MONASH University

Parallel Star Join Query Processing in Column-Oriented Data Storage

Prajwol Sangat

MIT, BE

Candidate's ORCID

A thesis submitted for the degree of Doctor of Philosophy

(0190) at

Monash University in 2020

Information Technology

Copyright Notice

I would like to dedicate this thesis to my parents Mr. and Mrs. Purna Sangat. I hope that this achievement will fulfill the dream that you had for me all those many years ago when you chose to give me the best education you could.

Abstract

Today, an ever-increasing number of researchers and businesses collect and analyse massive amounts of data in database systems. This big data has posed two connected challenges to data management solutions - processing unprecedented volumes of data, and providing ad-hoc real-time analysis without compromising performance. At the same time, computer hardware systems are scaling out elastically, scaling up in the number of processors and cores, and increasing extensively the main memory capacity. The data processing challenges, combined with the rapid advancement of hardware systems, have entailed the evolution of a new breed of parallel star algorithms optimised for column-oriented data storage. In this thesis, we design, implement, and evaluate two new parallel algorithms: *Nimble Join* and *ATrie Group Join (ATGJ)*, and answer the star join queries that include join, group-by and aggregation operation. In addition, we formulate the cost models for these algorithms, and instantiate and evaluate them.

Nimble Join is a progressive parallel star join algorithm that avoids multiple-pass scans in column processing using *Multi-Attribute Array Table (MAAT)*. MAAT includes a reduced list of signed integer positions that serves as a dynamic filter to probe the indexed array, drastically reducing the number of array lookups.

ATrie Group Join (ATGJ) integrates join, grouping and aggregation operations, performs a single scan of the fact columns, and uses hybrid parallelism for the optimal use of computing resources. It employs a novel grouping and aggregation technique using *Aggregate Trie* or *ATrie*, and facilitates the processing of data in tight loops, thereby significantly improving the performance of the algorithm on modern hardware.

ATGJ is extended so that it can be applied in a distributed environment and is known as *Distributed ATire Group Join (DATGJ)*. DATGJ uses the divide and broadcast-based approach to perform a single scan of fact columns and join data, avoiding cross-communication between workers. It uses *Fast Hash Table (FHT)* to perform join and ATrie to perform group-by and aggregation. DATGJ uses multiple workers, and the distributed processing improves scalability, fault tolerance, resource sharing and efficient computation of tasks.

The algorithms have been evaluated using the Star Schema Benchmark (SSBM) to demonstrate their superiority over the current, most popular approaches. The cost model accuracy has been verified by comparing it with the results of detailed experiments using different hardware parameters.

Declaration

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, financial support and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of Monash University, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Prajwol Sangat
Information Technology
Monash University
October 23, 2020

Publications

Some of the material in this thesis has previously appeared in the following publications:

- P. Sangat, M. Indrawan-Santiago, D. Taniar, Sensor data management in the cloud: Data storage, data ingestion, and data retrieval, Concurrency and Computation: Practice and Experience 30 (1) (2018) e4354.

- P. Sangat, D. Taniar, M. Indrawan-Santiago, C. Messom, Nimble Join: A parallel star join for main memory column-stores, Concurrency and Computation: Practice and Experience (2019) e5616. doi: https://doi.org/10.1002/cpe.5616

- P. Sangat, D. Taniar, C. Messom, ATrie Group Join: A Parallel Group Join and Aggregation for In-Memory Column-Stores, IEEE Transactions on Big Data. doi: https://doi.org/10.1109/tbdata.2020.3004520

- P. Sangat, D. Taniar and C. Messom, Distributed ATrie Group Join: Towards Zero Network Cost, IEEE Access 8 (2020) pp. 111598-111613.

The following paper has been published, but not included as it is not directly related to the research topic of this thesis:

- P. Sangat, M. Indrawan-Santiago, D. Taniar, B. Oh, P. Reichl, Processing high-volume geospatial data: A case of monitoring heavy haul railway operations, Procedia Computer Science (80) (2016) pp.2221-2225.

- L. Liang , P. Sangat, D. Taniar, Parallel Left Outer Joins in Distributed Environment: Does size and side of the table matter?, IEEE Access.
  Submitted on: 16 June 2020
  Status: Under Review

Acknowledgments

This research was supported by an Australian Government Research Training Program (RTP) Scholarship.

I am exceedingly grateful to my supervisors Assoc. Prof. David Taniar, Dr. Maria Indrawan-Santiago, and Dr. Christopher Messom for their supervision, friendship, support and advice during my PhD candidature. David, thank you for believing in me and giving me the opportunity to work under your supervision. I am grateful for your insights and immense knowledge that shed light on my path and enabled me to complete this PhD thesis. Working with you truly helped me to grow as a researcher. I would also like to thank you for providing me with the opportunity to join your teaching team which helped my academic career. Maria, I want to thank you for all your support and help during my candidature. You have been more than a supervisor to me. Your continuous academic support and incredible encouragement accompanied me throughout the years of my candidature. Chris, I want to thank you for all your invaluable advice and constructive comments. Your advice during every meeting helped me broaden my research scope. Overall, words are not enough to express my gratitude to my supervisors. They not only advised me on research methodology but also are role models for me, being a researcher like them will be my life credo.

I am greatly indebted to the graduate research student community in the Faculty of Information Technology, on both Caulfield and Clayton campuses. The community has been a great source of support for me during this journey. I would like to thank Ariesta, Dharshini, Ethan, Liang, Megan, Paras, Tennyson, Tian and Yuri for all the tips, jokes, laughter, lunch-times, help and moral support that you have given me. I would like to thank Allison, Helen, Julie, and all the other faculty administration staff: a huge thank you for the help and advice given, as well as support from the Faculty. I would like to express my gratitude to Bruna Pomella for helping me to proofread my thesis.

Finally, I would like to thank my family, especially my beloved wife, and my parents for their endless support, patience, encouragement, and love. Every week, calling my parents have always been the happiest moment, especially whenever I told them that my papers had been accepted. I also hope that happiness, harmony, and health remain forever in my family.

Contents

List of Figures

List of Tables

List of Abbreviations

ATGJ	**AT**rie **G**roup **J**oin
ATrie	**A**ggregate **Trie**
BI	**B**usiness **I**ntelligence
CAT	**C**oncise **A**rray **T**able
CHT	**C**oncise **H**ash **T**able
CPU	**C**entral **P**rocessing **U**nit
DATGJ	**D**istributed **AT**rie **G**roup **J**oin
DFS	**D**epth- **F**irst **S**earch
DGK	**D**ense **G**rouping **K**ey
DIRA	**D**ata- **I**ndependent **R**andom **A**ccess
DSM	**D**ecomposition **S**torage **M**odel
DSRM	**D**esign **S**cience **R**esearch **M**ethodology
EM	**E**arly **M**aterialisation
FHT	**F**ast **H**ash **Table**
FK	**F**oreign **K**ey
GAO	**G**roup **A**ggregation **O**bject
GPU	**G**raphical **P**rocessing **U**nit
I/O	**I**nput **O**utput
ICCS	**I**nternational **C**onference on **C**omputational **S**cience
ICDE	**I**nternational **C**onference on **D**ata **E**ngineering
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IGDC	**I**nternational Journal of **G**rid and **D**istributed **C**omputing
IMA	**I**n- **M**emory **A**ccumulator
IRT	**I**nitial **R**esponse **T**ime
IS	**I**nformation **S**ystems
JIT	**J**ust **I**n **T**ime

JPDC	**J**ournal of **P**arallel and **D**istributed **C**omputing
LAN	**L**ocal **A**rea **N**etwork
LM	**L**ate **M**aterialisation
MAAT	**M**ulti- **A**ttribute **A**rray **T**able
MFRJ	**M**ulti- **F**ragment- **R**eplication **J**oin
MRJ	**M**apReduce- **I**nvisible **J**oin
MSIL	**M**icrosoft **I**ntermediate **L**anguage
NSM	**N**-ary **S**torage **M**odel
RDD	**R**esilient **D**istributed **D**ataset
RLE	**R**un **L**ength **E**ncoding
SBFCJ	**S**park **B**loom **F**iltered **C**ascade **J**oin
SBJ	**S**park **B**roadcast **J**oin
SCHT	**S**tandard- **C**hain **H**ash **T**able
SEM	**S**tandard **E**rror of **M**ean
SF	**S**cale **F**actor
SIGMOD	**S**pecial **I**nterest **G**roup on **M**anagement of **D**ata
SIMD	**S**ingle **I**nstruction, **M**ultiple **D**ata
SIMT	**S**ingle **I**nstruction, **M**ultiple **T**hreads
SQL	**S**tructured **Q**uery **L**anguage
SSBM	**S**tar **S**chema **B**enchmark
TET	**T**otal **E**xecution **T**ime
TLB	**T**ransaction **L**ookaside **B**uffer
TPC	**T**ransaction **P**rocessing Performance **C**ouncil
VLDB	**V**ery **L**arge **D**ata **B**ase
WAN	**W**ide **A**rea **N**etwork

List of Symbols

F	Size of the table
$\mid F \mid$	Cardinality of the table
F_i	Size of the i-th table column, i = 1 ... n
$\mid F_i \mid$	Cardinality of the i-th table column
N	Number of processors
P	Page size
H	Hash Table size
π_i	Projectivity ratio of the i-th table
σ_i	Selectivity ratio of the i-th table
IO	Time to read a page from the disk
t_w	Time to write the record to the main memory
t_r	Time to read a record in the main memory
t_d	Time to compute destination
m_p	Message protocol cost per page
m_l	Message latency for one-page
$\lceil \rceil$	Ceiling function
$\lfloor \rfloor$	Floor function
$\wedge$	Minimum value
$\vee$	Maximum value
$\bowtie$	Inner Join
$\star$	Star Join
$\mp$	Column Join
$\rightarrow$	Single Join Mode
$\Rightarrow$	Parallel Join Mode
$\ll$	Distributed Join Mode

Chapter 1

Introduction

1.1 Motivation

An increasing number of companies rely on the results of big data analytics to improve their operations, planning, customer service and risk management, and increase their revenue. For example, in a survey of 476 executives around the world, more than half confirmed that their data has made existing services and products more profitable [2]. However, the large volume of data generated by companies have posed two connected challenges to data management solutions - processing unprecedented volumes of data, and providing ad-hoc real-time analysis in mainstream production data stores without compromising the performance.

The unprecedented volume of data generated by companies results in the failure of traditional relational database management systems (a.k.a. row-stores) as they can no longer process the data efficiently [3]. An alternative to row-stores are column-stores that have a novel layout tailored for analytical query processing. In a column-store, information about a logical entity is stored as separate columns in multiple locations in memory or disk [4]. For example, information about a customer such as name, address, city can be stored as a separate column in main memory or disk as shown in Figure 1.1. This data storage model is known as a Decomposed Storage Model (DSM) [5]. DSM makes column-stores more I/O efficient for read-only queries as they can read only those columns from the memory or disk that is accessed by the query [4, 6, 7].

In addition, computer hardware systems are scaling out elastically, scaling up in the number of processors and cores, and substantially increasing main memory capacity [8]. The use of multi-core parallelism can be beneficial in the context of query optimisation and execution for reasons such as increased system throughput and decreased response time [9].

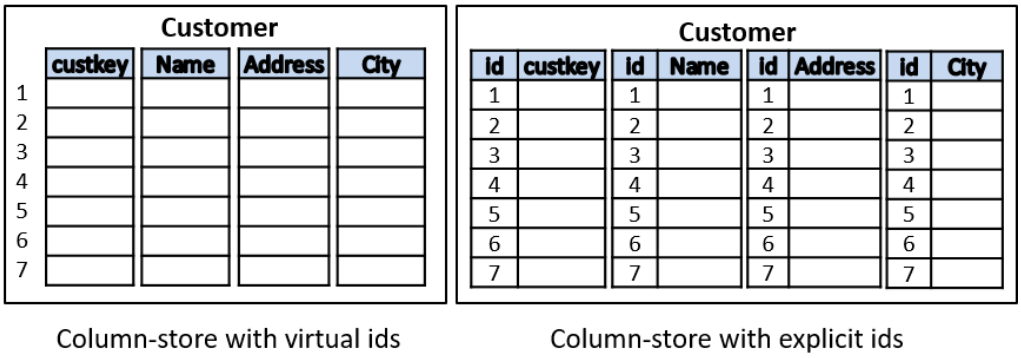


Figure 1.1: Physical layout of column-store

Also, most column-stores such as Hyper [10], SAP HANA [11], IBM DB2 with BLU Acceleration [12], Microsoft SQL Server [13, 14] and Oracle In-Memory Database [15, 16] are being tailored to run as main memory database systems which avoids the latency associated with secondary storage [10, 17–19]. Therefore, the data processing challenges combined with the rapid development of hardware systems have necessitated and motivated the evolution of a new breed of parallel star algorithms optimised for column-oriented data storage.

1.2 Scope of Research

The star queries or analytical queries spend most of the time on performing scans, predicate evaluation, joins, grouping and aggregation. In-Memory databases have vastly improved the scan and predicate evaluation cost by using single instruction, multiple data (SIMD) instructions to process columnar data by orders of magnitude [16]. A study of customer queries in DB2 [12, 20] and other surveys [21] have found that the group-by constructs occur in a large proportion of the analytical queries [22]. The queries in the benchmarks such as TPC-H [1] and the Star Schema Benchmark (SSBM) [23] spend more than 50% CPU time on joins, group-by and aggregation [16]. These queries often aggregate large portions of the data, which can lead to performance issues with very large data sets. Therefore, the scope of this research is the efficient processing and optimisation of star queries with join, group-by and aggregation using parallel and distributed processing to improve and maintain the performance as shown in Table 1.1.

There are two broader categories of star queries: 1) Star Join Queries - queries involving join operation, and 2) Star Group-By Join Queries - queries involving join, group-by and

[1] http://www.tpc.org/tpch/

Minimise Bits Read	Column-Stores Design Feature Research Areas
Skip loading of not-selected attributes	Column-oriented data storage [5]
Skip non-qualified values	Projections Cracking [24]
Skip redundant bits	Per-column compression [25, 26]
Adaptive / partial indexing	Database Cracking [27]
Work on selected attributes only (per-operator)	Materialisation strategies [28, 29]
Minimise CPU Time	
Minimise processing for each bit read	Joins [30]
	Group-by and Aggregation operations [31]
	Operating on compressed columns [28]
Minimise instructions and data misses	Vectorised execution [25]
Tailored operators	RISC style algebra [31]

Table 1.1: Taxonomy of design features that define modern column-stores. The highlighted areas indicate the scope of this research. The features are discussed in Section 2.1.3.

aggregation operation. Next, we provide a brief research overview of the two kinds of analytical queries (a comprehensive literature review is presented in Chapter 2).

1.2.1 Star Join Queries

Star Joins: Since the columns are stored separately, even the tuple reconstruction requires a join between the columns. So, the joining of two or more tables will involve additional operations such as tuple reconstruction, projection, grouping and sorting in addition to joining columns specified in query join condition. Existing join algorithms such as [32–37] provide only a partial solution to processing star join queries in column-stores. They include steps on how to join two columns specified in the query join condition, but do not include steps to re-construct the tuples required for the query output, filter conditions and operations such as group-by and sort. Invisible Join [30] performs star-join in column-stores that include operations such as tuple reconstruction and grouping; however, we have identified three shortcomings of the algorithm:

1. It has a performance bottleneck of multi-pass scan for column processing and increased memory consumption with the increasing number of tables in the join query.

2. It incurs a significant memory overhead cost because it creates multiple intermediate position lists.

3. It follows a task-parallel approach resulting in sub-optimal use of resources.

Memory and Initial Response: Mainstream data warehouses today hold several terabytes of data with table sizes over the one billion row threshold [38]. Therefore, the analytical queries need to be processed in parallel to gain performance improvements. The recent works on efficient parallel join algorithms show that carefully-tuned join implementation can perform well regardless of the data size [35, 39–43]. However, they assume an unlimited reserve of main memory and focus on minimising the total execution. The main memory is finite and will eventually be exhausted when the input tables or intermediate results exceed the available memory because of the increase in data. In addition, from a user's perspective, it is ideal to generate the first few results quickly with minimal response time so that the data processing can begin immediately. Therefore, an optimal solution is a star join algorithm for column-stores that has a fast response time, a fast query execution time, consumes less memory, and operates in parallel.

1.2.2 Star Group-By Join Queries

Star Group-By Join: Performing group-by aggregation when a join over two or more tables is involved in the query is the cornerstone of almost all star join queries. However, Invisible Join is not designed for grouping and aggregation operations. It uses a traditional approach, *Hash Join/ Hash Group-By*, where column-oriented hash join (using bloom filters, hash table build and probe) is followed by row-oriented hash group-by (using serial aggregation).

The unconventional approach is *In-Memory Aggregation* where group-by expressions are pushed down into the scans of dimension tables [16]. They create a unique key per distinct group called a *Dense Grouping Key (DGK)* and map the join keys to DGKs using a *Key Vector*. The key vector is used to filter non-matching rows during fact table scans where the aggregation result is stored in a multidimensional array known as *In-memory Accumulator (IMA)*.

Although the unconventional approach reduced the execution time substantially more than did the traditional approach, we have identified three shortcomings of the algorithm:

1. 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. The disadvantage of this approach is that the data is not processed in tight loops [36], which results in considerable performance deterioration on modern hardware.

2. This algorithm is efficient only if the IMA does not become too large [16].

3. With the increasing number of dimensions and grouping 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.

Excessive Network Communication and Disk Spill: Although parallel star joins improve the performance of the system, they are limited by the hardware as the join operation is performed using a single computer [44, 45]. On the other hand, distributed star joins involve multiple worker computers and the distributed computing improves scalability, fault tolerance and resource sharing, and helps perform computation tasks efficiently [46, 47]. However, large-scale data shuffling is inevitable in analytical queries such as distributed join between two large tables which are less popular as research topics or are left for data-centric generic distributed systems such as Apache Spark [48] for batch processing [49, 50]. Although Spark facilitates joins and group-by using Resilient Distributed Datasets (RDDs) [51], Spark SQL [52] and Spark DataFrame [52] operations, we have identified three shortcomings:

1. 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 [53]. 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.

2. It incurs excessive disk access and network communication because of cross-communication between the worker nodes. Unnecessary disk access is often the result of disk spill; i.e. the data is spilled into disk due to an overflowing memory buffer.

3. Excessive shuffling of records not only significantly increases the network communication cost, but also prevents further processing of the algorithm [46]. Therefore, naive Spark implementation fails to handle the issues such as multiple scans of data, excessive network communication and disk spill.

1.3 Research Aim and Questions

The aim of this research is to investigate, propose, design, implement and validate column-oriented parallel and distributed join algorithms. We have formulated the following research questions to address the problems described in the previous section:

In Chapter 4, we focus on the following research question (RQ):

RQ 1 - *How to **develop** and **optimise** parallel **joins** for main memory column-stores?*

In Chapter 5, we focus on the following research question (RQ):

RQ 2 - *How to **develop** and **optimise** parallel **group-by joins** for main memory column-stores?*

In Chapter 6, we focus on the following research question (RQ):

RQ 3 - *How to **extend** and **optimise** the parallel group-by joins to **distributed** main-memory column stores?*

1.4 Research Contribution and Impact

In answering **RQ1**, our main contributions presented in Chapter 4 are as follows:

- We extend the concise array table [54] known as **Multi-Attribute Array Table (MAAT)** that handles multiple attributes and facilitates probing required by the join algorithm.
- We propose a novel materialisation strategy based on MAAT known as **Progressive Materialisation**.
- We propose a new progressive parallel star join algorithm for the main memory column-stores known as **Nimble Join** that is significantly better than its competing column-store join algorithm. It uses a multi-attribute array table to hold attributes required by join query processing that facilitates progressive materialisation.
- We develop an analytical model to understand and predict the query performance of Nimble Join.

These contributions were published as a full research paper in CCPE 2019 [44].

In answering **RQ2**, our main contributions presented in Chapter 5 are as follows:

- We propose a novel approach to perform group-by and aggregation operations influenced by the concept of trie or prefix tree [55] using a data structure called **Aggregate Trie** or **ATrie**. The grouping and aggregation can be seen as a tree-shaped deterministic finite automation.
- We propose a new parallel star group join and aggregation for in-memory column-stores known as **ATrie Group Join (ATGJ)** that is a variation of Hash Join/Hash Group-By and uses both techniques but solves the problem of grouping and aggregating data using a rather novel approach with the help of *ATrie*.

- We propose an analytical model to understand and predict the query performance of ATGJ.

These contributions were published as a full research paper in ICCS TBD 2020 [45].

In answering **RQ3**, our main contributions presented in Chapter 6 are as follows:

- We present a new optimisation technique for efficient search in the hash table. The key idea is to use Robin Hood hashing [56] with an upper limit on the number of probes which are implemented in the **Fast Hash Table (FHT)**.

- 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.

- We propose an analytical model to understand and predict the query performance of DATGJ.

These contributions were published as a full research paper for publication in ICCS ACCESS 2020 [57].

In general, the main contributions of this PhD study to the current body of knowledge are:

- The design of a series of new column-oriented serial and parallel join algorithms that draws on the parallel processor architecture and develop cost models for them that are significantly better than the state-of-the-art algorithms.

- The design of a new column-oriented distributed join algorithm that draws on the distributed computing architecture and has improved performance than the current most popular approaches.

- The instantiation of column-oriented parallel and distributed join algorithms as a proof of concept or development.

- The extension and empirical validation of the column-oriented join algorithms by considering standard benchmarks such as SSBM.

1.5 Thesis Organisation

In this chapter, we discussed the research background, explained the research problem, stated the research aims and questions and the contributions made to tackle the problem. The rest of the thesis is organised as follows: Chapter 2 presents an overview of the related work on joins and aggregations. Chapter 3 describes the research methodology and evaluation methods. In Chapter 4, we explore the parallel star join for column-store. In Chapter 5, we investigate the parallel star group join and aggregation for column-stores. In Chapter 6, we explain the working of star group join and aggregation in distributed environment. Finally, Chapter 7 summarises the thesis contributions and suggests future research directions.

Chapter 2

Literature Review

Parallel star joins use parallel processing and execute join tasks simultaneously using multiple processors, whereas distributed star joins use distributed processing to divide a single join task among multiple worker computers in order to complete the join operation. Although parallel star joins improve the performance of column-stores, they are limited by the hardware as the join operation is performed using a single computer [44, 45]. On the other hand, distributed star joins involve multiple worker computers and the distributed computing improves scalability, fault tolerance, resource sharing and helps perform computation tasks efficiently [46,47]. In this chapter, we review the latest research on column-stores, parallel hash joins, parallel star joins and distributed star joins.

2.1 Column-Store

Column-stores vertically partition the data into a collection of individual columns and store them separately in memory or disk. This architecture enables queries to read just the attributes that are required to produce results, rather than having to read the entire record from disk, improving the utilisation of I/O and memory bandwidth [5–7, 17, 58].

The roots of column-stores can be traced back to the 1970s when transposed files first appeared [59]. One of the earliest systems that resembled modern column-stores was Cantor [60, 61] that included data compression features such as delta encoding and run-length encoding (RLE).

In 1985, the Decomposition Storage Model (DSM) [5], a predecessor of the column-store was proposed. It was the first comprehensive comparison for row versus column-oriented storage structure. At that time, row-store was popular as N-ary Storage Model (NSM) and was the standard architecture used in relational database systems. In DSM,

each column of a table was stored separately, and for each attribute value within a column, it stored a copy of the corresponding surrogate key (or record id).

DSM could speed up the data scan time compared to NSM when only a few columns were projected at the expense of extra storage space as it stored each column with surrogate keys [5]. However, this research did not examine the compression techniques; nor did it evaluate any benefits of column orientation for relational operators other than scans. In addition, inter-operation (or task) parallelism (discussed in Section 2.2) can be enhanced leveraging the DSM format [62]; moreover, join and projection indices can further strengthen the advantages of DSM over NSM [63].

Although DSM research demostrated several advantages of the column over the row storage model, it was not until much later, in the 2000s, that the column-oriented storage structure became a popular storage format for data warehousing and business intelligence (BI) applications. The following sections discuss the data layout, access pattern and features of column-stores.

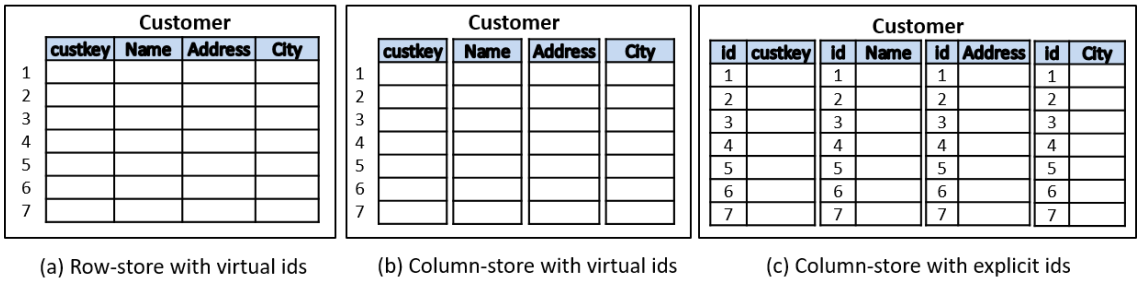


Figure 2.1: Physical layout of column-store versus row-store

2.1.1 Data Layout and Access Pattern

Figure 2.1 illustrates the basic differences between the physical layout of column-stores and that of row-stores. It shows three different ways to store a `CUSTOMER` table which contains several attributes.

In the column-oriented approach (Figure 2.1 (b) and Figure 2.1 (c)), each column is stored independently as a separate data object. Since the data is typically read from storage and written to storage in blocks, in this approach, each block that holds data for the `CUSTOMER` table holds data for only one of the columns. In this case, a query such as *find the number of customers from a particular city* would need to access the `city` column, and only the data blocks corresponding to that column would be accessed.

In the row-oriented approach (Figure 2.1 (a)), the entire table is stored as a single data object holding data from all the columns. Hence, it is impossible to read just the

attributes required for the query without transferring all the attributes. Therefore, for the aforementioned query, this approach will be forced to read a significant amount of redundant data as the required attributes are stored together with all other attributes in the same block. The disadvantage of this approach is that the data transfer cost is often the major bottleneck. Therefore, with fat tables (i.e. tables with hundreds of attributes) being common, the column-oriented data layout is likely to be more efficient when executing queries that require only a subset of the table's attributes [30].

2.1.2 Materialisation Strategies

As mentioned earlier, column-stores vertically partition the database tables and store each column separately on the disk. Although this is a physical modification of storage layout, logically it is still the same as row-stores. The application involved with the database, column-oriented or row-oriented, treats the interface as row-oriented. At some point in time, column-stores must stitch multiple attributes together to generate tuples and execute the rest of the query plan using row-store operators. This process of adding attributes to generate the result is called *materialisation* [28]. There are two different materialisation strategies for column-stores: early materialisation and late materialisation.

Consider a simple example: Suppose a query has three selection operators σ_1, σ_2 and σ_3, over columns $R.a$, $R.b$ and $R.c$ respectively, where all the columns are sorted in the same order and stored in separate files. Let σ_1 be the most selective and σ_3 be the least selective predicate.

- **Early materialisation:** This is the technique of stitching columns into partial tuples as early as possible, i.e. during column scan in query processing [28, 29]. The early materialisation strategy would process the query above as follows:
  - Read $R.a$, $R.b$ and $R.c$ from disk and stitch them together to create a row-store style tuple $< R.a,\ R.b,\ R.c >$.
  - Apply σ_1, σ_2 and σ_3 to get the final results.

  In some cases, columns must be accessed more than once in a query plan. For example, the case where a column is accessed once to get positions of the column that match a predicate and again downstream in the query plan for its values. In cases where the positions that match the predicate cannot be answered directly from an index, the column values must be accessed twice. If the query is properly pipelined, the re-access will not have a disk cost component (the disk block will still be in the buffer cache); however, a CPU cost is incurred when scanning through the block to

find the set of values corresponding to a given set of positions. This cost will be even higher if the positions are not in sorted order. However, for the early materialization strategy, as soon as a column is accessed, its values are added to the tuple being constructed and the column will not need to be re-accessed.

- **Late materialisation:** This is the technique of fetching columns on demand for each operator in the query plan [28, 29]. For example, in a query with multiple joins, when evaluating any join, only the columns needed for that join are fetched. The output from each join is only the set of matching row ids, which are used to fetch other columns as needed for the remaining operators in the query. The late materialisation strategy would process the query above as follows:

  - Access $R.a$ and output the list of positions satisfying σ_1. Similarly, access $R.b$ and $R.c$ and output the list of positions satisfying σ_2 and σ_3 respectively.
  - Use position-wise AND operations to intersect the position lists.
  - Re-access $R.a$, $R.b$, and $R.c$ and extract the values of the records that satisfy all predicates and stitch them together to create a row-store style tuple $< R.a, R.b, R.c >$.

  The late materialisation approach can potentially be more CPU-efficient because it requires fewer intermediate tuples to be stitched together (which is an expensive operation as it can be considered as a join on position) and position lists are a small, very compressible data structure that can be operated on directly with very little overhead. However, it requires re-scanning the base columns to form tuples, which can be a slow process.

2.1.3 Features of Column-Stores

The features and concepts of column-store are similar to those of early research on vertical partitioning [5, 59, 60]; however, column-stores include many architectural features beyond those proposed in previous studies. In addition, they are designed to maximise the performance on analytical workloads on modern hardware. Below, we briefly describe seven of the most important features.

1. **Virtual IDs:** The easiest way to identify every value in a column in a column-store is to attach an identifier (e.g. surrogate key [5], object-identifier [36] or recordIDs [4]). The explicit inclusion of this key (as in Figure 2.1 (c)) increases the size of data on the disk, and reduces I/O efficiency; therefore, using the position offset as a virtual

identifier (as in Figure 2.1 (b)) can avoid the problem [7]. The idea is to store each attribute as an array, and each record can be stored in the same (array) position across all columns of a table.

2. **Late Materialisation:** This refers to the postponement of the construction of tuples into wider tuples unless required [28]. This means that columns are not only stored individually; they are also processed individually whenever possible which helps to improve memory bandwidth efficiency [24, 28].

