Nimble join: A parallel star join for main memory column-stores

Prajwol Sangat  |  David Taniar  |  María Indrawan-Santiago  |  Christopher Messom

Faculty of Information Technology, Monash University, Clayton, VIC, Australia

Correspondence
David Taniar, Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia.
Email: david.taniar@monash.edu

Summary
Column-stores perform significantly better than row-stores on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications. As mainstream data warehouses are growing into multi-terabyte range, decision support queries should be processed in parallel to achieve adequate performance. Researchers of the column-oriented join queries assume an unlimited reserve of main memory and focus on minimising execution time. However, some analytics require a large amount of memory to calculate intermediate results, and some interactive analytics require a fast initial response time even though queries need to process a large amount of data. Motivated by these requirements, we present a new progressive parallel star algorithm for main memory column-stores known as “Nimble Join.” Equipped with multi-attribute array table and a novel progressive materialisation technique, Nimble Join requires half the memory space and has two times faster initial response whilst having comparable execution time to the existing algorithm.

KEYWORDS
column-stores, joins, main memory, parallel processing

1 | INTRODUCTION

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. For example, information about a customer such as name, address, and phone is stored as separate columns on disk. This data storage model is known as Decomposed Storage Model (DSM). 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.

With the column-oriented storage architecture, there are two main challenges in join operation. The first challenge is to join two columns. There has been a lot of research regarding efficient column-oriented joins, but most of these algorithms assume an unlimited reserve of main memory and focus on minimising the total execution time. However, the main memory will eventually be exhausted when the input tables or intermediate results will exceed the available memory because of the increase in data. Furthermore, from a user’s perspective, it is ideal if the first few results can be generated quickly with minimal response time so that they can start data processing immediately.

The second challenge is to join two tables in column-stores. Since the columns are stored separately, the joining of two tables will involve additional join operations such as tuple reconstruction, projection, grouping, and sorting in addition to joining columns specified in query Join condition. Existing join algorithms such as Jive Join and Radix Join provide only a partial solution to process queries involving tables join 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, filter conditions, and operations such as group-by and sort. Abadi et al proposed an algorithm to answer the table join queries in column-stores that will 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.

Mainstream data warehouses today hold several terabytes of data with table sizes passing the one billion row threshold. Therefore, the decision support queries need to be processed in parallel to gain performance improvements. The recent works on efficient parallel join algorithms show that carefully tuned join implementations can exhibit good performance regardless of the data size.
In this paper, we present a progressive parallel star join algorithm known as "Nimble Join" that 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.\textsuperscript{10} We have equipped nimble join with an extended version of the concise array table\textsuperscript{27} 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 the work of Abadi et al.\textsuperscript{10} It is much thinner than the hash table and packs better into cache lines. Furthermore, it stores only 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 used in the work of Abadi et al.\textsuperscript{10} and significantly lowers the query processing time in our join algorithm. Third, it facilitates the design of nimble join in such a way that the join results can be produced faster with the help of progressive materialisation.

In summary, we make the following contributions in this paper:

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, 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 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.

The rest of this paper is organised as follows: First, we describe the related works in Section 2. Then, we describe the problem statement in Section 3. After that, we describe the proposed algorithm in Section 4 and experimental evaluation in Section 5. Next, we describe our analytical evaluation in Section 6. Finally, we conclude in Section 7.

## 2 RELATED WORKS

In this section, we will present the existing work in three different areas of research: parallel hash joins, parallel star joins, and joins for column-stores.

### 2.1 Parallel hash joins

Main memory hash joins have been the focus of recent research in the domain of column stores. The groundbreaking work of MonetDB\textsuperscript{9} lead to radix hash join that aimed at overcoming the bottleneck of cache and Translation Lookaside Buffer (TLB) misses. Kim et al.\textsuperscript{6} postulated the radix hash join\textsuperscript{9} for parallel processing based on repeatedly partitioning the input relations. They compared hash join and sort-merge join that are optimised for modern multi-core systems. Their experimental results show that sort-merge joins will become faster with wider SIMD instructions and limited per-core bandwidth.

Blanas et al.\textsuperscript{12} extended the categories of main memory multi-core join algorithms and introduced no-partitioning hash join. They suggested that a simple, hardware oblivious non-partitioning hash join can offer superior performance to the partitioning joins. They compared their algorithm with the work of Kim et al.\textsuperscript{6} and demonstrated that 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.\textsuperscript{13} extended the design space of the main memory sort-merge join algorithm focusing on NUMA systems. They advocated a sort-merge algorithm for the main memory parallel joins, whereas others, such as the works of Balkesen et al.\textsuperscript{14,15} 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.

