

ATrie Group Join: A Parallel Star Group Join and Aggregation for In-Memory Column-Stores

Prajwol Sangat¹, David Taniar, and Christopher Messom

Abstract—This article presents a new holistic and efficient approach to big data analysis. We introduce a new parallel algorithm, known as *ATrie Group Join (ATGJ)*, that integrates join, grouping and aggregation operations to accelerate big data analytical workloads in in-memory column-stores. ATGJ performs a single scan of the fact columns and uses a mixture of data and task parallelism for the optimal use of computing resources. It uses a novel technique to perform group-by and aggregation realising the grouping attributes as a tree shaped deterministic finite automation known as *Aggregate Trie* or *ATrie*. ATrie facilitates grouping of attributes and processing of data in tight loops that significantly improves the performance on modern hardware. Unlike other competing algorithms, use of ATrie avoids the creation of multiple data structures with the increasing number of dimension tables and grouping attributes. Also, we demonstrate that ATGJ performs efficiently even when the ATrie becomes bushy. We evaluated the algorithm using Star Schema Benchmark (SSBM) to show that it is significantly faster and scales better than other algorithms for the number of concurrent threads, the number of group-by attributes, the data set size and the query complexity.

Index Terms—Big data processing, column-stores, parallel group-by, parallel-join, parallel processing

1 INTRODUCTION

BIG Data and its analysis are the decision making drivers for success of the business and companies. Business analytics queries routinely perform scans, predicate evaluation, joins, grouping and aggregation. A study of customer queries in DB2 has found that the group-by constructs occur in a large fraction of the analytical queries [1], [2]. The queries in the benchmarks such as TPC-H and the Star Schema Benchmark (SSBM) [3] spend more than 50 percent CPU time in joins, group-by and aggregation [4]. These queries often aggregate large portions of the data, which can lead to performance issues with very large data sets.

Column-stores have been proposed with a novel layout of data storage where information about a logical entity is stored as separate columns in multiple locations on the disk or the main memory [5]. They are efficient for business analytics queries as they can read only those columns from the disk or main memory that are accessed by the query [5], [6], [7], [8]. However, there are three important questions that arise. First, “Can we analyse structured Big Data using Column-Stores?”. Researches conducted on the use of modern column-stores for big data processing [9], [10] have concluded that parallel column-stores and fine-tuned algorithms can solve big data analytics problems.

Second, “How can we achieve fast processing for Big data in Column-Stores?”. Growing main memory capacity has fueled the development of in-memory big data management

and processing [11], [12]. Most column-stores are being tailored to run as main memory database systems which avoid the latency associated with secondary storage such as Hyper [13], SAP HANA [14], IBM DB2 with BLU Acceleration [1], Microsoft SQL Server [15], [16] and Oracle In-Memory Database [4], [17]. ATGJ target those in-memory databases to process business analytics queries more efficiently. Therefore, with in-memory column stores, faster processing of big data is possible.

Third and the most important question is, “Given a novel storage layout of column-stores, when and how should the aggregate columns be processed with respect to the join and grouping columns in business analytics queries?” The traditional approach is “Hash Join/Hash Group-By” where we perform column-oriented hash join (using bloom filters, hash table build and probe) and row-oriented hash group-by (using serial aggregation) [8], [18]. An unconventional approach is “In-Memory Aggregation” where group-by expressions are pushed down into the scans of dimension tables [4]. They create a unique key per distinct group called *Dense Grouping Key (DGK)* and map the join keys to DGK’s using *Key Vector*. Key vector is used to filter non-matching rows while performing fact table scans where the aggregation result is stored in a multidimensional array known as *In-memory Accumulator (IMA)*.

Although the unconventional approach has drastically reduced the execution time than the traditional approach, we have found three problems in the algorithm. First, the algorithm reads the value of a single row, one column after another, computes the aggregates, and stores them in their respective position in IMA. This approach is known to have a disadvantage that the data is not processed in tight loops, which results in considerable performance deterioration on modern hardware [19]. Second, this algorithm is only efficient provided that the IMA does not grow too large [4]. Third, with the increasing number of dimensions and grouping

• The authors are with the Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia. E-mail: {prajwol.sangat, david.taniar, christopher.messom}@monash.edu.

Manuscript received 12 Aug. 2019; revised 10 May 2020; accepted 22 June 2020. Date of publication 24 June 2020; date of current version 9 July 2022.

(Corresponding author: Prajwol Sangat.)

Recommended for acceptance by D. Puthal.

Digital Object Identifier no. 10.1109/TBDA.2020.3004520

attributes, this algorithm creates additional *key vectors* and *temporary tables* to process the join, grouping and aggregation which results in a significant increase in the execution time.

In this paper, we present *ATrie Group Join (ATGJ)*, a variant of Hash Join/Hash Group-By algorithm with a novel grouping technique and parallel processing design that fully integrate join, grouping and aggregation to accelerate big data analytical workloads. We propose a tree-based data structure known as the aggregate trie or ATrie influenced from a “trie” or “prefix tree” that facilitates the grouping of attributes and processing of data in tight loops. ATGJ performs a single scan of the fact columns and uses a mixture of data and task parallelism for optimal use of computing resources. It avoids the creation of multiple data structures with the increase in the number of dimension tables and grouping attributes. It performs efficiently even when the ATrie becomes bushy, as shown in Fig. 9, unlike In-Memory Aggregation where the performance degrades if IMA grows too large [4].

Our approach to optimising group join and aggregation was evaluated against the SSBM benchmark. The performance results show that our approach can reduce the total execution time by 2X to 6X. Furthermore, we also show that our approach scales better than other algorithms for the number of concurrent threads, the number of group-by attributes, the data set size and the query complexity. In summary, we make the following contributions in this paper:

- 1) We propose a novel approach to perform group-by and aggregation operations influenced by the concept of “trie” or “prefix tree” [20] using a data structure called *Aggregate Trie* or *ATrie*. The grouping and aggregation can be seen as a tree shaped deterministic finite automation. To the best of our knowledge, there is no such technique proposed for column-stores.
- 2) We propose a new parallel group join and aggregation for in-memory column-stores known as *ATrie Group Join* that is significantly faster than its competing column-store join algorithm. ATGJ is a variation of Hash Join/Hash Group-By that uses both of these techniques but solves the problem of grouping and aggregating data using a rather novel approach with the help of ATrie.
- 3) We propose an *analytical model* to understand and predict the query performance of ATGJ. The model accuracy has been verified by detailed experiments with different hardware parameters.

The rest of the paper is organised as follows: First, we describe the related works in Section 2. Then, we outline our proposed grouping method in Section 3. After that, we propose our new algorithm in Section 4 and experimental evaluation in Section 5. Finally, we present our conclusion in Section 6. We have included our analytical evaluation in the Appendix, which can be found on the Computer Society Digital Library at <http://doi.ieeecomputersociety.org/10.1109/TBDATA.2020.3004520>.

2 RELATED WORKS

In this section, we present the existing work in two different areas of research: Parallel Hash Joins and Parallel Star Joins.

2.1 Parallel Hash Joins

A significant number of research studies on column joins conclude that hash join is more efficient than sort-merge join. The ground breaking work of MonetDB [21] lead to a radix hash join that aimed at overcoming the bottleneck of cache and Translation Lookaside Buffer (TLB) misses. Kim *et al.* [22] postulated the radix hash join [21] for parallel processing based on repeatedly partitioning the input relations. They optimised hash join and sort-merge join for modern multi-core systems and showed that sort-merge joins would become faster with wider SIMD instructions and limited per-core bandwidth.

Blanas *et al.* [23] introduced no-partitioning hash join that offers superior performance to the partitioning join. They compared their algorithm with the radix hash join [22] and demonstrated that no-partitioning hash join outperformed all partition-based hash joins for almost all data distributions and is only slightly slower than parallel radix hash joins with uniform datasets.

Albutiu *et al.* [24] advocated a sort-merge algorithm for the main memory parallel joins, while others, such as Balkesen *et al.* [25], [26] directly compared heavily optimised versions of the sort-merge and hash join algorithms and concluded that hash joins still held a competitive advantage in the main memory scenario.

Balkesen *et al.* [25], [26] achieved further improvements on the radix hash join [22] and the no-partitioning hash join [23] by improving the cache efficiency of the hash table implementation and adopting a better skew handling mechanism for parallel radix hash join [22]. They showed that tuned parallel radix hash joins exhibits superior performance than the no-partitioning hash joins on their experimental hardware which contradicts [23].