3. **Block-oriented and vectorised processing:** Rather than using a conventional tuple-at-a-time iterator, column-stores use cache-sized blocks of tuples and operate on multiple values simultaneously which improves the cache utilisation and CPU efficiency [17]. Also, the use of vectorised CPU instructions on these blocks helps improve the throughput [25].

4. **Column-specific compression:** Column-stores use a suitable compression method on each of the columns to reduce the total size of the data on the disk. By storing data from the same attribute which has the same data type, column-stores can obtain good compression ratios even with simple compression techniques [25, 26].

5. **Direct operation on compressed data:** Many column-stores delay decompressing the data until it is required (which usually is at the time of producing the result). Late materialisation along with direct operation on compressed data can significantly improve memory bandwidth utilisation which is often the primary bottleneck [28].

6. **Database cracking and adaptive indexing:** The column-store is designed to store multiple copies of each column sorted by attributes heavily used in an application's query workload such that the tuples are already appropriately organised in multiple different orders across the various columns [7]. Database cracking avoids this prior sorting of columns; instead, it adaptively and incrementally sorts (index) columns as a by-product of query processing. Each query partially reorganises the columns it accessed to allow future queries to access data faster [27]. Also, the overhead of tuple reconstruction for multi-attribute queries can be dealt with using partial sideways cracking [24] that minimises the tuple reconstruction in a self-organising way.

7. **Efficient loading architectures:** One of the main concerns with column-stores is that they may be slower to load and update than row-stores because data is compressed and each column must be written separately [7]. To curtail the length

of time required for loading and updating, many column-stores write data into an uncompressed and write-optimised buffer and then flush data periodically in large and compressed batches to the "write-store" [64].

2.1.4 Research Gaps

A common approach to analytic queries is to use joins for filtering [28]. Late materialisation provides significant performance advantages for such queries since, after each join, there are fewer row ids to fetch from memory or disk. For large tables, this results in less random memory access or significant disk I/O savings. However, this benefit is gained at the cost of complexity. The research gaps in materialisation strategies are threefold:

- Tracking which columns to materialise at what point involves a lot of bookkeeping in the optimiser and execution engine, and has to be accounted for in the cost model during query optimization.
- It is difficult to implement partial aggregation before joins [29], because the optimiser needs to weigh the benefit of cardinality reduction (provided by the pushed-down aggregation operation) against the cost of fetching extra columns needed for aggregation.
- It is difficult to accurately estimate the number of distinct values of a number of columns, especially in the presence of predicates, which makes it difficult for the optimiser to make the correct choice.

2.2 Parallel Processing

Parallel processing is the process of taking a large task that may take a long time to complete and dividing it into smaller sub-tasks, each of which can be worked on separately but simultaneously by different processors in a parallel system as shown in Figure 2.2 [9]. This section outlines the motivation behind parallel processing, together with its advantages and the strategies used.

Motivation: Digital sources of data have seen an immense growth in the past decade [65]. For instance, in 2008 - 2009, the digital universe grew by 62% or 0.8 zettabytes. At the time of writing, the digital universe was expected to grow 44 times more than in 2009, reaching approximately 35 zettabytes. Although there has been some debate regarding the accuracy of these figures, the point is that this growth strains the ability of a single-processor system to handle all the processing required by the workload.

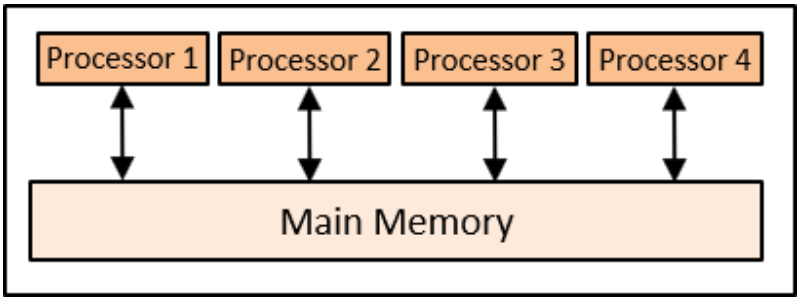


Figure 2.2: A parallel system: each processor has a direct access to a shared memory.

The performance of processors with a calculating engine was anticipated to double every two years in accordance with Moore's law [66]. But, they are achieving only a low percentage increase [9, 67]. Also, it is expected that mechanical delays will restrict the advancement of disk access time or disk throughput. Therefore, it is becoming increasingly difficult to use the disk capacity to its full potential because disk input/output (I/O) has become a bottleneck as a result of skewed processing speed and disk throughput. This inevitable I/O bottleneck is the motivation for the research on parallel processing.

Objective: It is crucial for a variety of applications, for instance, that traditional data warehouse applications be able to query a large set of data in an efficient manner [68]. The main reason for parallel processing is to improve performance, which can be measured in two ways:

i. *Throughput:* This is the number of tasks that can be completed within a given time interval.

ii. *Response Time:* This is the amount of time taken to complete a single task from the time it was submitted.

A system that processes many small transactions can improve throughput by processing many transactions in parallel, or a system that processes large transactions can improve response time as well as throughput by performing sub-tasks of each transaction in parallel. Parallel processing improves query processing and I/O speed by using multiple processors and disks in parallel. Multiple processors work simultaneously on several parts of a task to complete it quickly [9]. So, parallel processing enables the processing of a huge amount of data within a short period of time, and also performs complex computations in real time.

Obstacles: Single-threaded query execution cannot handle all the processing load because of the massive increase in data size for processing [40]. An alternative is to use multiple processors in parallel as data can be processed more quickly than with a single processor. However, the parallelizing of sequential algorithms and queries for processing are not easy. The obstacles in parallel processing are as follows:

1. **Start up:** In parallel processing, the start-up time may overshadow the actual processing time, especially if thousands of processes must be initiated [43, 68]. Even when there is a small number of parallel processes to be started, if the actual processing time is very short, the start-up cost may dominate the overall processing time [43].

2. **Consolidation cost:** Parallel processing normally starts with breaking up the main task into multiple sub-tasks, each of which is carried out by a different processor [9]. After these sub-tasks have been completed, it is necessary to consolidate the results produced by each sub-task for presentation to the user. Since the consolidation process is usually carried out by a single processor, normally by the host processor, no parallelism is applied, and consequently this affects the speed-up of the overall process [9, 43, 45, 68].

3. **Interference Cost:** Since processes executing in a parallel system often access shared resources, a slowdown may result from the interference of each new process as it competes with existing processes for commonly-held resources [9, 44, 45].

4. **Communication Cost:** Very often, one process may have to communicate with other processes. In a synchronised environment, the process wanting to communicate with others may be forced to wait for other processes to be ready for communication [44, 45, 57]. This waiting time may affect the whole process, as some tasks are idle waiting for other tasks [9].

5. **Complexity:** Parallel processing requires totally new algorithms and parallelisation strategies for efficient query processing [43]. In addition, troubleshooting and diagnostic issues are more complex and challenging in parallel processing.

Strategies: Parallel processing of algorithms and queries can be achieved by using an appropriate partitioning technique. Partitioning refers to the distribution of either workload or data to all available processors such that all processors can execute tasks simultaneously [9]. An algorithm or a query can be executed in parallel using two partition strategies: 1. data parallelism and 2. task parallelism.

1. **Data parallelism** refers to the strategy of partitioning and distributing data to available processors such that all processors can execute tasks simultaneously [9]. The same operation is applied to a different dataset. The processing of a query is boosted by parallelizing the execution of each individual operation (such as parallel sorting and searching).

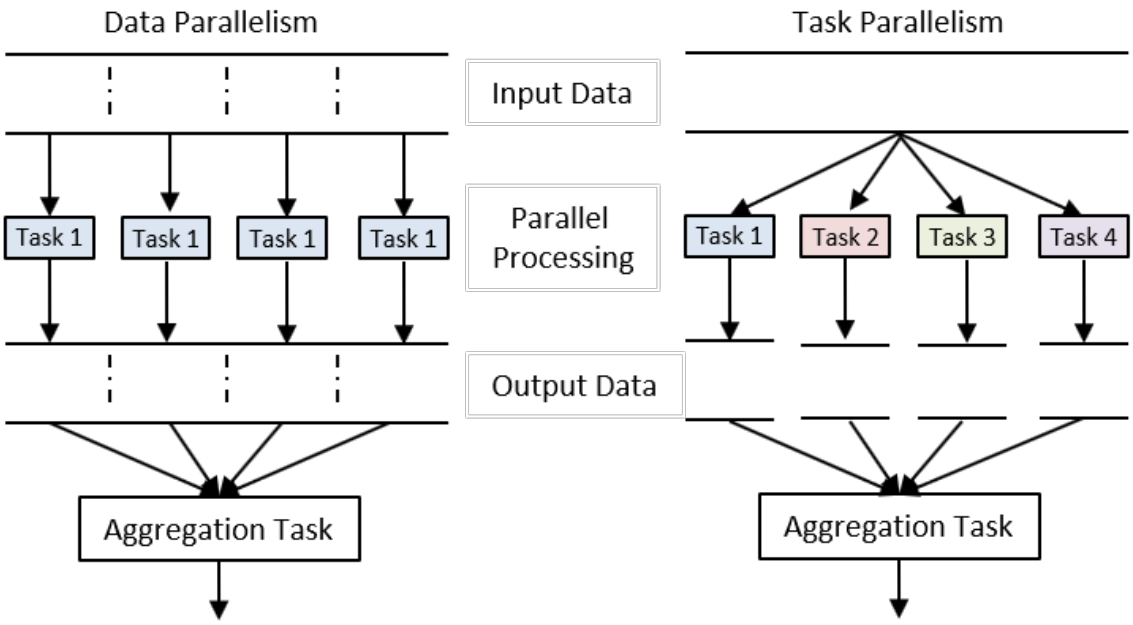


Figure 2.3: Data Parallelism vs Task Parallelism

Let us consider an example. We want to sum the contents of an array of size N. For a single-core system, one thread would sum the elements [0] . . . [N-1]. For a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2-1] and thread B, running on core 1, could sum the elements [N/2] . . . [N-1]. Therefore, the two threads would be running in parallel on separate computing cores. In Figure 2.3, the same task, Task 1 is performed on different chunks of data by multiple processors.

2. **Task parallelism** refers to the strategy of distributing independent task to all available processors such that each processor can execute a task simultaneously with other processors [9]. Each task operates on the same data. The processing of a query is boosted by executing in parallel different operations in a query expression (such as simultaneous sorting and searching).

   Again, consider the example above. An example of task parallelism might involve two threads, each performing a unique statistical operation (e.g. sum and multiplication) on the array of elements. The threads are operating in parallel on separate computing cores, but each of them is performing a unique operation. In Figure 2.3, the multiple tasks are performed on the same data by multiple processors.

Parallel join algorithms follow task parallelism or data parallelism or a combination of both to achieve more parallelism.

2.2.1 Parallel Hash Joins

Main memory hash joins have been the focus of recent research, particularly in the domain of column-stores. The ground-breaking work of MonetDB led to a Radix Hash Join [34,69] that aimed at overcoming the bottleneck of cache and Translation Lookaside Buffer (TLB) misses. Based on the principles of radix hash join, Partitioning Join [35] pushed the boundaries of hash join performance by introducing parallel processing of hash joins 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 single instruction, multiple data (SIMD) instructions and limited per-core bandwidth.

The no-partitioning Hash Join [39] was superior in performance to Partitioning Join [35]. No-partitioning hash join was compared with the partitioning hash join and the experiment results show that a no-partitioning hash join outperforms 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. [40] advocated a sort-merge algorithm for the main memory parallel joins, while others such as Balkesen et al. [41,42] 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. [41,42] achieved further improvements on partitioning hash join [35] and no-partitioning hash join [39] by improving the cache efficiency of the hash table implementation and adopting a better skew-handling mechanism for parallel partitioning hash join [35]. They showed that tuned parallel radix hash joins exhibits better performance than the no-partitioning hash joins on their experimental hardware which contradicts [39].

Traditional partitioned joins [34, 35, 41], as well as non-partitioned joins [39], have serious limitations in terms of I/Os and parallelism, respectively [54]. A memory-efficient Concise Hash Table (CHT) [54] and Concise Array Table (CAT) [54] was used to develop an equijoin algorithm that significantly reduces memory consumption compared to leading in-memory join algorithms such as [39,41]. The experimental result showed that it could reduce the memory consumption by one to three orders of magnitude with competitive performance. This equijoin scanned the outer table in a pipelined fashion and benefited from the build-side partitioning even when the probe side was non-partitioned.

MCJoin [70] was designed to perform efficiently under tight memory constraints. It utilised a block nested loop approach, performing hash join between blocks that uses a compact hash table to reduce the memory footprint. They improved the algorithm and

implemented the parallel version as PaMeCo Join [37] while being mindful of operating under a tight memory constraint. In scenarios without memory constraints, the algorithm offered performaned competitively against other contemporary non-hardware tuned hash joins, whereas in memory-constrained scenarios, it was three times faster than state-of-the-art memory-constrained hash joins.

Jha et al. [71] compared the modified variants of no-partitioning hash join [39] and parallel partitioning hash join [35] 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.

Summary: Joins are an enduring performance challenge for query processors. A significant number of research studies on column joins conclude that the hash join is more efficient than the sort-merge join. Given the overall high-performance of hash joins, and our ability to utilise the nature of hash tables when storing grouping attribute values, we have based our algorithms on the hash join algorithm.

2.2.2 Parallel Star Joins

Star-join consists of one fact table F referencing several dimension tables $D_1, D_2, ..., D_n$. 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 and those that store quantitative information for analysis. 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. Throughout this thesis, we will use the following definitions:

Definition 2.2.1. *D_i has the primary key PK_i that is associated with the foreign key FK_i of F where i is the dimension identification number of D_i.*

Definition 2.2.2. *Fact table has format: $F(fk_1, fk_2, ..., fk_n, m_1, m_2, ..., m_n)$ where fk_i is the value of the foreign key FK_i and m_i is a measure value.*

Definition 2.2.3. *The star-join query might have restrictions CD_i on D_i and CF_i on F.*

Generally, a star join has the following form:

```
SELECT D1.value, D2.value, F.value
FROM D1, D2, F
WHERE D1.pk1 = F.fk1,
AND D2.pk2 = F.fk2
AND CD1
AND CF1;
```

In this section, we outline the works related to Row-Oriented Joins, Materialisation Strategies in Column-Stores and Column-Oriented Joins.

Row-Oriented Joins: Bitmap star join [72] used bitmap indices to perform join and showed that an index lookup in the dimension table could be faster than hash join, whereas, hierarchical physical clustering [73] was proposed as an alternative to the use of indices and aimed at limiting the number of I/O access to the fact table.

Many other techniques such as index union and semi-join reduction plans [74] with bitmap filters have been proposed for efficient execution of star schema joins. Aguilar et al. [23] revisited the star join techniques to analyse the most up-to-date strategy for ad-hoc star join query processing. They proposed the hybrid solution that improved the features of the bitmap star join [72] and the hierarchical physical clustering [73], and showed near-optimal results in multiple use-cases. 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 [38] have also been proposed, demonstrating that the optimisation strategies improved the star join performance. Zhuhe et al. [75] 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.

Several of the approaches are based on the row-oriented storage architecture and mainly focus on joins rather than grouping and aggregation. All these works have been motivational attempts to developing star join algorithms for column-stores. Our algorithms are inspired by [38, 72, 76], although they have been designed to address queries with join, group and aggregation operations for column-stores.

Column-Oriented Joins: In column-stores, the columns are stored separately. Therefore, even tuple re-construction requires a join between the columns. The easiest way to implement a column-oriented join is to include only those columns in the join predicate as the input. The output of join is a set of pairs of positions in the two input relations for

which the predicate succeeded. For instance, Figure 2.4 depicts the result of a join between ColA (size 5) and ColB (size 4).

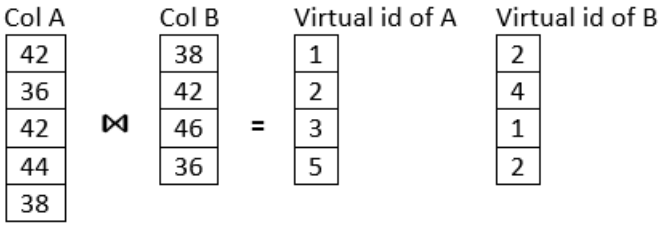


Figure 2.4: Simple column-oriented join

The output positions (virtual ids) may be sorted (virtual id of A) or may not be sorted (virtual id of B). Unsorted output positions are problematic because the extraction of values from a column (e.g. ColB) in this unordered fashion requires multiple random reads of data for each position, causing significant slow-down since random access is much slower than sequential access for most storage devices. Several improvements have been proposed to solve the problem of multiple random reads. One of the solutions is the Jive Join [32,77]. This algorithm allows sequential access to columns at the cost of adding two sorts of join output data. Most of the database systems have implemented fast external sort algorithm; therefore, this join can achieve remarkable performance relative to random access of data.

Additional research in the field has shown that complete sort is not necessary to mitigate random reads during value extraction of join output. Since most storage media are divided into contiguous blocks of storage, and random reads within a block are significantly cheaper than across the blocks, the idea is to partition the data into blocks of storage in which the positions can be found. This idea has been implemented in Radix Join [33] which provides a fast mechanism for performing the partition of column positions into the blocks before the column extraction. It also reorders the intermediate data back to the original join order after the extraction has been completed. However, the joins discussed consider only two columns, whereas the star join queries involve joining more than two columns (in fact we need to join two or more tables) along with operations such as group-by and sort. Therefore, limited research exists in the field of parallel star joins for column-stores and the available works have several limitations. Next, we discuss two main algorithms that are the foundation of this research, and their limitations.

1. **Invisible Join [30]** is an extended work on improving the performance for star joins [72, 74] by taking advantage of the column-oriented layout and rewriting the

predicates to avoid the hash lookups. Next, we briefly describe the Invisible Join phases.

Phase 1: Predicates are applied on dimension tables to build the respective intermediate hash tables.

Phase 2: Fact table columns are joined with their respective dimension table using an intermediate hash table. The result of the join operation is the lists of positions that passed the join predicates. At the end of this phase, we have multiple lists of positions one for each dimension table. These lists are intersected to generate the final list of positions satisfying all the predicates.

Phase 3: According to the final list of positions, the required fact table columns are *re-scanned* to construct the final result.

We have identified three problems with this algorithm:

i. It has the performance bottleneck of multi-pass scan for column processing and increases 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.

Yuan et al. [78] 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, Zhou et al. [79] 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 based on Invisible Join [30] and modified to work on GPU which is outside of the scope of this research.

Performing group-by aggregation when a join of two or more tables is involved in the query is the cornerstone of almost all star join queries. However, Invisible Join is not designed for grouping and aggregation operations. It uses a traditional approach, Hash Join/ Hash Group-By, where a column-oriented hash join (using bloom filters,

hash table build and probe) is followed by row-oriented hash group-by (using serial aggregation). There are two traditional approaches that can be used to improve the performance of group-by aggregation queries.

a. **Materialised View:** Materialised view can be used to serve the aggregation query [80]. However, this incurs the overhead of maintaining the materialised view [81]. In addition, it cannot support the queries that are not able to leverage the materialised view.

b. **De-normalising the schema:** In traditional join processing, the group-by and aggregation is performed after join [9]. Thus, another common approach involves the de-normalisation of the schema so that the join can be converted to a scan. However, this approach incurs additional storage cost as the data is nested and has repeated fields. Nested and repeated fields can maintain relationships without the performance impact of preserving a relational (normalised) schema. The storage savings from normalised data are less of a concern in modern systems. Increases in storage costs are worth the performance gains from denormalising data. [81].

2. **In-Memory Aggregation [16]** is optimised for aggregation over joins for star join queries 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. Next, we briefly describe the In-Memory Aggregation phases.

   Phase 1: Dimension tables 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 (DGK).

   Phase 2: The key vectors are used to create an additional multi-dimensional array known as *In-Memory Accumulator (IMA)*. Each dimension of IMA will have as many entries as the number of non-zero grouping keys corresponding to that dimension.

   Phase 3: For each entry in the fact table that matches join condition based on the key vector entries for the dimension values, a corresponding aggregate value is added to an appropriate cell in IMA. In the end, IMA receives the results of the aggregation. In this phase, the processing is done row-by-row which is a disadvantage as the data is not processed in tight loops. This results in considerable performance deterioration on modern hardware [36].

   We have identified three problems with this algorithm:

   i. This algorithm is efficient only if the IMA does not become too large.

ii. 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.

iii. The design of this algorithm is not ideal for parallel implementation.

2.2.3 Research Gaps

The research gaps in parallel star joins are three-folds:

- Much of the research on efficient column-oriented joins assumes an unlimited reserve of main memory and focuses on minimising the total execution time [7,33,35–37,70]. However, the main memory will eventually be exhausted when the input tables or intermediate results exceed the available memory because of the increase in the amount of data. In addition, from a user's perspective, the ideal is to generate the first few results quickly with minimal response time so that the data processing can begin immediately.

- Mainstream data warehouses today hold several terabytes of data with table sizes passing the one billion row threshold [38]. The recent works on efficient parallel join algorithms show that carefully-tuned join implementations demonstrate good performance [35, 39–43]. However, not all star join algorithms have been designed for parallel implementation.

- Performing group-by aggregation with join over two or more tables is the cornerstone of almost all star join queries [16]. However, most research [30,38,72,76,78] focuses on join rather than on grouping and aggregation, and the current technique optimised for aggregation than joins has a number of limitations.

2.3 Distributed Processing

Distributed processing is a loosely-coupled form of parallel processing where several computers are used, instead of just one main computer, to process data [9]. The computers in a distributed processing system can be physically located at the same place or close together, connected via a Local Area Network (LAN) or a Wide Area Network (WAN) as shown in Figure 2.5. Most distributed processing systems contain sophisticated software that detects idle CPUs on the network and parcels out programs to utilise them [43, 52].

Advantages: Parallel processing helps to process large amounts of data in a short period of time, and can perform complex computation in real time. In addition to these

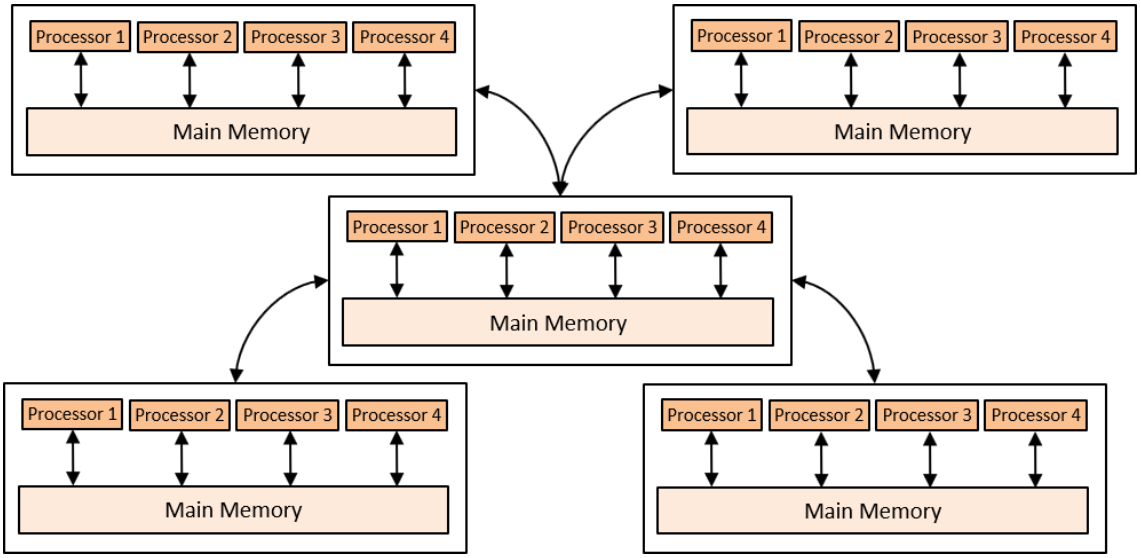


Figure 2.5: A distributed system: each computer has its own local memory, and information is exchanged by passing messages from one computer to another by using local area network (LAN).

benefits of parallel processing, further benefits are achieved using distributed processing as below:

1. **Cost Effectiveness:** Previously, companies invested in expensive mainframe and super computers to function as a centralised server. These super computers are very expensive compared to commodity computers. In addition, there are cases where the use of a single computer would be possible in principle, but the use of distributed processing is beneficial for practical reasons. For instance, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several commodity computers rather than a single super computer.

2. **Fault Tolerance:** Hardware glitches and software anomalies can cause a super computer to malfunction and fail, resulting in a complete system breakdown. Distributed processing is more fault-tolerant as there is no single point of failure [52]. A glitch in any one computer does not impact the network since another computer takes over its processing capability. Faulty computers can be quickly isolated and repaired [82].

3. **Scalability:** A distributed system may be easier to expand and manage than a monolithic super computer [52, 83]. For example, adding more nodes or computers to the network increases processing power and overall system capability, while removing computers from the network decreases processing power [82].

4. **Efficiency:** Distributed processing works on the principle that a job gets done faster if multiple machines are handling it in parallel, or synchronously [52]. For instance, complicated statistical problems are broken into modules and allocated to different machines where they are processed simultaneously. This significantly reduces processing time and improves performance [82].

Disadvantages: Some of the disadvantages of distributed processing are:

1. **Complexity:** There is an added complexity to ensure proper co-ordination among the sites involved in distributed processing. This increased complexity takes various forms:

   a) Software Development Cost: Distributed processing systems are difficult to deploy, maintain and troubleshoot/debug than the centralised servers. The increased complexity is not only limited to the hardware; distributed systems also need software capable of handling the security and communications.

   b) Increased Processing Overhead: The exchange of information and additional computation required to achieve inter site co-ordination are overheads that are not incurred in centralised systems.

2. **Security Concerns:** Data access can be controlled easily in centralised servers, but it is difficult to manage security of distributed processing systems. The communication between the computers take place via the network. However, not only does the network have to be secured; we also need to control the replication of data in multiple locations.

Summary: Distributed processing improves processing and analysis of big data by combining the power of multiple machines. It is much more scalable and allows other computers to be added easily to cope with the demands of an increasing workload. Although distributed computing has its own disadvantages, it offers unmatched scalability, better overall performance and greater reliability, which makes it a better solution for dealing with high workloads and big data.

2.3.1 Distributed Star Joins

With on-demand hardware and the advent of convenient parallel frameworks, distributed processing has been extensively applied to implement decision support systems [84, 85]. Several algorithms have been proposed to solve star joins in a distributed environment. In

this section, we outline the works related to Row-Oriented Joins and Column-Oriented Joins.

Row-Oriented Joins: Datta et al. [86] proposed a parallel star join algorithm based on the vertical partitioning of data in a distributed environment. Aguilar et al. [76] 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 [87, 88] have proposed the use of bloom filters [89] for map-side joins. The use of a bloom filter is based on an allowable error. It is known that an m bits filter has a false positive rate of $p = (1 - e^{-kn/m})^k$, where n is the number of tuples, and k is the number of hash functions. An error rate of $p = 0.001$ (0.1% false positives) requires k = 10 hash functions to be executed against every tuple from the fact table to decide whether or not it should be filtered out [89]. This process becomes computationally expensive with a large amount of data.

Purdilua et al. [53] 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. [90] 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 [46, 91] 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.

Column-Oriented Joins: In order to effectively deal with star join, Concurrent Join [92] and Scatter-Gather Merge [87] were proposed. In concurrent join, a query is split into sub-queries over multiple data-sets. These sub-queries are executed concurrently by distinct map-reduce phases. In the reduce phase, the temporary results are joined. Scatter-Gather Merge join partitions the fact table according to the dimension table, join for each partition, and then the intermediate results are joined. The by-product of these algorithms is a large intermediate result and high I/O cost. In addition, the intermediate result needs to be redistributed which produces a great deal of data replication.

HdBmp Join [93] is a star join method used for column-oriented data stores in MapReduce environments. This join used the *HdBmp Index* which can filter out most of the unnecessary tuples in tables, thereby greatly reducing the network overload. Multi-

Fragment-Replication Join (MFRJ) and MapReduce-Invisible Join (MRJ) [94] are two cache-conscious algorithms in the MapReduce environment that avoids fact table data movement. The fact table is partitioned into several column groups for cache optimization. As discussed previously, in column-stores, each column is stored separately. MFRJ is not appropriate as it has frequent cache misses from one column to another [94]. Also, it has a critical memory requirement that not all nodes can meet. Hence, these researchers [94] acknowledged that MRJ works well on column-oriented architecture and a small memory node. However, it is based on Invisible Join [30] and therefore, the problems discussed in Section 2.2.2 remain. In addition, these algorithms deal with join operations only, and not group-by and aggregation operations.

2.3.2 Research Gaps

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 [48] for batch processing [49, 50]. The research gaps regarding distributed star joins are threefold:

- Although Spark facilitates joins and group-by using Resilient Distributed Datasets (RDDs) [51], Spark SQL [52] and Spark DataFrame [52] operations, it can process only two tables at a time, inducing multiple scans of data for star joins and requiring one or two map-reduce iterations per join [53]. This means that the analytical queries will need $n-1$ or $2*(n-1)$ map-reduce iterations where n is the number of tables used by the query.

- Apache Spark produces excessive disk access and network communication because of cross-commutation between the worker nodes. Unnecessary disk access is often the result of disk spill where the data is spilled into the disk due to an overflowing memory buffer.

- Excessive shuffling of records not only significantly increases the network communication cost, but also prevents further processing of the algorithm [46]. Therefore, naive Spark implementation fails to handle the issues such as multiple scans of data, excessive network communication and disk spill.

2.4 Summary

We reviewed previous studies on or related to parallel hash join, parallel star joins and distributed star joins. A summary of the most relevant research is presented in Table 2.1. The fields of parallel and distributed star joins are active areas of research and both areas

Table 2.1: A summary of most relevant research work. $\star$ represents star join, $\mp$ represents column join, $\rightarrow$ represents single join mode, $\Rightarrow$ represents parallel join mode and $\ll$ represents distributed join mode.