Further improvements on the works of Kim et al.\textsuperscript{6} and Blanas et al.\textsuperscript{12} have been achieved by Balkesen et al.\textsuperscript{14,15} They achieved higher throughputs by improving the cache efficiency of the hash table implementations for both algorithms and adopting a better skew handling mechanism for parallel radix hash join.\textsuperscript{8} They showed that tuned parallel radix hash joins exhibit superior performance than the no-partitioning hash joins on their experimental hardware, which is contradicting to the work of Blanas et al.\textsuperscript{12}

Barber et al.\textsuperscript{17} focused on the memory efficiency of the hash join methods. They proposed a highly memory efficient Concise Hash Table (CHT) and a Concise Array Table (CAT). CAT stores only the payload values, and keys are stored separately as bitmaps. This system works best on moderately dense keys. They compared their join method with no-partitioning hash join,\textsuperscript{12} and the parallel radix hash join.\textsuperscript{14} They showed that they could reduce the memory usages by one to three orders of magnitude with competitive performance. We have based our multi-attribute array table (MAAT) on the concepts of CAT. However, MAAT supports both sparse and dense keys, and the position list of keys facilitates pruning.
Jha et al.\textsuperscript{18} compared the modified variants of no-partitioning hash join\textsuperscript{12} and parallel radix hash join\textsuperscript{6} on the Intel Xeon Phi coprocessors. They demonstrated that under a wider range of parameters, no-partitioning hash joins can match and even outperform parallel radix hash joins and suggested that the main memory hash joins need to be revisited as processor technology changes.

Given the overall high-performance level of hash joins and our ability to utilize the nature of hash tables in storing intermediate values, we have partially based nimble join on the hash join algorithms.

2.2 Parallel star joins

Bitmap star join using bitmap indices proposed by O’Neil and Graefe\textsuperscript{19} suggested that an index lookup in the dimension table could be faster than hash join, whereas Markl et al.\textsuperscript{20} proposed hierarchical physical clustering as an alternative to the use of indices and aimed at limiting the number of I/O access to the fact table. Aguilar et al.\textsuperscript{21} proposed a star hash join based on the use of bloom filters in cluster architectures to reduce both I/O and data traffic communication.

Datta et al.\textsuperscript{22} proposed new parallel star join algorithms based on the vertical partitioning of data in the distributed environment. Weininger\textsuperscript{23} proposed index union and semi-join reduction plans with bitmap filters for efficient execution of star schema joins. Aguilar-Saborit et al.\textsuperscript{24} revisited the star join techniques to analyse the most up-to-date strategy for ad-hoc star join query processing in a cluster of computers. They concluded by proposing the hybrid solution that improved the aspects of O’Neil and Graefe\textsuperscript{19} and Markl et al.\textsuperscript{20} and showed near-optimal results in multiple use-cases. Galindo-Legaria et al.\textsuperscript{25} proposed novel execution strategies for star join queries such as index intersections, dimension cross-product with fact table lookup, and semi-join reduction using bitmap filters. They showed that the optimisation strategies improved the star join performance.

These works are based on the row-oriented storage architecture. However, they have been motivational approaches towards developing star join algorithms for column-stores. Nimble join is inspired from other works\textsuperscript{11,19,21} and follows Volcano style processing,\textsuperscript{25} but it has been designed for column-stores (ie, “once-a-column” style processing for I/O and cache efficiency) as each column is stored as a separate binary file on disk.

2.3 Column-store joins

Li and Ross\textsuperscript{8} proposed a join algorithm with sequential access to the columns at the cost of adding two sorts of the join output data. Boncz et al.\textsuperscript{5} provided a fast mechanism for performing the partition of column positions into the blocks and reordering the intermediate data back to the original join order after the column extraction.

Abadi et al.\textsuperscript{10} extended the work on improving the performance for star joins\textsuperscript{19,23} by taking advantage of the column-oriented layout and rewriting the predicates to avoid the hash lookups. They followed the similar query execution model as Volcano model\textsuperscript{25} enabling parallel query execution. This work forms an essential foundation for the join algorithm proposed in this paper. However, the downside of this algorithm is that it includes a significant memory overhead cost because it creates multiple intermediate position lists and requires more disk I/O compared to our proposed join algorithm.

