Towards Multi-way Join Aware Optimizer in SAP HANA

Sungheun Wi [†] Wook-Shin Han [‡*] Chuho Chang [†] Kihong Kim [†]

SAP Labs Korea[†], POSTECH[‡]
South Korea
{sung.heun.wi, chuho.chang, ki.kim}@sap.com[†], wshan@dblab.postech.ac.kr[‡]

ABSTRACT

Existing binary join based plans may be suboptimal for important, emerging applications. Typical query optimizers enumerate plans using binary joins only. In this paper, we introduce the multi-way join aware optimizer in SAP HANA. The naive way to extend the existing query optimizer to be aware of multi-way joins (m-way joins for short) is to enumerate m-way joins on top of a traditional binary join enumeration framework. However, many different binary joins correspond to the same m-way join. Thus, unnecessary join enumerations would be required for such naive integration. To solve this problem, we introduce the new concept of an m-way join unit and explain how the construction of join units is plugged into the SAP HANA query optimizer. We also provide a series of optimizer enhancements by exploiting m-way join unit characteristics. Using TPC-H and our customer workloads, we showcase the superiority of our m-way join aware optimizer.

PVLDB Reference Format:
Sungheun Wi, Wook-Shin Han, Chuho Chang, and Kihong Kim. Towards Multi-way Join Aware Optimizer in SAP HANA. *PVLDB*, 13(12): 3019-3031, 2020.
DOI: https://doi.org/10.14778/3415478.3415531

1. INTRODUCTION

Several fast multi-way join algorithms have recently been developed. Existing binary join (i.e., 2-way join) based plans may be suboptimal for important, emerging applications. One example is OLAP-style analysis, where queries are often star-shaped (aka star-join) as illustrated in Figure 1(a). For instance, when breaking down revenue by the combination of country, month, and product category, a big sales-record table, called a fact table, is joined three times with so-called dimension tables representing store locations, sales dates, and product categories and then aggregation is executed for the joined results. An m-way star-join algorithm scans the fact table once [6, 27] and quickly performs $m-1$ joins and aggregations. The other example is for graph analysis, where queries have cycles. Consider a query in Figure 1(b), which lists all triangles in a graph where the graph is stored in an edge table $E(src, dest)$. Then, the triangle query is a three-way, self-join on E, and thus, any self binary join on E would generate $|E|^2$ intermediate results in the worst case. However, the maximum output size is bounded by $O(|E|^{1.5})$, which indicates that any binary join plan could be asymptotically suboptimal for graph queries [15, 26].

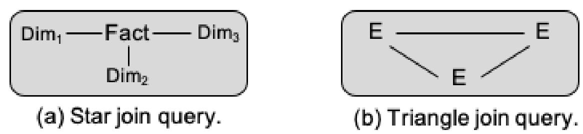


Figure 1: Multi-way join queries.

The naive way to extend the query optimizer to be aware of m-way joins is to enumerate m-way joins on top of a traditional binary join enumeration framework. However, many different binary join orders correspond to the same m-way join, and if the m-way join is faster than those binary joins, we end up performing unnecessary join enumeration, which should be avoided. Note that this problem looks similar to federated query optimization at first glance (aka optimization in a mediator or capability-based optimization), in that it tries to push large portions of subqueries down to subordinators to minimize overall processing cost including the network cost. However, the problem is very different from federated query optimization in that traditional query optimization in the mediator still assumes that both the mediator (or coordinator) and subordinators process binary joins only.

A natural and challenging question is whether it is possible to extend the query optimizer with a little effort so that it can enumerate traditional binary joins as well as m-way joins efficiently. For this, we propose the novel concept of m-way *join unit*, which is a new operator executing m-way join. This join unit is expanded by merging with another logical operator, such as join and group-by, or other join unit. We treat this join unit as a special physical operator, and the join unit enumeration is performed when we generate a physical operator for a given logical operator. In this way, changes to logical enumeration can be minimized.

The contributions of the paper are summarized as follows.

- **Cost-based m-way join enumeration**: We present an m-way join aware optimizer which considers multiple m-way join algorithms together with group-by op-

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415531

erator pushdown or eager aggregation [30] during logical/physical enumeration for cost-based decisions. This paper focuses on the *global* optimization, which coordinates various m-way join algorithms and group-by operators in cost-based enumeration, and how it coordinates with the *local* optimization specific to each m-way join algorithm.

- **Single enumeration framework for both m-way and binary join algorithms**: Our m-way join aware optimizer supports m-way joins for column tables, binary joins for row tables, and mixed joins between column and row tables in a single enumeration framework, which significantly reduces the maintenance overhead.

- **Easy extension of transformation-based enumeration**: The origin of the SAP HANA optimizer is P *TIME [3], which is one of the foundations of SAP HANA. It supports only binary join algorithms for row tables without supporting m-way join algorithms. Previously, m-way join algorithms had been triggered via an imperative programming interface without optimizer consideration. This paper presents how the existing transformation-based optimizer can be easily but still effectively extended to support m-way join algorithms for column tables.

Although SAP HANA does not currently support m-way join algorithms such as LeapFrog TrieJoin (LFTJ) [26] for graph queries, supporting such m-way join algorithms would be interesting future work for guaranteeing the worst case optimality. The global optimization framework can accommodate such m-way join algorithms, so that there is no need to change the enumeration framework.

The rest of this paper is organized as follows. Section 2 compares our approach with related works, and Section 3 describes technical background for detailed m-way join algorithms available in SAP HANA. In Section 4, we explain the unique challenges of the SAP HANA optimizer, and Section 5 describes the approach of the SAP HANA optimizer to find an optimal query plan considering m-way join algorithms. Section 6 explains how we handle mixed joins between column tables and row tables when we enumerate m-way join algorithms. Section 7 provides the experimental evaluation results. Finally, Section 8 concludes the paper.