Barber [27] proposed a memory efficient Concise Hash Table (CHT) and a Concise Array Table (CAT). They compared their join method with the no-partitioning hash join [23], and the parallel radix hash join [25]. They showed that they could reduce the memory usages by one to three orders of magnitude with competitive performance.

Jha *et al.* [28] compared the modified variants of the no-partitioning hash join [23] and the parallel radix hash join [22] on the Intel Xeon Phi co-processors. They demonstrated that under a wider range of parameters, no-partitioning hash joins can match and even outperform parallel radix hash joins and suggested that the main memory hash joins need to be revisited as processor technology changes.

Given the overall high-performance of hash joins and our ability to utilise the nature of hash tables in storing grouping attribute values, we have based ATGJ on the hash join algorithms.

2.2 Parallel Star Joins

We start by briefly discussing the Star Schema Benchmark and then with the help of Query 3.1 in SSBM, we discuss three major join algorithms proposed in the column-stores. We devote a large section to these works since they form an important foundation of the join algorithm proposed in this paper.

2.2.1 Star Schema Benchmark (SSBM)

Star Schema Benchmark [29] has been widely used in data warehousing research studies [4], [8], [30], [31]. Sanchez [32]

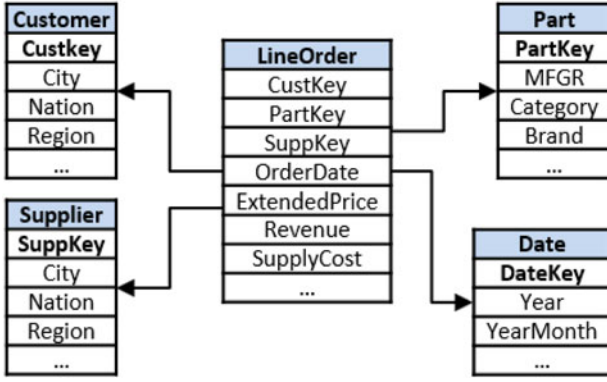


Fig. 1. Schema of SSBM.

reviewed SSBM and concluded that SSBM is a better benchmark than TPC-H that offers a much simpler schema and query execution set. It consists of a single fact table `LINEORDER` table that has foreign key references to four dimension tables `CUSTOMER`, `SUPPLIER`, `PART` and `DATE` table as shown in Fig. 1. A fact table is the central table in a star schema that typically has two types of columns: those that are the foreign keys to dimension tables such as `CustKey` in `LINEORDER` table and those that store quantitative information for analysis such as `Revenue` in `LINEORDER` table. Dimension tables are companion tables to a fact table that contains descriptive attributes and serve two critical purposes: query filtering and query result set labelling.

The SSBM consists of thirteen queries divided into four flights:

Flight 1 consist 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 consist 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 consist 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.

Flight 4 consist 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.

