, we divide all the fact columns into the same number of chunks as the number of worker nodes. Each processor reads the required fact column chunks from the main memory and probes the hash table. The cost for this phase is obtained with the following equation.

$$PC = \vee_{i=1}^{n_p} \left(\frac{|F_i|}{N} \times n_d \times t_r \right) + \left(\log_2 \left(\frac{|F_i|}{N} \right) \times n_d \times t_p \right) \quad (8)$$

$$Probe\ Cost = \vee_{j=1}^{N} PC_j \quad (9)$$

During the *Group-By using ATrie* phase, we hash grouping attributes to find the path in ATrie and perform on-the-fly aggregation. The cost for this phase is obtained with the following equation.

$$CA = \vee_{i=1}^{n_p} \frac{|F_i|}{N} \times \sigma_{f_i} \times (n_g \times (t_h + t_p + t_w) + t_a) \quad (10)$$

$$Create\ ATrie\ Cost = \vee_{j=1}^{N} CA_j \quad (11)$$

The ATries are sent back to the master for merging. Therefore, we encounter two different costs: *ATrie Transfer Cost* and *ATrie Receive Cost* given by the following equations.

$$ATrie\ Transfer\ Cost = \sum_{i=1}^{n_p} \frac{|A_i|}{P} \times (m_p + m_l) \quad (12)$$

$$ATrie\ Receive\ Cost = \sum_{i=1}^{n_p} \frac{|A_i|}{P} \times m_p \quad (13)$$

During the *Merge ATries* phase, we navigate the right ATrie and insert/append attributes or aggregate value to the left ATrie. The cost for this phase is obtained with the following equation.

$$Z = \lceil \log_2(n_p) \rceil \quad (14)$$

$$t_m = (n_g \times (t_r + t_h + t_p) + t_a) + \log_{3Q}(Q) \quad (15)$$

$$Merge\ ATries\ Cost = \vee_{i=1}^{\lfloor n_p/2 \rfloor} t_{m1i} + \sum_{j=2}^{Z} \vee_{i=1}^{\lfloor n_{pj}/2 \rfloor} t_{mji} \quad (16)$$

where $n_{p_j} = \lceil n_{p_{j-1}}/2 \rceil$ and Q is the number of keys in the ATrie.

C. MODEL EVALUATION

To evaluate the cost model and determine its time prediction accuracy, we compare the model with benchmark experiment results.

Effect of Data size and Number of Nodes: Figure 14 (a) and (b) shows the comparison between the elapsed time predicted by the model and the actual time required by the experiment using varying data sizes and number of worker nodes. As shown in both figures, the estimated elapsed time from the cost model is close to the actual elapsed time from the experiment, which demonstrates the effectiveness of our cost model.

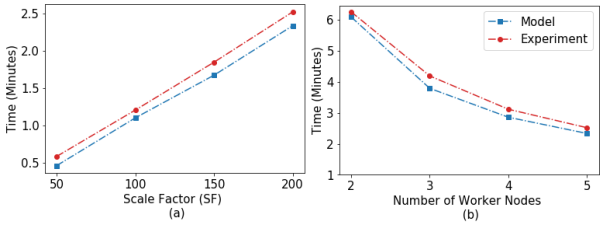


FIGURE 14. (a) Comparison of experiment result and cost model result for varying data sizes (N = 5). (b) Comparison of experiment result and cost model result for a varying number of worker nodes (SF = 200).

SSBM Queries: When evaluating our model for SSBM queries, we define the error rate as

$$error\_rate = \left| \frac{experiment\_time - model\_time}{experiment\_time} \right| \quad (17)$$

Table 6 shows the comparison between the execution time predicted by the model and execution time from the experiment for three query flights in SSBM. The estimated execution time for the cost model is close to the actual execution time obtained by the experiment in all cases, which again demonstrates the effectiveness of our cost model.

TABLE 6. Comparison of experiment results and cost model results for SSBM queries and error rate of estimated performance (N = 5, SF = 200).