3 PROBLEM STATEMENT

In this section, we will describe the benchmark used in our experiment and explain the existing algorithm based on a query proposed in the benchmark.

3.1 Star schema benchmark

In this paper, we use the Star Schema Benchmark (SSBM),\textsuperscript{26} which has been widely used in data warehousing research studies.\textsuperscript{10,27} It consists of a single fact table, \texttt{LINEORDER} table, which has foreign key references to four dimension tables, \texttt{CUSTOMER}, \texttt{SUPPLIER}, \texttt{PART}, and \texttt{DATE} tables, as shown in Figure 1. The SSBM consists of thirteen queries divided into four flights:

\textbf{Flight 1:} It consists of three queries. These queries have a restriction on one dimension attribute, as well as the \texttt{DISCOUNT} and \texttt{QUANTITY} columns of the \texttt{LINEORDER} table. These queries measure the increase in revenue (the product of \texttt{EXTENDEDPRICE} and \texttt{DISCOUNT}) that would be achieved if various levels of discount were eliminated for various order quantities in a given year. The selectivity of \texttt{LINEORDER} table for these three queries are $1.9 \times 10^{-2}$, $6.5 \times 10^{-4}$, and $7.5 \times 10^{-5}$, respectively.

\textbf{Flight 2:} It consists of three queries. These queries have a restriction on two dimension attributes and calculate the revenue for particular product classes in particular regions, grouped by product class and year. The selectivity of \texttt{LINEORDER} table for these three queries are $8.0 \times 10^{-3}$, $1.6 \times 10^{-3}$, and $2.0 \times 10^{-4}$, respectively.

\textbf{Flight 3:} It consists of four queries. These queries have a restriction on three dimension attributes and calculate the revenue in a particular region over a time period, grouped by customer nation, supplier nation, and year. The selectivity of \texttt{LINEORDER} table for these four queries are $3.4 \times 10^{-2}$, $1.4 \times 10^{-3}$, $5.5 \times 10^{-3}$, and $7.6 \times 10^{-7}$, respectively.
Flight 4: It consists of three queries. These queries restrict on three dimension attributes and calculate profit \((\text{REVENUE} - \text{SUPPLYCOST})\) grouped by year, nation, and category for query 1, and for queries 2 and 3, region and category. The selectivity of \text{LINEORDER} table for these three queries are \(1.6 \times 10^{-2}, 4.5 \times 10^{-3}, \) and \(9.1 \times 10^{-5}\), respectively.

This schema provides a base “Scale Factor (SF)” that can be used to scale the size of the data. In our experiments, we use a scale factor of 10 yielding the fact table with 60 million tuples unless stated otherwise.

3.2 | Invisible join: An existing work

Let us consider Query 3.1 in Flight 3 from SSBM.

\[
\begin{align*}
\text{SELECT} & \quad \text{c.nation}, \text{s.nation}, \text{d.year}, \\
& \quad \text{sum(lo.revenue)} \text{ AS revenue} \\
\text{FROM} & \quad \text{customer AS c}, \text{lineorder AS lo}, \\
& \quad \text{supplier AS s, [Date] AS d} \\
\text{WHERE} & \quad \text{lo.custkey} = \text{c.custkey} \\
& \quad \text{AND} \quad \text{lo.suppkey} = \text{s.suppkey} \\
& \quad \text{AND} \quad \text{lo.orderdate} = \text{d.orderdate} \\
& \quad \text{AND} \quad \text{c.region} = \text{"ASIA"} \\
& \quad \text{AND} \quad \text{s.region} = \text{"ASIA"} \\
& \quad \text{AND} \quad \text{d.year BETWEEN 1992 and 1997} \\
\text{GROUP BY} & \quad \text{c.nation}, \text{s.nation}, \text{d.year} \\
\text{ORDER BY} & \quad \text{d.year asc, revenue desc};
\end{align*}
\]

This query has a restriction on three dimension attributes and calculate the revenue in a particular region over a period, grouped by customer nation, supplier nation, and year. We will use this query as a running example to explain the invisible join algorithm and to demonstrate the problem.

A query plan for executing the Query 3.1 using invisible join is as follows:

**Phase 1:** Apply a predicate on the dimension table and extract the keys that pass the prediction to build the hash table. \(\text{c.region} = \text{"ASIA"}\) is applied to customer dimension. The customer key is extracted at the positions that matched the predicate and inserted into the corresponding intermediate hash table.