Let us consider the Query 3.1 (shown below) as a running example to describe the phases in the 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;
```

This query finds the revenue volume for the line order transactions by customer nation, supplier nation and year within a region 'Asia' in the period of 1992 and 1997.

2.2.2 Row-Store Joins

A number of approaches developed are based on the row-oriented storage architecture and mainly focus on joins rather than grouping and aggregation. O'Neil *et al.* [33] 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.* [34] 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. Aguilar *et al.* [35] proposed a star hash join based on the use of bloom filters in cluster architectures to reduce both I/O and data traffic communication.

Datta *et al.* [36] proposed new parallel star join algorithms based on the vertical partitioning of data in the distributed environment. Weininger [37] proposed index union and semi-join reduction plans with bitmap filters for efficient execution of star schema joins. Aguilar *et al.* [3] revisited the star join techniques to analyse the most up-to-date strategy for ad-hoc star join query processing in a cluster of computers. They proposed the hybrid solution that improved the aspects of the bitmap star join [33] and the hierarchical physical clustering [34] and showed near-optimal results in multiple use-cases. Galindo *et al.* [38] 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.

All of these works have been motivational approaches towards developing star join algorithms for column-stores. ATGJ is inspired from [33], [35], [38] but it has been designed to address join, group and aggregation queries for column-stores.

2.2.3 Column-Store Joins

Limited research exists in the field of parallel star joins for column-stores and they have a number of limitations. We next discuss these algorithms and their limitations.

1) *Invisible Join* [8]: Invisible join is an extended work on improving the performance for star joins [33], [37] by taking advantage of the column-oriented layout and rewriting the predicates to avoid the hash lookups. We next briefly describe the Invisible Join phases.

Phase 1: Predicates `c.region = 'ASIA'`, `s.region = 'ASIA'` and `d.year BETWEEN 1992 and 1997` are applied on dimension tables `CUSTOMER`, `SUPPLIER` and `DATE` to build the intermediate hash tables.

Phase 2: `LINEORDER` table columns are joined with the respective dimension table using an intermediate hash table. The result of the join operation are the lists of positions that passed the join predicates (`c.region = 'ASIA'`, `s.region = 'ASIA'` and `d.year BETWEEN 1992 and 1997`). At the end of this phase, we have three intermediate lists of positions one each for `CUSTOMER`, `SUPPLIER` and `DATE` dimension tables that are intersected to generate the final list of positions satisfying all the predicates.

Phase 3: According to the final list of positions, required fact table columns are *re-scanned* to construct the final result. `lo.revenue` is scanned whereas `lo.custkey`, `lo.supkey` and `lo.orderdate` are re-scanned.

We have identified three problems with the algorithm: i) It has the performance bottleneck of multi-pass scan for column processing and increased memory consumption with the increasing number of tables in the join query. ii) It incurs a significant memory overhead cost because it creates multiple intermediate position lists. iii) It follows a task parallel approach resulting in sub-optimal use of resources.

ATGJ, on the other hand, performs a single scan of the fact columns and uses a mixture of data and task parallelism for optimal use of computing resources.

2) *Nimble Join* [18]: Nimble join is a progressive parallel join algorithm for column-stores and their multi-attribute array table (MAAT) is based on the concept of Concise Array Table [27]. 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) [39]. This configuration permits the good use of CPU cache and hardware data prefetch while simultaneously saving memory space. We next briefly describe Nimble Join phases.

Phase 1: Predicates `c.region = 'ASIA'`, `s.region = 'ASIA'` and `d.year BETWEEN 1992 and 1997` are applied on dimension tables CUSTOMER, SUPPLIER and DATE to build the intermediate hash tables.

Phase 2: Each foreign key column of LINEORDER table is joined with the respective dimension table using intermediate hash table and inserted into MAAT. This process continues for all the FK columns of the LINEORDER table involved in the query. The probe sequence for all the FK columns is determined by sorting the join selectivity and performing the least selective join first. As probes are performed in MAAT, the size of the MAAT reduces as the items that do not satisfy the predicate are removed. The position of items satisfying the predicate is stored in the position list of MAAT.

Phase 3: For all the positions in the position list of MAAT, the *position* is used as an index of an array to perform the position look up and direct value extraction from the columns required in the query (e.g., `lo.revenue`).

MAAT holds the intermediate attributes required for the join query processing. It eliminates the problem of re-scanning of fact table columns performed by invisible join [8]. However, this joining technique is not optimised for group-by and aggregation operation whereas ATGJ focuses on improving group-by and aggregation operation using ATrie.

3) *In-Memory Aggregation* [4]: In-Memory Aggregation optimised aggregations over joins for star schemas 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. We next briefly describe the In-Memory Aggregation phases.

Phase 1: CUSTOMER, SUPPLIER and DATE dimensions are scanned and a new data structure called a *Key Vector* is created. A key vector maps a qualifying dimension key to a dense grouping key. For example, each customer from region 'ASIA' will have non-zero grouping key numbered 1..M, each supplier from region 'ASIA' will have a non-zero

grouping key numbered 1..N and each year between 1992 and 1997 will have a non-zero grouping key numbered 1..O.