Query	Model (min)	Experiment (min)	Error Rate (%)
2.1	2.276	2.433	6.421
2.2	2.216	2.323	4.565
2.3	2.171	2.281	4.812
3.1	1.870	2.027	7.713
3.2	2.123	2.25	5.618
3.3	2.047	2.117	3.298
3.4	1.990	2.093	4.915
4.1	2.874	3.13	8.172
4.2	2.840	2.996	5.206
4.3	2.611	2.743	4.804

To check whether there is a significant difference between the model's results and those obtained by the experiment, we conducted a *two-tailed t-test*. In this t-test, a sample size of 10 model values was compared with corresponding experimental values. The *p-value* obtained for the test was 0.4182, which is much larger than the significance level of 0.05. Therefore, we accept the null hypothesis, and conclude that there is no significant difference between the values of the model and those obtained by the experiment.

D. ANALYSIS

Three factors account for the difference between the estimated and the actual elapsed time:

1) The processors executing the task in parallel need to be initiated at each worker node. The initiation time of the processors varies, making it difficult to estimate the time accurately and include it in the cost model. In addition, if the actual processing time is very short, the *start-up time* may dominate the overall processing time.

2) Worker nodes use the local area network or internet to communicate with each other to send and receive the message. Communication efficiency is directly dependent on the *network latency* in real time and is very difficult to account for in the cost model.
3) Distributed processing normally starts with the breaking up of the main task into multiple sub-tasks, where each sub-task is carried out by different processors in a worker node. After these sub-tasks have been completed, it is necessary to consolidate the results produced by each sub-task. Therefore, we encounter the *consolidation cost* associated with the master node collecting results obtained from each worker node.

VIII. CONCLUSION

The main contribution of this paper is to propose a fast-distributed star group join algorithm for in-memory column-stores called Distributed ATrie Group Join (DATGJ). We improved the hash table for fastest lookup, while having fast inserts and deletes. The key idea is to use Robin Hood hashing with an upper limit for the number of probes which were implemented in the Fast Hash Table (FHT). DATGJ utilises FHT for fast single scan join and a novel technique to perform the group-by and aggregation operations using ATrie. We leveraged the technique of progressive materialization to represent grouping attributes on the edges and accumulated aggregates on the leaf nodes of ATrie. This enabled us to perform join, grouping and aggregation operations on the fly.

Experimental results show that DATGJ outperforms all the competing algorithms. For all the queries evaluated, on average, DATGJ is 1.5X to 6X faster than the competing algorithms. Furthermore, we also demonstrated that DATGJ has zero disk spills, zero data shuffle and minimal network transfer, and performs well with the addition of resources and under memory-constrained conditions. We also proposed an analytical model to understand and predict the query performance of DATGJ. Our evaluation shows that the model can predict performance with 95% confidence.