**Phase 2:** Fact table columns are joined with the respective dimension table. Customer key column in the fact table is joined with customer dimension table using this intermediate hash table. The results of the join operation are the lists of positions that passed the join predicate \((\text{c.region} = \text{"ASIA"})\). Similar joins are performed for the supplier and date tables. At the end of this phase, we have three intermediate list of positions, one each for customer, supplier, and date dimension tables. The entire list of positions is intersected together to generate the final list of positions satisfying all the predicates.

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

We have identified three problems with the algorithm: (1) The number of position lists in this algorithm increases with the increasing number of columns. It creates multiple position lists for each dimension table that increases memory consumption. (2) It has to re-scan the columns, which can significantly increase the query execution time because of increased disk I/O. The values need to be extracted in out-of-position order, which
can have significant cost.\textsuperscript{28} (3) The parallelism of the algorithm has not been properly examined to understand how well the algorithm performs in multi-core architecture.

4 | NIMBLE JOIN: PROPOSED ALGORITHM

In this section, we will discuss in detail the data structure and materialisation technique used in the proposed new join algorithm known as “Nimble Join” followed by the details of the join and its implementation in parallel.

4.1 | Multi-attribute array table

Usually, a join operation uses widely known data structure such as hash table or its variant to store intermediate results. Hash tables typically store 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.\textsuperscript{17}

Concise Hash Tables (CHT) and Concise Array Tables (CAT)\textsuperscript{17} consume less memory and has been used to develop memory efficient hash joins that are usually faster than leading in-memory hash joins from the work of Balkesen et al.\textsuperscript{14} We introduce a variant of CAT known as Multi-Attribute Array Table (MAAT) that consists of an indexed array storing multiple attributes as shown in Figure 2. The key advantage offered by an indexed array is the elimination of nodes and pointers that are used in Standard-Chain Hash Table (SCHT).\textsuperscript{29} This configuration permits the good use of CPU cache and hardware data prefetch whilst simultaneously saving memory space. Instead of using a bitmap, MAAT uses a list of signed integer positions that satisfy the predicate.

Bitmaps have been used to store the position values with an objective to reduce the memory consumption.\textsuperscript{17,20,31} However, 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 the selectivity of less than 1% (87% of SSBM queries have < 1% selectivity). If we have 6000 records, the selectivity of the query is 1%, ie, 60 records. Therefore, 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, ie, 4 bytes each, would only need 240 bytes.

It is a common practice in physical modelling of the database to assign primary keys as a serially increasing counter, ie, we could map these keys directly to the positions of an array. Working along the lines of this assumption, in the process of creating tuples progressively, we eliminate the use of primary key values and instead use the key positions. MAAT embeds a list of successful key positions to speed up the final lookups, which directly maps to the positions of an 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 involves only a single DIRA round of N access in an indexed array. Finally, the indexed array has entries only for the positions in the position list.

To facilitate parallel processing of algorithms, MAAT implements the spinlock mechanism that is used to ensure the atomic insert and updates of the data items in the indexed array making it a thread-safe collection. As opposed to other locking mechanism (such as mutex), spinlock is an excellent choice because the critical section in our algorithm such as inserting or updating items in MAAT is considered a minimal amount of work and spinlock works best in such scenarios (for comparison, on average, spinlock was 50ns faster than POSIX mutex). However, locking the entire MAAT for adding or updating an item is not a good idea. Therefore, we employ item level locking similar to the concept of row-level locking in the database. This approach allows different threads to work on different items in MAAT simultaneously giving the true sense of parallelism.

We experimented to identify the 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 total memory used by the programme. The numbers reported are the averages of ten iterations. Figure 3 shows that hash tables are not always the best choice to store intermediate results regarding memory usages and performance.
MAAT offers three main benefits: (1) The memory space MAAT uses is much thinner than other data structures (refer to Figure 3), so they pack better into cache lines. (2) It completely avoids the use of slow hash functions, and (3) 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.

4.2 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 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. Abadi et al. introduced two different materialisation strategies for column-stores: early materialisation (EM) and late materialisation (LM).