2. RELATED WORK

Star Join Optimization. Optimizing star-shaped queries using a series of binary joins has existed for a decade in disk-based row stores such as IBM DB2, Microsoft SQL Server, and Oracle. When we have m-1 dimension tables, all hash tables for those dimension tables can be built in-memory simultaneously, so probing for the fact table can be done in a pipeline fashion where the output of each join operator is passed to the parent join operator without materializing the intermediate result (i.e., pipelined hash join). Bitmap filtering from each dimension table to the fact table can be regarded as semi-join reduction. That is, such row stores simulate m-way join by using a series of binary joins. However, such optimization is applied to star patterns in the initial query via query rewrite, while our framework detects such patterns during enumeration. Thus, we are able to detect any subtree patterns during enumeration. Furthermore, some m-way join algorithms can not be simulated by binary joins [26].

Semi-Join Optimization. [24] extends dynamic programming query optimizers to generate a good plan with semi-joins. However, group-by operators were not considered for cost-based optimization. In [9], the semi-join reduction order is decided with a variant of the A* search algorithm. Thus, the order decision method is faster than dynamic programming, since it exploits a guided search rather than an uninformed, exhaustive search [9]. Our semi-join algorithm uses this order decision method.

Worst-case optimal multi-way join algorithms. Recently, a series of worst-case optimal algorithms have been proposed. LFTJ [26] is a representative one. It performs multi-way join and uses Trie to index relations. For a given query, it first orders the attributes and selects a value for each attribute in order. For instance, assume that it selects $< x, y, z >$ as the order for an example triangle query $R(x, y) \bowtie S(y, z) \bowtie T(x, z)$. Then, LFTJ finds all candidate values of x using $\Pi_x R(x, y) \cap \Pi_x T(x, z)$. Here, the intersection is performed using a Trie on each table, where the values of the first and second attributes are stored at height one and two, respectively. For each value a for x, LFTJ then finds candidate values of y using $\Pi_y \sigma_{x=a} R(x, y) \cap \Pi_y S(y, z)$. Again, for each value b for y, LFTJ finds the values for z using $\Pi_z \sigma_{y=b} S(y, z) \cap \Pi_z \sigma_{x=a} T(x, z)$. For each value c for z, LFTJ reports (a, b, c) as an output. Then, it backtracks to y and searches for the next b. The process continues until it backtracks to x and there is no next a.

EmptyHeaded [1] generates a query plan by decomposing a (hyper) query graph into a set of subgraphs where each subgraph corresponds to a worst-case join. However, the supported query is very limited; for example, it does not handle group-by queries. Thus, in order to handle group-by queries, one can apply group-by pushdown heuristic first. This baseline can be simulated by our framework using the group-by pushdown heuristic with m-way join units. However, this baseline could lead to generation of suboptimal plans since it does not consider m-way joins together with group-by operators in cost-based enumeration.

3. SAP HANA

The SAP HANA database is an in-memory data management system that leverages the capabilities of modern hardware, especially with huge amounts of main memory and multi-core CPUs, in order to improve the performance of analytical and transactional applications [5, 6, 12, 20, 21].

SAP HANA supports both in-memory column tables and in-memory row tables. While row tables, which take a row-major layout, have been studied extensively, column tables have not been studied much. SAP HANA invented various storage techniques and query processing algorithms for column tables.

3.1 Dictionary Encoding

Figure 2 illustrates dictionary encoding of column tables. Figure 2(a) shows a conceptual layout of a sales record table. It has three columns, *date*, *amount* and *customer*. Figure 2(b) shows the column table representation with dictionary encoding. Each column consists of two arrays, **value id (vid) array** and value array. The vid array stores one value id per record. The value array maps vids into values. For instance, the first record has vid 0 in *date*, which refers to

date	amount	customer
2018-12-31	100	BMW
2018-12-31	200	Ford
2018-12-31	150	KIA
2019-01-02	100	GM
2019-01-02	500	VW
2019-01-03	200	KIA
2019-01-03	400	FORD
2019-01-03	300	GM

(a) sales records

date (2 bits)
0
0
0
1
1
2
2
2

vid	date dict.
0	2018-12-31
1	2019-01-02
2	2019-01-03

amount (3 bits)
0
2
1
0
5
2
4
3

vid	amount dict.
0	100
1	150
2	200
3	300
4	400
5	500

customer (3 bits)
0
1
3
2
4
3
1
2

vid	customer dict.
0	BMW
1	Ford
2	GM
3	KIA
4	VW

(b) column table with dictionary encoding

Figure 2: Dictionary encoding.

the first element, 2018-12-31, in the value array. The value array, called dictionary, stores distinct values in the sorted manner.

The vids are sequential contiguous integers starting from 0. Such *dense* vids make dictionaries small and dictionary lookups very fast. Dictionaries don't have to store vids, shown in dotted boxes, because vids are array indexes. Dictionaries don't need index structures like tree indexes, especially for fixed-size value types [2]. For instance, the date column's dictionary is a densely packed array of fixed-size values. Consequently, small dictionaries are likely to fit in the CPU cache, and their access by vid becomes a fast array lookup without incurring a last-level CPU cache miss.

Dense vids often lead to an excellent compression ratio. The vid array of *date* in this example is just two bits wide, which is enough to represent three distinct values in its dictionary. Since a date value usually takes four bytes, the compression ratio is 16 times. In practice, when a sales table is partitioned by year, each partition has at most 365 distinct dates, which needs nine bits to represent [28]. Therefore, the compression ratio becomes around 3.4 times. Further compression can be achieved by applying other schemes, such as run-length encoding after dictionary-encoding.

Although dictionaries can be global and shared among tables, SAP HANA prefers column-specific dictionaries. It makes vids denser, vid arrays smaller, and dictionaries smaller. Consequently, it reduces both the memory footprint and CPU cache misses.