			Join Type		Join Mode		
Work	**Year**	**Publication**	$\star$	$\mp$	$\rightarrow$	$\Rightarrow$	$\ll$
Weininger [74]	2002	SIGMOD		✓	✓		
Zukowski et al. [69]	2005	IEEE		✓		✓	
Abadi et al. [30]	2008	VLDB	✓		✓	✓	
Tsirogiannis et al. [77]	2009	VLDB		✓	✓		
Kim et al. [35]	2009	VLDB		✓		✓	
Balnas et al. [39]	2011	SIGMOD		✓		✓	
Begley et al. [70]	2011	SIGMOD		✓	✓		
Albutiu et al. [40]	2012	VLDB		✓		✓	
Balkensen et al. [41]	2013	ICDE		✓		✓	
Zhang et al. [88]	2013	IGDC	✓			✓	✓
Begley et al. [37]	2016	IS		✓	✓	✓	
Brito et al. [46]	2016	ICCS	✓			✓	✓
Cheng et al. [95]	2017	JPDC		✓			✓
Chavan et al. [16]	2018	ICDE	✓		✓	✓	

have been investigated in this research project. Compared to the state-of-the art approaches, we conclude that this thesis is distinctive from them as it offers the following innovations:

- For parallel star joins: 1) We extend the concise array table [54] and name it *multi-attribute array table (MAAT)*. It handles multiple attributes and facilitates probing required in the join algorithm. 2) We propose a novel materialisation strategy based on MAAT known as *Progressive Materialisation*. To the best of our knowledge, there is no such strategy proposed for the column-stores. 3) We propose a new progressive parallel star join algorithm for the main memory column-stores named *Nimble Join* that is significantly better than its competing column-store join algorithm. It uses a multi-attribute array table to hold the attributes required in join query processing that facilitates progressive materialisation. 4) We propose *an analytical model* to understand and predict the query performance of the Nimble Join. The accuracy of the model has been verified by comprehensive experiments with different hardware parameters.

- For parallel star joins with group-by and aggregation: 1) We propose a novel approach to perform group-by and aggregation operations influenced by the concept of *trie* or *prefix tree* [55] 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, to date, no such technique has been proposed for column-stores. 2) We propose a new parallel star group join and aggregation for in-memory column-stores named *ATrie Group Join (ATGJ)* that is significantly faster than its competing column-store join algorithm. ATGJ is a variation of Hash Join/Hash Group-By that uses both techniques but solves the problem of grouping and aggregating data using a rather novel approach with the help of the *ATrie*. 3) We propose *an analytical model* to understand and predict the query performance of ATGJ. The model accuracy is verified by comprehensive experiments with different hardware parameters.

- For distributed star joins: 1) We present a new optimisation technique for efficient search in the hash table. The key idea is to use Robin Hood hashing with the upper limit on the number of probes which is implemented in *Fast Hash Table (FHT)*. 2) 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 the ATrie. 3) 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 prominent approaches. 4) We propose *an analytical model* to understand and predict the query performance of DATGJ. The model accuracy has been verified by comprehensive experiments with different hardware parameters.

Chapter 3

Research Methodology

In this chapter, we discuss the research methodology adopted for the project, and provide an overview of the research method and evaluation.

3.1 Research Method

The objective of this PhD project was to design, develop and implement new parallel join algorithms for column-stores, and to develop cost models for them. Therefore, Design Science Methodology [1, 96] was considered the most appropriate for this project.

Design science is an outcome-based information technology research methodology that is concerned with producing an artifact to achieve the desired goal. The research development process follows DSRM Process Model [1], which is an iterative process shown in Figure 3.1. This model involves (in order) identifying the problem and motivation, defining objectives of a solution, designing and developing an artifact, demonstrating a suitable context to solve a problem, and evaluating the artifact from which it can iterate back to design and development or proceed to communicating the research.

Based on the research framework in information technology [97, p. 255-258], there are four main research outputs:

- *Constructs* or concepts form the vocabulary of the problem domain used to describe problems and their solutions.
- A *model* is a set of propositions, statements or models that expresses the relationships between the constructs.
- A *method* is a set of steps (an algorithm or guideline) that are used to perform a task, and is based on a set of underlying constructs and a model of the solution.

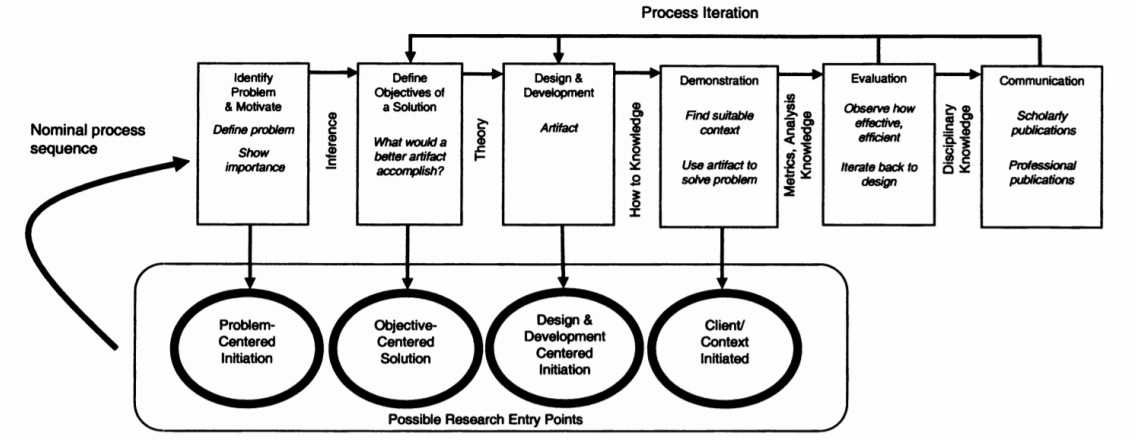


Figure 3.1: DSRM Process Model [1, p. 54]

- An *instantiation* is the realisation of an artifact in its environment.

In this thesis, three research outputs have been built (i.e. model, method, and instantiation) and the instantiation has been evaluated, theorised and justified.

The details of each of the research output are as below:

- Model: The models take the form of cost models delivered with the algorithms.

- Methods: The methods take the form of proposed column-oriented join and group-by algorithms.

- Instantiation: The instantiation is the implementation of column-oriented join and group-by algorithms, and is evaluated using controlled experiments.

Given an artifact whose performance has been evaluated, it is important to determine why and how the artifact worked and did not work. Therefore, we theorised and justified assumptions about those artifacts. The summary of research activities and outputs are demonstrated in Table 3.1.

Table 3.1: Research Activities and Outputs

	Build	**Evaluate**	**Theorize**	**Justify**
Constructs				
Model	✓			
Method	✓			
Instantiation	✓	✓	✓	✓

3.2 Experimental Evaluation

The research used **Controlled Experiment** evaluation method as described in the Design Evaluation Methods [96, p. 86] to evaluate the artifacts. The experiments were performed using Star Schema Benchmark (SSBM) [98] which is described next.

3.2.1 Star Schema Benchmark (SSBM)

The Star Schema Benchmark (SSBM) [98] is used widely in various data warehousing research studies [16, 30, 75, 78, 79, 94]. It is derived from TPC-H[1] but consists of fewer queries and has less stringent requirements on what forms of tuning are and are not allowed. Sanchez [99] 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 [98]. This uniform distribution is facilitated by a tool called SSB-DBGEN that makes populating the benchmark data easier and enables 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 [99]. To generate all SSBM tables we used:

```
dbgen -s <n> -T a -v

where -s <n> -- set Scale Factor (SF) to <n>
- T a -- generate all SSBM tables
-v -- enable VERBOSE mode
```

Schema: The benchmark consists of a single fact table, the `LINEORDER` table, that combines the `LINEITEM` and `ORDERS` table of TPC-H. This is a 17-column table with information about individual orders, with a composite primary key consisting of the `ORDERKEY` and `LINENUMBER` attributes. Other attributes in the `LINEORDER` table include foreign key references to the `CUSTOMER`, `PART`, `SUPPLIER`, and `DATE` tables as well as attributes of each order, including its priority, quantity, price and discount. The dimension table contains information about their expected respective entities. Figure 3.2 shows the schema of the tables.

Queries: The SSBM consists of thirteen queries divided into four categories, or "flights":

[1] http://www.tpc.org/tpch

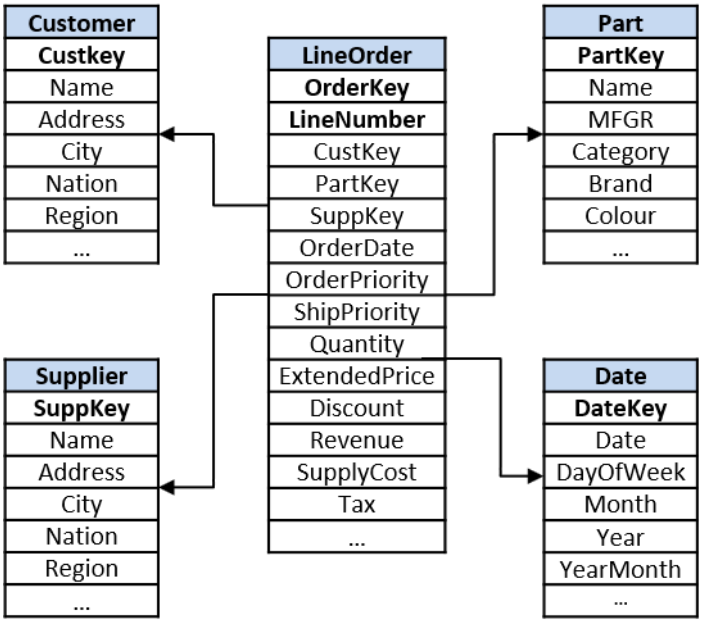


Figure 3.2: Schema of SSBM benchmark

- **Flight 1** contains three queries. Queries have a restriction on one dimension attribute, as well as the `DISCOUNT` and `QUANTITY` columns of the `LINEORDER` table. Queries measure the gain in revenue (the product of `EXTENDEDPRICE` and `DISCOUNT`) that would be achieved if various levels of discount were eliminated for various order quantities in a given year. The `LINEORDER` selectivity for the three queries are 1.9×10^{-2}, 6.5×10^{-4}, and 7.5×10^{-5} respectively.

- **Flight 2** contains three queries. Queries have a restriction on two of the dimension attributes and compute the revenue for particular product classes in particular regions, grouped by product class and year. The `LINEORDER` selectivity for the three queries are 8.0×10^{-3}, 1.6×10^{-3}, and 2.0×10^{-4} respectively.

- **Flight 3** consists of four queries, with a restriction on three dimensions. Queries compute the revenue in a particular region over a time period, grouped by customer nation, supplier nation, and year. The `LINEORDER` selectivity for the four queries are 3.4×10^{-2}, 1.4×10^{-3}, 5.5×10^{-5}, and 7.6×10^{-7} respectively.

- **Flight 4** consists of three queries. Queries restrict on three of the dimension columns and compute profit (`REVENUE - SUPPLYCOST`) grouped by year, nation, and category for query 1; and for queries 2 and 3, region and category. The `LINEORDER` selectivity for the three queries is 1.6×10^{-2}, 4.5×10^{-3}, and 9.1×10^{-5} respectively.

As with TPC-H, this benchmark provides a base "Scale Factor (SF)" to scale the size of the data. The size of each of the tables is defined according to this scale factor. 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 the PARTS table, where the data is scaled logarithmically [98]. Table 3.3 summarises the major characteristics of the SSBM queries. SSBM query definitions are available in the Appendix A. Let us consider the Query 3.1 (shown below) from SSBM.

```
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 1992 and 1997. We will use this query as a running example to discuss the phases of the algorithms proposed in this thesis.

3.3 Analytical Evaluation

Analytical models or cost models are sets of parametric cost equations or formulas that are used to calculate the elapsed time of a query using a specific parallel algorithm for processing. A cost equation comprises variables, which are substituted with specific values at runtime of the query. These variables denote the cost components of the parallel query processing. In this section, we discuss the motivation to create cost models and introduce our modelling methodology used to create the cost models and predict the cost of a join operation.

3.3.1 Motivation

New algorithms are being developed to offer novel methods and approaches for efficient big data processing. To apply these algorithms in real life, it is very important to know their behaviour under specific conditions depending on factors such as the characteristics of hardware, the number of processors, quantities and features of processing data. It is very expensive, time-consuming and infeasible to analyse behaviour of algorithms by constructing systems with all possible characteristics encountered in real life. Cost models are cheap and time-saving solutions to this problem, allowing the analysis of an algorithm without the need to physically build the system. Although the model cannot fully reflect the real-life situation, it can give some idea of the trends and patterns in algorithm behaviour under different conditions.

Estimating the elapsed time of an algorithm is an important, but not the only way to use cost models. Cost models permit the analysis of the algorithms at different levels of abstraction. This makes it possible to understand the examined algorithm in terms of its individual components in order to find potential problems and bottlenecks. This provides an opportunity to compare the performance of different algorithms conducting the same task under an equivalent set of conditions that can improve our understanding of which algorithm is best suited to a particular condition.

3.3.2 Model Methodology

To construct the cost model, the algorithms have been divided into logical steps, and each step is described by a formula based on the parameters that determine the execution time for this step. Before building the model, all parameters necessary for its construction are specified as shown in Table 3.2. We have followed the approach for constructing the cost model described in [9]. The cost model includes the following components: System Parameters and Data Parameters, Query Parameters, Time Unit Cost and Communication Cost.

- *Data Parameters* includes the number of records in the table ($|F|$) and table size in bytes (F). The number of records is used to describe in-memory processing, which is a record-based procedure. The size of the table in bytes is intended to determine the process of loading and writing data to/from the disk.

  In a parallel processing, the table is fragmented into multiple processors. Therefore, the number of records and actual table size for each table are divided (evenly or skewed) among as many processors as there are in the system. To indicate fragment

Table 3.2: General cost model parameters and notations

Symbol	Description
System and data parameters	
F	Size of the table
$\mid F \mid$	Cardinality of the table
F_i	Size of the i-th table column, i = 1 ... n
$\mid F_i \mid$	Cardinality of the i-th table column
N	Number of processors
P	Page size
H	Hash Table size
Query Parameters	
π_i	Projectivity ratio of the i-th table
σ_i	Selectivity ratio of the i-th table
Time Unit Cost	
IO	Time to read a page from the disk
t_w	Time to write the record to the main memory
t_r	Time to read a record in the main memory
t_d	Time to compute destination
Communication Cost	
m_p	Message protocol cost per page
m_l	Message latency for one page

table size in a particular processor, a subscript is used. For example, F_i indicates the size of the table fragment on processor i . Subsequently, the number of records in table F on processor i is indicated by $|F_i|$. The same notation is applied to both the fact and dimension tables used in a query.

- *System Parameters* include the number of processors (N) used to process the query, data page size (P) and maximum hash table size (H) which fits into the memory. The number of processors determines the amount of information processed by each processor. Since the data is loaded and written to/from disk by pages, the data page size is required to calculate the number of pages of information being processed. The maximum size of the hash table, represented in a number of records, is used to estimate the time required to scan a complete hash table.

- *Query Parameters* define the selectivity ratio (σ_i) and projectivity ratio (π_i). The selectivity ratio is the number of rows in the query output divided by the total number of rows in the table. The projectivity ratio is the number of attributes selected by the query divided by the total number of attributes in the table.

- *Time Unit Cost* are the parameters related to technical characteristics of the system, such as time to read and write page to/from disk (IO), time to read to/from main

memory (t_r), time to write to main memory (t_w) and time to send, receive and calculate a destination for the record that is to be sent from one processor to another (t_d).

- *Communication Cost* works at a page level and includes costs such as message protocol cost (m_p) and message latency cost (m_l). The message protocol cost is the cost associated with the initiation for a message transfer; whereas, message latency is associated with the actual message transfer time.

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

There are many factors such as the cost of start-up, interference and communication (refer Section 2.2) that account for the difference between the estimated time of the model and the time taken by the experiment. When evaluating our model, we define the error rate as

$$error\ rate = \left| \frac{experiment\ time - model\ time}{experiment\ time} \right| \tag{3.1}$$

3.4 Summary

We stated that Design Science Methodology [1] is the research method used for this project. We described the star schema benchmark (SSBM) used to perform experimental evaluation using controlled experiment evaluation method explained in Design Evaluation Methods [96, p. 86]. We also discussed the motivation behind analytical models and described the model methodology used in this thesis.

Table 3.3: 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

Chapter 4

Parallel Star Joins

In this chapter, we focus on the parallel star join for column-stores. First, we introduce the technical challenges of answering star join queries for column-oriented data. Then, we discuss *multi-attribute array table (MAAT)* and propose a novel materialisation strategy known as *Progressive Materialisation*. We use MAAT and progressive materialisation technique to develop a new parallel star join algorithm known as *Nimble Join*. Experiments on SSBM dataset is conducted to verify the efficiency of our proposed solution.

4.1 Overview: Challenges and Solution

Column-stores have gained popularity as a promising physical design alternative to improve query performance in analytical workloads such as those found in data warehouses, decision support and business intelligence applications. In a column-store, information about a logical entity is stored as separate columns in multiple locations on disk [4]. For example, information about a customer such as name, address, phone is stored as separate columns on disk. This data storage model is known as Decomposed Storage Model (DSM) [5]. DSM makes column-stores more I/O efficient for read-only queries as they can read only those columns from the disk that is accessed by the query [4, 6, 7]. With the column-oriented storage architecture, the main challenge is how to execute star join queries that includes two or more tables.

4.1.1 Star Joins

Since the columns are stored separately, even the tuple reconstruction requires a join between the columns. Therefore, the joining of two or more tables will involve additional

operations such as tuple reconstruction, projection, grouping and sorting in addition to joining columns specified in query join condition. Existing join algorithms such as [32–37] provide only a partial solution to process star join queries in column-stores. They include steps on how to join two columns specified in the query join condition but do not include steps to re-construct the tuples required in the query output, filter conditions and operations such as group-by and sort. Invisible Join [30] performs star-join in column-stores that include operations such as tuple reconstruction and grouping. However, the algorithm has the performance bottleneck of multi-pass scan for column processing increasing disk I/O and increased memory consumption with increasing number of tables in the join query as discussed in Section 2.2.2.

4.1.2 Memory and Initial Response

Mainstream data warehouses today hold several terabytes of data with table sizes passing the one billion row threshold [38]. Therefore, the decision support queries need to be processed in parallel to achieve performance improvements. The recent works on efficient parallel join algorithms show that carefully-tuned join implementation demonstrate good performance regardless of the data size [35, 39–43]. However, they assume an unlimited reserve of main memory and focus on minimising the total execution. The main memory is finite and will eventually be exhausted when the input tables or intermediate results exceed the available memory because of the increase in data. In addition, from a user's perspective, it is ideal to generate the first few results quickly with minimal response time so that data processing can begin immediately.

Therefore, an optimal solution is a star join algorithm for column-stores that: 1) has a fast response time, 2) has a fast query execution time, 3) consumes less memory, and 4) operate in parallel.

4.1.3 Technical Contributions

To address the challenges mentioned above, we developed a progressive parallel star join algorithm known as *Nimble Join*. We equipped Nimble Join with an extended version of the concise array table [54] known as *multi-attribute array table (MAAT)*, which facilitates progressive materialisation and offers three main improvements in the algorithm. First, it eliminates the memory consumed to hold intermediate data structures in Invisible Join [30]. It is much thinner than the hash table and packs better into cache lines. Further, it stores only those positions that satisfy the join conditions, which remarkably reduces the memory consumption. Second, it holds the intermediate attributes required

for the join query processing that eliminates multi-pass scans for column processing and significantly lowers the query processing time in our join algorithm. Third, it facilitates the design of Nimble Join such that the join results can be produced faster with the help of progressive materialisation. Experiments show that Nimble Join has 2X faster initial response time, 10% - 25% better execution time, 40% - 50% reduced memory consumption and approximately 50% reduced disk I/O time compared to competing Invisible Join.

In summary, we make the following technical contributions:

1. We extend the concise array table and name it *multi-attribute array table (MAAT)*. It handles multiple attributes and facilitates probing required in the join algorithm.

2. We propose a novel materialisation strategy based on MAAT known as *Progressive Materialisation*. To the best of our knowledge, to date, no such strategy has been proposed for the column-stores.

3. We propose a new progressive parallel star join algorithm for the main memory column-stores known as *Nimble Join* that is significantly better than its competing column-store join algorithm. It uses a multi-attribute array table to hold attributes required in join query processing that facilitates progressive materialisation.

4. We propose *an analytical model* to understand and predict the query performance of the Nimble Join. The model accuracy has been verified by detailed experiments with different hardware parameters.

4.2 Multi-Attribute Array Table (MAAT)

A join operation uses a widely-known data structure such as hash table or its variants to store intermediate results [37, 54, 100]. Hash tables typically stores both *keys* and *payloads* (referred to as *attributes*). Often during the implementation, the size of the hash tables is doubled to handle collisions. For linear probing, this overhead arises from the fill factor whereas, for chaining, the overhead comes from memory fragmentation and pointer storage. In addition, most of the hash tables also round up the number of slots to a power of two, making it unappealing for memory efficient joins [54].

Concise Hash Tables (CHT) and Concise Array Tables (CAT) [54] consume less memory and are used to develop memory-efficient hash joins faster than leading in-memory hash joins such as in [41]. *Multi-Attribute Array Table (MAAT)* is a variant of CAT that consists of two pieces as shown in Figure 4.1:

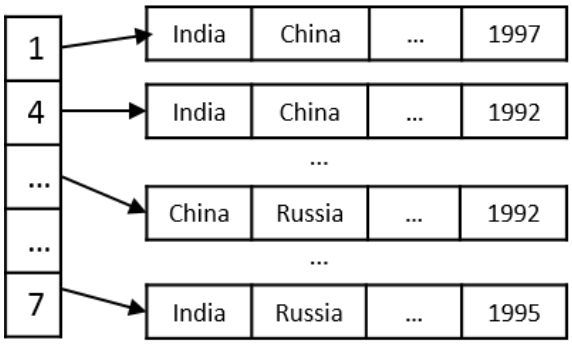


Figure 4.1: Multi-Attribute Array Table

- A *list* of signed integer positions that also serves as a dynamic filter in our join algorithm.

- 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) [101]. This configuration permits the efficient use of the CPU cache and hardware data pre-fetch while simultaneously saving memory space.

4.2.1 Avoiding Collisions

Collision avoidance is based on the observation that the database administrators (DBAs) usually design join keys to produce a dense domain. In fact, it is common practice in the physical modelling of the database to assign primary keys as a serially increasing counter so that we can map these keys directly to the positions of an array. Based on this assumption, in the process of creating tuples progressively, we use key positions instead of primary keys. MAAT embeds a list of successful key positions to speed up final look-ups, which directly maps to the positions in an indexed array. These look-ups have pure data-independent random access (DIRA) pattern - each look-up can be issued before the previous one finishes. N look-ups in MAAT involve only a single DIRA round of N access in an indexed array. Finally, the indexed array has entries only for positions in the position list.

4.2.2 Memory Consumption

Bitmaps have been used to store the position values in order to reduce memory consumption [37, 54, 70]. If max_i is the maximum value of key_i in the set S_{key}, we need NB bits to store

all key_i values within the filter:

$$NB = 2^{log_2(max_i)+1} \tag{4.1}$$

The bits are usually stored as an array of `long` data type, and setting/getting individual bit values are performed using bit operations. Let us consider a simple example. Assume that S_{key} has the following values: $S_{key} = \{1, 5, 6, 7, 9, 14\}$. Then, $max_i = 14$ so $NB = 16$ from equation 4.1. Adding six key_i values to the filter means setting six bits within the bit set as follows: 0100001011100010. To check whether a key_i is stored within the filter, we check a single bit value within the bit set. This is very fast and performed in four-bit operations. However, if the numerical values are big with large gaps between them, the bit set will be very sparse and many bits will be wasted. For instance, assume S_{key} has the following values: $S_{key} = \{1, 68, 11, 45, 12, 442, 110, 4, 11110, 352, 111109, 15, 1234\}$. Then, max_i = 111,109 and $NB = 2^{17}$ i.e. we need 131,072 bits or 16,384 bytes. This is very inefficient as only 13 distinct values are to be encoded within the bit set.

Therefore, instead of using a bitmap, MAAT uses *a list* of signed integer positions that satisfy the predicate. If the output position list is small, we can achieve significant savings in memory consumption using the regular list of signed integers. For instance, the real-world queries have a selectivity of less than 1% (87% of SSBM queries have $< 1\%$ selectivity) [12, 22, 98]. If we have 6000 records, and the selectivity of the query is 1%, i.e. 60 records. So, 1 bit to save 6000 positions equals 6000 bits or 750 bytes (in case of bitmaps the position values could either be true or false depending on the predicate). We do not need all those 6000 bits, as a *false* value in bitmap means the predicate was not satisfied. For the same scenario, assuming a position list of signed integers, i.e. 4 bytes each, this would need only 240 bytes. Therefore, significant amount of memory is saved using a list of signed integer positions in MAAT.

4.2.3 Parallelism in MAAT

To facilitate the parallel processing of algorithms, MAAT implements the *spinlock* mechanism that is used to ensure the atomic insertion and updates of the data items in the indexed array, making it a thread-safe collection. As opposed to other locking mechanisms (such as mutex), *spinlock* is an excellent choice because the critical section in our algorithm such as the insertion or updating of items in MAAT requires a minimal amount of work and *spinlock* works best in such scenarios (for comparison, on average, spinlock was 50 ns faster than POSIX mutex). However, locking the entire MAAT for at the addition or updating of an item is not a good idea. Therefore, we employ item-level locking like the

concept of row-level locking in the database. This approach allows different threads to work on different items in MAAT simultaneously, thereby providing true parallelism.

4.2.4 Evaluation

We conducted an experiment to identify any differences in the performance and memory consumption of Standard-Chain Hash Table (SCHT), Concise Hash Table (CHT), Concise Array Table (CAT) and Multi-attribute Array Table (MAAT). A total of 300,000 records were inserted, and the same amount of data were retrieved and deleted. Each dataset was composed of $< key, value >$ where *key* is hash key and *value* is its associated value. We recorded the memory usages using `System.Diagnostic` class that provided us with the garbage collector's total memory used by the program. The numbers reported are the averages of ten iterations.

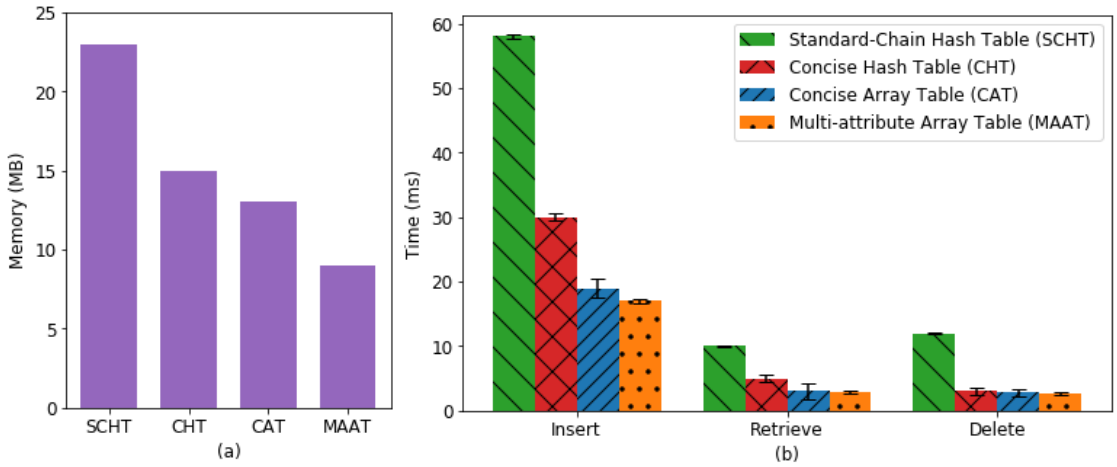


Figure 4.2: (a) Memory usages comparison of various data structures (b) Performance comparison of various data structures to `insert` a new key-value pair and `retrieve` or `delete` the value associated with a key.

Figure 4.2 shows that hash tables are not always the best choice for storing intermediate results regarding memory usages and performance. MAAT offers three main benefits:

1. **Small Memory Footprint:** It uses memory space that is thinner than other data structures (refer Figure 4.2), so they pack better into cache lines.

2. **Faster Look-ups:** It includes a reduced list of positions used to probe against the indexed array that drastically minimises the number of array lookups depending on the join selectivity.

3. It completely obviates the need for slow hash functions.

4.3 Progressive Materialisation

Column-stores vertically partition the database tables and store each column separately on the disk. Although this is a physical modification of storage layout, logically it is still the same as row-stores. The application involving the database, whether column-oriented or row-oriented, treats the interface as row-oriented. At some point in time, column-stores must stitch multiple attributes together to generate tuples and execute the rest of the query plan using row-store operators [28]. This process of adding attributes to generate the result is called *materialisation*. In Section 2.1.2, we discussed two different materialisation strategies for column-stores: *early materialisation (EM)* and *late materialisation (LM)* [28]. Here, we introduce a novel materialisation strategy called *Progressive Materialisation*. The operation and advantages of progressive materialisation is described next.

4.3.1 Operation

Progressive materialisation adopts the notion of late materialisation to push the tuple construction as late as possible, but reduces memory usages, disk access and avoids multiple intermediate data structures by carrying the attribute values required for join processing throughout the query plan. This is made possible by an in-house data structure, MAAT, discussed in Section 4.2.

Consider a simple example: Suppose a query has three selection operators σ_1, σ_2 and σ_3, in columns $R.a$, $R.b$ and $R.c$ respectively, where all columns are sorted in the same order and stored in separate files. Let σ_1 be the most selective and σ_3 be the least selective predicate. *Progressive materialisation* strategy would process the query as follows:

- Access $R.a$ and output attribute values satisfying σ_1 to MAAT.
- Access $R.b$ and $R.c$ and output attribute values satisfying σ_2 and σ_3 respectively to MAAT only if MAAT has attributes stored from the previous step.

4.3.2 Advantages

MAAT holds the join output with records stitched together and the position list of records that satisfy all predicates in the query. The use of MAAT in progressive materialisation offers two benefits over late materialisation:

1. **Avoids multiple access:** For the progressive materialisation strategy, as soon as a column is accessed, the attribute value satisfying the predicate is added to MAAT and the column will not need to be re-accessed. Thus, the fundamental trade-off between

progressive materialisation and late materialisation is the following: while late materialisation enables several performance optimisations such as direct operation on column-oriented compressed data and high value iteration speeds, if the column re-access cost at tuple reconstruction time is high, a performance penalty is incurred.

2. **Saves memory space:** For the late materialisation strategy, the process of tuple construction becomes interesting as soon as predicates are applied to different columns. The result of predicate application are the different subsets of positions for different columns. In many cases, these position representations can be operated on directly without using column values. For example, an AND operation of three single column predicates in the WHERE clause of an SQL query can be performed by applying each predicate separately on its respective column to produce three sets of positions for which the predicate matched. These three position lists can be intersected to create a new position list that contains a list of all positions of tuples that passed every predicate. However, as discussed in Section 4.2.2, we save significant memory space by using progressive materialisation. Instead of maintaining intermediate position lists, the progressive materialisation technique helps MAAT to maintain a single list of signed integers that include positions that satisfy the predicate.

4.4 Nimble Join

Nimble Join is a progressive parallel star join algorithm for column-stores equipped with a multi-attribute array table that facilitates progressive materialisation. In this section, we discuss the details of the join and its implementation in parallel.

4.4.1 Join Processing Method

Nimble Join performs join in three phases: 1. Key Hashing, 2. Probing, and 3. Value Extraction.

1. **Key Hashing:** The predicates are applied to the appropriate dimension table, and the dimension keys and values required by the query are extracted. Using the keys, we create a hash table to test whether a particular key from the fact columns satisfy the predicate. An example of this phase for Query 3.1 is shown in Figure 4.3 (a).

2. **Probing:** The hash tables created in the key hashing phase are used to match keys in the fact table that satisfy the predicate. Each value in the foreign key (FK) column of

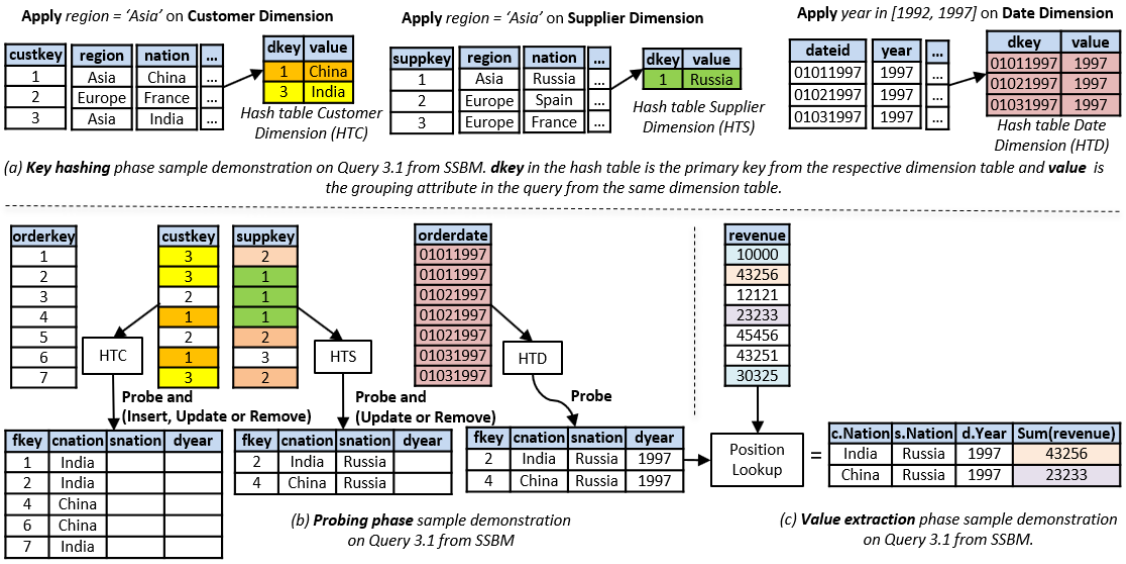


Figure 4.3: Phases of Nimble Join to execute Query 3.1 from SSBM on some sample data.

the fact table is probed against the respective hash table and inserted into MAAT that satisfies the predicate. The probing process continues for all FK columns of the fact table involved in the query. A probe sequence for all 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 MAAT decreases as we remove the items that do not satisfy the predicate. Then, we save the position of items satisfying the predicate in the position list of MAAT. During this process, we output the records satisfying all join conditions producing results much earlier than Invisible Join. The probing phase terminates when we have probed all the FK columns in the query. An example of this phase for Query 3.1 is shown in Figure 4.3 (b).

3. **Value Extraction:** The *position* can be used to look up the remaining required columns in the query (e.g. revenue). The *position* is used as an index of an array to perform the position look-up and direct value extraction from those columns.

   MAAT holds intermediate attributes required for the join query processing, thereby eliminating the re-scanning of fact table columns performed by Invisible Join in phase 3 to reconstruct output tuple. For Query 3.1, in phase 3 of Invisible Join, it requires scanning of column `lorevenue` and re-scanning of columns `locustkey`, `losuppkey` and `loorderdate`. If each scanning takes t time, then the total time spend in disk I/O for Invisible Join is $4t$ whereas t for Nimble Join (scanning `lorevenue`). Finally, arithmetic operations can be performed on the join results to obtain the required aggregation. An example of this phase for Query 3.1 is shown in Figure 4.3 (c).

4.4.2 Parallelizing Nimble Join

In row-stores, the records used to join contain redundant information and the use of proper data partitioning technique can significantly improve the parallelism of join queries [9]. Initially, we attempted to create an algorithm based on a single instruction, multiple threads (SIMT) (i.e. an algorithm using an execution model where single instruction, multiple data (SIMD) is combined with multi-threading), that distributed data equally amongst all the threads.

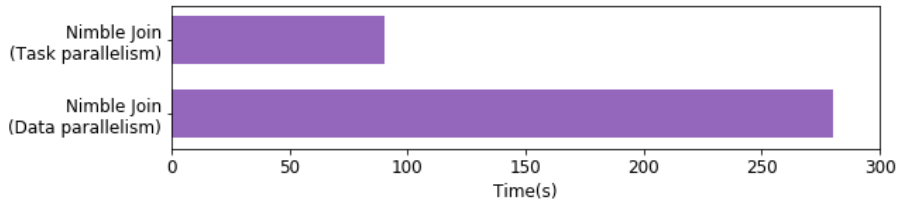


Figure 4.4: Data Parallelism versus Task Parallelism

However, we realised that data parallelism using SIMT is not cost-free. In profiling the code, we noticed that the overhead cost of partitioning data amongst different threads (e.g. setup cost) exceeded the amount of work done in each thread, especially when partitioning dimension table columns because they are significantly smaller than fact columns.

The difference in performance between data parallelism and task parallelism is shown in Figure 4.4. The result presented in the figure is for Query 3.1 in SSBM with SF = 1. The task performed by the algorithm was not sophisticated enough compared to the management of parallelism, thus slowing down the entire process. Therefore, our algorithm favours task parallelism over data parallelism (although this does not mean that it never uses data parallelism).

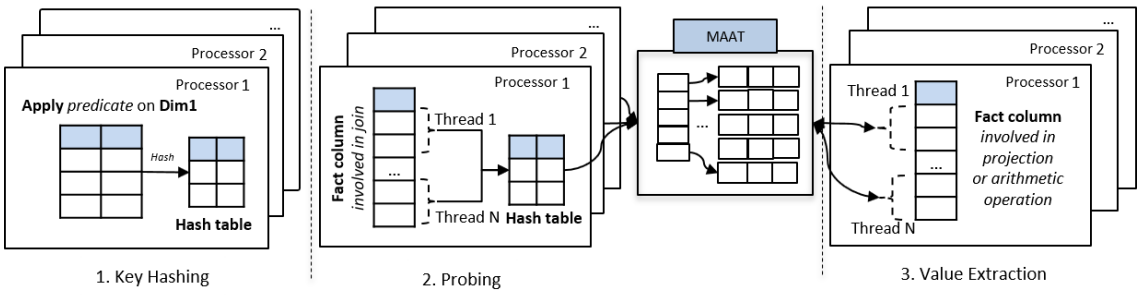


Figure 4.5: Nimble Join parallel processing model

Implementation: Let N be the number of processors (P) that will be used in the join query execution (i.e. $P_1, P_2, ..., P_N$). The maximum number of processors that are activated depends on the number of processors available in the system (`max number of`

`processes = processor count - 2`), or if the number of tasks (n) is less than the maximum number of processors, the algorithm will activate only the required number of processors. For instance, if $N = 4$ and $n = 4$ then, each task will be handled by four different processors; whereas, if $N = 4$ and $n = 2$, only two processors will be activated. If $n = 6$, the 5^{th} and 6^{th} task will be handled as soon as a processor completes its job. In other words, the algorithm specifies the actions that are to be run concurrently, and the run-time handles all processors scheduling details, including automatic scaling to the number of processors (processor count) on the host computer if required. Figure 4.5 gives an overview of the parallel processing model employed by Nimble Join which will be used to discuss each phase in the algorithm.

1. **Key Hashing:** The predicates are applied to the appropriate dimension table, and the dimension keys and values required by the query are extracted. For each predicate on the dimension table, we create an individual hash table. The size of the hash table is calculated (whenever possible) to precisely fit the tuples from smaller tables, leading to the hash table having a 100% fill factor. Each process P_i reads one of the dimension tables included in the query and creates the hash table.

Algorithm 1: Key Hash

Data: Column[] C, Predicate p

Result: Intermediate Hash Table