Consider a simple example: Suppose a query has three selection operators, $\sigma_1$, $\sigma_2$, and $\sigma_3$, over columns $R.a$, $R.b$, and $R.c$ respectively, where all columns are sorted in the same order and stored in separate files. Let $\sigma_1$ be the most selective and let $\sigma_3$ be the least selective predicate. An Early materialisation strategy would process the query as follows: Read $R.a$ and output the list of positions satisfying $\sigma_1$. Similarly, access $R.b$ and $R.c$ and output the list of positions satisfying $\sigma_2$ and $\sigma_3$ respectively. Use position-wise AND operations to intersect the position lists. Finally, 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 $<R.a, R.b, R.c>$.

Late materialisation strategy would process the query as follows: Access $R.a$ and output the list of positions satisfying $\sigma_1$. Then, access $R.b$ and $R.c$ and output attribute values satisfying $\sigma_2$ and $\sigma_3$ respectively, to MAAT only if MAAT has attributes stored from the previous step. MAAT holds the join output with the records stitched together and the position list of the records that satisfied all the predicates in the query. Instead of re-accessing columns $R.a$, $R.b$ and $R.c$, progressive materialisation maintains MAAT. Use of MAAT in progressive materialisation offers two benefits over late materialisation: (1) The memory space it uses is much smaller than maintaining intermediate position list because MAAT includes only values that satisfy the predicate. (2) MAAT avoids excessive disk access improving the query execution time.

4.3 Join processing methods

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

**Key Hashing:** The predicates are applied to the appropriate dimension table, and the dimension keys and values required in the query are extracted. Using the keys, we create a hash table to test whether a particular key from fact columns satisfy the predicate. An example of this phase for the query mentioned above is shown in Figure 4.

**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 FK column of the fact table is probed against the respective hash table and inserted into MAAT that satisfies the predicate.

The probing process continues for all the FK columns of the fact table involved in the query. A probe sequence for all the FK columns is performed in MAAT, the size of the MAAT reduces as we remove the items that do not satisfy the predicate and save the position of items satisfying the predicate in the position list of
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 performed probing with all the FK columns in the query. An example of this phase for the query mentioned above is shown in Figure 5.

**Value Extraction:** For all the positions in the position list of MAAT, the position can be used to look up for the rest of the 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 also holds the 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 the 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 spent in disk I/O for invisible join is $4t$ whereas it is $t$ for nimble join (scanning `lorevenue`). Finally, arithmetic operations can be performed on the join results to get the required aggregation. An example of this phase is shown in Figure 6.

### 4.4 Parallelising nimble join

In row-stores, the records to join contain redundant information and the use of proper data partitioning technique can significantly improve the parallelism of the join queries. Initially, we attempted to create a full data parallel algorithm that distributed data equally amongst all the threads.
However, we realised that data parallelism is not cost-free. In profiling the code, we noticed that the overhead cost of partitioning the data amongst different threads (e.g., setup cost) exceeded the amount of work done in each thread especially for partitioning dimension table columns because they are significantly smaller in size than fact columns.

The difference in the performance between data parallelism and task parallelism is shown in Figure 7. The result in the Figure is for Query 3.1 in SSBM with SF = 1. The task performed in 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 (however, it does not mean that it never uses data parallelism).

### 4.4.1 Implementation

Let $N$ be the number of process ($P$) that will be used in the join query execution (i.e., $P_1, P_2, \ldots, P_N$). The number of processes that are activated depends on the number of processors available in the system ($\text{Max Number of processes} = \text{Processor Count}$), or if the number of tasks ($n$) is less than processor count, the algorithm will activate only required number of processes.

The algorithm specifies the actions to run concurrently, and the run-time handles all process scheduling details, including scaling automatically to the number of processors (processor count) on the host computer if required. Figure 8 shows the basic overview of the parallel processing model employed by the nimble join. We will discuss each phase in the join based on this parallel processing model below.
**Key Hashing:** The predicates are applied to the appropriate dimension table, and the dimension keys and values required in 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. Since the dimension tables are significantly smaller than the fact table, we do not employ data parallelism because for a smaller set of data, if the operations are not substantial, partitioning the data can be a significant overhead. If the dimension table(s) is large, nimble join chooses to mix data and task parallelism whilst processing that dimension table(s). In the case of the example used above, three processes get activated as we have three dimension tables to hash. The pseudocode for this phase is shown in Algorithm 1. Once all the processes $P_1, P_2,$ and $P_3$ have completed the task of hashing, we move on to the Probing Phase.