3.2 Motivating M-way Join Example

Figure 3 shows a star-schema query example for a fact table, *Sales*, and for dimension tables, *Date* and *Customer*. Suppose that each column is stored as an array. For instance, *Date.Y* is an array of [2018, 2018, 2019, 2019, 2019] and *Customer.nation* is an array of [DE, US, US, KR, DE]. *Date* and *Customer* have two deliberately chosen properties. 1) Join columns, *Date.d_id* and *Customer.c_id*, are dense integers starting from 0. 2) *Date* and *Customer* records are sorted by the join columns.

When tables are stored in this way, a join becomes simply array operations. For instance, consider the last *Sales*

Sales

s_id	d_id	c_id	amount
0	1	0	100
1	1	1	200
2	1	3	150
3	3	2	100
4	3	4	500
5	4	3	200
6	4	1	400
7	4	2	300

Date

d_id	Y	M	D
0	2018	12	30
1	2018	12	31
2	2019	1	1
3	2019	1	2
4	2019	1	3

Customer

c_id	name	nation
0	BMW	DE
1	Ford	US
2	GM	US
3	KIA	KR
4	VW	DE

```
select sum(s.amount) as revenue
  from Sales s, Date d, Customer c
 where s.d_id = d.d_id and s.c_id = c.c_id
       and c.nation = 'DE' and d.Y = 2019
```

Figure 3: Example tables and a star-schema query.

record, (7, 4, 2, 300). Joining it with *Date* and finding the sales year costs just one array operation, $Date.Y[4]$, which returns 2019.

When dimension tables are small enough to fit in the CPU cache memory, query processing is extremely fast. Join simply looks up small dimension table arrays, so random array access doesn't incur CPU cache misses. The query shown in Figure 3, which calculates the total revenue from German customers in 2019, can be executed as follows. It scans the *Sales* table once and performs two joins without building any auxiliary index structures, such as hash tables.

```
sum = 0;
for i in |Sales|
  d_id = Sales.d_id[i];
  c_id = Sales.c_id[i];
  if (Date.Y[d_id] == 2019)
    if (Customer.nation[c_id] == 'DE')
      sum += Sales.amount[i];
```

Note that this is a three-way join and can easily be extended to an m-way join. Note also that *Sales.c_id* and *Sales.d_id* are logical ids, but they are close to physical pointers. It means no index or hash table is needed to map logical id into physical pointers.

3.3 M-way Star Join

When applying the idea from Section 3.2, we often encounter that tables are not sorted by join columns, and join columns are not dense integers. SAP HANA overcomes these issues by introducing dictionary-based join indexes (**DJI**) [25].

Figure 4 shows a DJI example. Figure 4(a) shows *Sales* and *Date* tables when *d_id*, the key column of *Date*, has the form of YYYYMMDD. Figure 4(b) shows the *Sales* table after dictionary encoding. *Sales.d_id* has three distinct values, 20181231, 20190102, and 20190103, which are encoded with three vids, 0, 1, and 2, respectively. Note a difference from Figure 3, where *d_id* has 1, 3 or 4, which are not vids but values.

Figure 4(c) shows a per-query DJI, built for the Figure 3 query. When joining the last record with *Date* and getting its year, Figure 3 does $Date.Y[4]$, and Figure 4 does $Date.Y[d\_id\_idx[2]]$. Note that the inner array lookup returns 4. DJI *d_id_idx* can be built as follows. For each dictionary entry of *d_id_dict*, look up *Date*. If a matching record is found, check whether the record satisfies the where condition, $Y = 2019$. If satisfied, store its record offset. Otherwise, store -1.

DJI has two variations, per dictionary and per query. Per-dictionary DJIs are incrementally built when executing

Sales

s_id	d_id	c_id	amount
0	20181231	0	100
1	20181231	1	200
2	20181231	3	150
3	20190102	2	100
4	20190102	4	500
5	20190103	3	200
6	20190103	1	400
7	20190103	2	300

Date

d_id	Y	M	D
20181230	2018	12	30
20181231	2018	12	31
20190101	2019	1	1
20190102	2019	1	2
20190103	2019	1	3

(a) when *d_id* is YYYYMMDD

Sales

s_id	d_id	c_id	amount
0	0	0	100
1	0	1	200
2	0	3	150
3	1	2	100
4	1	4	500
5	2	3	200
6	2	1	400
7	2	2	300

(b) after dictionary-encoding of *d_id*

d_id_dict

d_id	value
0	20181231
1	20190102
2	20190103

d_id_idx

d_id	idx
0	-1
1	3
2	4

(c) join index for *Sales.d_id = Date.d_id and Date.Y = 2019*

Figure 4: Dictionary-based join index.

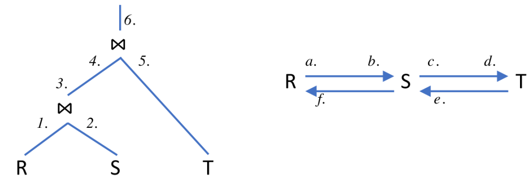
queries and cached for subsequent uses. Per-query DJIs are built for a query on the fly and discarded. Since it is specific to a query, two additional optimizations are often adopted. First, filters are pre-evaluated as shown in Figure 4(c). Second, DJI stores query-specific target-column values or vids. For instance, the example query joins with table $Date$ to access column Y. Thus, d_id_idx is further optimized into an array of [n/a, 2019, 2019], instead of [-1, 3, 4], where n/a means no join pair. Then, $Date.Y[d\_id\_idx[2]]$ is optimized as $d\_id\_idx[2]$.