Phase 2: The key vectors are used to create an additional multi-dimensional array known as *In-Memory Accumulator*. Each dimension of IMA will have as many entries as the number of non-zero grouping keys corresponding to that dimension; in the above example, it will be an $(M \times N \times O)$ array.

Phase 3: For each entry in the LINEORDER table that matches the join conditions based on the key vector entries for the dimension values, the corresponding `lo.revenue` amount is added to the appropriate cell in the IMA. In the end, the IMA has the results of the aggregation. Processing row-by-row is a disadvantage as the data is not processed in tight loops, which results in considerable performance deterioration on modern hardware [19].

This algorithm is only efficient provided that the IMA does not grow too large [4]. In addition, with the increasing number of dimensions and grouping attributes, this algorithm creates additional *key vectors* and *temporary tables* to process the group join with a significantly increased execution time, as shown in Fig. 9.

ATGJ, on the other hand, uses ATrie to facilitate grouping and processing the data in tight loops. It avoids creation of additional data structures with the increase of group-by attributes as well as performs efficiently even when ATrie becomes bushy unlike [4] where the performance degrades if IMA grows too large.

3 AGGREGATE TRIE: PROPOSED GROUPING METHOD

Aggregate Trie is a highly efficient data structure for grouping and aggregation operation. The idea of the aggregate trie is based on a "trie" or "prefix tree" [20] and it facilitates faster grouping and aggregation of the attributes than other existing techniques such as vector group-by [4], [40]. In this section, we focus on the terminologies used, formal definition of the aggregate trie and the core operational concepts.

3.1 Terminologies

An *Aggregate Trie* (a.k.a. *ATrie*) is a collection of grouping attributes 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 the ATrie that has both *parent* and *child nodes*. Similarly, the bottom most node in the ATrie that does not have child node is called *leaf node*. The *height* of an ATrie is the height of root node.

3.2 Formal Definition

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

Definition 3.1 (Group Aggregation Object (GAO)). *Group Aggregation Object* is a list data structure that represents a set of grouping attributes. It includes the aggregation attribute at the end. Formally, the semantics of the GAO is defined as follows:

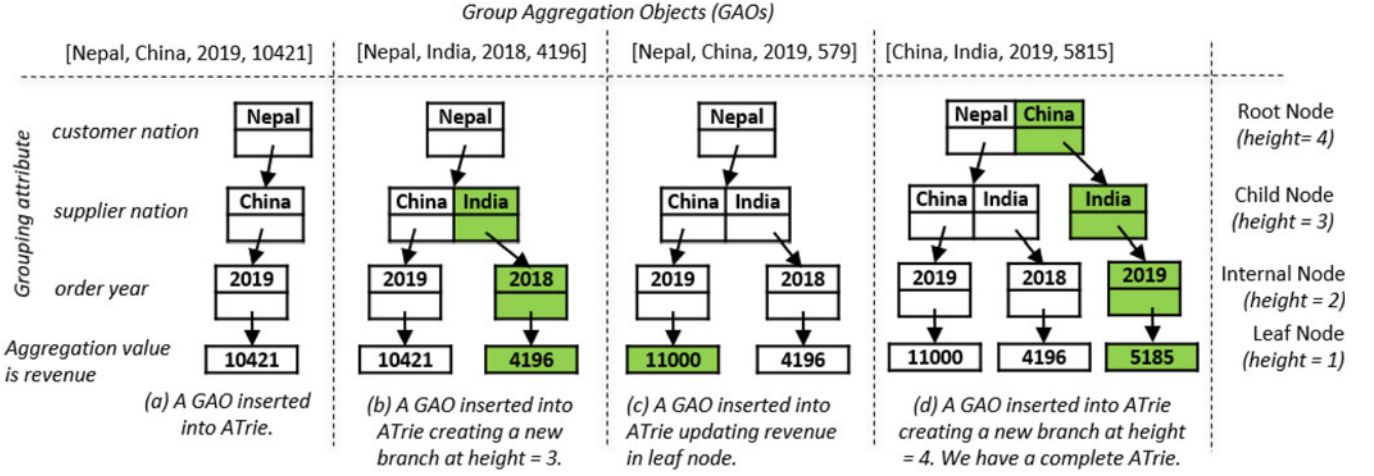


Fig. 2. A step-wise insertion of GAOs in the ATrie. The new insertion of the group attribute or update of aggregate value has been highlighted after each insertion of a GAO.

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

Lets 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 3.2 (Aggregate Trie). The aggregate trie (a.k.a ATrie) is a deterministic tree of GAOs with height $h = \text{sizeOfGAO}()$. The root node in ATrie describes level h , while nodes at level 1 are the leaf nodes and holds 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, the ATrie has the 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:** 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 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 Materialization:** ATGJ extends the idea of progressive materialization [18] by using ATrie as a vessel to perform materialization and aggregation on the fly while 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 [18].

3.3 Physical Data Structure

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

3.4 ATrie Operations

Fig. 2 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 $K = \text{maximum number of the distinct attributes in all the level of the ATrie}$.

3.4.1 Reading or Searching ATrie

To read or search ATrie, we follow the path designated by addresses advancing to indicated height of the ATrie each time we move to a new grouping attribute. At each height, we examine for the existence of the 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 of this operation is $\mathcal{O}(h * K)$.

Lets try to read or search attributes in a $GAO = ["Nepal", "China", "2019", 11000]$ in a complete ATrie (refer Fig. 2d). 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 are no pointers to the hash table, therefore, we read the aggregated value revenue = 11000.

3.4.2 Insert a GAO into ATrie

To insert a GAO into 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

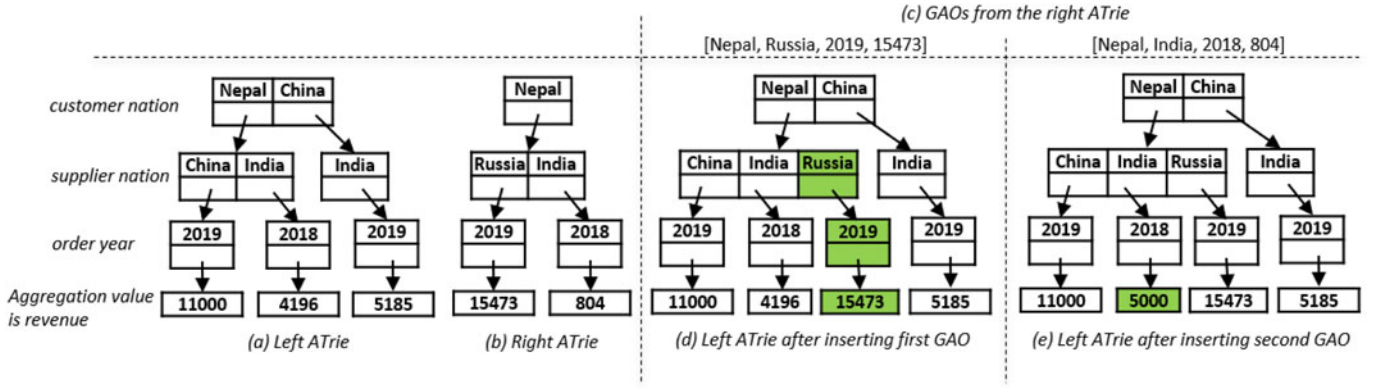


Fig. 3. A step-wise merging of two ATries. The new insertion of the group attribute or update of aggregate value has been highlighted after each insertion of a GAO.

allows for shorter access time, greater ease of addition of nodes or updating the values, and 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 of this operation is $\mathcal{O}(h + h * K) \approx \mathcal{O}(h * K)$.

Fig. 2 shows an example of the step-wise insertion of GAOs into the ATrie. We discuss two examples of insertion shown in Figs. 2b and 2c. Lets try to insert a GAO = ["Nepal", "India", "2018", 4196] (Fig. 2b). 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 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.

Lets try to insert another GAO = ["Nepal", "China", "2019", 579] (Fig. 2c). 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 this example, all the grouping attributes already exist as a result of first insertion procedure (Fig. 2a). Therefore, we update the aggregation value in the leaf node revenue = 10421 + 579 = 11000.

The pseudocode to insert data into an ATrie is given in Algorithm 1.

3.4.3 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 of this operation is $\mathcal{O}((h * K)^2)$.

Fig. 3 shows an example of the step-wise merging of GAOs into the Left ATrie. First, we read the right ATrie (Fig. 3b) 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 Fig. 3d and Fig. 3e respectively.

When we insert GAO = ["Nepal", "India", "2018", 804] into the left ATrie (refer Fig. 3e), 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 (refer Fig. 3a). Therefore, we update the aggregation value in the leaf node revenue = 4196 + 804 = 5000.

Algorithm 1. Inserting into an ATrie