**Algorithm 1 Key Hash (Parallel)**

```plaintext
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
```

**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 the 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., data parallelism), which probes respective hash tables to update and synchronise MAAT concurrently. As probes are performed in MAAT, the size of the MAAT reduces as we remove the items that do not satisfy the predicate and save 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 producing results much earlier than that in the work of Abadi et al. The pseudocode for this phase is shown in Algorithm 2.

**Algorithm 2 Probe (Parallel)**

```plaintext
Data: Column[] C
Result: MAAT maat

/* operates columns <= N in parallel */
for column c ∈ C do
  for block b ∈ c do
    /* operates in parallel */
    READ b from disk by pages
    for tuple t ∈ b do
      READ from page and WRITE to memory
      GET key
      if probe(key) is successful then
        ADD to maat and pos. list
      else
        REMOVE from maat
      end
    end
  end
end
```

Since the blocks are equal sized, there are less chances of skew. However, if data parallel probing approach results in an unbalanced workload per thread in case of skew, it can be handled using techniques explained in the works of Leis et al. and Psaroudakis et al.
Value Extraction: Once all the processes $P_1, P_2, \ldots, P_N$ have completed the task of synchronising MAAT, each column required to answer the query is accessed by the threads to retrieve the 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 no write operation after the Probing phase.

MAAT holds the intermediate attributes required for the join query processing, thereby eliminating the re-scanning of fact table columns performed by invisible join\textsuperscript{10} to reconstruct the output tuple. If the query requires additional attributes added to MAAT, it can be done easily via AddorUpdate operation in MAAT. Finally, arithmetic operations are performed on the join results to get the required aggregation. The pseudocode for this phase is shown in Algorithm 3.

### Algorithm 3: Extract Value (Parallel)

**Data:** Column[] C, POSLIST pl

**Result:** MAAT maat

```plaintext
/* operates columns <= N in parallel */
for column c ∈ C do
  for block b ∈ c do
    READ b from disk by pages
    for pos. ∈ pl do
      JUMP to pos. in b, OUTPUT value(s)
      ADD or UPDATE maat
    end
  end
end
```

5 | EXPERIMENTAL EVALUATION

In this section, we will briefly describe the environment used in the experiment. Then, we will present the detailed analysis of the results obtained from the experiment. We have conducted all our experiments on the NeCTAR\textsuperscript{*} server equipped with 12 Intel Xeon E3-12xx (Ivy Bridge) processors clocked at 2.6 GHz, 48 GB of 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.

5.1 | Experimental results

We implemented invisible join in serial and parallel fashion as mentioned in the paper of Abadi et al.\textsuperscript{10} and we propose "Nimble Join" and its parallel implementation. The implementation of the algorithm was purely focused on Join technique without application of column-store optimisation such as column-specific compression\textsuperscript{35} and database cracking and adaptive indexing.\textsuperscript{36} We believe that, with this optimisation, nimble join will perform more effectively.

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. Therefore, a first run was executed to prime the .Net Framework JIT Compiler, and the result was discarded. We forced the garbage collector to run after each iteration so that it will not distort the results. For all the join algorithms, the standard deviation was less than 10% of the average time. We discuss the detailed analysis of the results below.

5.1.1 | Initial response time

Initial response time is the time it takes to yield the first join result to the standard output. Let $t_s$ be the starting time, and let $t_f$ be the time at which the first join result was yielded to standard output; then, the initial response time is $\Delta t_f = t_f - t_s$.

The detail results of initial response time broken down by query flight is shown in Figure 9A, with average results in Figure 9B. It is immediately apparent that the average response time of nimble join is faster than its competing join algorithm in both serial and parallel versions. This performance difference can be attributed to the design of nimble join and its use of MAAT as discussed before. In some of the queries (query 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 waiting for the join query processing to be completed.

\* https://www.nectar.org.au/about-nectar
5.1.2 Total execution time

Total execution time 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 let \( t_e \) be the time at which the last join result was yielded to standard output; then, the total execution time is \( \Delta t_{te} = t_e - t_s \).

The detail results of total execution time broken down by query flight are shown in Figure 10A, with average results across all queries in Figure 10B. 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 5.1.3).

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 \( t_{bs} \) be the starting time, and let \( t_{de} \) be the time at which all disk I/O is completed; then, the time taken for disk I/O is \( \Delta t_{de} = t_{de} - t_{bs} \).
Figure 11 shows the average time taken for disk I/O for all the algorithms across all SSBM queries drilled down according to the phases of the algorithm. We have achieved significant reduction (≈ 50%) in disk I/O time because of MAAT. MAAT holds the intermediate attributes required for the join query processing, thereby eliminating the bottleneck of multi-pass scans (additional disk I/O) performed by invisible join and significantly lowering the total execution time in the nimble join.