Consider a star join, which is an m-way join, where a large fact table F is joined with m-1 smaller dimension tables, D_1, D_2, ..., D_{m-1}. A fact table stores dimension table keys, and is joined with dimension tables via these key columns. When processing an m-way star join, SAP HANA first builds $m-1$ DJIs, as described above, for D_1 to D_{m-1}. Then, SAP HANA scans the large fact table and performs $m-1$ joins by looking up the DJIs.

The time complexity of the m-way star join algorithm is $O(|F| + |D_1|log|D_1| + ... + |D_{m-1}|log|D_{m-1}|)$, where $|F|$ denotes the cardinality of table F. The strength of this algorithm comes from the cost coefficient in its time-complexity. Consider a two-way join of table D and table F. The hash join complexity is $O(|F| + |D|)$. Suppose that $a_h|F| + b_h|D|$ denotes the hash join cost, where $a_M|F| + b_M|D|log|D|$ denotes the m-way star join cost. The m-way star join wins when $|D|$ is small, since a_M is significantly smaller than a_h, typically by orders of magnitude. The m-way star join reduces the cost coefficient of the table F term at the cost of increasing the complexity of the table D term, from $O(|D|)$ to $O(|D|log|D|)$. Thus, as $|D|$ increases, this algorithm gets less attractive. This trade-off works well for SAP HANA's in-memory column tables, especially for star(-schema) and snowflake(-schema) queries [22].

3.4 M-way Column Join

SAP HANA uses another m-way join algorithm [9,10,23], called m-way column join. This algorithm is a good alternative when each join reduces intermediate results.



1. build $H_x: R_x \rightarrow R$
2. find $S' \equiv \{s \in S' | s.x \in R_x\}$
3. build$\{(r,s) | r.x = s.x\}$
4. build $H'_y: S'_y \rightarrow R \bowtie S'$
5. find $T' \equiv \{t \in T' | t.y \in S'_y\}$
6. find $\{((r,s),t) | s.y = t.y\}$

(a) left-deep hash join tree

a. build $I_x: R_x \rightarrow R$
b. find $S' \equiv \{s \in S' | s.x \in R_x\}$
c. build $I'_y: S'_y \rightarrow S'$
d. find $T' \equiv \{t \in T' | t.y \in S'_y\}$
e. find $\{(s,t) | s.y = t.y\}$
f. find$\{(r,(s,t)) | r.x = s.x\}$

(b) m-way column join

Figure 5: Left-deep hash join vs. m-way column join.

Consider a three-way join, $R \bowtie S \bowtie T$, with the join conditions, $R.x = S.x$ and $S.y = T.y$, respectively. Figure 5 compares a binary hash join scheme and the m-way column join of SAP HANA for this three-way join. For ease of explanation, assume $|R| < |S| < |T|$, where $|R|$ denotes the cardinality of R, and $|R \bowtie S \bowtie T| < |R \bowtie S| < |R|$, meaning that each join reduces intermediate results.

Figure 5(a) sketches a left-deep hash join tree.

- Step 1 is the build phase. It builds a hash table H_x, which is a mapping from R_x to R, where R_x is the set of distinct values in $R.x$.

- Step 2 is the probe phase. It scans S, looks up H_x for each row in S, and checks the join condition, $R.x = S.x$. Let S' denote the subset of S that satisfies the join condition.

- Step 3 materializes the join pairs from Step 2. Usually, Step 2 and Step 3 are intermixed and thus, they are indistinguishable. For the purpose of explaining the m-way column join, we intentionally distinguish Step 2 from Step 3.

- Steps $4-6$ repeat the process for the next join. S'_y in Step 4 denotes the set of distinct y values in S'. Note that S'_y is a subset of S_y. Unlike Step 3, Step 6 doesn't have to materialize the join pairs (or triples). Whether to materialize the join pairs depends on the consumer operation. Since Step 4 builds a hash table, its input or the output of Step 3 needs to be materialized. The consumer operation of Step 6 is left unknown and thus, it is left open whether to materialize.

Figure 5(b) sketches how the m-way column join works.

- Step a is equivalent to Step 1. The difference is that Step a doesn't build a hash table but rather a proprietary index structure, denoted I_x, to best leverage dictionary encoding. This index structure is similar to the one in Figure 4(c). The details are omitted because the focus of this paper is not the algorithm itself but optimizer integration of the algorithm.

- Step b is similar to Step 2 in the sense that it finds S', which is the semi join, $S \ltimes R_x$. This step can be regarded as performing semi-join reduction, reducing S into S' leveraging R_x.

- For the next join, Steps c and d repeat Steps a and b, respectively, so Step c is dissimilar to Step 3 but similar to Step 4, and Step d is similar to Step 5. Instead of completing the first join $R \bowtie S'$ to find the join pairs, Step c moves to the next join $S' \bowtie T$ and builds I'_y. Step d performs the second semi join reduction to find T', which satisfies the join condition, $S'.y = T.y$.

- Step e is similar to Step 6. It finds the join pairs between S' and T'. Step e can be combined with Step d. Thus, it doesn't have to look up I_y additionally.

- Step f is similar to Step 3. It finds the join pairs between R and $S' \bowtie T'$ by looking up I_x.

The m-way column join doesn't perform $m-1$ joins one by one. It first builds a proprietary index per join, as shown in Steps a and c. Each index building is followed by a semi-join reduction, as shown in Steps b and d. Then, it finds the join tuples of m record ids, as shown in Steps e and f.

This looks similar to a pipelined hash join plan, which builds hash tables for S and R and then performs $T \bowtie S$ and $(T \bowtie S) \bowtie R$ in a pipeline. They are similar in the sense that both avoid materializing intermediate results. The difference is that the m-way column join first performs a semi-join with S and then builds an index on S'. If this semi-join is not selective, then the pipelined hash join has a better chance to win. However, in such a case, it needs to compete with the m-way star join. Note that the m-way star join is regarded as an enhanced pipelined hash join plan leveraging dense vids and DJIs.