REFERENCES

[1] K. Sridhar, "Modern column stores for big data processing," in *Proc. Int. Conf. Big Data Anal.* Hyderabad, India: Springer, 2017, pp. 113–125.
[2] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, *The Design and Implementation of Modern Column-Oriented Database Systems*, vol. 5, no. 3. Washington, DC, USA: Now, 2013, doi: 10.1561/1900000024.
[3] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. Row-stores: How different are they really?" in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2008, pp. 967–980.
[4] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang, "DB2 with blu acceleration: So much more than just a column store," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1080–1091, 2013.
[5] M. Eich, P. Fender, and G. Moerkotte, "Efficient generation of query plans containing group-by, join, and groupjoin," *VLDB J.*, vol. 27, no. 5, pp. 617–641, Oct. 2018.
[6] J. Aguilar-Saborit, V. Muntés-Mulero, C. Zuzarte, and J.-L. Larriba-Pey, "Star join revisited: Performance internals for cluster architectures," *Data Knowl. Eng.*, vol. 63, no. 3, pp. 997–1015, Dec. 2007.
[7] S. Chavan, A. Hopeman, S. Lee, D. Lui, A. Mylavarapu, and E. Soylemez, "Accelerating joins and aggregations on the oracle in-memory database," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 1441–1452.
[8] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 165–178.
[10] O. Polychroniou, W. Zhang, and K. A. Ross, "Distributed joins and data placement for minimal network traffic," *ACM Trans. Database Syst. (TODS)*, vol. 43, no. 3, p. 14, 2018.
[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.*, 2012, p. 2.
[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[13] V. Purdilă and Ş.-G. Pentiuc, "Single-scan: A fast star-join query processing algorithm," *Softw., Pract. Exper.*, vol. 46, no. 3, pp. 319–339, Mar. 2016.
[14] J. J. Brito, T. Mosqueiro, R. R. Ciferri, and C. D. D. A. Ciferri, "Faster cloud star joins with reduced disk spill and network communication," *Procedia Comput. Sci.*, vol. 80, pp. 74–85, Jan. 2016.
[15] P. Sangat, M. Indrawan-Santiago, D. Taniar, and C. Messom, "Atrie group join: A parallel star group join and aggregation for in-memory column-stores," *TBD*, vol. 30, no. 1, Art. no. e5616, 2020.
[16] P. Sangat, D. Taniar, M. Indrawan-Santiago, and C. Messom, "Nimble join: A parallel star join for main memory column-stores," *Concurrency Comput., Pract. Exper.*, vol. 30, no. 1, 2019, Art. no. e4354.
[17] J. I. Munro and P. Celis, "Techniques for collision resolution in hash tables with open addressing," in *Proc. ACM Fall Joint Comput. Conf.*, Nov. 1986, pp. 601–610.
[18] P. Flajolet, P. Poblete, and A. Viola, "On the analysis of linear probing hashing," *Algorithmica*, vol. 22, no. 4, pp. 490–515, Dec. 1998.
[19] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing," in *Proc. 26th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1985, pp. 281–288.
[20] K. Sridhar, "Big data analytics using SQL: Quo vadis," in *Proc. Int. Conf. Res. Practical Issues Enterprise Inf. Syst.* Shanghai, China: Springer, 2017, pp. 143–156.
[21] P. O'Neil and G. Graefe, "Multi-table joins through bitmapped join indices," *ACM SIGMOD Rec.*, vol. 24, no. 3, pp. 8–11, Sep. 1995.
[22] V. Markl, F. Ramsak, and R. Bayer, "Improving OLAP performance by multidimensional hierarchical clustering," in *Proc. IDEAS Int. Database Eng. Appl. Symp.*, Aug. 1999, pp. 165–177.
[23] A. Weininger, "Efficient execution of joins in a star schema," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2002, pp. 542–545.
[24] C. A. Galindo-Legaria, T. Grabs, S. Gukal, S. Herbert, A. Surna, S. Wang, W. Yu, P. Zabback, and S. Zhang, "Optimizing star join queries for data warehousing in microsoft SQL server," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 1190–1199.
[25] Z. Fang, Z. He, J. Chu, and C. Weng, "SIMD accelerates the probe phase of star joins in main memory databases," in *Int. Conf. Database Syst. Adv. Appl.* Chiang Mai, Thailand: Springer, 2019, pp. 476–480.
[26] J. Aguilar-Saborit, V. Muntés-Mulero, C. Zuzarte, and J.-L. Larriba-Pey, "Ad hoc star join query processing in cluster architectures," in *Proc. Int. Conf. Data Warehousing Knowl. Discovery*. Copenhagen, Denmark: Springer, 2005, pp. 200–209.
[27] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on GPU devices," *Proc. VLDB Endowment*, vol. 6, no. 10, pp. 817–828, Aug. 2013.
[28] Z. Guoliang and W. Guilan, "GBFSJ: Bloom filter star join algorithms on GPUs," in *Proc. 12th Int. Conf. Fuzzy Syst. Knowl. Discovery (FSKD)*, Aug. 2015, pp. 2427–2431.
[29] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe, "Memory-efficient hash joins," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 353–364, Dec. 2014.