Effect of varying number of processors: In this experiment, both the algorithms were granted unrestricted memory access and the processor access in the increasing order by two. We stop at the maximum of 10 processors. Figure 12 shows the performance comparison between nimble join and invisible join with varying number of processors for Query 3.1. It is immediately apparent that nimble join has a better performance than invisible join for both initial response and total execution time.

We can see a significant improvement in performance from 2 processors to 4 processors. At $N = 4$, the cost of work done in each process by multiple threads was significantly lower than splitting the work amongst different threads yielding remarkable improvement in the performance. However, with 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 used 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 the algorithms.

5.1.3 Memory consumption

Memory consumption is the amount of memory used by the algorithm whilst executing the join query. Let $m_s$ be the amount of memory available at the start and let $m_e$ 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 13A, with average results across all queries in Figure 13B. 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 even more than 50%. The reduction in memory consumption is because of two main reasons: (1) The nimble join avoids additional data structures to hold multiple lists of positions, and (2) it does significantly less disk I/O (discussed in Section 5.1.2) to re-access the columns resulting in no need to store those columns in memory for query processing.

6 ANALYTICAL EVALUATION

In this section, we will introduce our modelling methodology and describe the cost model to predict the cost of the join. We will also present our model evaluation and statistical analysis to understand the difference between model and experiment.
6.1 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. We have followed the approach for constructing the cost model described in the work of Taniar et al.\textsuperscript{32} The cost model includes the following components: System and Data Parameters, Query Parameters, and Time Unit Cost.

**Data Parameters** includes the number of records in the tables and table size in bytes. 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.

**System Parameters** includes the number of processors used to process the query, data page size, and maximum hash table size, which fit into the memory. The number of processors determines the amount of information processed by each processor. Since the data are loaded and written to/from disk by pages, the data page size is required to calculate the number of pages in the 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 and projectivity ratios. 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 a total number of attributes in the table.

**Time Unit Cost and Communication Cost** are the parameters related to the technical characteristics of the system, such as time to read and write page to/from disk, time to read to/from main memory, time to execute a particular function in the main memory, and time to send, receive, and calculate a destination for the record that is to be sent from one processor to another.

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

6.2 Cost models

To be able to describe the components of the 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 1. The symbols used in the formula: ⌈⌉ is a ceiling function, ∧ means minimum, and ∨ means maximum.

When building the hash table, a read operation is performed three times. We read the required dimension table column from disk by pages, read from the page, and write to the memory and finally read and hash. We apply the predicate to the data and save intermediate results into
TABLE 1  The cost model parameters and notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System and data parameters</td>
<td></td>
</tr>
<tr>
<td>$D_i$</td>
<td>Size of the i-th dimension table, $i = 1 \ldots n$</td>
</tr>
<tr>
<td>$</td>
<td>D_i</td>
</tr>
<tr>
<td>$F_i$</td>
<td>Size of the i-th fact table column</td>
</tr>
<tr>
<td>$</td>
<td>F_i</td>
</tr>
<tr>
<td>$n_d$</td>
<td>Number of dimension tables</td>
</tr>
<tr>
<td>$n_f$</td>
<td>Number of fact table column involved in the query</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of processors</td>
</tr>
<tr>
<td>$P$</td>
<td>Page size</td>
</tr>
</tbody>
</table>

Query parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi_i$</td>
<td>Projectivity ratio of the i-th dimension table</td>
</tr>
<tr>
<td>$\pi_f$</td>
<td>Projectivity ratio of the fact table</td>
</tr>
<tr>
<td>$\sigma_i$</td>
<td>Selectivity ratio of the i-th dimension table</td>
</tr>
<tr>
<td>$\sigma_{fi}$</td>
<td>Selectivity ratio of the i-th column in the Fact table</td>
</tr>
</tbody>
</table>