This m-way join algorithm has three advantages. First, it doesn't materialize intermediate join tuples as Step 3 does. Second, it indexes a smaller number of entries by doing a semi-join reduction. Third, its index operations, build and look up, are more efficient than hash table operations. Dense vids are used as index keys. When comparing to hash tables, hash key generation is not needed, hash collision doesn't exist, the number of index directory entries is optimal, and index keys don't have to be stored.

The disadvantage of this algorithm, far smaller than the advantages, is more index lookup operations. Steps 3, 6, and e can be intermixed with the preceding steps, Steps 2, 5, and d, respectively. Thus, they can reuse the index lookup results of the preceding steps. However, Step f and Step b are not adjacent and thus, they cannot share index lookup results.

4. SAP HANA OPTIMIZER CHALLENGES

This section explains the unique challenges of SAP HANA for m-way join enumeration in the optimizer.

SAP HANA currently supports multiple m-way join algorithms for column tables only. Since they effectively exploit vids, as explained in the previous section, they are faster than typical binary join algorithms for column tables. Therefore, the SAP HANA optimizer enumerates only m-way join algorithms for column tables, while binary join algorithms are enumerated for row tables. Both types of algorithms are enumerated and compared for a mixed join between column tables and row tables. The challenge is to handle these variations together in a single enumeration framework to avoid duplicate code maintenance overhead.

Another challenge is that group-by operators also need to be considered together with m-way joins in a cost-based enumeration. For example, pushing down a group-by operator through a join operator might be beneficial in certain scenarios. However, it is not always beneficial especially when a join operator reduces the intermediate result significantly and the group-by operator does not reduce it. When it is not applicable to push down a whole group-by operator, the partial group-by/aggregation pushdown by eager-aggregation approach [30] needs to be similarly considered together with m-way joins in a cost-based enumeration. In Section 7, we compare the cost-based enumeration with the group-by pushdown heuristics and show the effectiveness of the cost-based enumeration.

The last challenge is that SAP HANA must support complex analytical queries for Hybrid Transactional/Analytical Processing (HTAP). We examine a typical complex query from an S/4HANA customer database [19], which is a HTAP application that runs complex analytical queries directly against OLTP tables without a separate ETL step. After compiling this query, its initial query plan has 46 leaf nodes referring to 11 distinct tables, 40 binary join nodes, three union-all nodes, 31 group-by nodes, and many intermediate filters and projection nodes. After query optimization, the resulting query plan has 15 join units with a mixture of different m-way join algorithms. Some queries from S/4HANA are used for the experiments in Section 7.

5. M-WAY JOIN AWARE OPTIMIZER

This section explains the approach of the SAP HANA optimizer to find an optimal query plan considering m-way join algorithms explained in Section 3. We first define some terminologies used throughout this paper. Since the HANA optimizer is based on Volcano/Cascade [7,8], the terminologies are similar.

- A *logical alternative*, which is the same as an equivalent logical algebra expression in Volcano/Cascade, is a logical operator tree created from another logical alternative by applying a transformation rule. A logical operator tree translated from an initial query is also referred to as a logical alternative.

- An *equivalence class* is the same terminology as that of Volcano/Cascade and consists of logical alternatives and corresponding physical operators (algorithms).

- The *search space* for a given query graph is the set of all equivalence classes, logical alternatives, and physical operators explored during logical and physical enumeration.

5.1 M-way Join Unit

We first define a new operator called a *join unit* which executes an *associated* m-way join algorithm over m tables. For this, the join unit internally stores a join graph for an m-way join. The join unit optionally contains a post-join filter and a group-by operator followed by its m-way join. Without loss of generality, we assume that both single table operator (i.e., scan operator) and binary join operators are regarded as special cases of m-way join units (i.e. when m=1 or 2). In this paper, we focus on the join and group-by operators only, since these two types of operators play important roles on join unit creation; handling other types of operators are straightforward. For example, a filter operator is already pushed down to a target table in the query rewriting phase

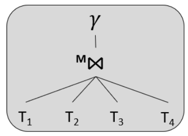


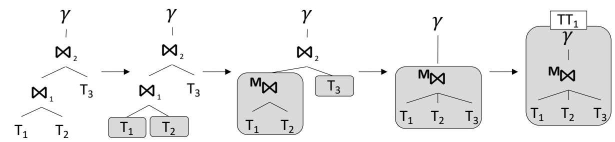
Figure 6: Basic unit of m-way join unit.

prior to the logical/physical enumeration phase. Thus, the filter operator can be treated together with a corresponding base table. In general, the join unit can include filter, order by, and limit operators as well as join and group-by operators. Another common operator is an union-all operator, and it is handled as a separate physical operator outside the m-way join unit in HANA. Thus, it is not further described in this paper because it is orthogonal to join unit construction.

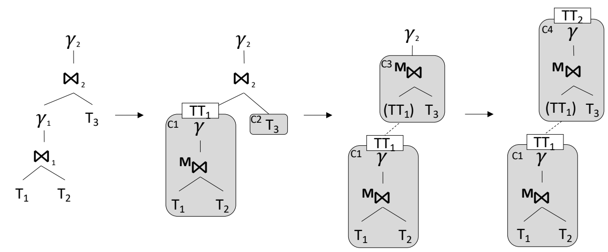
Figure 6 illustrates an example of the join unit that executes a 4-way join over $T_1 \sim T_4$ and then performs a group by operation. Note that γ denotes a group-by operator in this paper. This join unit is expanded by combining with another logical operator, such as join and group-by, or other join unit. SAP HANA uses the following three main procedures to enlarge small join units into larger join units in a bottom-up manner.