```

Data: ATrie root, GAO gao
Result: complete ATrie
1: node < - root
2: height < - sizeofGAO()
3: value < - gao.pop(-1)
4: for attribute  $\in$  gao do
5:   if attribute NOT IN node.children then
6:     node.children.Add(attribute)
7:   end
8: node.height = --height
9: node = node.children[attribute]
10: end
11: node.height = --height
12: node.value += value

```

The pseudocode for merging two ATries is shown in Algorithm 2.

Algorithm 2. MergeATries

```

Data: ATrie atrie1, ATrie atrie2
Result: ATrie atrie1
/* atrie2 gets merged to atrie1 */
1: for Key k in atrie2.children.Keys do
2:   attributes < - k /* attributes is a global GAO */
3:   tempAtrie < - atrie2.children[k]
4:   if tempAtrie.children is NOT NULL then
5:     Insert (atrie1, attributes)
6:     attributes.Clear()
7:   break
8: end
9: MergeATries(atrie1, tempAtrie)
10: end
11: return atrie1

```

3.4.4 Deleting an Attribute From ATrie

To delete an attribute from ATrie, we first read the ATrie. Through reading, we establish that the attribute to be deleted

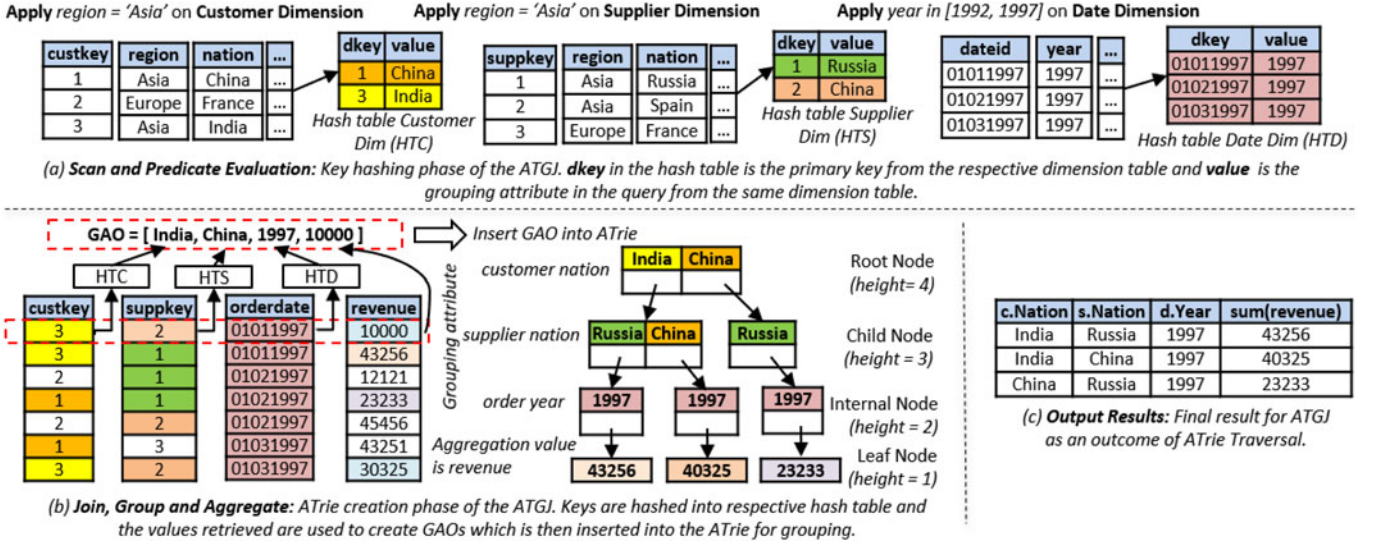


Fig. 4. Stages of ATGJ to execute Query 3.1 from SSBM on some sample data.

is actually 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 examine not only the attribute in our entry but also make sure that there are no other attributes in that level. If we find that there are other entries, it means that the edges have been shared by multiple attributes and we must not delete these shared paths. Note that deleting attribute from the ATrie is not required for the ATGJ.

4 ATrie GROUP JOIN (ATGJ): JOIN QUERY PROCESSING

We start by giving an overview of ATrie Group Join, describing the key stages. Later in this section, we describe each stage in more detail. The description of ATGJ presented in this section is for a single-threaded implementation. We extend ATGJ into multi-threaded group join algorithm in Section 4.4.

ATrie Group Join is the variation of Hash Join/Hash Group-By that efficiently groups and aggregates the data by realising the grouping attributes as a tree shaped deterministic finite automation. This unique approach is inspired by the concept of "trie" or "prefix tree" [20]. In the first phase, predicates are applied to the appropriate dimension table columns to create hash tables. In the second phase, foreign key columns in the fact table are scanned and probed into the hash table and qualifying rows are inserted into ATrie. The insert operation will either add the new nodes to ATrie or update the aggregation value if all the nodes exist. ATrie is an important facet in ATGJ to perform grouping and aggregation on the fly while scanning the fact columns. We next explain in detail the working of ATGJ using Query 3.1 from SSBM.

4.1 Scan and Predicate Evaluation

Hash - Predicates are applied on the appropriate dimension tables (CUSTOMER, SUPPLIER and DATE) to create the respective hash tables. The primary key of the dimension tables acts as the key in the hash table (i.e., dkey) and the grouping attributes in the query from the same dimension

table act as value. The output of this phase is the hash tables that help to efficiently prune out non-qualifying rows. An example of the execution of this phase on a sample data is displayed in Fig. 4a.

4.2 Join, Group and Aggregate

Probe and Create ATrie - Foreign key columns of the fact table (LINEORDER) are accessed row-by-row and the keys are probed into the corresponding hash table to create GAOs. For simplicity, the order of the grouping attributes while creating a GAO is decided based on the query grouping requirement (for example, in Query 3.1, we group by based on customer nation then by supplier nation and finally by order year). Changing the order of the attributes does not affect the performance of the ATrie as shown in Fig. 10. The GAOs are then inserted into the ATrie. Insertion proceeds by walking the ATrie according to the attributes in GAO, then appending the new node for the attribute that is not contained in the ATrie. We start with the root node and insert each GAO into the ATrie to build up the required branches as we move through the internal nodes of the ATrie. Leaf node holds the aggregated value.

Fig. 4b shows the creation of the ATrie. For the first row in the LINEORDER, customer key = 3, supplier key = 2 and order date = 01011997. These keys are probed into the respective hash tables to get a GAO = ["India", "China", "1997", 10000]. These attributes will act as edges to guide the grouping. In the root node, it searches for attribute "India". Since it does not exist, it adds the attribute to the node. As this is a new attribute added to the node, rest of the attributes will not exist. We create a new node that has a pointer to attribute "India" and add the attribute "China" to the node. After that, we create another node that has a pointer to attribute "China" and add the attribute "1997". Finally, we add the 10000 revenue = 10000 in the leaf node that has a pointer from attribute "1997".

Following the same procedure ATGJ accesses row-by-row, probes the keys into the corresponding hash table to create GAO that are inserted into the ATrie for grouping and aggregation. Now let's look at the last row. We have customer key = 3, supplier key = 2 and order date = 01031997.

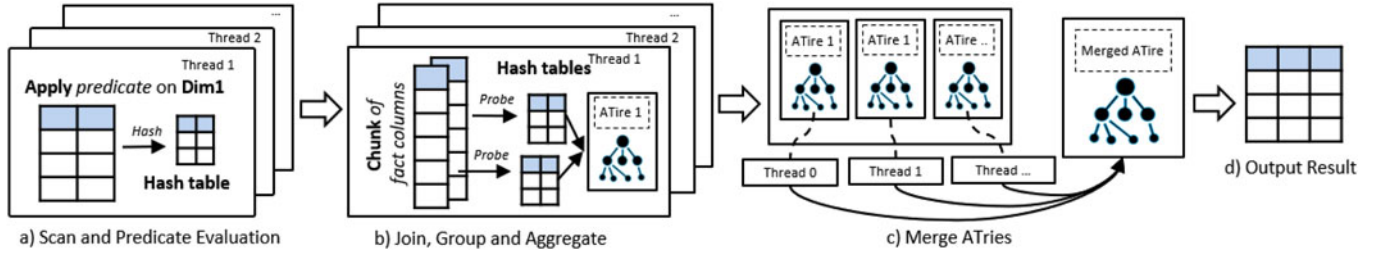


Fig. 5. Atrie group join parallel model.

When these keys are probed into the respective hash table, we get a GAO = ["India", "China", "1997", 30325]. Since all the attributes already exist in each level of the ATrie, we simply traverse those nodes to update the aggregated value of $lo.revenue = 10000 + 30325 = 40325$.

The output of this phase is a complete ATrie with grouping attributes on its edges to guide the grouping process and aggregated value on the leaf node.

Algorithm 3. Recursive Algorithm to Get Final Result