Time unit cost

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$IO$</td>
<td>Time to read a page from the disk</td>
</tr>
<tr>
<td>$t_w$</td>
<td>Time to write the record to the main memory</td>
</tr>
<tr>
<td>$t_r$</td>
<td>Time to read a record in the main memory</td>
</tr>
<tr>
<td>$t_j$</td>
<td>Time to join records</td>
</tr>
<tr>
<td>$t_p$</td>
<td>Time to probe</td>
</tr>
<tr>
<td>$t_h$</td>
<td>Time to hash</td>
</tr>
<tr>
<td>$t_c$</td>
<td>Time to perform computation in the main memory</td>
</tr>
<tr>
<td>$t_d$</td>
<td>Time to access the record directly</td>
</tr>
<tr>
<td>$t_{del}$</td>
<td>Time to delete the record</td>
</tr>
</tbody>
</table>

the memory. The approximate cost of Phase 1 is

$$ \text{Phase}_1 = \left\lceil \frac{n_d}{N} \right\rceil \times (n_d \land N) \times \left( \sum_{i=1}^{n} |D_i| \times \frac{\pi_i}{P} \times IO \right)$$

$$ + \left( \sum_{i=1}^{n} |F_i| \times (t_r + t_w + t_j) \right)$$

$$ + \left( \sum_{i=1}^{n} |D_i| \times \sigma_i \times (t_r + t_j + t_h) \right). $$

(1)

Whilst probing the hash table, the read operation is performed three times. We read the required fact table column from disk by pages, read from the page, 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 not satisfying the join conditions. The approximate cost of Phase 2 is

$$ \text{Phase}_2 = \left\lceil \frac{n_f}{N} \right\rceil \times (n_f \land N) \times \left( \sum_{i=1}^{n} |F_i| \times \frac{\pi_f}{P} \times IO \right)$$

$$ + \left( \sum_{i=1}^{n} |F_i| \times (2t_r + t_w + t_j) \right)$$

$$ + \left( \sum_{i=1}^{n} |F_i| \times \sigma_{fi} \times (t_r + t_j + t_p) \right)$$

$$ + \left( \sum_{i=1}^{n} |F_i| \times \sigma_{fi} \times (1 - \sigma_{fi}) \times t_r \times t_{del} \right). $$

(2)

Whilst 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, 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 Phase 3 is

$$ \text{Phase}_3 = \left\lceil \frac{n_f}{N} \right\rceil \times (n_f \land N) \times \left( \sum_{i=1}^{n} |F_i| \times \frac{\pi_f}{P} \times IO \right)$$

$$ + \left( \sum_{i=1}^{n} |F_i| \times (t_r + t_w) \right)$$

$$ + \left( \sum_{i=1}^{n} |F_i| \times \sigma_{fi} \times (t_d + t_j + t_r) \right). $$

(3)

6.3  Model evaluation

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

6.3.1  Varying number of processors

Figure 14 shows the comparison between the execution time predicted by the model and actual execution time from the experiment for varying numbers 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 actual execution time: (1) The processes executing in parallel often access shared resources and the slow down results from the interference between the processes to access the shared resources. (2) 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.

Similar to the experiment, we have varied the number of processors by the power of two. In the experiments, we stopped at 10 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.

### 6.3.2 SSBM queries

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

\[
\text{error_rate} = \frac{\text{experiment_time} - \text{model_time}}{\text{experiment_time}}. \tag{4}
\]

Table 2 shows the comparison between the execution time predicted by the model and actual execution time from the experiment for all SSBM queries. The estimated execution time from the cost model is close to the actual execution time from the experiment in all the cases, which again demonstrates the effectiveness of our cost model. The differences between the estimated and the actual cost can be attributed to the Startup cost associated with initiating multiple processors. The startup time of the processes varies, making it difficult to estimate and include 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. Therefore, we accept the null hypothesis, and we conclude that there is no significant difference between the values of the model and the experiment.

### 7 CONCLUSION

In this paper, we proposed a new progressive join algorithm for column-stores known as “Nimble Join.” It uses a multi-attribute array table to hold attributes required in join query processing that facilitates progressive materialisation. We showed that our algorithm produces results faster, has comparatively better execution time, and dramatically reduce the memory consumption than the competing column-store join. We attribute
this improvement primarily 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.

We hope that this paper leads to more research into column-store joins as we have shown for possible improvements and anticipate that more is achievable. Future work may include extending nimble join to use MAAT in all phases making the join operation more memory friendly. We are also excited to extend nimble join to include more features of column-stores such as column-specific compression and database cracking and adaptive indexing.

**ORCID**

David Taniar [ORCID: https://orcid.org/0000-0002-8862-3960]

**REFERENCES**

10. Abadi DJ, Madden SR. Hachem N. Column-stores vs. row-stores: how different are they really? In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data; 2008; Vancouver, Canada.