- **CombineBaseTable(T_c)**: T_c is a base table operator, and a special join unit is created with a base table T_c.
- **CombineJoin($\bowtie_c, C_{i1}, C_{i2}$)**: $\bowtie_c$ is a binary logical join operator, and it has two child join units - C_{i1} and C_{i2}. This function creates a larger join unit that executes a multi-way join over tables in C_{i1} and C_{i2}. Here, two join graphs from C_{i1} and C_{i2} respectively and binary join $\bowtie_c$ are combined into a new join graph, and they are stored in an expanded join unit.
- **CombineGroupBy(γ_c, C_i)**: γ_c is a logical group-by operator, and C_i is a child join unit of γ_c. A new join unit is constructed by copying the join graph of C_i and group-by information of γ_c. The output of the new join unit is materialized in a temporary table, since SAP HANA materializes the output of the group-by operator into a temporary table. Once a group-by operator is combined into the join unit, it cannot be further expanded with another join or group-by operator. Instead, a new join unit is created, which can be combined with another join or group-by operator by referring to the materialized temporary table.

Figure 7 illustrates how two different logical alternatives in the leftmost figures are combined into m-way join units with these procedures. Note that we execute combine operations in a bottom-up manner. In Figure 7(a), CombineBaseTable creates special join units for $T1$ and $T2$ first. Then, a new join unit for $\bowtie_1$ is created by calling CombineJoin. The join units for $T3$ and $\bowtie_2$ are created in a similar way. CombineGroupBy is called for γ to create a bigger join unit that contains the group by operator on top. Here, TT_1 is a temporary table constructed by this final join unit. In Figure 7(b), a join unit is constructed in a similar way to Figure 7(a) until C_1 and C_2 are created. During the creation of join unit C_3, C_1 cannot be further expanded because its output needs to be materialized into a temporary table. Therefore, a new join unit C_3 is created,



(a) Sequence of combine operations into a single join unit.



(b) Sequence of combine operations into two join units.

Figure 7: Sequence of combine operations of m-way join units.

and it refers to the temporary table TT_1. Correspondingly, the join condition for $\bowtie_2$ is changed to refer to TT_1 instead of T_1 or T_2. In this case, the final plan has two join units, while Figure 7(a) has one join unit.

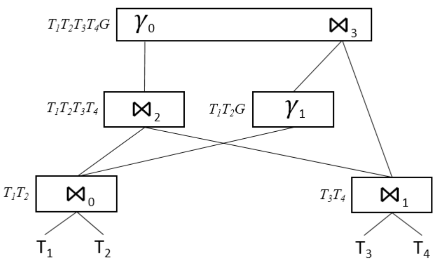
Now, we introduce a heuristic to limit search space when we consider join units during plan enumeration.

- *Larger join unit heuristic*: Due to the characteristics of the m-way join implementations in SAP HANA, our optimizer prefers an m-way join unit rather than selecting a plan performing a binary join between k-way join and (m-k)-way join for any k. Note that, when k = 1, k-way join corresponds to a table scan operator. Otherwise, we need to generate all *logical* alternatives by varying k and make a cost-based decision. Hence, we expand m-way join units as much as possible without cost-based comparison with other binary join alternatives. However, this heuristic is not applied to a group-by operator. Instead a logical alternative is enumerated for cost-based decisions, and details are explained in the next section.

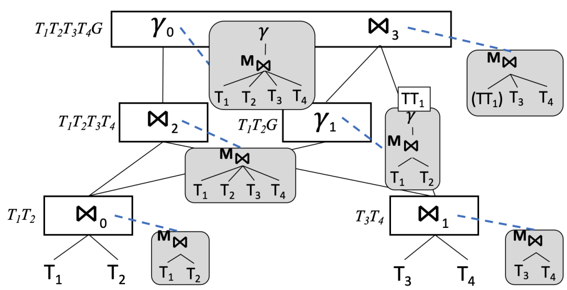
5.2 Search space enumeration

The previous section describes how m-way join units are constructed for a given logical alternative. This section explains the global optimization - i.e., how m-way join units can be incorporated with transformation-based search space enumeration in the SAP HANA query optimizer. Note that this paper focuses on m-way join enumeration, and the other aspects considered by the SAP HANA query optimizer during search space enumeration, such as input property, distributed query optimization, and branch-and-bound pruning, are not the scope of this paper. Thus, they are omitted in the following algorithm and explanation.

As explained before, the HANA optimizer is conceptually based on the Volcano/Cascade framework, but there is a slight difference in the internal representation. Instead of maintaining a MEMO table to avoid duplicate enumeration, the HANA optimizer keeps a DAG structure for a search space similar to [18]. Figure 8(a) shows an example of a search space. The box denotes an equivalence class, and



(a) Search space considering binary joins only.



(b) Search space with m-way join units.

Figure 8: M-way join unit enumeration.

operators inside the box represent logical alternatives that refer to child equivalence classes.

Our design principle for enumerating m-way join units is to minimize changes to the existing query optimizer targeted for binary join algorithms. For this, a join unit is enumerated as a special *physical* operator, and the join unit enumeration is performed when we generate a physical operator for a given logical operator. The m-way join unit has different aspects from other physical operators in that it has two choices of m-way join algorithms, and additional local join order optimization, which will be explained in Section 5.3, is needed inside the join unit. However, we chose a physical operator to localize m-way join unit changes inside physical operator implementations. In this way, changes to logical enumeration and global enumeration framework can be minimized. Additionally, it has an advantage in that the impact on the binary join enumeration, which is needed for row tables and a mixed join between row and column tables, is minimized.

In Figure 8(a), a part of the search space is extracted for an initial query plan $\gamma_0((T_1 \bowtie_0 T_2) \bowtie_2 (T_3 \bowtie_1 T_4))$ for ease of explanation. Figure 8(a) has another logical alternative created by applying the group-by push transformation rule to the initial query plan in the equivalence class $T_1T_2T_3T_4G$. Figure 8(b) illustrates how m-way join units are constructed for the given search space in Figure 8(a). The pseudo code on how m-way join units are constructed during the search space enumeration is described in Algorithms 1 to 4. Although logical enumeration and m-way join unit creation are not separate steps but interleaving steps, in Figure 8 and this section, they are explained as separate steps for easier understanding. Other parts of the search space, which are not illustrated in the figure, are explored in a similar way to what is described in the algorithm.