```

Data: ATrie root
1: if root.children is NULL then
2:   if root.height IN attributes then
3:     attributes[root.height] = root.value
4:   else
5:     attributes.Add(root.height, root.value)
6:   end
7:   PRINT (attributes)
8:   return
9: end
10: for child  $c \in$  root.Children do
11:   if root.height IN attributes then
12:     attributes[root.height] = c.Key
13:   else
14:     attributes.Add(root.height, c.Key)
15:   end
16:   GetResult(root.Children[c.Key])
17: end
  
```

4.3 Output Results

ATrie Traversal - Depth-first search (DFS) algorithm is used to produce the final result shown in Fig. 4c. We perform a minor change to the DFS algorithm and include a global hash table that tracks the grouped items at the particular height to facilitate the display of outputs. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. The recursive algorithm for *ATrie Traversal* is shown in Algorithm 3.

4.4 Parallelizing ATGJ

So far we have discussed the serial version of ATGJ. In this section, we present the solution to parallelizing ATGJ shown in Fig. 5.

Setup: Let N be the number of threads (T) that will be used in the query execution (i.e., $T_1, T_2, \dots, T_N$). The number of threads that are activated depends on the number of processors available in the system ($Max\ number\ of\ threads = Processor\ Count - 2$), or if the number of tasks (n) is less than processor count, the algorithm will activate only the required number

of threads. The algorithm specifies the actions to run concurrently, and the run-time handles all process scheduling details, including scaling automatically to the number of processors or processor count on the host if required.

4.4.1 Scan and Predicate Evaluation

Hash - Each thread T_i reads one of the dimension tables included in the query and creates the hash table. Since the dimension tables are significantly smaller than the fact table, we do not employ *data parallelism* because for a smaller set of data, if the operations are not substantial, partitioning the data can be a significant overhead [18]. If the dimension table (s) is large, ATGJ chooses to mix *data and task parallelism* while processing that dimension table(s). In the example Query 3.1, three threads get activated as we have three dimension tables to hash. Once all the threads T_1, T_2 , and T_3 have completed the task of hashing; we move on to the *Join, Group and Aggregate Phase*.

4.4.2 Join, Group and Aggregate

The size of the fact table is significantly large compared to dimension tables. But since we assume in-memory column-stores, the fact table columns reside in memory and we can access those columns using the indexes same as the array lookups. ATGJ uses *getPartitionIndexes* (Algorithm 4) to retrieve the logical partitioning indexes instead of physically partitioning the columns.

Algorithm 4. Get Partitioning Indexes

```

Data: int totalNumberOfRecords, int processorCount
Result: List of partition indexes
1: boundaries < - null
2: min < - 0
3: max < - totalNumberOfRecords / processorCount
4: for  $i < - 0$  to processorCount do
5:   if  $i$  equals processorCount - 1 then
6:     max < - totalNumberOfRecords - 1
7:   end
8:   boundaries < - min, max
9:   min < - max + 1
10:  max < - max + totalNumberOfRecords / processorCount
11: end
12: return boundaries
  
```

Algorithm 4 returns the index boundaries of the sub-range that includes all the indexes in the range. It means that we have an equal number of items in each logical partition such that there is no skewness in the join and group-by processing. For example, if we have total 100 records and

we use 3 threads to perform Join, Group and Aggregate operation, `getPartitionIndexes` returns (0, 33), (34, 66), (67, 99) inclusive ranges.

Probe - Each thread T_i works on the logically partitioned data based on indexes. These indexes serve as a starting and ending point in the algorithm where foreign key columns of the fact table (LINEORDER) is accessed row-by-row based on index values. The foreign keys in those positions are probed into the corresponding hash table to create a GAO. Probing on a hash table is a thread-safe operation because even though hash table allows only single writer thread, it supports multiple reader threads concurrently.

Create ATrie - During the *Probe* each thread T_i works on the logically partitioned data based on indexes. We create a thread local ATrie and the GAO is inserted into these ATries. Therefore, the data is processed in a tight loop. Then, we merge all the thread local ATries in the global merge phase to get the final result.

4.4.3 Merge ATries

Merging ATries can be serial or parallel depending on the number of ATries or the number of threads. For example, let's say we have three ATries A1, A2 and A3. Merging these ATries will require two serial merging 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 the ATrie is much less compared to the number of records in the fact and dimension tables. Therefore, the cost of *merge ATries* is significantly less than *probe* and *create ATrie* operations.

5 EXPERIMENTAL EVALUATION

In this section, we briefly describe the environment used and present a detailed analysis of the results. We have conducted all our experiments on the NeCTAR¹ server equipped with 16 Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS) processors clocked at 2.6 GHz, 64 GB memory and 1 TB RedHat VirtIO SCSI Disk Device. The operating system is Windows Server 2016 Standard. The algorithms were implemented using C#. NET framework 4.5.1 supporting X64 architecture.

5.1 Algorithms Tested

The following column-oriented join algorithms are evaluated in this section.

- Invisible Join [8]: A late materialized join that minimises the value that needs to be extracted out-of-order by rewriting the joins into the predicates on the foreign key columns in the fact table.
- Nimble Join [18]: A progressive parallel star join algorithm for column-stores equipped with multi-attribute array table that facilitates progressive materialization.
- In-Memory Aggregation [4]: A novel technique to perform traditional group-by and aggregation operations across joins using *dense grouping keys* and *key vectors*.
- ATrie Group Join: A highly efficient group join based on ATrie presented in this paper.

TABLE 1
Data Characteristics Used in the Experiments Showing for Each Scale Factor (SF) the Number of Tuples in the Fact Table (#Tuples) and its Disk Size

SF	#Tuples	Size(GB)
25	152 Million	14.2 GB
50	302 Million	28.5 GB
75	451 Million	43.1 GB
100	603 Million	57.7 GB

5.2 Benchmark Dataset

We are using Star Schema Benchmark [29] for the experiment. Sanchez [32] reviewed SSBM and concluded that SSBM is a better benchmark than TPC-H that offers much simpler schema and query execution set. The data set generated using SSBM is uniformly distributed [29]. This uniform distribution is aided by a tool called SSB-DBGEN that makes populating the benchmark data easier and allow quick transitions between transaction tests. However, SSB-DBGEN is not easy to adapt to different data distributions as its metadata and actual data generation implementations are not separated [32].

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. SF determines the amount of information initially loaded into the benchmark tables. As the SF increases, the number of rows added to the tables increases. Data is generated proportionally to SF. All other tables are scaled linearly except for PARTS table, where the data is scale logarithmically [29]. Similar to [4], [8], [18], [30], we use scale factors of 1, 25, 50, 75 and 100 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 1.

5.3 Experimental Results

The numbers reported here are the average of 20 iterations. Before Microsoft Intermediate language (MSIL) can be executed, it must be converted by the .NET Framework Just in time (JIT) compiler to native code. So, a first run was executed to prime the .NET Framework JIT Compiler and the result was discarded. We forced the garbage collector to run after each iteration so that it will not distort the results. For all the algorithms, the standard deviation was less than 10 percent of the average time. We next discuss the detailed analysis of the results.

5.3.1 Total Execution Time

Fig. 6 shows the detailed results of the total execution time broken down by query flight with the standard error of mean (SEM) and the average result across all queries. For all the group-by queries in SSBM, ATGJ is 100 percent faster than the rest of the competing algorithms. For all the queries evaluated, on average, ATGJ is 6X faster than Invisible Join, 4X faster than In-Memory Aggregation and 2X faster than Nimble Join.

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

1. <https://www.nectar.org.au/about-nectar>

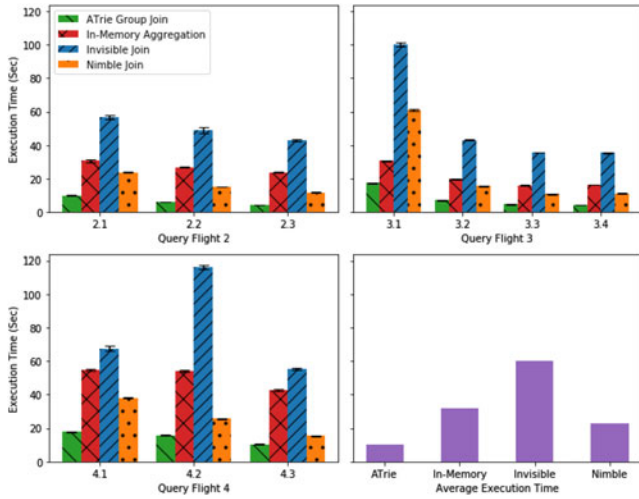


Fig. 6. Total execution time of all algorithms by SSBM query flights ($N = 14$ and $SF = 100$) and the average execution time of all the queries.

table as a single operation. After the probing phase in Join, Group and Aggregate stage, the original join-keys are replaced with the actual attribute values from the dimension table that are used both to perform the group-by efficiently and aggregate the rows using the proposed novel grouping technique and the progressive materialisation.

The size of the data set, query complexity and the number of concurrent processors as well as other factors such as hardware and software configuration, all affect the query performance. With all others being equal, ATGJ scales better than other algorithms for the number of concurrent processors, the number of group-by attributes, the data set size and the query complexity which is shown by the following experiments.

Effect of Number of Threads. In this experiment, all the algorithms were granted unrestricted memory access and processor access in an increasing order by two. We stop at a maximum of 14 processors. Fig. 7 shows the average execution time across all queries and the speed up of all the algorithms.

Fig. 7 shows a significant improvement in performance of ATGJ until $N = 6$ where the cost of work done by multiple threads was significantly higher than splitting the work amongst different threads. However, with increasing number of threads, the task scheduler does not necessarily activate all the processors. It observes whether increasing the number of threads improves or degrades overall throughput and adjusts the number of worker threads accordingly. Therefore, we created a synthetic dataset with 14 dimensional tables, predicate constraints on each dimensional table and increased workload.

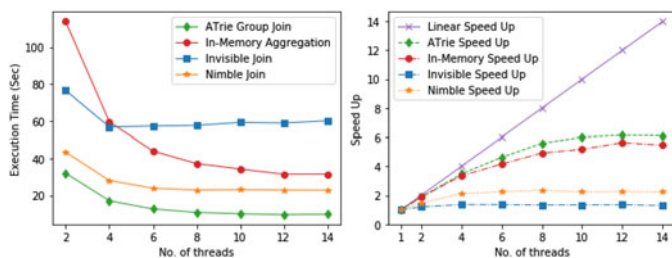


Fig. 7. A comparison of join algorithms while varying number of cores and the speed up of each algorithm ($SF = 100$).

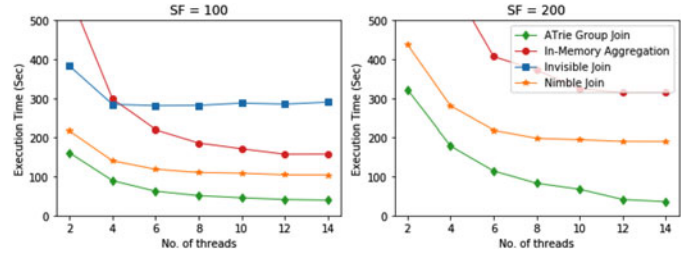


Fig. 8. A comparison of join algorithms while varying number of cores and the speed up of each algorithm.

Fig. 8 shows no significant performance improvement with the increase in the dimensional tables and predicate constraints for $SF = 100$. However, we see performance improvement with increasing number of threads for increased data size (i.e., $SF = 200$). Although data partitioning is the key to parallel processing [41], we follow task parallelism for *Scan* and *Predicate Evaluation* phase in ATGJ which is similar to the parallel processing technique applied by Invisible Join [8] and Nimble Join [18]. The dimension tables are significantly smaller than the fact table. For a smaller set of data, if the operations are not substantial, partitioning the data can be a significant overhead [18]. Therefore, we do not employ *data parallelism*. If the dimension table(s) is large, ATGJ chooses to mix *data and task parallelism* while processing that dimension table(s). We employ data parallelism for *Join*, *Group* and *Aggregate* phase in ATGJ similar to in-memory aggregation [4]. With the logical partitioning of data using `getPartitionIndexes` on the in-memory fact columns, it allows the optimal use of available resources in ATGJ and therefore, we can see a better speed up than the rest of the competing algorithms.

Effect of Number of Group-by Attributes. SSBM has approximately 25 categorical dimensional attributes that can be used in a grouping. In the SSBM queries, the maximum number of grouping attributes used is 4. However, we assume that 10 is the average number of attributes that could be used in a query. The experiment result shown in Fig. 9 uses a modified Query 4.3 in SSBM to include more grouping attributes. For each point in the graph, the number of group-by attributes is increased starting from 1 and continuing to 10 grouping attributes. The query with 10 grouping attributes is shown below.