[30] N. Askitis, "Fast and compact hash tables for integer keys," in *Proc. Thirty-Second Australas. Conf. Comput. Science*, vol. 91, Jan. 2009, pp. 113–122.
[31] A. Datta, D. VanderMeer, and K. Ramamritham, "Parallel star Join+DataIndexes: Efficient query processing in data warehouses and OLAP," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 6, pp. 1299–1316, Nov. 2002.
[32] H. Han, H. Jung, H. Eom, and H. Y. Yeom, "Scatter-gather-merge: An efficient star-join query processing algorithm for data-parallel frameworks," *Cluster Comput.*, vol. 14, no. 2, pp. 183–197, Jun. 2011.
[33] C. Zhang, L. Wu, and J. Li, "Efficient processing distributed joins with bloomfilter using mapreduce," *Int. J. Grid Distrib. Comput.*, vol. 6, no. 3, pp. 43–58, 2013.
[34] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[35] Y. Ramdane, N. Kabachi, O. Boussaid, and F. Bentayeb, "Skipsjoin: A new physical design for distributed big data warehouses in hadoop," in *Proc. Int. Conf. Conceptual Modeling*. Salvador, Brazil: Springer, 2019, pp. 255–263.
[36] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MaPreduce," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2010, pp. 975–986.
[37] H. Zhu, M. Zhou, F. Xia, and A. Zhou, "Efficient star join for column-oriented data store in the MapReduce environment," in *Proc. 8th Web Inf. Syst. Appl. Conf.*, Oct. 2011, pp. 13–18.
[38] G. Zhou, Y. Zhu, and G. Wang, "Cache conscious star-join in MapReduce environments," in *Proc. 2nd Int. Workshop Cloud Intell. Cloud-I*, 2013, pp. 1–7.
[39] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*. Lyon, France: Springer, 2009, pp. 237–252.
[40] D. Taniar, C. H. Leung, W. Rahayu, and S. Goel, *High Performance Parallel Database Processing and Grid Databases*, vol. 67. Hoboken, NJ, USA: Wiley, 2008.
[41] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2014, pp. 743–754.
[42] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling up concurrent main-memory column-store scans: Towards adaptive NUMA-aware data and task placement," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1442–1453, Aug. 2015.
[43] J. Sanchez, "A review of star schema benchmark," 2016, *arXiv:1606.00295*. [Online]. Available: http://arxiv.org/abs/1606.00295
[44] *Monitoring and Instrumentation*. Accessed: Mar. 20, 2019. [Online]. Available: https://spark.apache.org/docs/latest/monitoring.html
[45] F. Kastrati and G. Moerkotte, "Optimization of conjunctive predicates for main memory column stores," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1125–1136, Aug. 2016.

PRAJWOL SANGAT received the Masters of Information Technology by Research (MIT) degree from Monash University, Melbourne, VIC, Australia, in 2015, where he is currently pursuing the Ph.D. degree in computer science with the Caulfield School of Information Technology. His research interests include distributed and parallel database systems, main-memory column-stores, query processing, and cost models.

DAVID TANIAR received the B.Sc., M.Sc., and Ph.D. degrees in computer science, specialising in databases. He is currently an Associate Professor with the Faculty of Information Technology, Monash University. His research interests include parallel database and spatial/mobile query processing. He has published extensively in these areas, including a book in *High Performance Parallel Database Processing* (Wiley, 2008). He is the Founding Editor in-Chief of two SCIE journals, such as the *International Journal of Data Warehousing and Mining* and the *International Journal of Web and Grid Services*.

CHRISTOPHER MESSOM received the M.Sc. and Ph.D. degrees in computer science from Loughborough University, Loughborough, U.K., in 1992 and 1989, respectively. He was a Lecturer with Singapore Polytechnic, from 1993 to 1997, a Senior Lecturer with the Dubai University College, UAE, from 1998 to 1999, and a Senior Lecturer and the Director of the Centre for Parallel Computing, Massey University, Auckland, New Zealand, from 1999 to 2008. He was the Head of the School of IT and the Deputy President (Academic) of Monash University Malaysia, and the Deputy Head of School/Campus with the Faculty of IT at Caulfield, Monash University, from 2009 to 2017. More recently, he has been the Director of Graduate Programmes at the Faculty and has chaired the Graduate Programs Committee and the Undergraduate Programs Committee. He is currently an Academic Workforce Manager with the Faculty of IT, Monash University. He is the author of more than 120 research articles in various journals and conference proceedings. His research interests include intelligent systems, data mining, and high performance computing.