Algorithm 1 presents the main function for enumerating the search space, ENUMERATE. It takes a logical operator *op*

Algorithm 1: ENUMERATE(*op*)

```
   Input: A logical operator op
1  if op.IsPhysicalEnumFinished() then
2  |  return
3  foreach child equivalence class ce of op do
4  |  if ce.HasNotEnumeratedLogicalAlt() then
5  |  |  Enumerate(ce.GetTargetLogicalAlt())
6  if op.GetEquivClass().IsLogicalEnumFinished() ==
    false then
7  |  ApplyLogicalTransformations(op)
8  |  op.GetEquivClass().SetLogicalEnumFinished()
9  PhysicalEnum(op)
10 if op.HasNextLogicalAlt() then
11 |  Enumerate(op.GetNextLogicalAlt())
```

Algorithm 2: PHYSICALENUM(*op*)

```
Input: A logical operator op
new_mju := EnumerateMwayJoinUnit(op)
op.AddPhysicalOp(new_mju)
op.SetPhysicalEnumFinished()
```

as input and performs the search space enumeration (both logical and physical) for *op* starting from a root operator. It first checks if the physical enumeration for *op*, which is performed after logical enumeration, is finished. If it is not finished, the search space enumeration for *op*'s child operators needs to be recursively performed first as described in Lines 3-5 because enumeration is performed in a bottom up manner. By iterating each child equivalence class, ENUMERATE is called for the logical alternative whose logical or physical enumeration is not finished.

Once the enumeration for every child of *op* is finished, logical and physical enumeration for *op* itself are started if they have not been performed before. In Lines 6-8, logical enumeration is performed by applying all valid transformation rules between *op* and each logical alternative in its child equivalence class. For example, $\bowtie_3$ in Figure 8(a) is newly enumerated in this step by pushing down a group-by operator, and eager/lazy aggregation alternatives are enumerated in this step as well for cost-based comparison. Note that this part is similar to the other transformation-based optimizers [7, 17], group-by transformations [4, 29, 30], and join transformations [13]. Therefore, we omit the detailed explanation on logical transformations of the HANA optimizer in this paper. The current equivalence class is marked as logically enumerated to avoid duplicate enumeration (Line 8).

Once the logical enumeration is finished, physical enumeration for *op* is performed, and its details are described in Algorithm 2 and Algorithm 3. After that, we process the next available logical alternative for further enumeration, since the enumeration for *op* is finished.

In the physical enumeration for *op* in Algorithm 2, we need to perform physical enumeration for the m-way join unit, which is a unique part of the SAP HANA optimizer. Once an m-way join unit is created, it is attached to *op*. The details for m-way join unit enumeration are given in Algorithm 3. Since the entire enumeration is performed in a bottom up manner, the m-way join unit is also constructed in this order. Depending on the type of *op*, we have to set properties of the m-way join unit accordingly. When *op* is a base-table operator, a special join unit with a base table is created (Lines 2-3). If *op* is a join operator, the correspond-

Algorithm 3: ENUMERATEMWAYJOINUNIT(*op*)

```
   Input: A logical operator op
1  new_mju := CreateNewMJU()
2  if op == TABLE then
   |  /* CombineBaseTable(op)                        */
3  |  new_mju.join_graph := op.GetTable()
4  else if op == JOIN then
   |  /* CombineJoin(op, c1, c2)                     */
5  |  c1 := op.GetLeftChild().GetMinCostMJU()
6  |  c2 := op.GetRightChild().GetMinCostMJU()
7  |  j1 := GetJoinGraphForCombine(c1)
8  |  j2 := GetJoinGraphForCombine(c2)
9  |  new_mju.join_graph := CombineJoinGraph(op,j1,j2)
10 else if op == GROUP_BY then
   |  /* CombineGroupBy(op, c)                       */
11 |  c := op.GetChild().GetMinCostMJU()
12 |  new_mju.join_graph :=
   |    GetJoinGraphForCombine(c)
13 |  new_mju.group_by := GetGroupByInfo(op)
14 |  new_mju.temporary_table := MatIntoTempTable(op)
15 return new_mju
```

Algorithm 4: GETJOINGRAPHFORCOMBINE(c)

```
Input: An m-way join unit c
if c.group_by == NULL then
|  return c.join_graph
else
|  return c.temporary_table
```

ing code is described in Lines 4-9. First, the existing join unit with a minimum cost is chosen for each child equivalence class of *op* (i.e. $c1$ and $c2$). In function GETMINCOSTMJU, local optimization and cost estimations for join units in the equivalence class are performed when there is a join unit whose cost estimation was not done before. The details of cost estimation for a join unit will be explained in Section 5.3. The corresponding join graphs for the combine operation is stored in $j1$ and $j2$ respectively. These two join graphs are combined together with *op*, and the newly combined join graph is stored in a new join unit. For example, in Figure 8(b), join units for $\bowtie_0$, $\bowtie_1$, and $\bowtie_2$ are created in this way.

Lines 10-14 describe a combine operation for the group-by operator. The minimum cost of join unit c in the child equivalence class is chosen, and the join graph for the combine operation is copied from c (Lines 11-12). Information about the group-by operator is stored in a new join unit (Line 13). Since the output of a group-by operator is materialized into a temporary table, the temporary table name is stored in the new join unit as well. GETJOINGRAPHFORCOMBINE of Algorithm 4 returns either a join graph stored in the join unit or a temporary table where the output of the join unit is materialized with no join edge, depending on whether a group-by operator has already been combined in the join unit. Once a join graph is returned, the current join unit will be further expanded while a new join unit is started if a temporary table is returned.