```
SELECT d.year, d.month, s.nation,
       s.region, s.city, c.nation,
       c.region, c.city, p.brand, p.MFGR,
       sum(lo.revenue - lo.supplycost)
FROM date AS d, customer AS c, part AS p,
     supplier AS s, lineorder AS lo
WHERE lo.custkey = c.custkey
      AND lo.supkey = s.supkey
      AND lo.partkey = p.partkey
      AND lo.orderdate = d.datekey
      AND c.region = 'AMERICA'
      AND s.nation = 'UNITED STATES'
      AND (d.year = 1997 OR d.year = 1998)
      AND p.category = 'MFGR14'
GROUP BY d.year, d.month, s.nation,
         s.region, s.city, c.nation, c.region,
         c.city, p.brand, p.type;
```

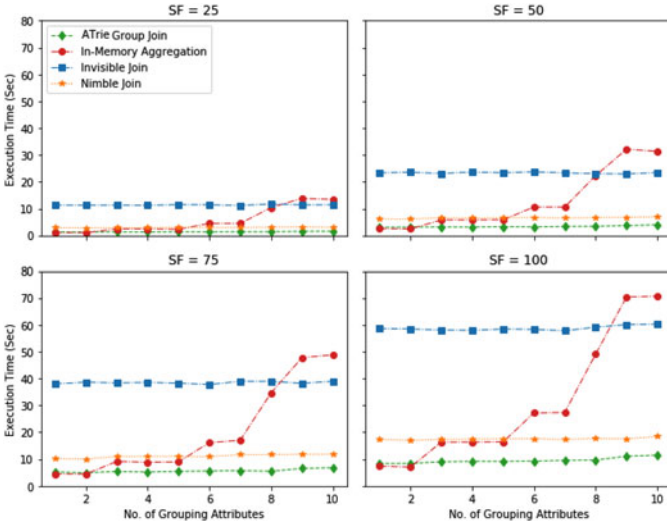


Fig. 9. A comparison of join algorithms for the scalability with an increasing number of grouping attributes.

Fig. 9 shows that performance of ATGJ is consistent with the increasing number of grouping attributes. This is mainly because of the deterministic property of the ATrie where there is no dynamic reorganisation of attributes for the insertion operation in the ATrie and the constant complexity in terms of the fill factor of the ATrie. The primary concern with ATrie was the increased height of the tree with the increasing number of grouping attributes. Although, creating additional internal nodes has incremental computational cost with the increasing number of grouping attributes, this is only a small portion of total execution time and the ATGJ performance decreases minimally with the addition of grouping attributes.

Rather, we have found that the In-Memory Aggregation has an atrocious performance with the increasing number of grouping attributes. We can see two different kinds of impact on In-Memory Aggregation algorithm. First, with the increasing number of dimensions, it creates additional key vectors and temporary tables for each added dimension to process the grouping and aggregation. The cost of adding new key vectors and temporary tables and carrying them through the join execution plan grew linearly with the increase in the number of the dimension table. Second, when the grouping attribute has a large number of distinct values (such as *c.city*. For comparison, *s.city* had 10 distinct value whereas *c.city* had 50 distinct values), this algorithm creates an equivalent number of dense grouping keys and the number of records in temporary table significantly increases. Therefore, the algorithm suffers most in the final stage which cannot be parallelized due to its algorithmic design.

Effect of Order of Group-by Attributes. In ATGJ, the order of the grouping attributes is decided based on the query grouping requirement. One might argue that if the number of dimension tables is large and the cardinality is very large, the ordering is very critical for the query performance. In general, the ordering of query-tree is the critical part in query optimization and the ordering is NP-hard problem. However, it is important to note that the benchmark queries are chosen to span the tasks performed by an important set

TABLE 2
Different Order of the Grouping Attributes
(Modified Query 4.1, # Grouping Attributes = 10)

Group #1	d.year, d.month, s.nation, s.region, s.city, c.nation, c.region, c.city, p.brand, p.type;
Group #2	s.nation, s.region, s.city, c.nation, c.region, c.city, p.brand, p.type, d.year, d.month;
Group #3	c.nation, c.region, c.city, p.brand, p.type, d.year, d.month, s.nation, s.region, s.city;
Group #4	p.brand, p.type, d.year, d.month, s.nation, s.region, s.city, c.nation, c.region, c.city;
Group #5	d.year, s.nation, d.month, s.region, c.nation, s.city, c.region, p.brand, c.city, p.type;

of star schema queries, so that prospective users can derive performance rating from the weighted subset that is expected to use in practice. It is difficult to provide true functional coverage with a small number of queries, but SSBM at least provides functional and selectivity coverage. To understand the effect of order of grouping attributes, we created five different groups of attributes for the query with 10 grouping attributes shown in Table 2.

Fig. 10 shows that changing the order of attributes does not affect the performance of ATrie. This is mainly because of the nature of insertion of attributes in ATrie (i.e., insert if not present) which allows for the shorter access time, greater ease of addition of nodes or updating the values and greater convenience in handling a varying group of attributes.

Effect of Data Size. For each point in the graph, the scale factor is increased starting from 1 and continuing to 100 with a step of 25. The number of records in each of the tables at different scale factors for SSBM are shown in Table 1.

Fig. 11 shows the average execution time across all queries and the scale-up of the algorithms for varying data sizes. ATGJ performs significantly better than the competing algorithms for varying data sizes. We can see the execution time for ATGJ is significantly lower compared to the competing algorithms for the larger data set. The number of records in dimension tables is significantly less compared to the fact table. When the predicate filters are applied, only a small percentage of these dimension table attributes are selected for grouping [41]. ATGJ uses ATrie to group the attributes and uses a hash table to track the edges and navigate through the ATrie to update the aggregation value. Hashing is relatively faster in comparison with other operations in ATGJ. While creating ATrie, the only concern we had was the height of the tree with the increase in the number of grouping

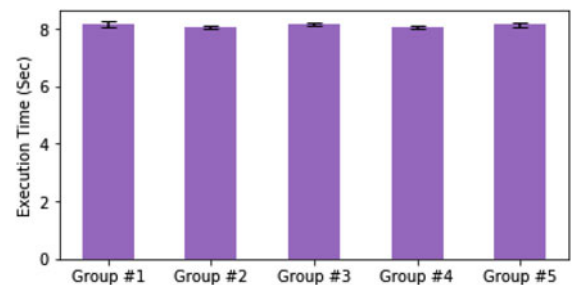


Fig. 10. Average execution time of ATGJ for varying order of grouping attributes (SF = 100, N = 14).

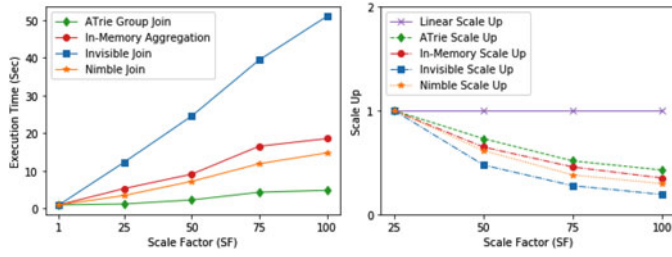


Fig. 11. Average execution time across all SSBM Queries for all algorithms ($N = 14$) and scale-up of the algorithms for varying data sizes. The processing resources are doubled when the data size is doubled.

attributes. However, in Section 5.3.1 we show that the height of the tree does not significantly impact the performance of the ATGJ. Therefore, increasing the data size will not have a significant effect on the performance of ATGJ as long as the computing resources are not exhausted. In addition, from the scale up graph, we see that ATGJ has a better scale up than the competing algorithms.

Effect of Selectivity Ratio. For each point in the graph, the selectivity ratio is decreased by 10 percent starting from 0.016 (high selectivity) and continuing to 0.0000016 (low selectivity).

Fig. 12 shows that for varying selectivity ratio ATGJ is only 2X to 3X times faster than Nimble Join but it is 6X to 8X faster than Invisible Join. Creating ATrie is the second most expensive step after probing in ATGJ. As the selectivity ratio decreases, the number of internal nodes in the ATrie significantly decreases. Therefore, it takes significantly less time to complete the *Create ATrie* phase. Previously discussed algorithms include complex steps in the group and join processing and thereby, can not take advantage of the decreasing selectivity ratio as much as the ATGJ.

5.3.2 Memory Consumption

Table 3 shows the memory used by the algorithms to create and store data in the internal data structures such as *key vectors* and *IMA* in In-Memory Aggregation, *ATrie* in ATrie Group Join, *intermediate hash tables* and *multiple position lists* in Invisible Join and *MAAT* in Nimble Join.

In-Memory Aggregation uses vector transformation to minimise the amount of data that must flow during the query execution and uses array structures to accumulate aggregate data. This strategy uses significantly less memory compared to other algorithms. Although ATGJ has higher memory consumption than In-Memory Aggregation, it consumes

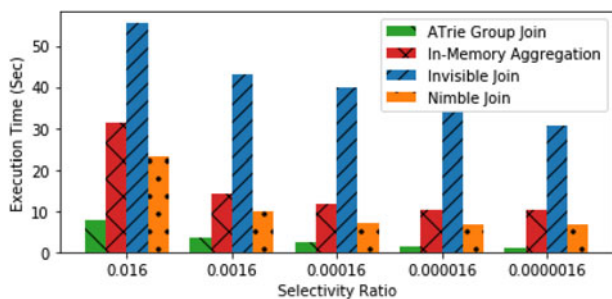


Fig. 12. Total execution time for all algorithms with different selectivity ratio ($N = 14$ and $SF = 100$).

TABLE 3
Memory Used by the Algorithms for Internal Data Structures
(Modified Query 4.1, # Grouping Attributes = 10,
 $N = 14$ and $SF = 100$)

Algorithms	Size(MB)	Memory Consumed (%)
In-Memory Aggregation	0.02	0.00003
ATrie Group Join	133.28	0.20291
Invisible Join	359.29	0.54772
Nimble Join	4869.69	7.42956

comparatively less memory than Invisible Join and Nimble Join because of the ATrie. ATrie features data compression by sharing the same grouping attributes allowing it to use less space than it would take to list out all the distinct results separately as done by Invisible Join and Nimble Join.

5.3.3 Analysing ATrie

We analyse the ATrie based on number of different factors.

Tree Height. A potential problem while creating the ATrie is the height of the tree (number of levels). The height of the ATrie is directly proportional to the `sizeofGAO()`. For example, if the `sizeofGAO()` is 15, the tree height $h = 15$. If the `sizeofGAO()` is 100, the tree height $h = 100$ i.e., the number of nodes to be accessed is high (we need to access 100 nodes). This problem has two facets, namely the number of accessed nodes in the ATrie and the number of created nodes. However, the experiment results (refer Fig. 9) show that the cost to create and access additional nodes in the ATrie is only a small portion of the total execution cost and therefore, can be safely overlooked in ATGJ.

Branching Factor. Another potential problem while creating the ATrie is the branching factor or the number of branches. The branching factor is highly dependent on the number of distinct values of the grouping attributes which can make the ATrie bushy. For example, if the supplier nation attribute (refer Fig. 2) has 100 distinct values, the maximum number of branches in each internal node at height = 3 is 100. However, the experiment results (refer Fig. 9) show that the branching factor has no significant impact in the performance of ATGJ. The performance of ATGJ is stable throughout the experiment even though the grouping attribute with a large number of distinct value was introduced such as *c.city* (For comparison, *s.city* had 10 distinct value whereas *c.city* had 50 distinct values).