```
/* operates columns <= N in parallel                    */
for column c ∈ C do
    for block b ∈ c do
        READ b from disk by pages
        for tuple t ∈ b do
            READ from page and WRITE to memory
            APPLY p to t
            SAVE intermediate result to hash table
        end
    end
end
```

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 of the data can incur a significant overhead. If the dimension table(s) is large, Nimble Join chooses to mix *data and task parallelism* while processing that dimension table(s). In the case of Query 3.1, three processors

are activated as we need to hash three of the dimension tables. The pseudocode for this phase is shown in Algorithm 1. Once all the processes P_1, P_2, and P_3 have completed the hashing task of, we move to the *Probing Phase*.

2. **Probing:** The hash tables created in the key hashing phase are used to match keys in the fact table that satisfy the predicate. For each column C_i that is used in the query, we activate process P_i. Each C_i in P_i is further partitioned using a static partitioner to break up the data into equal sized blocks B_j where $j = 1 \ldots N$. Each block B_j is processed using a separate thread *(i.e. SIMT parallelism)* which probes respective hash tables to update and synchronise *MAAT* concurrently. As probes are performed in MAAT, the size of MAAT decreases as we remove items that do not satisfy the predicate, and retain the position of items satisfying the predicate in the position list of MAAT. 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. During this process, we output the records satisfying all join conditions, thereby producing results much earlier than Invisible Join [30]. The pseudocode for this phase is shown in Algorithm 2.

Algorithm 2: Probe

```
   Data: Column[] C
   Result: MAAT maat
   /* operates columns <= N in parallel                    */
1  for column c ∈ C do
2      for block b ∈ c do
3          READ b from disk by pages
           /* operates in parallel                         */
4          for tuple t ∈ b do
5              READ from page and WRITE to memory
6              GET key
7              if probe(key) is successful then
8                  ADD to maat and pos. list
9              else
10                 REMOVE from maat
11             end
12         end
13     end
14 end
```

Since the blocks are of equal size, there is less chance of skew. Nevertheless, theoretically, the skew might still occur even when the blocks are partitioned equally. This problem arises from an imbalance in the number of results produced, such as the cost of join with the hash table. 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 case where a table has not been partitioned equally [9]. In addition, if the data parallel probing approach produces an unbalanced workload, it can be handled using techniques such as Morsel-driven parallelism [102] or Index Vector Partitioning (IVP) [83].

3. **Value Extraction:** Once all the processes $P_1, P_2, ..., P_N$ have completed the task of synchronising *MAAT*, each column required to answer the query is accessed by the threads to retrieve values based on a position list in MAAT. The *position* is used as an index of an array to perform the position look-up and direct value extraction from those columns. There is no need for a locking mechanism because reading from *MAAT* is a thread-safe operation as we guarantee that there is a no-write operation after the *Probing* phase.

   MAAT holds the intermediate attributes required for the join query processing, thereby eliminating the need to re-scan the fact table columns performed by Invisible Join [30] to reconstruct the output tuple. If the query requires other attributes to be added to MAAT, this can be done easily via the `AddorUpdate` operation in MAAT. Finally, arithmetic operations are performed on the join results to obtain the required aggregation. The pseudocode for this phase is shown in Algorithm 3.

Algorithm 3: Extract Value

```
Data: Column[] C, POSLIST pl
Result: MAAT maat
/* operates columns <= N in parallel                    */
1 for column c ∈ C do
2     for block b ∈ c do
3         READ b from disk by pages
4         for pos. ∈ pl do
5             JUMP to pos. in b, OUTPUT value(s)
6             ADD or UPDATE maat
7         end
8     end
9 end
```

4.5 Experiment Evaluation

In this section, we briefly describe the environment used in the experiment. Then, we present detailed analysis of the results obtained from the experiment.

4.5.1 Experiment Setup

We conducted the experiment on NeCTAR [1] server equipped with 12 Intel Xeon E3-12xx (Ivy Bridge) processors clocked at 2.6 GHz, 48 GB memory and 1 TB RedHat VirtIO SCSI Disk Device. The operating system is Windows Server 2012 Standard Build 9200. The algorithms were implemented using C# .NET framework 4.5.1 supporting X64 architecture. In our experiment, we used a scale factor (SF) = 10 yielding the fact table with 60 million tuples unless stated otherwise.

4.5.2 Algorithms Tested

The following column-oriented join algorithms are evaluated.

- Invisible Join [30]: A late materialised 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: A progressive parallel star join algorithm for column-stores equipped with a multi-attribute array table that facilitates the progressive materialisation presented in this chapter.

We implemented Invisible Join and Nimble Join in serial and parallel fashion. The implementation of the algorithms was focused purely on join technique without application of column-store optimisation such as column-specific compression [25] and database cracking and adaptive indexing [27]. We believe that with these optimisation, Nimble Join will perform more efficiently.

4.5.3 Experiment Results

The numbers reported here are the average of 20 iterations. Before Microsoft Intermediate language (MSIL) can be executed, it must be converted by .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

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

each iteration so that it would not distort the results. For all the join algorithms, standard deviation was less than 10% of the average time. Below, we discuss in detail the analysis results.

1. **Initial Response Time (IRT)** is the time it takes to yield the first join result to the standard output. Let t_s be the starting time, and t_i be the time at which the first join result was yielded to standard output, then the initial response time is $\Delta t_i = t_i - t_s$.

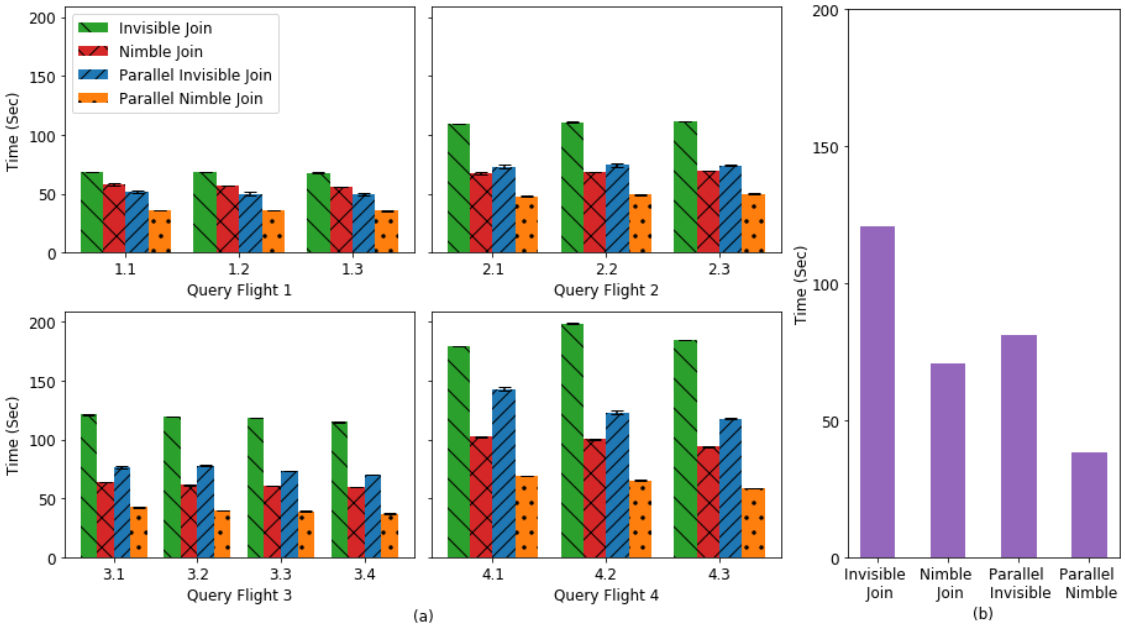


Figure 4.6: (a) Initial response time of all algorithms by SSBM query flights (N= 10 & SF = 10) (b) Average initial response time across all queries

The results of the initial response time broken down by query flight is shown in Figure 4.6 (a), with average results in Figure 4.6 (b). The average response time of Nimble Join is approximately 1.5x faster than its competing join algorithm in both serial and parallel versions. This difference in performance can be attributed to the design of Nimble Join and its use of MAAT in the probing phase as discussed in Section 4.4. In some of the queries in SSBM such as 3.1, 3.2, 3.3, 3.4, 4.1, 4.2 and 4.3, it is even 2x faster than Invisible Join. This improved response time can be beneficial for real-time analytics of the results instead of having to wait for the join query processing to be completed.

2. **Total Execution Time (TET)** is the time spent by the system executing the algorithm until all the output has been yielded to the standard output. Let t_s be the starting time, and t_e be the time at which the last join result was yielded to standard output, then the total execution time is $\Delta t_e = t_e - t_s$.

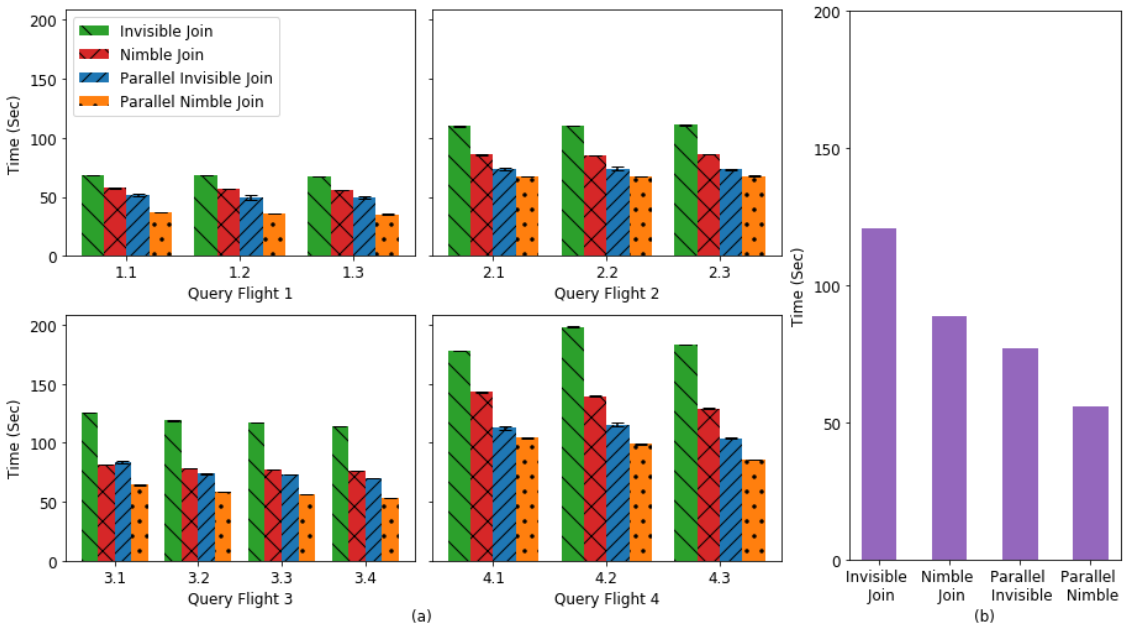


Figure 4.7: (a) Total execution time of all algorithms by SSBM query flights (N= 10 & SF =10). (b) Average total execution time across all queries.

The results of the total execution time broken down by query flight are shown in Figure 4.7 (a), with average results across all queries in Figure 4.7 (b). In addition to better response time, Nimble Join improves the performance by 10% - 25%. This overall performance does not come at the sacrifice of producing results quickly or consuming less memory (discussed in Section 4.5.3).

i. *Effect of MAAT on Disk I/O time:* Disk I/O time is the time spent by the system to transfer the data between secondary storage and the main memory. Let t_{ds} be the starting time, and t_{de} be the time at which all disk I/O is completed, then the time taken for disk I/O is $\Delta t_{de} = t_{de} - t_{ds}$.

   Figure 4.8 (a) shows the average time taken for disk I/O by all the algorithms for all SSBM queries drilled down according to the phases of the algorithm. We have achieved significant reduction ($\approx 50\%$) in disk I/O time because of MAAT. MAAT holds the intermediate attributes required for the join query processing, thereby eliminating the bottleneck produced by multi-pass scans (additional disk I/O) performed by Invisible Join and significantly lowering the total execution time in the Nimble Join.

ii. *Effect of varying number of processors:* In this experiment, both the algorithms were granted unrestricted memory access and the processor access in an increasing order of two. We stop at the maximum of 10 processors. Figure 4.8 (b) shows the performance comparison between Nimble Join and Invisible Join

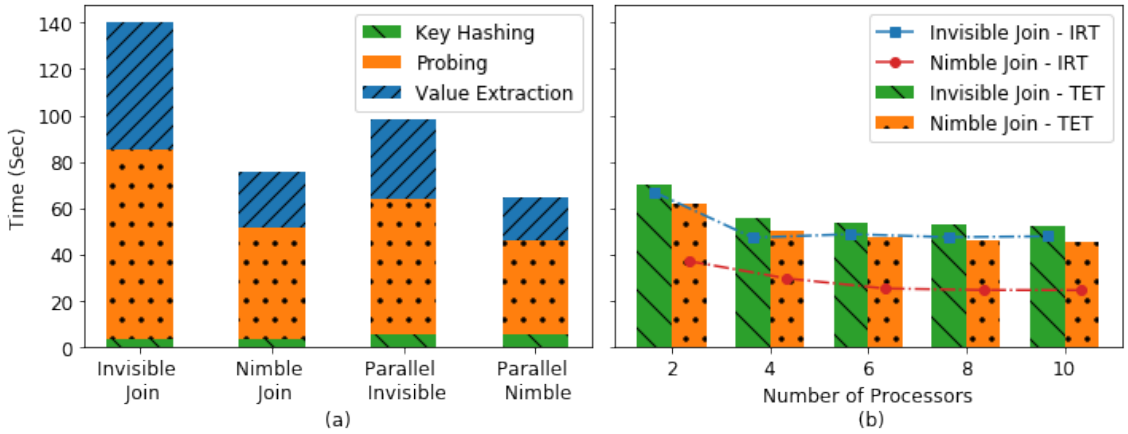


Figure 4.8: (a) Average time for disk I/O for all the algorithms (SF = 10). (b) Performance comparison between nimble and Invisible Join (Query 3.1, SF = 10)

with a varying number of processors for Query 3.1. It is immediately apparent that Nimble Join performs better than Invisible Join in terms of both initial response and total execution time.

We can see a significant improvement in performance from two processors to four processors. At $N = 4$, the cost of work done in each process by multiple threads was significantly lower than if the work were split amongst different threads, yielding remarkable improvement in the performance. However, with an increasing number of processors, the task scheduler does not necessarily activate all the processors. The primary purpose of the task scheduler is to keep all processors occupied as much as possible with some useful work. At runtime, the system observes whether increasing the number of processors improves or degrades overall throughput, and adjusts the number of worker processors accordingly. Therefore, there is only slight improvement in the performance of both algorithms.

3. **Memory Consumption** is the amount of memory used by the algorithm when executing the join query. Let m_s be the amount of memory available at the start and m_e be the amount of memory available at the end of the execution of the algorithm; then the total amount of memory consumed during the execution of the algorithm is $\Delta m_e = m_e - m_s$.

   The detail results of memory consumption broken down by query flight are shown in Figure 4.9 (a); the average results for all queries are shown in Figure 4.9 (b). On average, MAAT can decrease the memory consumption by a factor of 40%. For certain queries in SSBM such as 2.1, 2.2, 2.3, 3.1, 3.2, 3.3 and 4.1, the reduction is

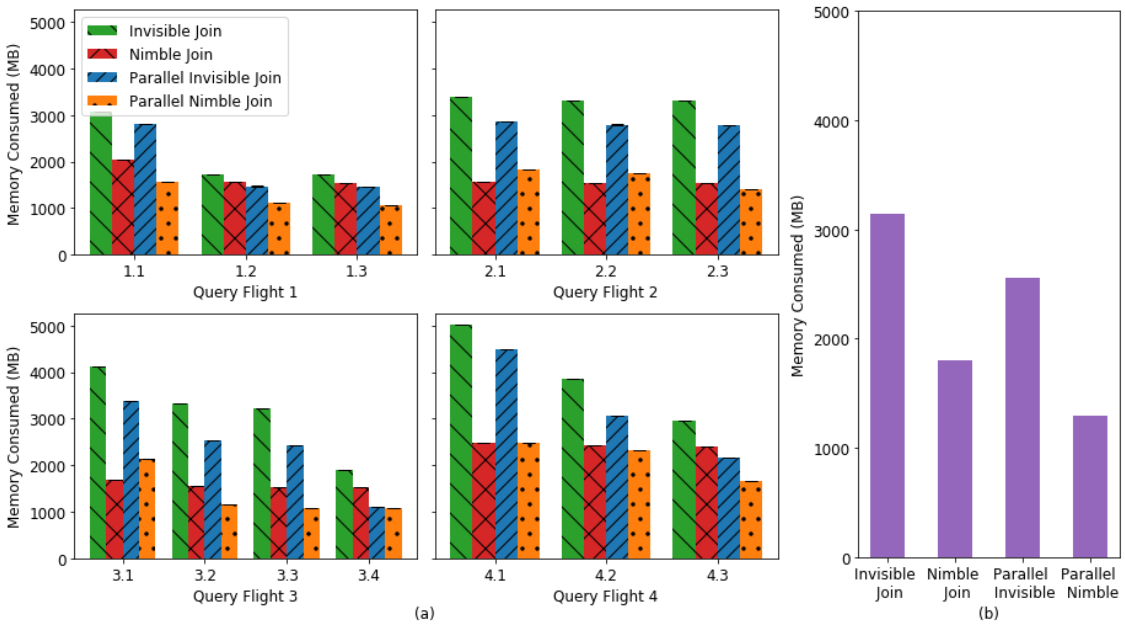


Figure 4.9: (a) Memory consumption of all algorithms by SSBM query flights (N= 10 & SF =10). (b) Average memory consumption across all queries.

even greater than 50%. There are two main reasons for this reduction in memory consumption:

i. Nimble Join avoids the need for additional data structures to hold multiple lists of positions.

ii. It does significantly reduce the disk I/O (discussed in Section 4.5.2i) required to re-access the columns; hence, there is no need to store those columns in memory for query processing.

4.6 Analytical Evaluation

In this section, we describe the cost model to predict the cost of the join operation. Then, we present our model evaluation and statistical analysis to determine the difference between the results of the model and those of the experiment.

4.6.1 Cost Models

To be able to describe the components of an algorithm using mathematical formulas, it is necessary to determine the parameters that define the system and affect the efficiency of the algorithm. The parameters used to create the cost model are listed in Table 4.1. The

symbols used in the formula are: $\lceil\rceil$ is a ceiling function, $\wedge$ means minimum and $\vee$ means maximum.

Table 4.1: The cost model parameters and notations

Symbol	Description
System and data parameters	
D_i	Size of i-th dimension table, i = 1 ... n
$\mid D_i \mid$	Cardinality of i-th dimension table
F_i	Size of the i-th fact table column
$\mid F_i \mid$	Cardinality of the i-th fact table column
n_d	Number of dimension tables
n_f	Number of fact table column involved in the query
N	Number of processors
P	Page size
Query Parameters	
π_i	Projectivity ratio of the i-th dimension table
π_f	Projectivity ratio of the fact table
σ_i	Selectivity ratio of the i-th dimension table
σ_{fi}	Selectivity ratio of the i-th column in the Fact table
Time Unit Cost	
IO	Time to read a page from the disk
t_w	Time to write the record to the main memory
t_r	Time to read a record in the main memory
t_j	Time to join records
t_p	Time to probe
t_h	Time to hash
t_c	Time to perform computation in the main memory
t_d	Time to access the record directly
t_{del}	Time to delete the record

When building the hash table, a read operation is performed three times. We read the required dimension table column from disk page by page, read from the page and write to the memory and, finally, read and hash. We apply the predicate to the data and save the intermediate results into the memory. The approximate cost of Key Hashing is:

$$\begin{aligned} \textit{Key Hashing Cost} = \left\lceil \frac{n_d}{N} \right\rceil \times (n_d \wedge N) \times \big(& \vee_{i=1}^{n_d} D_i \times \pi_i / P \times IO \\ & + \vee_{i=1}^{n_d} \mid D_i \mid \times (t_r + t_w + t_c) \\ & + \vee_{i=1}^{n_d} \mid D_i \mid \times \sigma_i \times (t_r + t_h + t_w) \big) \end{aligned} \tag{4.2}$$

When probing the hash table, the read operation is performed three times. We read the required fact table column from disk page by page. We read from the page and write to the memory, and read and probe to the corresponding hash table. We create MAAT to store the intermediate tuples or remove the tuples that do not meet the join conditions. The

approximate cost of probing is:

$$\begin{aligned} \textit{Probing Cost} = \left\lceil \frac{n_f}{N} \right\rceil \times \left(n_f \wedge N\right) \times \big(\vee_{i=1}^{n_f} F_i \times \pi_f / P \times IO \\ + \vee_{i=1}^{n_f} \mid F_i \mid \times (2t_r + t_w + t_p) \\ + \vee_{i=1}^{n_f} \mid F_i \mid \times \sigma_{f_i} \times \left(t_r + t_j + t_p\right) \\ + \vee_{i=1}^{n_f} \mid F_i \mid \times \sigma_{f_i} \times \left(1 - \sigma_{f_i}\right) \times t_w \times t_{del}\big) \end{aligned} \tag{4.3}$$

When extracting the value and projecting the join results, the read operation is performed three times. We read the required fact table column from disk by pages, read from the page and write to the memory, and look up and read the value from the column with the key value position. We then add the value to the final result and project it. The approximate cost of Value Extraction is:

$$\begin{aligned} \textit{Value Extraction Cost} = \left\lceil \frac{n_f}{N} \right\rceil \times \left(n_f \wedge N\right) \times \big(\left(\vee_{i=1}^{n_f} F_i \times \pi_f / P \times IO \right) \\ + \vee_{i=1}^{n_f} \mid F_i \mid \times (t_r + t_w) \\ + \vee_{i=1}^{n_f} \mid F_i \mid \times \sigma_{fi} \times \left(t_d + t_r + t_j\right) \big) \end{aligned} \tag{4.4}$$

4.6.2 Model Evaluation

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

1. *Effect of number of processors:* Figure 4.10 shows the comparison between the execution time predicted by the model and actual execution time from the experiment for varying number of processors for Query 3.1 in SSBM. As shown in the figure, the estimated execution time from the cost model is close to the actual execution time from the experiment in all the cases, which demonstrates the effectiveness of our cost model. Two factors account for the difference between the estimated and the actual execution time:

    i. The processors executing in parallel often access shared resources and the slowdown results from the *interference* between the processors to access the shared resources.

    ii. Usually, processes communicate with each other and the process wanting to *communicate* with others may be forced to wait for other processes to be ready for communication. These two factors are extremely difficult to account for in the cost model.

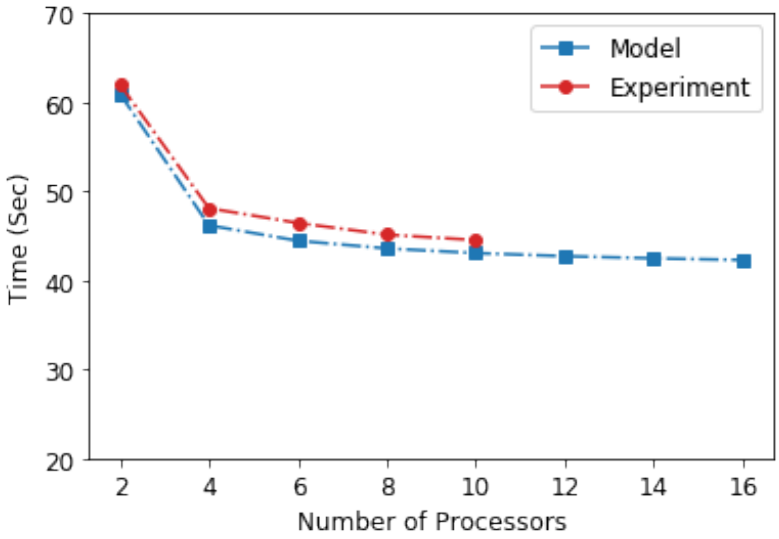


Figure 4.10: Evaluation result with varying number of processors (SF = 10)

Similar to the experiment, we have varied the number of processors by the power of two. In the experiments, we stopped at ten processors due to hardware restrictions. However, we do not have that restriction in the cost model. Therefore, we can verify the results from the experiments and affirm that there will be no significant improvement with the increasing number of processors for the join algorithm based on our cost model.

2. *SSBM queries:* When evaluating our model for SSBM queries, we define the error rate as

$$error\ rate = \left| \frac{experiment\ time - model\ time}{experiment\ time} \right| \tag{4.5}$$

Table 4.2 shows the comparison between the execution time predicted by the model and the actual execution time from the experiment for all SSBM queries. In all cases, the estimated execution time of the cost model is close to the actual execution time of the experiment, which again demonstrates the effectiveness of our cost model. The differences between the estimated and the actual cost can be attributed to the *Start-up cost* associated with initiating multiple processors. The start-up time of the processes varies, making it difficult to estimate and include it in the cost model accurately.

To check whether there is a significant difference between the model and the experiment, we conducted a *two-tailed t-test*. In this t-test, a sample size of 13 model values was compared with corresponding experimental values. The *p-value* obtained for the test was 0.8664 which is much larger than the significance level of 0.05.

Table 4.2: Evaluation result for SSBM queries and error rate of estimated performance (N = 10, SF = 10)

Query	Model (Sec)	Experiment (Sec)	Error Rate (%)
1.1	35.769	36.712	2.568
1.2	35.011	35.592	1.632
1.3	34.321	35.325	2.842
2.1	66.890	67.369	0.711
2.2	67.000	67.181	0.269
2.3	67.259	67.781	0.770
3.1	62.125	64.414	3.553
3.2	55.128	58.265	5.384
3.3	55.411	56.475	1.884
3.4	52.132	53.415	2.401
4.1	99.985	104.138	3.985
4.2	98.110	98.937	0.835
4.3	83.298	85.864	2.988

Therefore, we accept the null hypothesis, and we conclude that there is no significant difference between the values of the model and those of the experiment.

4.7 Summary

In this chapter, we proposed a new progressive parallel star join algorithm for column-stores know as *Nimble Join*. It used a multi-attribute array table to hold attributes required in join query processing that facilitated progressive materialisation. We showed that our algorithm produces results faster, has comparatively better execution time, and dramatically reduces the memory consumption more so than the competing column-stores join. This improvement is mainly due to the design of the Nimble Join and its use of MAAT and progressive materialisation. We also proposed an analytical model to understand and predict the query performance of the Nimble Join. Our evaluation shows that the model can predict the performance with 95% confidence.

Chapter 5

Parallel Star Group-By Join

In this chapter, we focus on the parallel star group-by join for column-stores. First, we introduce the technical challenges associated with answering star group-by join queries for column-oriented data. Then, we discuss *Aggregate Trie (ATrie)*: our novel grouping technique. We use ATrie and progressive materialisation technique to develop a new parallel star group-join algorithm known as *ATrie Group Join (ATGJ)*. Experiments on an SSBM dataset is conducted to verify the efficiency of our proposed solution.

5.1 Overview: Challenges and Solution

Big Data and its analysis are the decision-making drivers of business and company success. In this section, we discuss the challenges posed by big data and how to address them. We focus on addressing this issue: When and how should the aggregate columns be processed to join and group big data in star join queries?

5.1.1 Big Data, Big Problems

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 [12, 22]. The queries in the benchmarks such as TPC-H and the Star Schema Benchmark (SSBM) [23] spend more than 50% CPU time on joins, group-by and aggregation [16]. These queries often aggregate large portions of the data, which can lead to performance issues with very large data sets.

Column-stores 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 [6, 7, 30, 31]. However, three important questions arise. Firstly: Can we analyse structured big data using column-stores? Researches conducted on the use of modern column-stores for big data processing [84, 85] have concluded that parallel column-stores and fine-tuned algorithms can successfully analyse structured big data and solve big data analytics problems.

Secondly: How can we achieve fast processing for big data in column-stores? The need to increase the capacity of main memory has fuelled the development of in-memory big data management and processing [103, 104]. Most column-stores are being tailored to run as main memory database systems which avoid the latency associated with secondary storage such as Hyper [10], SAP HANA [11], IBM DB2 with BLU Acceleration [12], Microsoft SQL Server [13, 14] and Oracle In-Memory Database [15, 16]. Therefore, with in-memory column stores, faster processing of big data is possible. In addition, our algorithm targets those in-memory databases to help them process star queries more efficiently.

5.1.2 Star Group-By Join

The third and 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 star group-by join 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) [30, 44]. An unconventional approach is *In-Memory Aggregation* where group-by expressions are pushed down into the scans of dimension tables [16]. They create a unique key for each distinct group called *Dense Grouping Key* (DGK) and map the join keys to DGKs using a *Key Vector*. Key vector is used to filter non-matching rows when performing fact table scans where the aggregation result is stored in a multidimensional array known as *In-memory Accumulator (IMA)*.

Although the unconventional approach drastically reduced execution time more so than the traditional approach, we found three main problems in this approach: 1) In-Memory Aggregation 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 in that the data is not processed in tight loops, which results in considerable performance deterioration on modern hardware [36]. 2) It is efficient only provided that the IMA does not become too large [16]. 3) The increasing number of

dimensions and grouping attributes produces additional *key vectors* and *temporary tables* to process join, group and aggregation resulting in a significant increase in the execution time.

5.1.3 Technical Contributions

To address the abovementioned challenges, we developed *ATrie Group Join (ATGJ)*, a variant of Hash Join/Hash Group-By algorithm with a novel grouping technique and parallel processing design that fully integrates join, grouping and aggregation to accelerate big data analytical workloads. We propose a novel tree-based data structure known as the *aggregate trie* or *ATrie* influenced by a trie or prefix tree [55] 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 Figure 5.8, unlike In-Memory Aggregation where the performance degrades if IMA grows too large [16]. Experiment results show that our approach can reduce the total execution time by 2X to 6X. Furthermore, 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 technical contributions:

1) We propose a novel approach to perform group-by and aggregation operations 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, to date, no such technique has been proposed for column-stores.

2) We propose a new parallel star group join and aggregation for in-memory column-stores known as *ATrie Group Join (ATGJ)* that is significantly faster than its competing column-stores join algorithm. ATGJ is a variation of Hash Join/Hash Group-By that uses both techniques but solves the problem of grouping and aggregating data using a rather novel approach with the help of the *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.

5.2 Aggregate Trie (ATrie)

An *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* [55] and it facilitates faster grouping and aggregation of the attributes than do other current techniques such as vector group-by [16, 105]. In this section, we focus on the terminologies used, formal definition of the *aggregate trie* and the core operational concepts.

5.2.1 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 *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 child node is called *leaf node*. The *height* of the ATrie is the height of the root node.

5.2.2 Formal Definition

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

Definition 5.2.1. *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 GAO is:*

$$GAO = \{[x_1, x_2, ..., x_{n-1}, x_n] \mid x_n \in \mathbb{Z} \wedge (x_1, x_2, ..., x_{n-1}) \in \textit{grouping attributes}\} \quad (5.1)$$

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

Definition 5.2.2. *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 leaf nodes and hold aggregated value. Each node of 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.*

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 operation.
- *Data Compression:* TheATrie 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 GAOs 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 Materialization:* Recall from Section 4.3, Progressive Materialization adopts the notion of late materialization to push the tuple construction as late as possible but carries attribute values required in query processing throughout the query plan. ATGJ maneuver the idea of progressive materialization 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.

5.2.3 Physical Data Structure

The basic form of implementing ATrie is by using the 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 in this chapter.

5.2.4 ATrie Operations

Figure 5.1 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.

1. **Reading or Searching the ATrie:** To read or search the ATrie, we follow the path designated by addresses advancing to 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

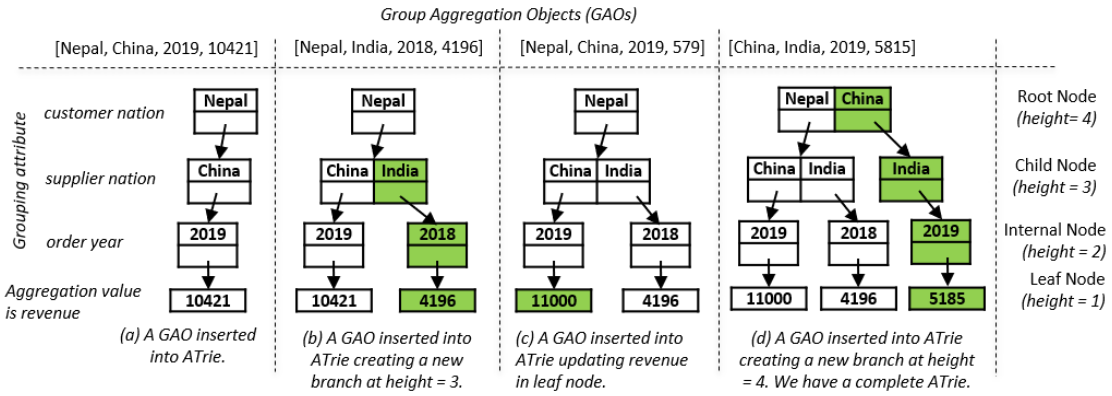


Figure 5.1: A step-wise insertion of GAOs in the ATrie. The new insertion of grouping attribute or update of aggregation value has been highlighted after each insertion of a GAO.

contains no address, then we have reached the leaf node that holds the aggregated value. The worst-case time complexity of this operation is $\mathcal{O}(h * K)$.

Let us try to read or search attributes in a GAO = [“Nepal”, “China”, “2019”, 11000] in a complete ATrie (refer Figure 5.1 (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 are no pointers to the hash table, therefore, we read the aggregated value `revenue` = 11000.

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

   Figure 5.1 shows an example of the step-wise insertion of GAOs into the ATrie. We discuss two examples of insertion shown in Figure 5.1 (b) and (c). Let us insert a GAO = [“Nepal”, “India”, “2018”, 4196] (Figure 5.1 (b)). 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.

Let us try to insert another GAO = [“Nepal”, “China”, “2019”, 579] (Figure 5.1 (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 because of first insertion procedure (Figure 5.1 (a)). Therefore, we update the aggregation value in leaf node `revenue` = 10421 + 579 = 11000. The pseudocode for the insertion of data into an ATrie is given in Algorithm 4.

Algorithm 4: Inserting into an ATrie