Order of Grouping Attributes. Another potential problem while inserting a GAO into the ATrie is the order of grouping attributes in the GAO. We want to understand whether the ordering of group-by attribute has any impact on the performance of ATGJ. For example, “Does grouping-by customer nation, supplier nation, and order date show the same performance as grouping-by order date, customer nation, and supplier nation?” We have found that changing the order of the grouping attributes has no significant impact in the creation of ATrie as shown in Fig. 10.

Differences With B+ Tree. Although, ATrie and B+ Tree look similar, the differences are as follows. In B+ Tree, the root node must have at least 2 pointers to other nodes when it is not the leaf node [42]. However, in the ATrie, the root node can have 1 to many pointers to other nodes depending on the data item in the root node. In B+ Tree, the entries in

the internal node point to the data classified as a greater than or equal to the corresponding reference value and its extra pointer references data classified as less than the node's smallest reference value [42]. However, in the ATrie, each height represents a different category of data. The internal nodes at the same height can have same keys but an internal node will not have duplicate keys. In B+ Tree, the leaf node contains an extra pointer reference to the next leaf node in the tree ordering; the leaves are linked and known as neighbours [42]. However, in the ATrie, the leaf node holds the aggregation value and there is no relation between the two leaf nodes.

6 CONCLUSION

The main contribution of this paper is to propose a parallel star group join algorithm for in-memory column-stores called ATrie Group Join. The algorithm utilises a novel technique to perform traditional group-by and aggregation operations across joins by using the *aggregate trie* (a.k.a *ATrie*). We leveraged the technique of progressive materialization to represent grouping attributes on the edges and accumulated aggregates on the leaf node of the ATrie. This helped us perform join, grouping and aggregation operation on the fly.

Experimental results show that ATGJ outperforms all the competing algorithms. For all the queries evaluated, on average, the ATGJ is 6X faster than Invisible Join, 4X faster than In-Memory Aggregation and 2X faster than Nimble Join. Furthermore, we also demonstrated that ATGJ scales better than other algorithms for the number of concurrent threads, the number of group-by attributes, the data set size and the query complexity.

We hope that this paper leads to more research into the column-store group joins as we have shown for possible improvements, and anticipate that more are achievable. Future work may include adapting ATGJ to operate on cluster computing frameworks such as Apache Spark [43].

REFERENCES

- [1] V. Raman *et al.*, "DB2 with BLU acceleration: So much more than just a column store," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1080–1091, 2013.
- [2] M. Eich, P. Fender, and G. Moerkotte, "Efficient generation of query plans containing group-by, join, and groupjoin," *The VLDB J.—The Int. J. Very Large Data Bases*, vol. 27, no. 5, pp. 617–641, 2018.
- [3] 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, 2007.
- [4] 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.*, 2018, pp. 1441–1452.
- [5] D. Abadi *et al.*, *The Design and Implementation of Modern Column-Oriented Database Systems*, vol. 5. Delft, The Netherlands: Now, 2013. [Online]. Available: <http://dx.doi.org/10.1561/19000000024>
- [6] R. MacNicol and B. French, "Sybase IQ multiplex-designed for analytics," in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 1227–1230.
- [7] M. Stonebraker *et al.*, "C-store: A column-oriented DBMS," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 553–564.
- [8] 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*, 2008, pp. 967–980.
- [9] C. Ordóñez, "Can we analyze big data inside a DBMS?" in *Proc. 16th Int. Workshop Data Warehousing OLAP*, 2013, pp. 85–92.
- [10] K. Sridhar, "Modern column stores for big data processing," in *Proc. Int. Conf. Big Data Analytics*, 2017, pp. 113–125.
- [11] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.
- [12] Q. Cai *et al.*, "Memepic: Towards a unified in-memory big data management system," *IEEE Trans. Big Data*, vol. 5, no. 1, pp. 4–17, Mar. 2019.
- [13] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.
- [14] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: Data management for modern business applications," *ACM SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, 2012.
- [15] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos, "Real-time analytical processing with SQL server," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1740–1751, 2015.
- [16] P.-Å. Larson, E. N. Hanson, and M. Zwillig, "Evolving the architecture of SQL server for modern hardware trends," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1239–1245.
- [17] T. Lahiri *et al.*, "Oracle database in-memory: A dual format in-memory database," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 1253–1258.
- [18] P. Sangat, M. Indrawan-Santiago, D. Taniar, and C. Messom, "Nimble join: A parallel star join for main memory column-stores," *Concurrency Comput. Pract. Experience*, vol. 30, no. 1, 2019, Art. no. e5616.
- [19] S. Idreos *et al.*, "MonetDB: Two decades of research in column-oriented database architectures," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 35, no. 1, pp. 40–45, 2012.
- [20] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960. [Online]. Available: <http://doi.acm.org.ezproxy.lib.monash.edu.au/10.1145/367390.367400>
- [21] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 4, pp. 709–730, Jul./Aug. 2002.
- [22] C. Kim *et al.*, "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.
- [23] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 37–48.
- [24] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *Proc. VLDB Endowment*, vol. 5, no. 10, pp. 1064–1075, 2012.
- [25] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 85–96, 2013.
- [26] Ç. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on modern processor architectures," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1754–1766, Jul. 2015.
- [27] R. Barber *et al.*, "Memory-efficient hash joins," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 353–364, 2014.
- [28] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh, "Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach," *Proc. VLDB Endowment*, vol. 8, no. 6, pp. 642–653, 2015.
- [29] 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*, 2009, pp. 237–252.
- [30] 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, 2013.
- [31] 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.*, 2019, pp. 476–480.
- [32] J. Sanchez, "A review of star schema benchmark," *CoRR*, vol. abs/1606.00295, 2016. [Online]. Available: <http://arxiv.org/abs/1606.00295>
- [33] P. O'Neil and G. Graefe, "Multi-table joins through bitmapped join indices," *ACM SIGMOD Rec.*, vol. 24, no. 3, pp. 8–11, 1995.
- [34] V. Markl, F. Ramsak, and R. Bayer, "Improving OLAP performance by multidimensional hierarchical clustering," in *Proc. Int. Symp. Database Eng. Appl.*, 1999, pp. 165–177.
- [35] 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. Discov.*, 2005, pp. 200–209.
- [36] 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./Dec. 2002.

- [37] A. Weininger, "Efficient execution of joins in a star schema," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 542–545.
- [38] C. A. Galindo-Legaria *et al.*, "Optimizing star join queries for data warehousing in Microsoft SQL server," in *Proc. IEEE 24th Int. Conf. Data Eng.*, 2008, pp. 1190–1199.
- [39] N. Askitis, "Fast and compact HASH tables for integer keys," in *Proc. 22nd Australas. Conf. Comput. Sci.*, 2009, pp. 113–122.
- [40] O. Polychroniou and K. A. Ross, "Vectorized bloom filters for advanced SIMD processors," in *Proc. 10th Int. Workshop Data Manage. New Hardware*, 2014, Art. no. 6.
- [41] 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.
- [42] J. Jannink, "Implementing deletion in B+-trees," *ACM SIGMOD Rec.*, vol. 24, no. 1, pp. 33–38, 1995.
- [43] M. Zaharia *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.



Prajwol Sangat received the master's of information technology by research (MIT) degree from Monash University, Melbourne, Australia, in 2015. He is currently working toward the PhD degree in computer science with the Caulfield School of Information Technology, Monash University, Melbourne, Australia. His research interests include distributed and parallel database systems, main-memory column-stores, query processing, and cost models.



David Taniar received the BSc, MSc, and PhD degrees, all in computer science, specialising in databases. His research is mainly in 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 (*Data Warehousing and Mining*, and the *Web and Grid Services*). He is currently an associate professor at the Faculty of Information Technology, Monash University, Melbourne, Australia.



Christopher Messim received the MSc and PhD degrees in computer science from Loughborough University, Loughborough, United Kingdom, in 1992 and 1989, respectively. He was a lecturer at Singapore Polytechnic from 1993 to 1997, senior lecturer at the Dubai University College, U.A.E, from 1998 to 1999, senior lecturer and director of the Centre for Parallel Computing at 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 as well as deputy head of School/Campus with the Faculty of IT at Caulfield, Monash University, Australia, 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 and is currently Academic Workforce manager at the Faculty of IT at Monash University, Australia. His research spans intelligent systems, data mining and high performance computing. He is the author of more than 120 research papers in various journals and conference proceedings.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.