```
Data: ATrie root, GAO gao
Result: complete ATrie
node <- root
height <- sizeofGAO()
value <- gao.pop(-1) /* last item in gao is value */
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
```

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 $\mathcal{O}((h * K)^2)$.

   Figure 5.2 shows an example of the step-wise merging of GAOs into left ATrie. Firstly, we read right ATrie (Figure 5.2 (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 5.2 (d) and Figure 5.2 (e) respectively.

   When we insert GAO = [“Nepal”, “India”, “2018”, 804] into the left ATrie (refer Figure 5.2 (e)), 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 Figure

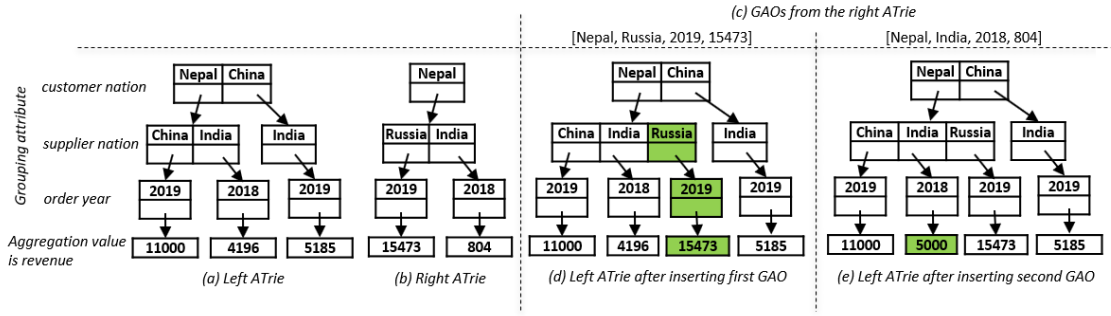


Figure 5.2: A step-wise merging of two ATries. The new insertion of the group attribute or update of an aggregate value has been highlighted after each insertion of a GAO.

5.2 (a)). Therefore, we update the aggregation value in the leaf node revenue = 4196 + 804 = 5000.

The pseudocode for merging two ATries is shown in Algorithm 5.

Algorithm 5: 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
```

4. **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 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 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 ATGJ.

5.3 ATrie Group Join (ATGJ)

ATrie Group Join (ATGJ) is a 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 the trie or prefix tree [55]. In this section, we describe in detail the main stages of the algorithm. The description of ATGJ presented in this section is for a single-threaded implementation. We extend ATGJ to the multi-threaded group join algorithm in Section 5.3.2.

5.3.1 Join Processing Method

We explain in detail the working of ATGJ using Query 3.1 from SSBM. ATGJ has 3 phases: 1. Scan and Predicate Evaluation, 2. Join, Group and Aggregate and 3. Output Results.

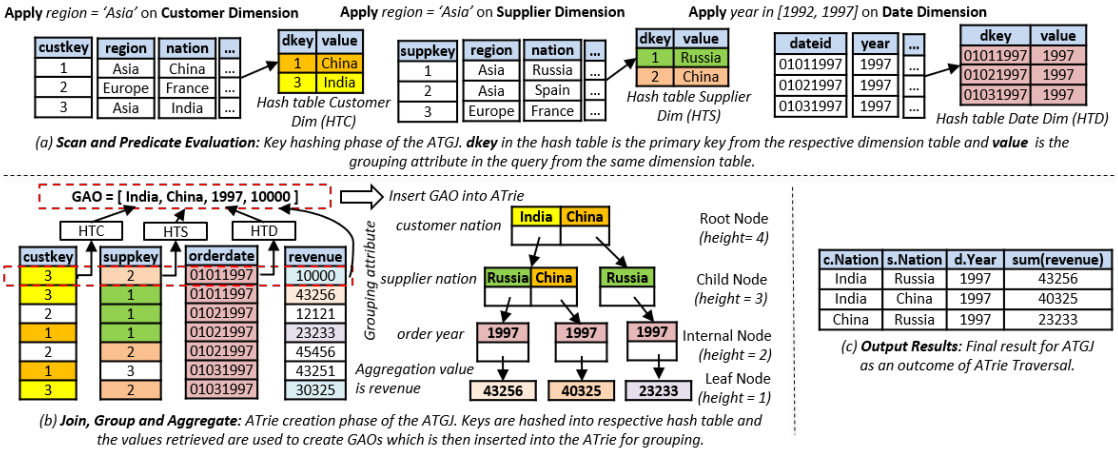


Figure 5.3: Stages of ATGJ to execute Query 3.1 from SSBM on some sample data

1. **Scan and Predicate Evaluation**

   Hash - Predicates are applied on the appropriate dimension tables (i.e. `CUSTOMER`, `SUPPLIER` and `DATE`) to create the respective hash tables. The primary key of the dimension tables acts as 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 Figure 5.3 (a).

2. **Join, Group and Aggregate**

   Probe and Create ATrie - Foreign key columns of the fact table (i.e. `LINEORDER`) are

accessed row-by-row and the keys are probed into the corresponding hash table to create GAOs. For simplicity, when creating a GAO, the order of grouping attributes is decided based on the query grouping requirement (for instance, in Query 3.1, we group-by based on `customer nation` then by `supplier nation` and finally by `order year`). Changing the order of attributes does not affect the performance of ATrie as shown in Figure 5.9. The GAOs are then inserted into the ATrie. Insertion proceeds by walking the ATrie according to the attributes in GAO, then appending a new node for an attribute that is not contained in 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. The leaf node holds the aggregated values.

Figure 5.3 (b) shows the creation of ATrie. For the first row in `LINEORDER` table, 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, we search for attribute “India”. Since it does not exist, we add the attribute to the node. As this is a new attribute added to the node, the remaining attributes will no longer 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 `lo.revenue` = 10000 in the leaf node that has a pointer from attribute “1997”.

Following the same procedure as above, ATGJ accesses row-by-row, probes the keys into the corresponding hash table to create GAO that are inserted into ATrie for grouping and aggregation. Now let us look at the last row. We have customer key = 3, supplier key = 2 and order date = 01031997. When these keys are probed into the respective hash table, we obtain 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 the aggregation of values on the leaf node.

3. **Output Results**

ATrie Traversal - Depth-first search (DFS) algorithm is used to produce the final result shown in Figure 5.3 (c). We perform a minor change to the DFS algorithm and include a global hash table that tracks the grouped items at a 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 6.

Algorithm 6: 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 ∈ 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
```

5.3.2 Parallelizing ATGJ

So far, we have discussed the serial version of ATGJ. In this section, we present the solution to parallelizing ATGJ as shown in Figure 5.4.

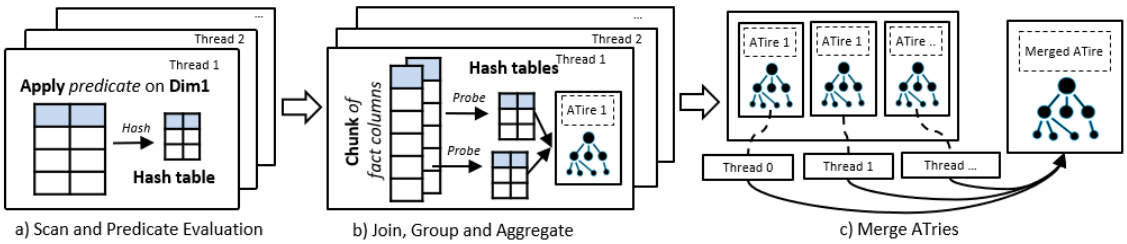


Figure 5.4: ATrie Group Join Parallel Model

Implementation: Let N be the number of threads (T) that will be used for the query execution (i.e. $T_1, T_2, ..., T_N$). 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 the maximum number of threads, the algorithm will activate only the required number of threads. The algorithm specifies 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. Figure 5.4 gives a basic overview of the parallel processing model employed by ATGJ which will be used to explain each phase in the algorithm.

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, the partitioning of the data can incur a significant overhead [44]. 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 are activated as we need to hash three of the dimension tables. 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.

Algorithm 7: Get Partitioning Indexes

```
Data: int totalNumberOfRecords, int processorCount
Result: List of partition indexes
boundaries <− null
min <− 0
max <− totalNumberOfRecords/ processorCount
for i <− 0 to processorCount do
    if i equals processorCount - 1 then
        max <− totalNumberOfRecords - 1
    end
    boundaries <− min, max
    min <− max + 1
    max <−max + totalNumberOfRecords/processorCount
end
return boundaries
```

2. **Join, Group and Aggregate**

   The size of the fact table is significantly large compared to that of dimension tables. However, since we assume in-memory column-stores, the fact table columns reside in memory and we can access those columns using the indexes the same as with the

array lookups. ATGJ uses `getPartitionIndexes` (refer Algorithm 7) to retrieve the logical partitioning indexes instead of physically partitioning the columns.

Algorithm 7 returns the index boundaries of the sub-range that includes all indexes within the range. This gives us an equal number of items in each logical partition so that there is no load imbalance in the join and group-by processing. For example, if we have a total of 100 records and we use three 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 the foreign key columns of the fact table (i.e. `LINEORDER`) are 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 the hash table allows only a single writer thread, it supports multiple reader threads concurrently.

Create ATrie - During *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 thread local ATries in the global merge phase to get final result.

3. **Merge ATries**
   Merging ATries can be serial or parallel depending on the number of ATries or the number of threads. For example, let us say that 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 found that the number of nodes in 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 the *probe* and *create ATrie* operations.

5.4 Experiment Evaluation

In this section, we briefly describe the environment used in the experiment. Then, we present a detailed analysis of results obtained from the experiment.

5.4.1 Experiment Setup

We conducted all our experiments on the high-performance NeCTAR [1] 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.

Benchmark Dataset: We used SSBM [98] for the experiment. Query Flight 1 in SSBM does not contain queries with the group-by. Therefore, we excluded Query Flight 1. This benchmark provides a base scale factor (SF) to scale the size of the data. Similar to [16, 30, 75, 78], 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 5.1.

Table 5.1: 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

5.4.2 Algorithms Tested

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

- Invisible Join [30]: A late materialized join that minimises the value that needs to be extracted out-of-order by rewriting joins into predicates on the foreign key columns in the fact table.

- Nimble Join: A progressive parallel star join algorithm for column-stores equipped with a multi-attribute array table that facilitates progressive materialization. The details can be found in Section 4.4.

- In-Memory Aggregation [16]: A novel technique used to perform traditional group-by and aggregation operations across joins using *dense grouping keys* and *key vectors*.

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

- ATrie Group Join: A highly efficient group join based on ATrie presented in this chapter.

We implemented Invisible Join, Nimble Join, In-Memory Aggregation and ATGJ in parallel. The implementation of the algorithms were focused purely on join technique without the application of column-store optimisations such as column-specific compression [25], database cracking [25] and adaptive indexing [27]. We believe that with these optimisations, ATGJ will perform more efficiently.

5.4.3 Experiment 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 would not distort the results. For all the algorithms, the standard deviation was less than 10% of the average time. Next, we discuss in detail the results of the analysis.

1. **Total Execution Time (TET)**
   TET is the time spent by the system executing the algorithm until all the output has been yielded to the standard output. Let t_s be the starting time, and t_e be the time at which the last join result was yielded to standard output, then the total execution time is $\Delta t_e = t_e - t_s$.

   Figure 5.5 shows the results of the total execution time broken down by query flight with the standard error of mean (SEM) and the average result for all queries. For all the group-by queries in SSBM, ATGJ is 100% faster than the rest of 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 ATGJ coalesces the joins and applies all the joins to the fact table in a single operation rather than constructing the rows to be grouped by processing the fact data through a successive series of joins. 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 group-by efficiently, and aggregate rows using the proposed novel grouping technique and progressive materialisation.

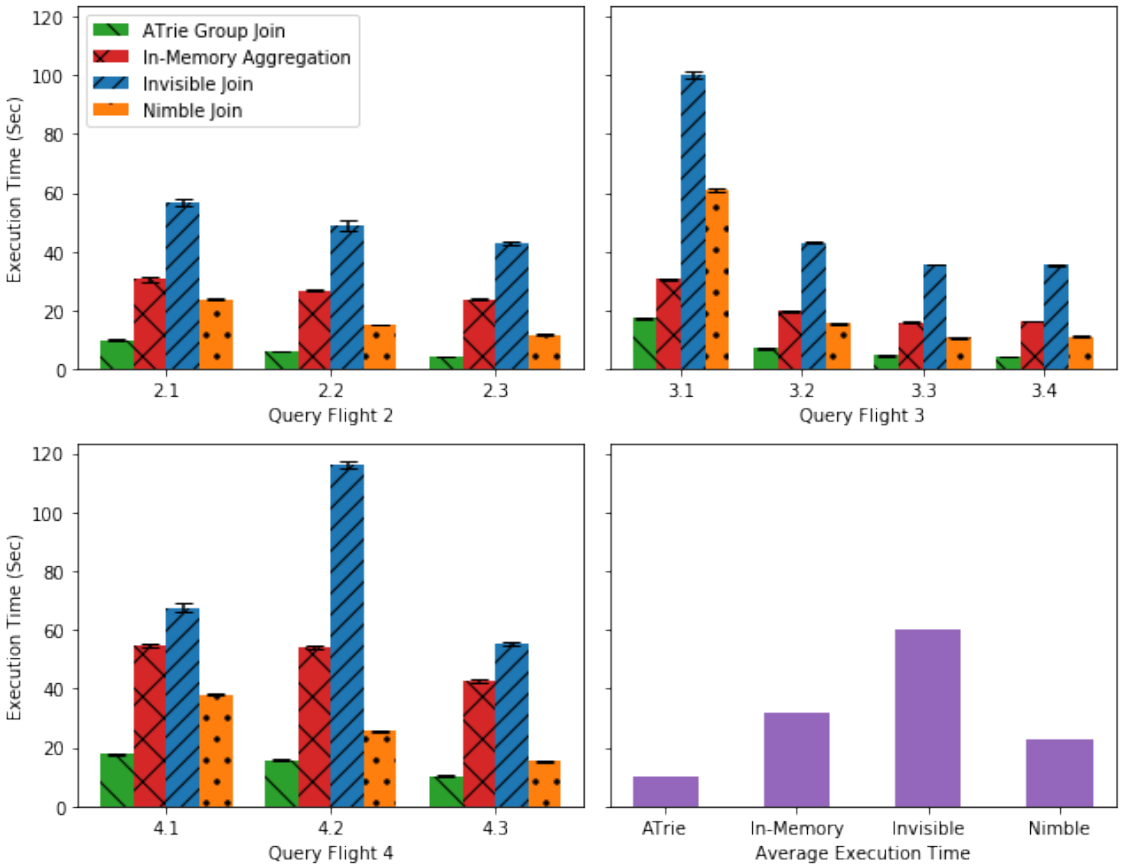


Figure 5.5: Total execution time of all algorithms by SSBM query flights (N = 14 and SF = 100) and the average execution time of all the queries.

The size of data set, query complexity and 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.

i. *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. Figure 5.6 shows the average execution time for all queries and the speed-up of all the algorithms.

   Figure 5.6 shows a significant improvement in performance of ATGJ until N = 6 where the cost of work done by multiple threads was significantly higher than when splitting the work amongst different threads. However, when increasing the number of threads, the task scheduler does not necessarily activate all the processors. It observes whether increasing the number of threads improves or degrades the overall throughput and adjusts the number of worker threads

accordingly. Therefore, we created a synthetic dataset with 14 dimensional tables, with predicate constraints on each dimensional table, and increased the workload.

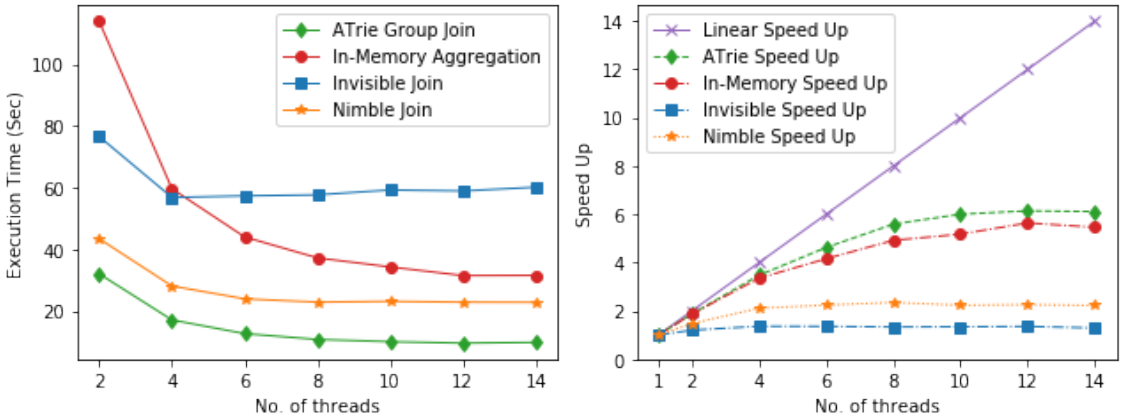


Figure 5.6: A comparison of join algorithms while varying number of cores and the speed up of each algorithm (SF = 100).

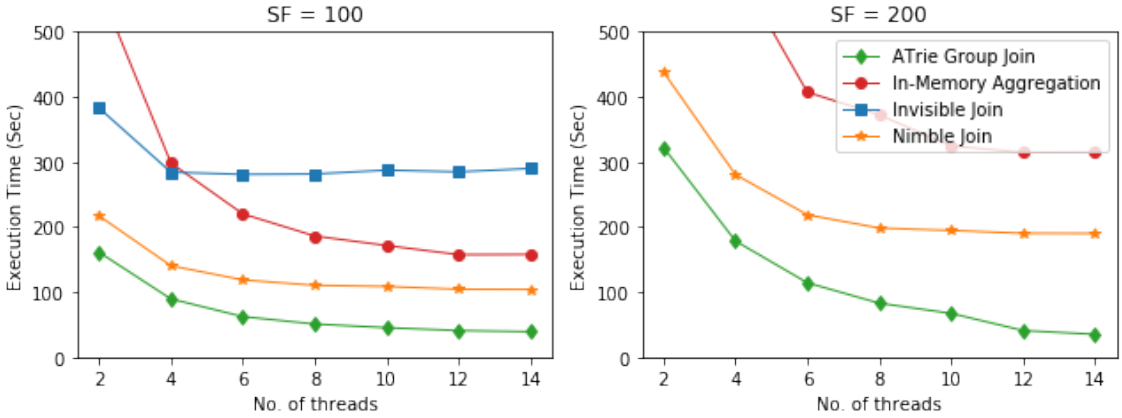


Figure 5.7: A comparison of join algorithms while varying number of cores and the performance improvement with the increased workload.

Figure 5.7 shows no significant performance improvement with the increase in the dimensional tables and predicate constraints for SF = 100. However, we see performance improvement when the number of threads is increased to deal with an increased data size (i.e. SF = 200). Although data partitioning is the key to parallel processing [9], we follow task parallelism for *Scan and Predicate Evaluation* phase in ATGJ which is similar to the parallel processing technique applied by Invisible Join [30] and Nimble Join [44]. The dimension tables are significantly smaller than the fact table. For a smaller set of data, if the operations are not substantial, the partitioning of the data can incur a

significant overhead [44]. Therefore, we do not employ *data parallelism*. If the dimension table(s) is large, ATGJ chooses to mix *data and task parallelism* when processing that dimension table(s). We employ data parallelism for *Join, Group and Aggregate* phase in ATGJ similar to in-memory aggregation [16]. The logical partitioning of data using `getPartitionIndexes` on the in-memory fact columns allows the optimal use of available resources in ATGJ; therefore, we can see a better speed-up than with the rest of the competing algorithms.

ii. *Effect of Number of Group-by Attributes:* SSBM has approximately 25 categorical dimensional attributes that can be used in a grouping. In SSBM queries, the maximum number of grouping attributes used is four. However, we assume that ten is the average number of attributes that could be used in a query. The experiment result shown in Figure 5.8 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 one and continuing to ten grouping attributes. The modified Query 4.3 with ten 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.suppkey = s.suppkey
    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 = 'MFGR#14'
GROUP BY d.year, d.month, s.nation, s.region, s.city,
c.nation, c.region, c.city, p.brand, p.type;
```

Figure 5.8 shows that the performance of ATGJ is consistent with the increasing number of grouping attributes. This is mainly because of the deterministic property of ATrie where there is no dynamic reorganisation of attributes for the insertion operation in ATrie, in addition to the constant complexity in terms

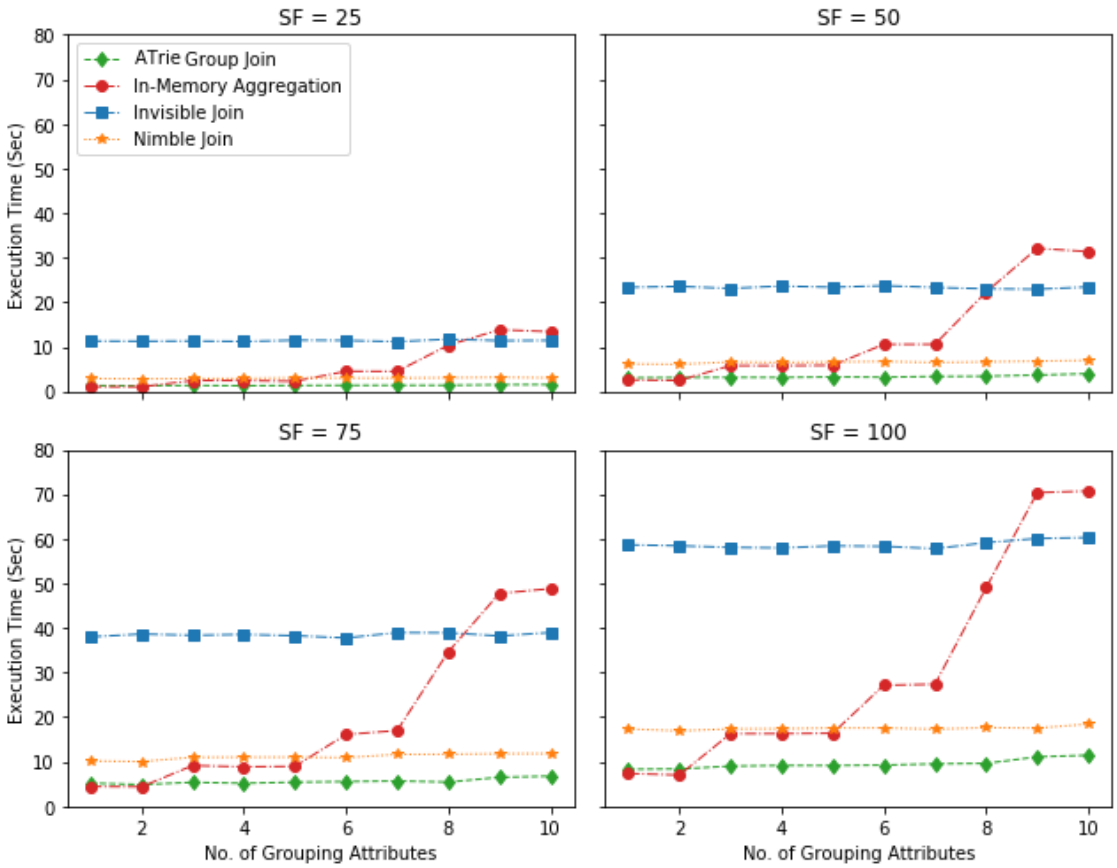


Figure 5.8: A comparison of join algorithms for the scalability with an increasing number of grouping attributes.

of the fill factor of ATrie. The primary concern with ATrie was the increased height of the tree with 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 the total execution time and the performance of ATGJ decreases minimally with the addition of grouping attributes.

Instead, we found that In-Memory Aggregation has atrocious performance with the increasing number of grouping attributes. We can see two different kinds of impact on the In-Memory Aggregation algorithm. First, the increasing number of dimensions 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 dimension tables. Second, when the grouping attribute has a large number of distinct values (such as `c.city`. For comparison, `s.city` had ten distinct values whereas `c.city` had 50 distinct

values), this algorithm creates an equivalent number of dense grouping keys and the number of records in temporary tables increases significantly. Therefore, the algorithm suffers most in the final stage which cannot be parallelized due to its algorithmic design.

iii. *Effect of Order of Group-by Attributes:* In ATGJ, the order of the grouping attributes depends 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 of star schema queries, so that prospective users can derive performance ratings 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 the order of grouping attributes, we created five different groups of attributes for the query with ten grouping attributes shown in Table 5.2.

Table 5.2: Different order of the grouping attributes (Modified Query 4.1, # Grouping Attributes = 10)

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

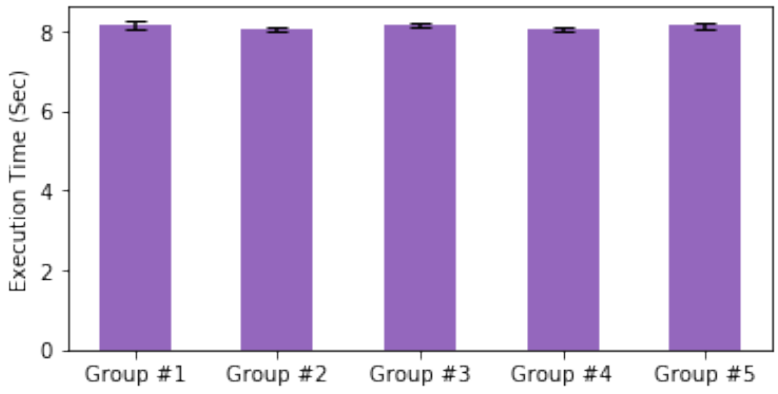


Figure 5.9: Average execution time of ATGJ for varying order of grouping attributes (SF = 100, N = 14).

Figure 5.9 shows that changing the order of attributes does not affect the performance of ATrie. This is mainly due to the nature of insertion of attributes in ATire (i.e. insert if not present) which allows shorter access time, greater ease of addition of nodes or updating values, and greater convenience in handling a varying group of attributes.

iv. *Effect of Data Size:* For each point in graph, the scale factor is increased starting from 1 and continuing to 100 with a step of 25. The number of records in the fact table for different scale factors for SSBM dataset are shown in Table 5.1.

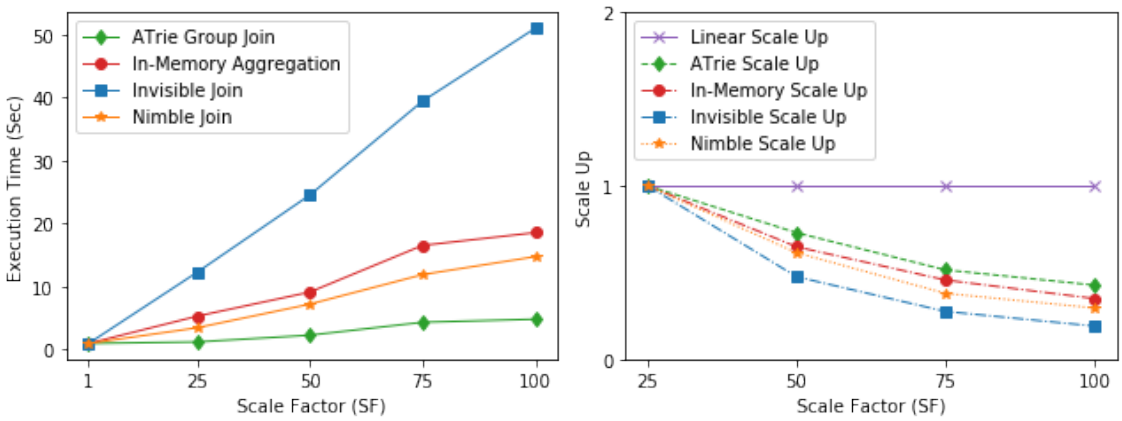


Figure 5.10: Average execution time across all SSBM Queries for all algorithms (N = 14) and scale-up of the algorithms for varying data sizes. The processing resources are doubled when the data size is doubled.

Figure 5.10 shows the average execution time of all queries and the scale-up of algorithms for varying data sizes. ATGJ performs significantly better than competing algorithms for varying data sizes. We can see that the execution time for ATGJ is significantly lower than that of competing algorithms for the larger data set. The number of records in the dimension tables is significantly less than that in the fact table. When the predicate filters are applied, only a small percentage of these dimension table attributes are selected for grouping [9]. ATGJ uses ATrie to group attributes and a hash table to track the edges and navigate through ATrie to update the aggregation value. Hashing is relatively faster than other operations in ATGJ. When creating ATrie, our only concern was the increase in the height of the tree with the increase in the number of grouping attributes. However, in Section 5.4.3.1ii, we showed that the height of the tree does not significantly impact the performance of 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.

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

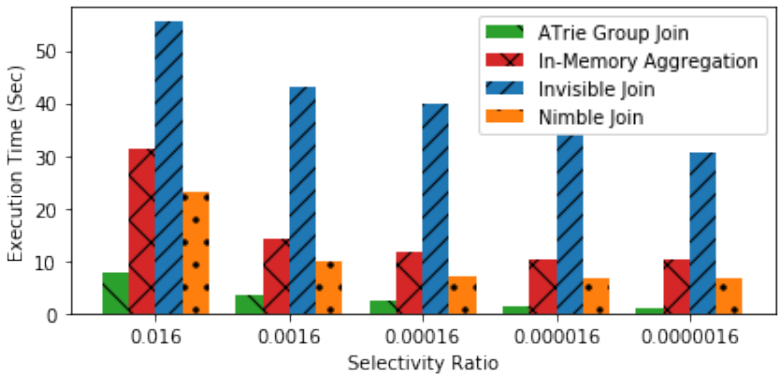


Figure 5.11: Total execution time for all algorithms with different selectivity ratio (N = 14 and SF = 100)

Figure 5.11 shows that for varying selectivity ratio, ATGJ is only 2X to 3X times faster than Nimble Join but 6X to 8X faster than Invisible Join. The creation of ATrie is the second most expensive step after probing in ATGJ. As the selectivity ratio decreases, the number of internal nodes in ATrie significantly decreases. Therefore, it takes significantly less time to complete the *Create ATrie* phase. The previously discussed algorithms include complex steps in the group and join processing and therefore cannot take advantage of the decreasing selectivity ratio as much as ATGJ.

2. **Memory Consumption**

Table 5.3 shows the memory used by algorithms to create and store data in 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 comparatively less memory than Invisible Join and Nimble Join because of ATrie. ATrie features data compression by sharing the same grouping

Table 5.3: 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

attributes, thereby allowing it to use less space than it would require to list all the distinct results separately as done by Invisible Join and Nimble Join.

3. **Analysing ATrie**

   We analyse ATire based on a number of different factors.

   - **Tree Height:** A potential problem when creating 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 ATrie and the number of created nodes. However, the experiment results (Figure 5.8) show that the cost to create and access additional nodes in ATrie is only a small portion of the total execution cost, and therefore, can be safely overlooked in ATGJ.
   - **Branching Factor:** Another potential problem when creating ATrie is the branching factor or the number of branches. The branching factor is highly dependent on the number of distinct values of grouping attributes which can make ATrie bushy. For example, if the supplier nation attribute (refer Figure 5.1) has 100 distinct values, the maximum number of branches in each internal node at height h = 3 is 100. However, the experiment results (refer Figure 5.8) show that the branching factor has no significant impact on the performance of ATGJ. The performance of ATGJ is stable throughout the experiment even though the grouping attribute with a large number of distinct values was introduced such as `c.city` (for comparison, `s.city` had ten distinct values whereas `c.city` had 50 distinct values).
   - **Order of Grouping Attributes:** Another potential problem when inserting a GAO into ATrie is the order of grouping attributes in a GAO. We need to understand whether the ordering of group-by attributes 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 grouping attributes has no significant impact in the creation of ATrie as shown in Figure 5.9.

- **Differences with B+ Tree:** Although, ATrie and B+ Tree look similar, there are significant differences between them. In B+ Tree, the root node must have at least two pointers to other nodes when it is not the leaf node [106]. However, in ATire, the root node can have one 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 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 [106]. However, in 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 [106]. However, in ATrie, the leaf node contains an aggregation value and there is no relationship between two leaf nodes.

5.5 Analytical Evaluation

In this section, we describe the cost model to predict the cost of group join. We also present our model evaluation and statistical analysis to understand the difference between model and experiment results.

5.5.1 Cost Models

The parameters used to create the cost model are listed in Table 5.4. The symbols used in the formula: $\lceil\rceil$ is a ceiling function, $\lfloor\rfloor$ is a floor function, $\wedge$ means minimum and $\vee$ means maximum.

During the *Hash* phase, we read the dimension table columns from main memory and create the hash table. Therefore, we encounter two different types of cost: *Scan Cost* and *Hash Cost* obtained by the following equations.

$$M = \left\lceil \frac{n_d}{N} \right\rceil \tag{5.2}$$

Table 5.4: The cost model parameters and notations

Symbol	Description
System and data parameters	
D_i	Size of i-th dimension table, i = 1 ... n
$\mid D_i \mid$	Cardinality of i-th dimension table
F_i	Size of the i-th fact table column chunk
$\mid F_i \mid$	Cardinality of the i-th fact table column chunk
n_d	Number of dimension tables
n_g	Number of attributes involved in the grouping
N	Number of processors
Query Parameters	
σ_i	Selectivity ratio of the i-th dimension table
σ_f	Selectivity ratio of the Fact table
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

$$Scan\ Cost = \sum_{j=0}^{M-1} \vee_{i=j\times N+1}^{(j\times N+N)\wedge n_d} \mid D_i \mid \times (t_r + t_f) \tag{5.3}$$

$$Hash\ Cost = \sum_{j=0}^{M-1} \vee_{i=j\times N+1}^{(j\times N+N)\wedge n_d} \mid D_i \mid \times \sigma_i \times (t_h + t_w) \tag{5.4}$$

During the *Probe* phase, we divide the entire fact columns into the same number of partitions as the number of processors. Each processor reads the required fact column partitions from the main memory and probes the hash table. The cost for this phase is obtained by the following equation.

$$Probe\ Cost = \vee_{i=1}^{N}(\mid F_i \mid \times n_d \times t_r) + (\log_2 \mid F_i \mid \times n_d \times t_p) \tag{5.5}$$

During the *Create 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 by the following equation.

$$Create\ ATrie\ Cost = \vee_{i=1}^{N} \mid F_i \mid \times \sigma_f \times (\ n_g \times (t_h + t_p + t_w) + t_a) \tag{5.6}$$

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

$$Z = \lceil \log_2(N) \rceil \tag{5.7}$$

$$t_m = (n_g \times (t_r + t_h + t_p) + t_a) + Q\log(Q) \tag{5.8}$$

$$Merge\ ATries\ Cost = \vee_{i=1}^{\lfloor N/2 \rfloor} t_{m_{1i}} + \sum_{j=2}^{Z} \vee_{i=1}^{\lfloor N_j/2 \rfloor} t_{m_{ji}} \tag{5.9}$$

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

5.5.2 Model Evaluation

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

1. *Effect of Number of Threads:* Figure 5.12 shows the comparison between execution time predicted by the model and actual execution time from the experiment for a varying number of processors. As shown in the figure, the estimated execution time from the cost model is close to the actual execution time from the experiment in all cases, which demonstrates the effectiveness of our cost model.

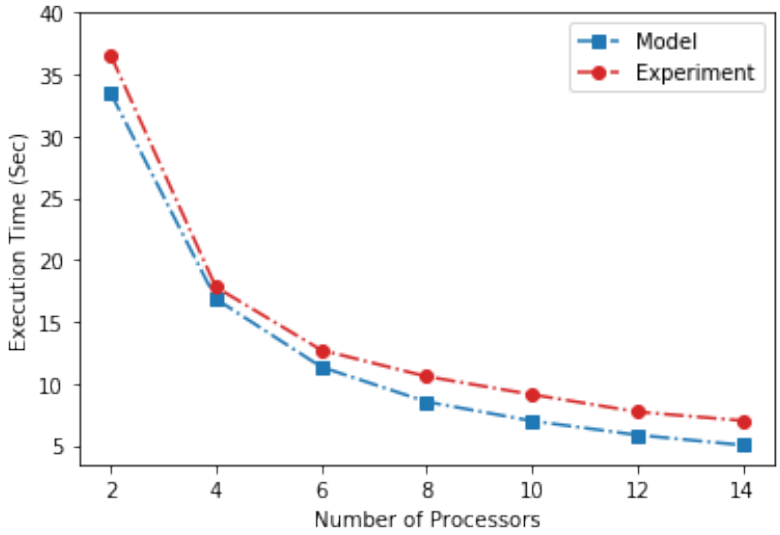


Figure 5.12: Comparison between experiment result and cost model result for a varying number of threads (SF = 100).

Two factors account for the difference between the estimated and actual execution time:

i. The threads executing in parallel often access shared resources and the slowing down is caused by the *interference* between the threads trying to access the shared resources.

ii. Usually, threads communicate with each other and the thread wanting to *communicate* with others may be forced to wait for other threads to be ready for communication. These two factors are extremely difficult to account for in the cost model.

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

$$error\ rate = \left| \frac{experiment\ time - model\ time}{experiment\ time} \right| \tag{5.10}$$

Table 5.5: Comparison between experiment result and cost model result for SSBM queries and error rate of estimated performance (N = 14, SF = 100)

Query	Model (Sec)	Experiment (Sec)	Error Rate (%)
2.1	10.42	10.5	4.4
2.2	5.56	6.3	5.41
2.3	3.7	4.5	3.67
3.1	16.55	17.38	4.77
3.2	6.46	6.97	7.3
3.3	4.91	5.09	3.53
3.4	4.87	5.04	3.36
4.1	17.46	16.97	2.88
4.2	14.63	15.12	3.24
4.3	5.8	7.107	18.3

Table 5.5 shows the comparison between execution time predicted by the model and execution time of the experiment for three query flights in SSBM. In all the cases, the estimated execution time of the cost model is close to the actual execution time obtained by the experiment, which again demonstrates the effectiveness of our cost model. The differences between the estimated and the actual cost can be attributed to *Start-up cost* associated with initiating multiple threads. The start-up time of the threads varies, making it difficult to estimate and include it in the cost model accurately.

To check whether there is a significant difference between the model and 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 0.8453 which is much larger than the significance level of 0.05.

Therefore, we accept the null hypothesis, and we conclude that there is no significant difference between the values of the model and the experiment.

5.6 Summary

In this chapter, we proposed a parallel star group join algorithm for in-memory column-stores called *ATrie Group Join (ATGJ)*. This 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 enabled us to perform join, grouping and aggregation operations on the fly.

Experimental results show that ATGJ outperforms all 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. Furthermore, we demonstrated that ATGJ scales better than other algorithms for the number of concurrent threads, the number of group-by attributes, data set size, and query complexity. We also proposed an analytical model to understand and predict the query performance of ATGJ. Our evaluation shows that the model can predict the performance with 95% confidence.

Chapter 6

Distributed Star Group-By Joins

In this chapter, we focus on the distributed star group-by join for column-stores. First, we first introduce the technical challenges of answering distributed star group-by join queries for column-oriented data. Then, we discuss a new distributed star group-by join algorithm know as *Distributed ATrie Group Join (DATGJ)* that uses ATrie (refer Section 5.2) and progressive materialisation technique (refer Section 4.3) to perform group-by after join. An experiment on a SSBM dataset is conducted to verify the efficiency of our proposed solution.

6.1 Overview: Challenges and Solution

An increasing number of companies rely on the results of massive data analytics for critical business decisions. Such analysis is crucial to improve service quality, support novel features, and detect changes in patterns over time. Usually, the volume of data to be stored and processed is so large that traditional relational database management systems (a.k.a. row-stores) are no longer practical [85]. An alternative to row-stores are column-stores which store information about a logical entity as separate columns in multiple locations on the disk [31]. This novel layout improves the query performance on analytical workloads [30, 31]. Subsequently, several companies have developed column-oriented distributed data storage and processing systems on large clusters of thousands of shared-nothing commodity computers [10, 48, 52].

Over the years, the underlying distributed engines have benefited from retrofitting well-known techniques from row-stores. For instance, query optimisation and several execution alternatives in distributed scenarios have been adapted and generalised from those in row-stores. One of the most common operations over data sources is the joining

on a set of attributes. Fortunately, many of the lessons learned over the years regarding the optimisation and execution of join queries can be translated directly to distributed processing. At the same time, some of the unique characteristics of distributed scenarios have produced new challenges and opportunities for efficient and reliable join processing. In this section, we discuss the challenges posed by big data in distributed join processing and how to address them by focusing on the specific issue of how to improve the performance of star join queries.

6.1.1 Excessive Network Communication and Disk Spill

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 [48] for batch processing [49, 50]. 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 [50]. Although Spark facilitates joins and group-by using Resilient Distributed Datasets (RDDs) [51], Spark SQL [52] and Spark DataFrame [52] operations, it can only process two tables at a time inducing multiple scans of data for star joins and requires one or two map-reduce iterations per join [53]. 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 between the worker nodes as shown in Figure 6.1.

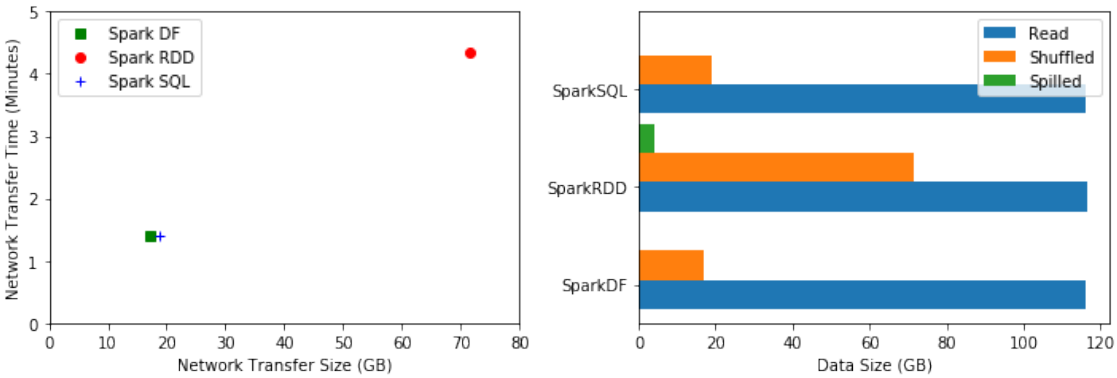


Figure 6.1: Average disk access and network transfers communication for SparkRDD, SparkDF and SparkSQL based joins for SSBM Queries [SF=200, Nodes=5, Number of Cores=35 (7 per node) and Total Memory=150GB (30GB per node)]

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 [46]. Therefore, naive Spark implementation fails to handle the issues such as multiple scans of data, excessive network communication and disk spill.

6.1.2 Technical Contributions

This chapter extends the work described in Chapter 5 where we presented *ATrie Group Join (ATGJ)* and compared its performance against parallel star join algorithms, Invisible Join [30], Nimble Join [44] and In-Memory Aggregation [16], in a multi-threaded environment. To solve the aforementioned challenges, we developed *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 [107], linear probing [108], Robin hood hashing [56], 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. Experiment 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 technical contributions:

1. We present a new optimisation technique for efficient search in the hash table. The key idea is to use Robin Hood hashing [56] with an upper limit imposed on the number of probes which is implemented in *Fast Hash Table (FHT)*.

2. 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.

3. 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.

4. 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.

6.2 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 cannot 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 [56] with an upper limit imposed on the number of probes. If an element must 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 of a hash table lookup. The FHT implementation involves open addressing [107], linear probing [108], Robin hood hashing [56], 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.

6.2.1 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 can 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). $log_2(1009) \approx 10$, 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 8: Search Fast Hash Table

```
Data: FindKey key
Result: EntryPointer ep
1  index <- hashPolicy.indexForHash(hashObject(key))
2  ep <- entries + index
3  distance <- 0
4  while true do
5      if ep.distanceFromDesired < distance then
6          return end
7      else if comparesEqual(key, ep.value) then
8          return ep
9      distance++
10     ep++
11 end
```

Algorithm 8 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.

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.

6.2.2 Evaluation

We experimented to identify the differences in the performance and memory consumption of Standard-Chain Hash Table (SCHT), Concise Hash Table (CHT) [54], Concise Array Table (CAT) [54], Multi-attribute Array Table (MAAT) [44] 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.

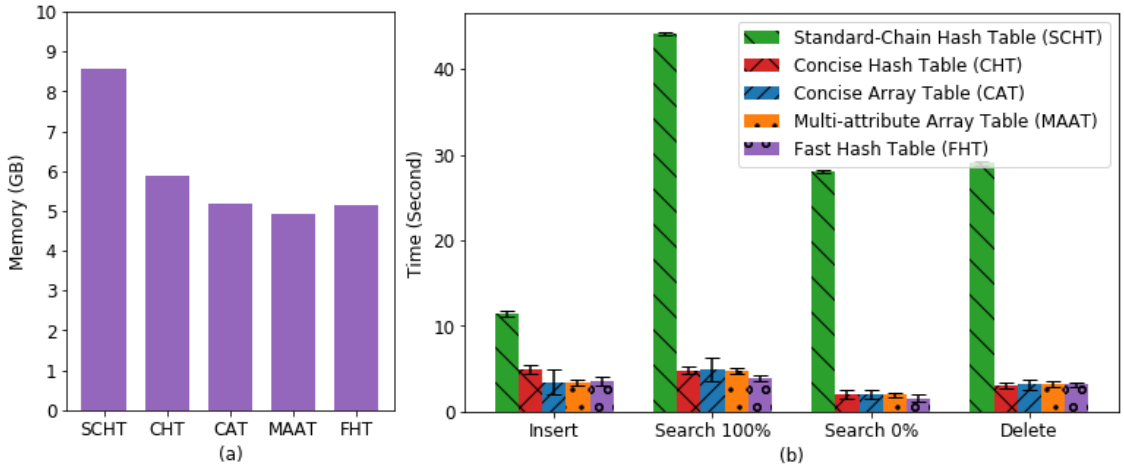


Figure 6.2: (a) Memory usages comparison of various data structures (b) Performance comparison of various data structures to `insert` a new key-value pair and `search` or `delete` the value associated with a key (Search 100% = 100% Successful and Search 0% = 100% Unsuccessful).

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 6.2 (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 6.2 (a).

2. **Insert Performance**
   Figure 6.2 (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 6.2 (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.

6.3 Distributed ATrie Group Join (DATGJ)

The Distributed ATrie Group Join (DATGJ) is a distributed single-scan join algorithm based on ATrie. In this section, we discuss the details of group-by join and its implementation in a distributed environment.

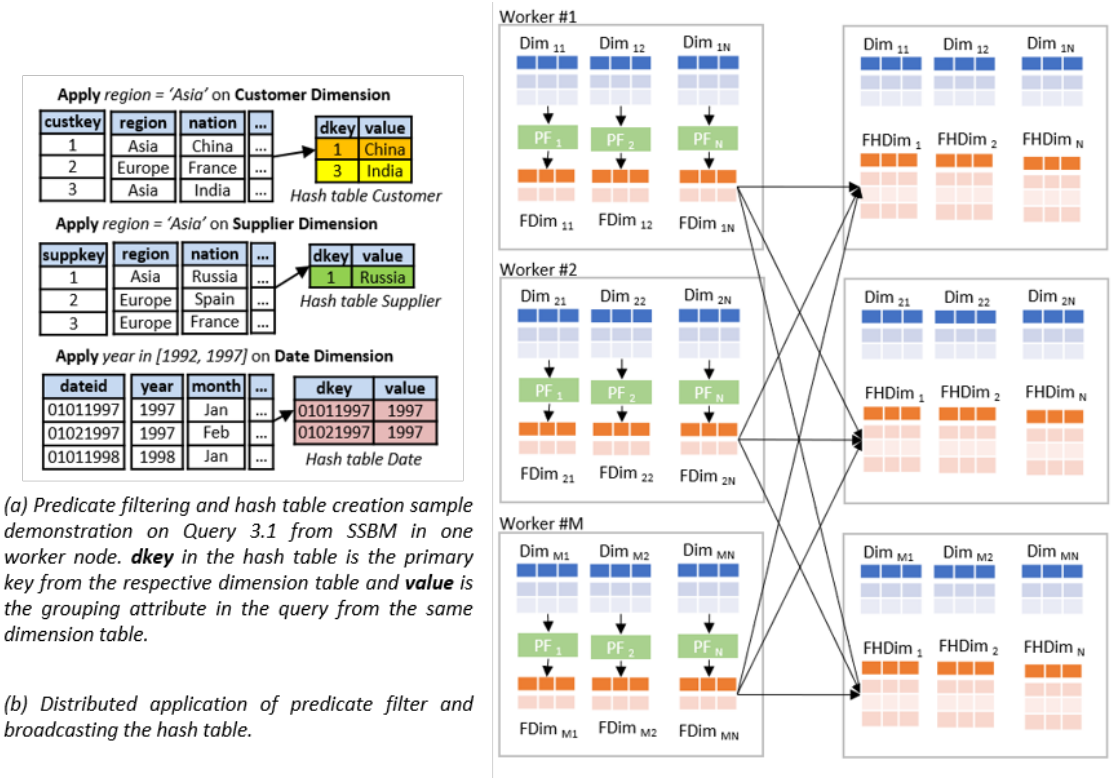
6.3.1 Join Processing Method

The Distributed ATrie Group Join (DATGJ) has four different phases: 1. Broadcast Phase, 2. Single Scan Hash Join, 3. Group-By using ATrie and 4. Merge ATries.

1. **Broadcast Phase:** In this phase, predicates are applied to the appropriate dimension tables to create 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 FHDims 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 [51,52]. 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 6.3 (a).

   In Figure 6.3 (b), 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 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.

   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.



*(a) 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.*

(b) Distributed application of predicate filter and broadcasting the hash table.

Figure 6.3: Applying the predicate filter and broadcasting the hash table

2. **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 an equal amount of data. Since each task has an equal number of records to work on, there is no load imbalance in the join and group-by processing. The single scan is performed on the fact table columns and broadcast FHTs are used to perform the join. Each task works on its respective data partition to retrieve the foreign key and probe it into the corresponding FHTs to create a group aggregation object (GAO) (refer Definition 5.2.1).

   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 [9]. In addition, if parallel

probing results in an unbalanced workload, it can be handled using techniques such as Morsel-driven parallelism [102] or Index Vector Partitioning (IVP) [83].

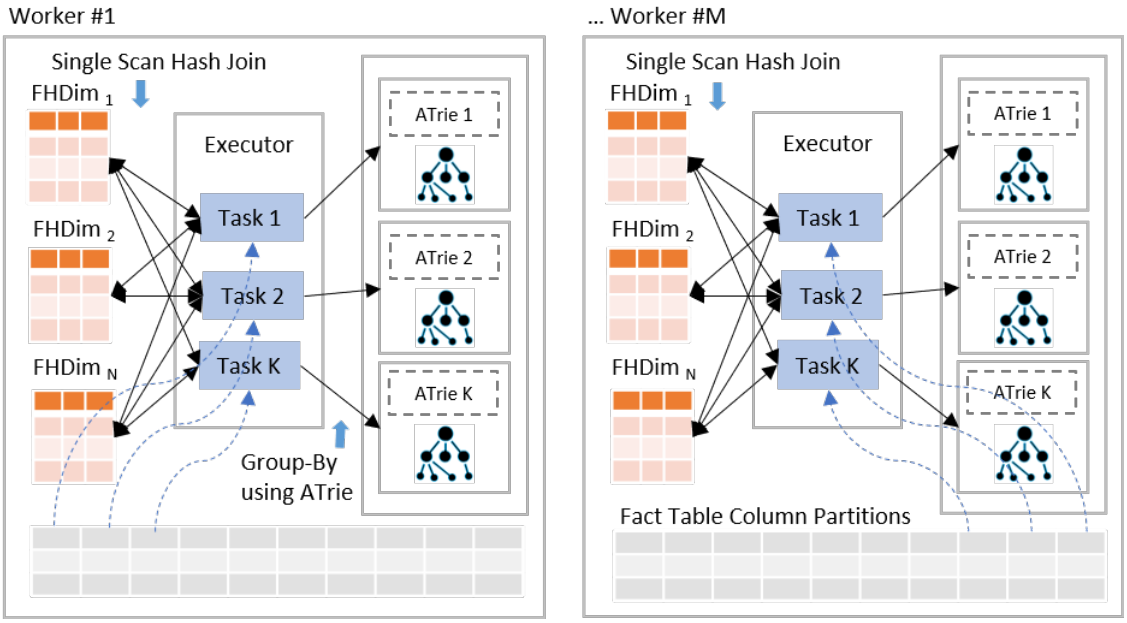


Figure 6.4: Join using a broadcast hash table and group-by using ATrie

In Figure 6.4, each Task $t = 1 \ldots K$ works on the independent partition of data from the fact table columns and performs join with all broadcast hash dimension tables $FHDim_i$ where $i = 1 \ldots N$. Each record in the fact table columns is scanned only once during the join process, unlike other algorithms such as Invisible Join [30] which reduces the disk access time in the algorithm.

3. **Group-By Using ATrie:** We create a task local ATrie and the GAO is inserted into these ATries. Insertion proceeds by walking the ATrie according to attributes in GAO, then appending the new node for an attribute that is not present in 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 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 6.4, each Task $t = 1 \ldots K$ work on a task local ATrie, $ATrie1$, $ATrie2$, $\ldots, ATrieK$. The GAOs created during *Single Scan Hash Join* are inserted into these ATries to group attributes on the fly.

4. **Merge ATries:** All the 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 the fact 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.

6.4 Experimental Evaluation

In this section, we give a brief description of the environment used and present a detailed analysis of the results.

6.4.1 Experimental Setup

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 6.5. The detailed instruction to set up the standalone cluster is available in Appendix B.

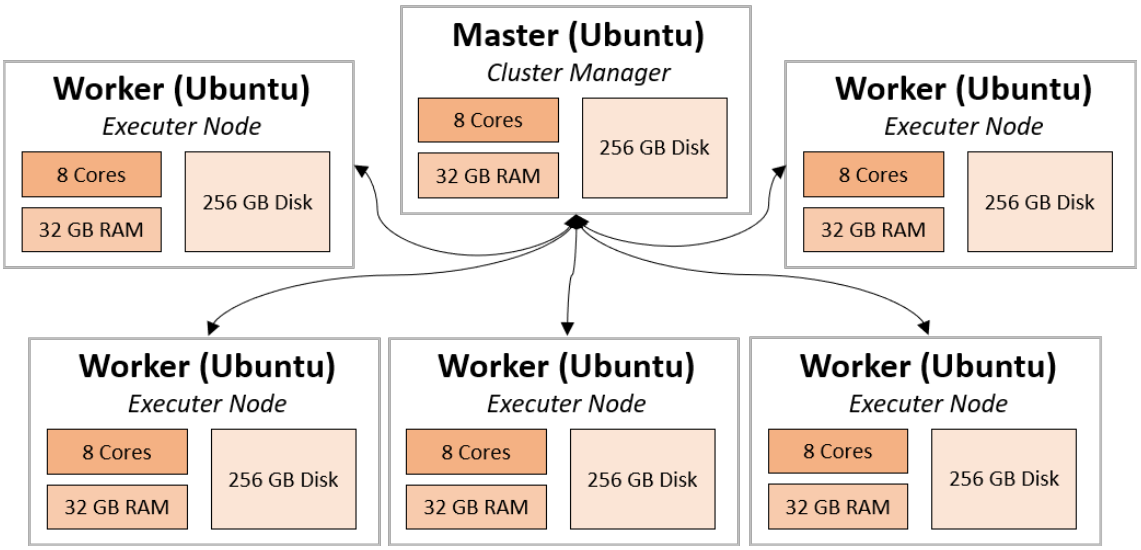


Figure 6.5: Apache Spark Standalone Cluster with one master and five worker nodes.

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

Benchmark Dataset: We used SSBM for the experiment. 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 other works [16, 30, 44, 46, 78], 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 6.1.

Table 6.1: 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)
50	300 Million	30 GB
100	600 Million	60 GB
150	900 Million	90 GB
200	1.2 Billion	120 GB

6.4.2 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) [46]:A join that processes star join queries using bloom filters, and is resilient when there is scant memory.
- Spark Broadcast Join (SBJ) [46]: 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 chapter.

6.4.3 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 35 cores (5 nodes) of the cluster on the SSBM dataset SF = 200.

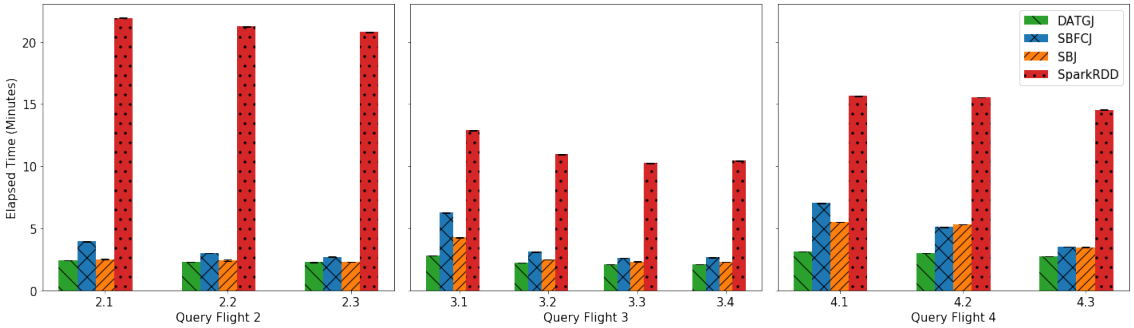


Figure 6.6: Elapsed time of all algorithms by SSBM query flights (# Worker Nodes = 5 and SF = 200).

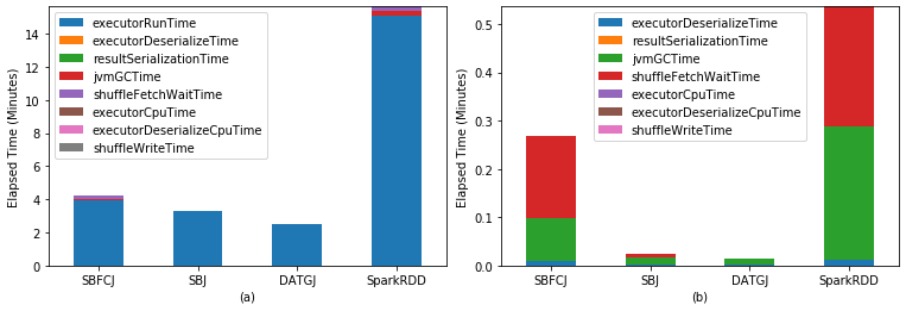


Figure 6.7: Average elapsed time of all algorithms (# Worker Nodes = 5 and SF = 200).

Figure 6.6 shows the results of total elapsed time broken down by the query flight; Figure 6.7 shows the average result for all SSBM queries. The definition of metrics in Figure 6.7 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.

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 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 proposed novel grouping technique and progressive materialisation.

In Figure 6.7 (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 the SBJ and the DATGJ. Upon further inspection, after removing *executorRunTime* (refer Figure 6.7 (b)), we observe the significant time taken for *jvmGCTime*

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

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 6.2. *executorRunTime* includes time to fetch shuffle data [82]. Therefore, we are unable to include time to fetch shuffle data in Table 6.2.

Table 6.2: 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 [46]. Table 6.2 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 [9]. 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 investigated 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 6.8 (a) shows the linear increase of 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.

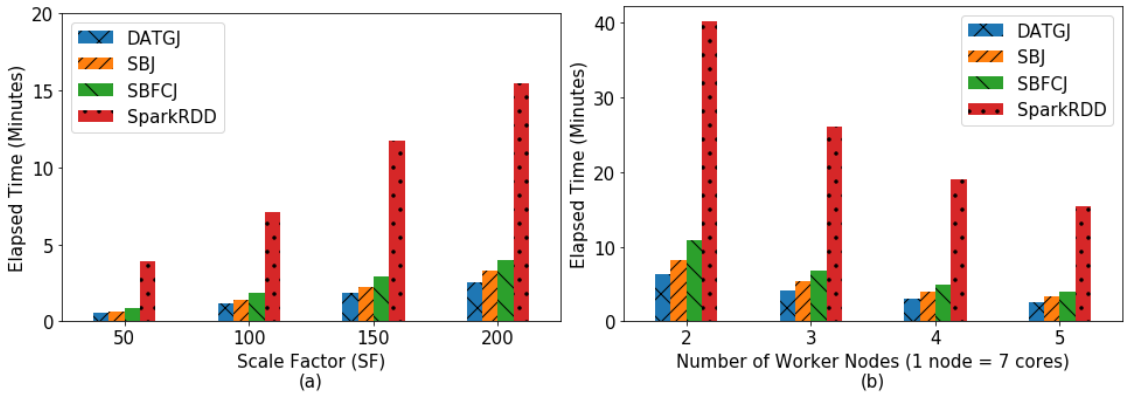


Figure 6.8: (a) Impact of Scale Factor (SF) on the performance of the algorithms (# Worker Nodes = 5). (b) Impact of the number of worker nodes on the scalability of the algorithms (SF = 200).

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 investigated 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 6.8 (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 [9]. 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 [44]. 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.

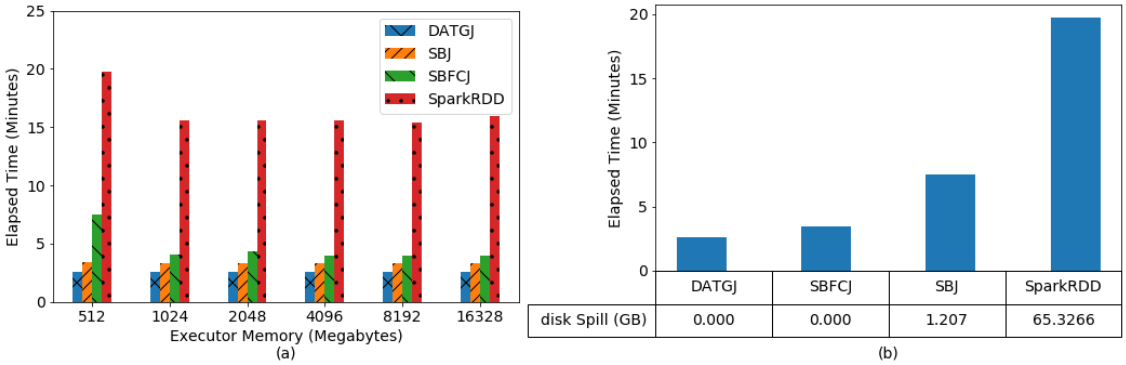


Figure 6.9: (a) Performance of Algorithms under different memory conditions (# Worker Nodes = 5 and SF = 200). (b) Disk spill for 512 MB memory.

Figure 6.9 (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).

In the 512 MB scenario, SBFCJ and SparkRDD algorithms cause disk spills as shown in Figure 6.9 (b). Spilling occurs when the data-storage memory is insufficient. Generally, there are three occasions when data spilling occurs:

a) Hash Table Broadcast: Broadcast methods usually demand more memory to allocate dimension tables [46]. If the memory is insufficient, hash tables are spilled into the disk [9].

b) 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 [52].

c) 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 [9, 109].

As discussed in Section 6.4.3.2, 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

Table 6.3: 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

queries such as 4.1 and 4.2 are above the threshold of 512 MB as shown in Table 6.3. 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 broadcast 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 6.3 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 3.3, 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.

6.5 Analytical Evaluation

In this section, we 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.

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

6.5.1 Cost Models

The parameters used to create the cost model are listed in Table 6.4. The symbols used in the formula: $\lceil\rceil$ is a ceiling function, $\lfloor\rfloor$ is a floor function and $\vee$ means maximum.

Table 6.4: 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.

$$SC = \vee_{i=1}^{n_d} \frac{\mid D_i \mid}{N} \times (t_r + t_f) \tag{6.1}$$

$$Scan\ Cost = \vee_{j=1}^{N} SC_j \tag{6.2}$$

$$HC = \vee_{i=1}^{n_d} \frac{\mid D_i \mid}{N} \times \sigma_{d_i} \times (t_h + t_w) \tag{6.3}$$

$$Hash\ Cost = \vee_{j=1}^{N} HC_j \tag{6.4}$$

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

$$Hash\ Table\ Transfer\ Cost = \sum_{i=1}^{n_d} \frac{|\ H_i\ |}{P} \times\ (m_p + m_l) \tag{6.5}$$

$$Hash\ Table\ Receive\ Cost = \sum_{i=1}^{n_d} \frac{|\ H_i\ |}{P} \times\ m_p \tag{6.6}$$

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.

$$PC = \vee_{i=1}^{n_p} \left(\frac{|\ F_i\ |}{N} \times\ n_d \times\ t_r \right) + \left(\log_2 \left(\frac{|\ F_i\ |}{N} \right) \times\ n_d \times\ t_p \right) \tag{6.7}$$

$$Probe\ Cost = \vee_{j=1}^{N} PC_j \tag{6.8}$$

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.

$$CA = \vee_{i=1}^{n_p} \frac{|\ F_i\ |}{N} \times\ \sigma_{f_i} \times\ (\ n_g \times (t_h + t_p + t_w) + t_a) \tag{6.9}$$

$$Create\ ATrie\ Cost = \vee_{j=1}^{N} CA_j \tag{6.10}$$

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.

$$ATrie\ Transfer\ Cost = \sum_{i=1}^{n_p} \frac{|\ A_i\ |}{P} \times\ (m_p + m_l) \tag{6.11}$$

$$ATrie\ Receive\ Cost = \sum_{i=1}^{n_p} \frac{|\ A_i\ |}{P} \times\ m_p \tag{6.12}$$

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.

$$Z = \lceil \log_2(n_p) \rceil \tag{6.13}$$

$$t_m = (n_g \times (t_r + t_h + t_p) + t_a) + \log_{3Q}(Q) \tag{6.14}$$

$$Merge\ ATries\ Cost = \vee_{i=1}^{\lfloor n_p/2 \rfloor} t_{m_{1i}} + \sum_{j=2}^{Z} \vee_{i=1}^{\lfloor n_{pj}/2 \rfloor} t_{m_{ji}} \tag{6.15}$$

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

6.5.2 Model Evaluation

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

1. *Effect of Data size and Number of Nodes:* Figure 6.10 (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.

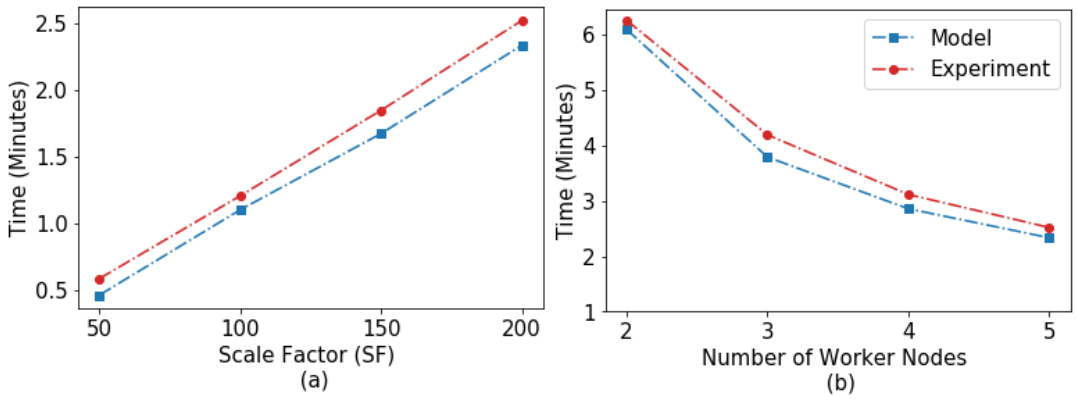


Figure 6.10: (a) Comparison of experiment result and cost model result for varying data sizes (N = 5). (b) Comparison of experiment result and cost model result for a varying number of worker nodes (SF = 200).

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

$$error\ rate = \left| \frac{experiment\ time - model\ time}{experiment\ time} \right| \tag{6.16}$$

Table 6.5: Comparison of experiment results and cost model results for SSBM queries and error rate of estimated performance (N = 5, SF = 200)

Query	Model (Minutes)	Experiment (Minutes)	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

Table 6.5 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 the cases, which again demonstrates the effectiveness of our cost model.

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.

6.5.3 Analysis

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

i. 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 ime 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.

ii. 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.

iii. 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.

6.6 Summary

In this chapter, we proposed 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.

Chapter 7

Conclusion

We have explored parallel and distributed join and group-by algorithms for column-oriented data storage. This chapter concludes the thesis by summarising the research contributions and offering suggestions for future research directions.

7.1 Summary of Contributions

In this section, we summarise the contributions made by this research, with an emphasis on the strengths of the developed algorithms and all associated new components that have been introduced to achieve the research goal.

7.1.1 Nimble Join

In Chapter 4, we presented Nimble Join: a new progressive join algorithm for column-stores. Nimble Join uses multi-attribute array table (MAAT) to hold attributes required to process join query and facilitates progressive materialisation. Some of the advantages that MAAT and progressive materialisation offer are:

1. **Small Memory Footprint:** MAAT used memory space that is thinner than that of other data structures, so they packed better into cache lines.

2. **Faster Look-ups:** MAAT included a reduced list of positions used to probe the indexed array, thereby drastically minimising the number of array lookups depending on the join selectivity.

3. **No Multiple Access:** For the progressive materialisation strategy, as soon as a column is accessed, the attribute value satisfying the predicate is added to MAAT

and the column will not need to be re-accessed, thereby avoiding the performance penalty incurred by having to re-access the column for tuple reconstruction.

Experiment results showed that Nimble Join has 2X faster initial response time, 10% - 25% better execution time, 40% - 50% reduced memory consumption and approximately 50% reduced disk I/O time compared to the competing column-store join algorithm. We also formulated the cost models for Nimble Join, and instantiated and evaluated these models.

7.1.2 ATrie Group Join (ATGJ)

In Chapter 5, we presented the ATrie Group Join (ATGJ): a parallel star group join algorithm for in-memory column-stores. ATGJ utilised 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 represented grouping attributes on the edges and accumulated aggregates on the leaf node of ATrie. This helped us perform join, grouping and aggregation operation on the fly.

ATrie has three important properties that improved the performance of ATGJ:

1. **Deterministic Property:** Each distinct grouping attribute object (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 operation resulting in good insertion performance while grouping the attributes.

2. **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 ATrie, allowing the representation to use less space than it would take to list all the distinct GAOs separately.

3. **Progressive Materialisation:** ATGJ maneuver the idea of progressive materialisation by using ATrie as a vessel to perform materialisation and aggregation on the fly while scanning the fact columns and inserting GAOs into ATrie, avoiding re-scanning and multiple access of data.

Experimental results showed that ATGJ outperformed all 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. Also, we demonstrated that ATGJ scales better than other algorithms for the number of concurrent threads, number

of group-by attributes, data set size and query complexity. We also formulated the cost models for ATGJ, and instantiated and evaluated these models.

7.1.3 Distributed ATrie Group Join (DATGJ)

In Chapter 6, we extended ATGJ to work in a distributed environment known as Distributed ATrie Group Join (DATGJ). DATGJ used the divide and broadcast-based approach which enabled it to perform a single scan of the fact columns, joined the data using fast hash table (FHT), and completely avoided the shuffling of data during the group join processing.

FHT implementation uses open addressing [107], linear probing [108], Robin Hood hashing [56], and the prime number amount of slots with the upper limit on the number of probes. These four methods are common in hash table implementation; however, our new contribution and the primary source of speed-up in a single scan hash join was the setting of an upper limit for the number of probes.

The main advantage of FHT was that it enabled **faster search**. The FHT search algorithm performed at most $log_2(n)$ iterations. Normally, the worst-case time complexity for search in a hash table is $\mathcal{O}(n)$. However, in FHT, it was $\mathcal{O}(log_2(n))$. This was significant because, with linear probing, it is highly likely that we encounter the worst-case scenario since linear probing usually groups elements together.

The search performance was achieved at the expense of additional memory. The memory overhead of the search operation was one byte per item. One byte was padded out to the alignment of the data type that was inserted. For instance, if we inserted `int`, the one byte received three bytes of padding. Hence, we had four bytes of overhead per item. If we inserted `pointers`, there were seven bytes of padding such that we had eight bytes of overhead per item. We could change the memory layout to solve this problem, but it would incur two cache misses for each lookup instead of one cache miss. Therefore, the memory overhead was one byte per item plus padding, as the performance advantages provided by additional memory usages made it worthwhile for us to implement it as such in FHT.

DATGJ used ATrie to perform group-by and aggregation, and process the data in tight loops. Although ATGJ improved performance, it was limited by the hardware as the join operation is performed using a single computer. DATGJ used a multiple worker computer and the distributed computing improved scalability, fault tolerance and resource sharing, and helped perform computation tasks efficiently.

Experimental results showed that DATGJ outperformed all the competing algorithms. For all the queries evaluated, on average, DATGJ was 1.5X to 5X faster than the competing

algorithms. Furthermore, we demonstrated that DATGJ has zero disk spills, zero data shuffle and minimal network transfer, and performs well in memory-constrained conditions. We also formulated the cost models for DATGJ, and instantiated and evaluated these models.

7.2 Future Research

We hope that this PhD research will lead to more research into column-store joins as we have shown the potential for possible improvements, and anticipate that more is achievable. Below, we suggest two interesting areas of research deserving future investigation.

7.2.1 Online Aggregation

Aggregation is an increasingly important operation in today's database management systems [110]. As data sets grow larger and both users and user interfaces become more sophisticated, there is an increasing emphasis on extracting not just specific data items, but also general characterisations of large subsets of the data. Users want this aggregate information right away, even though producing it may involve accessing and condensing enormous amounts of information. Unfortunately, aggregation processing closely resembles batch processing where users submit an aggregation query and are forced to wait without feedback while the system churns through billions of records [110, 111]. Only after a significant period of time does the system respond with the (usually small) final answer. A particularly frustrating aspect of this problem is that aggregation queries are typically used to obtain a "rough picture" of a large body of information; yet, they are computed with painstaking precision, even in situations where an acceptably precise approximation might be available very quickly. In Chapter 4, we touched briefly upon the aspect of online aggregation; however, there is the possibility of changing the interface to aggregation processing and, by extension, changing the aggregation processing itself. The idea is to perform aggregation online to allow users both to observe the progress of their queries and to control execution on the fly.

7.2.2 Column-Store Specific Features

Future work could extend the algorithms so as to include more features of column-stores such as column-specific compression and direct operations on compressed data.

Column-specific compression: The data stored in columns is more compressible than the data stored in rows. Compression algorithms perform better on data with low entropy (i.e. with high data value locality), and values from the same column tend to have more value locality than values from different columns [31]. By compressing each column using the compression method that is most effective for it, a substantial reduction in the total size of the data on disk or in memory can be achieved. By storing data from the same attribute (column) together, we can obtain good compression ratios using simple compression schemes such as Run-length Encoding, Bit-Vector Encoding and Dictionary Encoding [25].

Direct operation on compressed data: The column-oriented compression schemes mentioned above can be operated directly without decompression. We can delay decompression of the data until it is absolutely necessary, ideally until results need to be presented to the user. Working over compressed data can significantly improve the utilisation of memory bandwidth which is one of the major causes of bottlenecks [31]. It also gives the maximum boost to performance, since the system saves I/O by reading in less data without incurring the decompression cost [28].

Bibliography

[1] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee, A design science research methodology for information systems research, Journal of Management Information Systems 24 (3) (2007) 45–77.

[2] B. Marr, Big data facts: How many companies are really making money from their data? (Apr 2016).

[3] N. Marz, J. Warren, Big Data: Principles and best practices of scalable real-time data systems, New York; Manning Publications Co., 2015.

[4] D. J. Abadi, P. A. Boncz, S. Harizopoulos, Column-oriented database systems, Proceedings of the VLDB Endowment 2 (2) (2009) 1664–1665.

[5] G. P. Copeland, S. N. Khoshafian, A decomposition storage model, in: ACM SIGMOD Record, Vol. 14, ACM, 1985, pp. 268–279.

[6] R. MacNicol, B. French, Sybase iq multiplex-designed for analytics, in: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, VLDB Endowment, 2004, pp. 1227–1230.

[7] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al., C-store: a column-oriented dbms, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment, 2005, pp. 553–564.

[8] N. Mukherjee, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase, A. Holloway, H. Jin, J. Kamp, K. Kulkarni, et al., Distributed architecture of oracle database in-memory, Proceedings of the VLDB Endowment 8 (12) (2015) 1630–1641.

[9] D. Taniar, C. H. Leung, W. Rahayu, S. Goel, High performance parallel database processing and grid databases, Vol. 67, John Wiley & Sons, 2008.

[10] A. Kemper, T. Neumann, Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots, in: Data Engineering (ICDE), 2011 IEEE 27th International Conference on, IEEE, 2011, pp. 195–206.

[11] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, W. Lehner, Sap hana database: data management for modern business applications, ACM Sigmod Record 40 (4) (2012) 45–51.

[12] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al., Db2 with blu acceleration: So much more than just a column store, Proceedings of the VLDB Endowment 6 (11) (2013) 1080–1091.

[13] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, V. Papadimos, Real-time analytical processing with SQL server, Proceedings of the VLDB Endowment 8 (12) (2015) 1740–1751.

[14] P.-A. Larson, E. N. Hanson, M. Zwilling, Evolving the architecture of SQL server for modern hardware trends, in: 2015 IEEE 31st International Conference on Data Engineering, IEEE, 2015, pp. 1239–1245.

[15] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, et al., Oracle database in-memory: A dual format in-memory database, in: 2015 IEEE 31st International Conference on Data Engineering, IEEE, 2015, pp. 1253–1258.

[16] S. Chavan, A. Hopeman, S. Lee, D. Lui, A. Mylavarapu, E. Soylemez, Accelerating joins and aggregations on the oracle in-memory database, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, 2018, pp. 1441–1452.

[17] P. A. Boncz, M. Zukowski, N. Nes, Monetdb/x100: Hyper-pipelining query execution., in: CIDR, Vol. 5, 2005, pp. 225–237.

[18] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, S. Madden, Hyrise: a main memory hybrid storage engine, Proceedings of the VLDB Endowment 4 (2) (2010) 105–116.

[19] M. Heimel, M. Saecker, H. Pirk, S. Manegold, V. Markl, Hardware-oblivious parallelism for in-memory column-stores, Proceedings of the VLDB Endowment 6 (9) (2013) 709–720.

[20] A. Tsang, M. Olschanowsky, A study of database 2 customer queries, IBM Santa Teresa Laboratory, San Jose, CA, Technical Report TR-03-413 (1991).

[21] S. Chaudhuri, K. Shim, Including group-by in query optimization, in: VLDB, Vol. 94, 1994, pp. 354–366.

[22] M. Eich, P. Fender, G. Moerkotte, Efficient generation of query plans containing group-by, join, and groupjoin, The VLDB Journal—The International Journal on Very Large Data Bases 27 (5) (2018) 617–641.

[23] J. Aguilar-Saborit, V. Muntés-Mulero, C. Zuzarte, J.-L. Larriba-Pey, Star join revisited: Performance internals for cluster architectures, Data & Knowledge Engineering 63 (3) (2007) 997–1015.

[24] S. Idreos, M. L. Kersten, S. Manegold, Self-organizing tuple reconstruction in column-stores, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, ACM, 2009, pp. 297–308.

[25] D. Abadi, S. Madden, M. Ferreira, Integrating compression and execution in column-oriented database systems, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, ACM, 2006, pp. 671–682.

[26] M. Zukowski, S. Heman, N. Nes, P. Boncz, Super-scalar ram-cpu cache compression, in: 22nd International Conference on Data Engineering (ICDE'06), IEEE, 2006, pp. 59–59.

[27] S. Idreos, M. L. Kersten, S. Manegold, et al., Database Cracking., in: CIDR, Vol. 3, 2007, pp. 1–8.

[28] D. J. Abadi, D. S. Myers, D. J. DeWitt, S. R. Madden, Materialization strategies in a column-oriented dbms, in: 2007 IEEE 23rd International Conference on Data Engineering, IEEE, 2007, pp. 466–475.

[29] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, C. Bear, Materialization strategies in the vertica analytic database: Lessons learned, in: 2013 IEEE 29th International Conference on Data Engineering (ICDE), IEEE, 2013, pp. 1196–1207.

[30] D. J. Abadi, S. R. Madden, N. Hachem, Column-stores vs. row-stores: How different are they really?, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, 2008, pp. 967–980.

[31] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al., The design and implementation of modern column-oriented database systems, Foundations and Trends in Databases 5 (3) (2013) 197–280.
URL http://dx.doi.org/10.1561/1900000024

[32] Z. Li, K. A. Ross, Fast joins using join indices, The VLDB Journal—The International Journal on Very Large Data Bases 8 (1) (1999) 1–24.

[33] P. A. Boncz, S. Manegold, M. L. Kersten, et al., Database architecture optimized for the new bottleneck: Memory access, in: VLDB, Vol. 99, 1999, pp. 54–65.

[34] S. Manegold, P. Boncz, M. Kersten, Optimizing main-memory join on modern hardware, IEEE Transactions on Knowledge and Data Engineering 14 (4) (2002) 709–730.

[35] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, P. Dubey, Sort vs. hash revisited: fast join implementation on modern multi-core cpus, Proceedings of the VLDB Endowment 2 (2) (2009) 1378–1389.

[36] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al., Monetdb: Two decades of research in column-oriented database architectures, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 35 (1) (2012) 40–45.

[37] S. Begley, Z. He, Y.-P. P. Chen, Pameco join: A parallel main memory compact hash join, Information Systems 58 (2016) 105–125.

[38] C. A. Galindo-Legaria, T. Grabs, S. Gukal, S. Herbert, A. Surna, S. Wang, W. Yu, P. Zabback, S. Zhang, Optimizing star join queries for data warehousing in microsoft sql server, in: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, IEEE, 2008, pp. 1190–1199.

[39] S. Blanas, Y. Li, J. M. Patel, Design and evaluation of main memory hash join algorithms for multi-core cpus, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM, 2011, pp. 37–48.

[40] M.-C. Albutiu, A. Kemper, T. Neumann, Massively parallel sort-merge joins in main memory multi-core database systems, Proceedings of the VLDB Endowment 5 (10) (2012) 1064–1075.

[41] C. Balkesen, G. Alonso, J. Teubner, M. T. Özsu, Multi-core, main-memory joins: Sort vs. hash revisited, Proceedings of the VLDB Endowment 7 (1) (2013) 85–96.

[42] Ç. Balkesen, J. Teubner, G. Alonso, M. T. Özsu, Main-memory hash joins on modern processor architectures, IEEE Transactions on Knowledge and Data Engineering 27 (7) (2015) 1754–1766.

[43] P. Sangat, M. Indrawan-Santiago, D. Taniar, Sensor data management in the cloud: Data storage, data ingestion, and data retrieval, Concurrency and Computation: Practice and Experience 30 (1) (2018) e4354.

[44] P. Sangat, D. Taniar, M. Indrawan-Santiago, C. Messom, Nimble join: A parallel star join for main memory column-stores, Concurrency and Computation: Practice and Experience (2019) e5616.

[45] P. Sangat, D. Taniar, C. Messom, ATrie Group Join: A Parallel Star Group Join and Aggregation for In-Memory Column-Stores, IEEE Transactions on Big Data 30 (1) (2020) 1253–1258.

[46] J. J. Brito, T. Mosqueiro, R. R. Ciferri, C. D. de Aguiar Ciferri, Faster cloud star joins with reduced disk spill and network communication, Procedia Computer Science 80 (2016) 74–85.

[47] K. Sridhar, Big data analytics using sql: Quo vadis?, in: International Conference on Research and Practical Issues of Enterprise Information Systems, Springer, 2017, pp. 143–156.

[48] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al., Apache spark: a unified engine for big data processing, Communications of the ACM 59 (11) (2016) 56–65.

[49] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, ACM, 2009, pp. 165–178.

[50] O. Polychroniou, W. Zhang, K. A. Ross, Distributed joins and data placement for minimal network traffic, ACM Transactions on Database Systems (TODS) 43 (3) (2018) 14.

[51] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on

Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.

[52] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., Spark sql: Relational data processing in spark, in: Proceedings of the 2015 ACM SIGMOD international conference on management of data, ACM, 2015, pp. 1383–1394.

[53] V. Purdilă, Ş.-G. Pentiuc, Single-scan: a fast star-join query processing algorithm, Software: Practice and Experience 46 (3) (2016) 319–339.

[54] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, D. Sharpe, Memory-efficient hash joins, Proceedings of the VLDB Endowment 8 (4) (2014) 353–364.

[55] E. Fredkin, Trie memory, Commun. ACM 3 (9) (1960) 490–499. doi:10.1145/367390.367400.

[56] P. Celis, P.-A. Larson, J. I. Munro, Robin hood hashing, in: 26th Annual Symposium on Foundations of Computer Science (sfcs 1985), IEEE, 1985, pp. 281–288.

[57] P. Sangat, D. Taniar, C. Messom, Distributed ATrie Group Join: Towards Zero Network Cost, IEEE Access 8 (2020) 111598–111613.

[58] S. Watanabe, K. Fujimoto, Y. Saeki, Y. Fujikawa, H. Yoshino, Column-oriented database acceleration using fpgas, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 686–697.

[59] D. S. Batory, On searching transposed files, ACM Transactions on Database Systems (TODS) 4 (4) (1979) 531–544.

[60] I. Karasalo, P. Svensson, An overview of cantor-a new system for data analysis., in: SSDBM, Vol. 83, 1983, pp. 315–324.

[61] I. Karasalo, P. Svensson, The design of cantor-a new system for data analysis., in: SSDBM, Vol. 86, 1986, pp. 224–244.

[62] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, P. Valduriez, A query processing strategy for the decomposed storage model, in: Data Engineering, 1987 IEEE Third International Conference on, IEEE, 1987, pp. 636–643.

[63] S. Khoshafian, P. Valduriez, Parallel execution strategies for declustered databases, in: Database Machines and Knowledge Base Machines, Springer, 1988, pp. 458–471.

[64] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, P. Boncz, Positional update handling in column stores, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM, 2010, pp. 543–554.

[65] A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, D. Barton, Big data, The management revolution. Harvard Bus Rev 90 (10) (2012) 61–67.

[66] R. R. Schaller, Moore's law: past, present and future, IEEE spectrum 34 (6) (1997) 52–59.

[67] L. G. Roberts, Beyond Moore's law: Internet growth trends, Computer 33 (1) (2000) 117–119.

[68] A. Kumar, J. Naughton, J. M. Patel, X. Zhu, To join or not to join? Thinking twice about joins before feature selection, in: Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD, Vol. 16, 2016, pp. 54–65.

[69] M. Zukowski, P. A. Boncz, N. Nes, S. Héman, Monetdb/x100-a dbms in the cpu cache., IEEE Data Eng. Bull. 28 (2) (2005) 17–22.

[70] S. K. Begley, Z. He, Y.-P. P. Chen, Mcjoin: a memory-constrained join for column-store main-memory databases, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 121–132.

[71] S. Jha, B. He, M. Lu, X. Cheng, H. P. Huynh, Improving main memory hash joins on intel xeon phi processors: An experimental approach, Proceedings of the VLDB Endowment 8 (6) (2015) 642–653.

[72] P. O'Neil, G. Graefe, Multi-table joins through bitmapped join indices, ACM SIGMOD Record 24 (3) (1995) 8–11.

[73] V. Markl, F. Ramsak, R. Bayer, Improving olap performance by multidimensional hierarchical clustering, in: Database Engineering and Applications, 1999. IDEAS'99. International Symposium Proceedings, IEEE, 1999, pp. 165–177.

[74] A. Weininger, Efficient execution of joins in a star schema, in: Proceedings of the 2002 ACM SIGMOD international conference on Management of Data, ACM, 2002, pp. 542–545.

[75] Z. Fang, Z. He, J. Chu, C. Weng, Simd accelerates the probe phase of star joins in main memory databases, in: International Conference on Database Systems for Advanced Applications, Springer, 2019, pp. 476–480.

[76] J. Aguilar-Saborit, V. Muntés-Mulero, C. Zuzarte, J.-L. Larriba-Pey, Ad hoc star join query processing in cluster architectures, in: International Conference on Data Warehousing and Knowledge Discovery, Springer, 2005, pp. 200–209.

[77] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, G. Graefe, Query processing techniques for solid state drives, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, ACM, 2009, pp. 59–72.

[78] Y. Yuan, R. Lee, X. Zhang, The yin and yang of processing data warehousing queries on gpu devices, Proceedings of the VLDB Endowment 6 (10) (2013) 817–828.

[79] Z. Guoliang, W. Guilan, GBFSJ: Bloom filter star join algorithms on gpus, in: 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), IEEE, 2015, pp. 2427–2431.

[80] J. Goldstein, P.-Å. Larson, Optimizing queries using materialized views: a practical, scalable solution, ACM SIGMOD Record 30 (2) (2001) 331–342.

[81] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, C. Bear, The vertica analytic database: C-store 7 years later, Proceedings of the VLDB Endowment 5 (12) (2012).

[82] Monitoring and instrumentation (2019).
URL https://spark.apache.org/docs/latest/monitoring.html

[83] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, A. Ailamaki, Scaling up concurrent main-memory column-store scans: towards adaptive numa-aware data and task placement, Proceedings of the VLDB Endowment 8 (12) (2015) 1442–1453.

[84] C. Ordonez, Can we analyze big data inside a dbms?, in: Proceedings of the Sixteenth International Workshop on Data Warehousing and OLAP, ACM, 2013, pp. 85–92.

[85] K. Sridhar, Modern column stores for big data processing, in: International Conference on Big Data Analytics, Springer, 2017, pp. 113–125.

[86] A. Datta, D. VanderMeer, K. Ramamritham, Parallel star join+ dataindexes: Efficient query processing in data warehouses and olap, IEEE Transactions on Knowledge and Data Engineering 14 (6) (2002) 1299–1316.

[87] H. Han, H. Jung, H. Eom, H. Y. Yeom, Scatter-gather-merge: An efficient star-join query processing algorithm for data-parallel frameworks, Cluster Computing 14 (2) (2011) 183–197.

[88] C. Zhang, L. Wu, J. Li, Efficient processing distributed joins with bloom filter using MapReduce, International Journal of Grid and Distributed Computing 6 (3) (2013) 43–58.

[89] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970) 422–426.

[90] Y. Ramdane, N. Kabachi, O. Boussaid, F. Bentayeb, SkipSJoin: A New Physical Design for Distributed Big Data Warehouses in Hadoop, in: International Conference on Conceptual Modeling, Springer, 2019, pp. 255–263.

[91] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, Y. Tian, A comparison of join algorithms for log processing in MapReduce, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 975–986.

[92] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, S. Wu, Llama: leveraging columnar storage for scalable join processing in the MapReduce framework, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, 2011, pp. 961–972.

[93] H. Zhu, M. Zhou, F. Xia, A. Zhou, Efficient star join for column-oriented data store in the MapReduce environment, in: 2011 Eighth Web Information Systems and Applications Conference, IEEE, 2011, pp. 13–18.

[94] G. Zhou, Y. Zhu, G. Wang, Cache conscious star-join in MapReduce environments, in: Proceedings of the 2nd International Workshop on Cloud Intelligence, ACM, 2013, p. 1.

[95] L. Cheng, S. Kotoulas, T. E. Ward, G. Theodoropoulos, Improving the robustness and performance of parallel joins over distributed systems, Journal of Parallel and Distributed Computing 109 (2017) 310–323.

[96] R. H. Von Alan, S. T. March, J. Park, S. Ram, Design science in information systems research, MIS Quarterly 28 (1) (2004) 75–105.

[97] S. T. March, G. F. Smith, Design and natural science research on information technology, Decision support systems 15 (4) (1995) 251–266.

[98] P. O'Neil, E. O'Neil, X. Chen, S. Revilak, The star schema benchmark and augmented fact table indexing, in: Technology Conference on Performance Evaluation and Benchmarking, Springer, 2009, pp. 237–252.

[99] J. Sanchez, A review of star schema benchmark, arXiv preprint arXiv:1606.00295 (2016).

[100] G. Graefe, R. Bunker, S. Cooper, Hash joins and hash teams in Microsoft SQL Server, in: VLDB, Vol. 98, Citeseer, 1998, pp. 86–97.

[101] N. Askitis, Fast and compact hash tables for integer keys, in: Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91, Australian Computer Society, Inc., 2009, pp. 113–122.

[102] V. Leis, P. Boncz, A. Kemper, T. Neumann, Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, 2014, pp. 743–754.

[103] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, M. Zhang, In-memory big data management and processing: A survey, IEEE Transactions on Knowledge and Data Engineering 27 (7) (2015) 1920–1948.

[104] Q. Cai, H. Zhang, W. Guo, G. Chen, B. C. Ooi, K.-L. Tan, W.-F. Wong, Memepic: Towards a unified in-memory big data management system, IEEE Transactions on Big Data 5 (1) (2018) 4–17.

[105] O. Polychroniou, K. A. Ross, Vectorized bloom filters for advanced simd processors, in: Proceedings of the Tenth International Workshop on Data Management on New Hardware, ACM, 2014, p. 6.

[106] J. Jannink, Implementing deletion in b+-trees, ACM Sigmod Record 24 (1) (1995) 33–38.

[107] J. I. Munro, P. Celis, Techniques for collision resolution in hash tables with open addressing, in: Proceedings of 1986 ACM Fall joint computer conference, IEEE Computer Society Press, 1986, pp. 601–610.

[108] P. Flajolet, P. Poblete, A. Viola, On the analysis of linear probing hashing, Algorithmica 22 (4) (1998) 490–515.

[109] F. Kastrati, G. Moerkotte, Optimization of conjunctive predicates for main memory column stores, Proceedings of the VLDB Endowment 9 (12) (2016) 1125–1136.

[110] J. F. Naughton, Technical perspective: Optimized wandering for online aggregation, ACM SIGMOD Record 46 (1) (2017) 32–32.

[111] F. Li, B. Wu, K. Yi, Z. Zhao, Wander join: Online aggregation via random walks, in: Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 615–629.

Appendix A

SSBM Query Definitions

Query Flight (QF) 1: SSBM starts with a query flight having restrictions on only one dimension. The query quantifies the amount of revenue increase that would have resulted from eliminating certain company-wide discounts in a given percentage range for products shipped in a given year. This is a "what if" query to find possible revenue increases.

```
select sum(lo_extendedprice * lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_year = [YEAR] -- Specific values below
and lo_discount between [DISCOUNT] - 1 and [DISCOUNT] + 1
and lo_quantity < [QUANTITY];
```

In QF 1, lo_quantity is restricted, not just to the lower half of the range, but to different ranges with different filter factors. QF 1 has three queries.

Q 1.1 YEAR = 1993, DISCOUNT = 2, QUANTITY = 25, so predicates are d_year = 1993, lo_quantity $<$ 25, lo_discount between 1 and 3.

```
select sum(lo_extendedprice*lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_year = 1993
and lo_discount between1 and 3
and lo_quantity < 25;
```

Filter Factor (FF) = (1/7)*0.5*(3/11) = 0.0194805. Number of lineorder rows selected, for SF = 1, is 0.0194805*6,000,000 ≈ 116,883.

Q 1.2 d_yearmonthnum = 199401, lo_quantity between 26 and 35, lo_discount between 4 and 6.

```
select sum(lo_extendedprice * lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_yearmonthnum = 199401
and lo_discount between4 and 6
and lo_quantity between 26 and 35;
```

FF = (1/84)*(3/11)*0.2 = 0.00064935. Number of lineorder rows selected, for SF = 1: 0.00064935*6,000,000 ≈ 3896.

Q 1.3 d_weeknuminyear = 6 and d_year = 1994, lo_quantity between 36 and 40, lo_discount between 5 and 7.

```
select sum(lo_extendedprice * lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_datekey
and d_weeknuminyear = 6
and d_year = 1994
and lo_discount between 5 and 7
and lo_quantity between 26 and 35;
```

FF = (1/364)*(3/11)*0.1 = .000075. Number of lineorder rows selected, for SF = 1, is .000075*6,000,000 ≈ 450.

QF 2: For a second query flight, the queries have restrictions on two dimensions. The queries compares revenue for some product classes, for suppliers in a certain region, grouped by more restrictive product classes and all years of orders. QF 2 has three queries.

Q 2.1 p_category = 'MFGR#12', s_region = 'AMERICA'

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
```

```
and p_category = 'MFGR#12'
and s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

p_category = 'MFGR#12', FF = 1/25; s_region, FF=1/5. So LINEORDER FF = (1/25)*(1/5) = 1/125. Number of lineorder rows selected, for SF = 1, is (1/125)*6,000,000 $\approx$ 48,000

Q 2.2 p_category = 'MFGR#12' changed to p_brand1 between 'MFGR#2221' and 'MFGR#2228' and s_region to 'ASIA'.

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1 between 'MFGR#2221' and 'MFGR#2228'
and s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

So lineorder FF = (1/125)*(1/5) = 1/625. Number of lineorder rows selected, for SF = 1, is (1/625)*6,000,000 $\approx$ 9600.

Q 2.3 p_category = 'MFGR#12' changed to p_brand1 = 'MFGR#2339' and s_region = 'EUROPE'.

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey
and lo_partkey = p_partkey
and lo_suppkey = s_suppkey
and p_brand1 = 'MFGR#2221'
and s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

So lineorder FF = (1/1000)*(1/5) = 1/5000. Number of lineorder rows selected, for SF = 1, is (1/5000)*6,000,000 ≈ 1200.

QF 3: In the third query flight, restrictions are placed on three dimensions, including the remaining dimension, customer. The query provides revenue volume for lineorder transactions by customer nation and supplier nation and year within a given region, within a certain time period. QF 3 has four queries.

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_region = 'ASIA'
and s_region = 'ASIA'
and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Q 3.1 Q3 as written: c_region = 'ASIA' so FF = 1/5 for customer, FF = 1/5 for supplier, and 6-year period FF = 6/7 for d_year; Thus, LINEORDER FF = (1/5)*(1/5)*(6/7) = 6/175 and the number of lineorder rows selected, for SF = 1, is (6/175)*6,000,000 ≈ 205,714.

Q 3.2 Restriction is changed to a certain nation, and within that nation, revenue by customer city and supplier city, and year.

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and c_nation = 'UNITED STATES'
and s_nation = 'UNITED STATES'
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

The c_nation and s_nation restriction has FF = 1/25; so lineorder FF is (1/25)*(1/25)*(6/7) = 6/4375. The number of lineorder rows selected, for SF = 1, is (6/4375)*6,000,000 ≈

8,228.

Q 3.3 Restriction is changed to two cities in 'UNITED KINGDOM'; retrieve c_city and group by c_city.

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1'or c_city='UNITED KI5')
and (s_city='UNITED KI1' or s_city='UNITED KI5')
and d_year >= 1992 and d_year <= 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

The c_nation and s_nation restriction has FF = (2/10)*(1/25)= 1/125; so lineorder FF is (1/125)*(1/125)*(6/7) = 6/109375. The number of lineorder rows selected, for SF = 1, is (6/109375)*6,000,000 $\approx$ 329.

Q 3.4 Query drills down in time to just one month, to create a "needle-in-haystack" query.

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_orderdate = d_datekey
and (c_city='UNITED KI1' or c_city='UNITED KI5')
and (s_city='UNITED KI1' or s_city='UNITED KI5')
and d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

So lineorder FF is (1/125)*(1/125)*(1/84) = 1/1,312,500. The number of lineorder rows selected, for SF = 1, is (1/1,312,500)*6,000,000 $\approx$ 5.

QF 4: This query flight represents a "What-If" sequence, of the OLAP type. It starts with a group by on two dimensions and rather weak constraints on three dimensions, and measure the aggregate profit, measured as (lo_revenue - lo_supplycost).

```
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;
```

Q 4.1 Query QF 4 written as restriction on region FFs 1/5 each, p_mfgr restriction 2/5. FF on lineorder = (1/5)(1/5)*(2/5) = 2/125. So the number of lineorder rows selected for SF = 1 is (2/125)*6,000,000 $\approx$ 96000.

Q 4.2 Assume that in Q 4.1 output, we find a surprising growth of 40% in profit from year 1997 to year 1998, uniform across c_nation. (This need not be true for the data we actually examine.) We would probably want to pivot to group by year, s_nation and a further breakdown by p_category to see where the change arises.

```
select d_year, s_nation, p_category,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
and lo_partkey = p_partkey
and lo_orderdate = d_datekey
and c_region = 'AMERICA'
and s_region = 'AMERICA'
and (d_year = 1997 or d_year = 1998)
and (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;
```

This has the same FF as Q4.1 except in time and accesses 2/7 of the same lineorder data; for that data it simply has a different group by dimension breakout. Its FF = (2/7)*(2/125)

= 4/875. So the number of lineorder rows selected for SF = 1 is (4/875)*6,000,000 ≈ 27,428.

Q 4.3 Assume that as a result of Q 4.2, a great percentage of the profit increase from year 1997 to 1998 comes from s_nation = 'UNITED STATES' and p_category = 'MFGR1#4'. Now we might want to drill down to cities in the United States and into p_brand1 (within p_category).

```
select d_year, s_city, p_brand1,
sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey
and lo_suppkey = s_suppkey
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 = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1;
```

The FF for c_region is 1/5 and for s_nation is 1/25; the FF for d_year remains at 2/7, and the restriction on p_category is now 1/25. Thus the lineorder FF is: (1/5)*(1/25)*(2/7)*(1/25) = 2/21875. The number of lineorder rows retrieved for SF = 1 is (2/21875)*6,000,000 ≈ 549.

Appendix B

Setting up the Standalone Cluster

We used Ubuntu 18.04.02 LTS (Bionic Beaver) as the operating system and installed the following tools and technologies for the experiment.

1. **Python** as a programming language
2. **Jupyter Notebook** as an IDE for python development
3. **Apache Spark** as a big data processing tool

We will go through one by one the installation of required software for the experiment.

Install JAVA

We updated the local `apt` package index and then downloaded and installed the packages:

```
$ sudo apt update
```

Apache Spark needs JAVA to run. We installed JAVA by typing:

```
$ sudo apt install openjdk-8-jdk
```

Install Spark

We installed Apache Spark using the following commands:

```
$ wget http://.../spark-2.4.0-bin-hadoop2.7.tgz
$ tar zxvf spark-2.4.0-bin-hadoop2.7.tgz
$ sudo nano .bashrc
# Spark
export SPARK_HOME="/home/ubuntu/spark-2.4.0-bin-hadoop2.7/"
$ source .bashrc
```

Setting up the Spark Cluster (Key-less entry)

We set up password-less SSH access from the master machine to the others. This required having the same user account on all the machines, creating a private SSH key for it on the master via ssh-keygen, and adding this key to the .ssh/authorized_keys file of all the workers. We followed the commands below to setup the key-less entry:

```
# On master: run ssh-keygen accepting default options
$ ssh-keygen -t rsa
Enter file in which to save the key (/home/you/.ssh/id_rsa): [ENTER]
Enter passphrase (empty for no passphrase): [EMPTY]
Enter same passphrase again: [EMPTY]
```

```
# On workers:
# copy ~/.ssh/id_rsa.pub from your master to the worker, then use:
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys (on the workers)
$ chmod 644 ~/.ssh/authorized_keys
```

On Master, we edited the conf/slaves file on the master and fill in the workers' host names (ip address). Then we copied "conf/slaves.template" to create "conf/slaves". On Master and Slaves, we edited the conf/spark-env.sh.template to create conf/spark-env.sh and included the following lines:

```
export SPARK_MASTER_IP=<IP ADDRESS OF MASTER>
export SPARK_MASTER_HOST=<IP ADDRESS OF MASTER>
export SPARK_MASTER_PORT=7077
export SPARK_LOCAL_DIRS=/mnt/spark_tmp_dir
# We need to create the directory and
# change ownership to user ubuntu
```

```
export PYSPARK_PYTHON=/usr/bin/python3
export PYSPARK_DRIVER_PYTHON=/usr/bin/ipython
```

To start the cluster, we ran the following command on the master node. It is important to run it there rather than on a worker.

```
$ spark-2.4.0-bin-hadoop2.7/sbin/start-all.sh
# If everything started, the cluster manager's web UI should appear
# at http://<masternode i.e ipaddress of master>:8080 and show all
# your workers.
```

To stop the cluster, we ran the following command on the master node.

```
$ spark-2.4.0-bin-hadoop2.7/sbin/stop-all.sh
```

Install Jypyter Notebook

Anaconda is an open-source package manager, environment manager, and distribution of the Python and R programming languages that can be used to install jypter notebook. *Note: Installation is required in every node to avoid error 13: Permission denied.*

```
# Download Anaconda bash script:
$ wget https://repo.anaconda.com/archive/
Anaconda3-5.2.0-Linux-x86_64.sh
# Verify Data integrity of the installer
$ sha256sum Anaconda3-5.2.0-Linux-x86_64.sh
# Run the anaconda script
$ bash Anaconda3-5.2.0-Linux-x86_64.sh
# Activate installation
$ source ~/.bashrc
```

Once conda is installed, we created a conda virtual environment called jupyter:

```
$ conda create -n jupyter
$ source activate jupyter
(jupyter) ubuntu@mu-master:~$ conda install jupyter
```

We edited the config to make notebook accessible from outside. This needs to be done only on master node.

```
(jupyter) ubuntu@mu-master:~$ jupyter notebook --generate-config
(jupyter) ubuntu@mu-master:~$ sudo nano .jupyter/
jupyter_notebook_config.py
# Find #c.NotebookApp.ip = 'localhost' and replace with
c.NotebookApp.ip = '*'
```

Running the cluster

In Master, we ran the code below:

```
$ spark-2.4.0-bin-hadoop2.7/sbin/start-all.sh
```

This starts all the master and the slaves.

Tmux is a good option available to run Jupyter Notebook in the background. We executed the code below to start the jupyter notebook.

```
~$ source activate jupyter
(jupyter).....~$ jupyter notebook
```

PhD made me poorer, without money, but richer in thoughts.

- Lailah Gifty Akita