Note that COMBINEJOINGRAPH, which is called in Line 9 of Algorithm 3, modifies the join condition for *op* so that it refers to the materialized temporary table instead of the original tables in the query if $j1$ or $j2$ contains a temporary table. For example, during join unit enumeration for $\bowtie_3$ in Figure 8(b), the join condition of $\bowtie_3$ is changed to refer to TT_1 instead of T_1 or T_2.

5.3 Local optimization and cost estimation of m-way join unit

Before estimating the cost of a join unit, we first check which algorithms are available for a given m-way join unit, since available algorithms may be different depending on the join shape. For example, the *m-way star join* algorithm explained in Section 3.3 is available only for specific join shapes such as star join and snowflake join. If both *m-way star join* and *m-way column join* algorithms are possible, we need to estimate the cost for both and choose the cheaper one.

To estimate the cost of the m-way join unit, *local* join order optimization as opposed to global join order optimization needs to be performed inside the given join unit. For example, in the *star join* algorithm, we need to determine which table is a fact table. For snowflake join, there can be multiple candidates for a fact table, and the candidate with the lowest cost among them is chosen. Conversely, in the *column join* algorithm, the order of semi-join reduction needs to be determined. The semi-join reduction order is decided with a variant of the A* search algorithm, and the order with the lowest cost is stored inside the join unit. We refer readers to [9] for details of the algorithm which determines the semi-join reduction order. This local join order optimization is implemented inside a join unit operator as part of the cost estimation, so it doesn't have to change the global enumeration framework explained in the previous section. The cost of the join unit is the summation of the m-way join cost, group-by cost, and materialization cost. Once the local join order optimization is done, they can easily be calculated using the predefined cost models for each algorithms.

5.4 Optimizer enhancements by exploiting m-way join unit characteristics

As explained in Section 5.2, our approach for enumerating m-way join units starts from the binary joins and finds m-way join unit boundaries in a search space. In this section, we explain how the HANA optimizer reduces the overhead of search space enumeration by exploiting the characteristics of the m-way join unit. Even if these enhancements are applied, we guarantee that the optimal query plan is not overlooked.

5.4.1 Reduce unnecessary join enumeration

If there exist join operators only in a query, and only column tables are involved, we might be able to completely skip join enumeration for different binary join orders since a whole query plan can be mapped to an m-way join unit, and the local join order optimization can be performed inside the join unit. However, in general, we should consider queries that have group-by operators in many analytic applications. In transformation-based enumeration, how deep a group-by operator can be pushed down depends on what types of different binary join orders exist in the search space. For example, let's consider an initial query graph of $\gamma((T_1 \bowtie T_2) \bowtie T_3)$. Depending on the group-by operator, there is a case that the group-by operator can be pushed down, and a logical alternative of $(\gamma(T_1 \bowtie T_3)) \bowtie T_2$ is enumerated. To enumerate the logical alternative, $\gamma((T_1 \bowtie T_3) \bowtie T_2)$ is enumerated first with join reordering, and then the group-by operator is pushed down

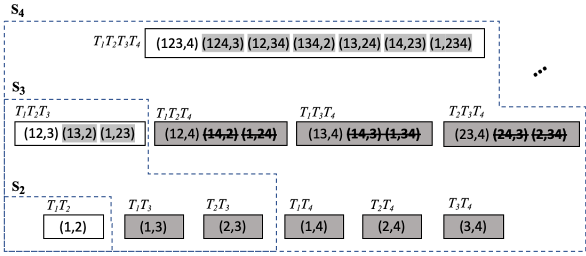


Figure 9: An example of join enumeration reduction.

Algorithm 5: ENUMERATE_ENHANCED(op)

```
7  ApplyLogicalTransformations(op)
8  op.GetEquivClass().SetLogicalEnumFinished()
9  if op.GetEquivClass().ConsistsOfJoinOnly() then
10     foreach logical alternative l_alt in current equivalence
       class do
11         foreach child equivalence class ce of l_alt do
12             ce.SetLogicalEnumFinished()
```

towards the equivalence class, T_1T_3. Thus, binary join reordering is still important for group-by operator pushdown, and complete join enumeration cannot be simply skipped when there is a group-by operator involved in the query. When it is not applicable to push down a whole group-by operator, an eager-aggregation [30] alternative is considered in the HANA optimizer. In the previous example, $\gamma_2((\gamma_1(T_1 \bowtie T_3)) \bowtie T_2))$ can be enumerated with the eager aggregation. The same binary join reordering is needed as well for eager aggregation, and the only difference is that a post group-by operator γ_2 is additionally needed. However, the required binary join reordering is the same for both alternatives, so there is no separate mention of the eager aggregation in the remaining sections.

For the group-by operator pushdown, we don't have to enumerate all binary join orders. Ideally it is enough to generate all possible equivalence classes with a minimal number of binary join enumerations so that the group-by operator can be pushed down to the appropriate equivalence class. The join enumeration that does not contribute to the generation of a new equivalence class is not needed for this purpose and can be skipped safely.

Our philosophy is to minimize the change to existing binary join enumeration as explained in Section 5.2 but to take still effective solution. The basic idea is that 1) all logical enumerations are still performed for the equivalence classes that exist in the initial query graph, and 2) logical enumeration for the newly generated equivalence classes can be skipped if logical alternatives for the equivalence class consist of join operators only, since its logical enumeration does not generate new equivalence classes.

Figure 9 illustrates which join alternatives are skipped for a query graph of $\gamma((((T_1 \bowtie T_2) \bowtie T_3) \bowtie T_4)... \bowtie T_n)$. Equivalence classes and logical join alternatives that are not shaded exist in an initial query graph, and newly generated ones during logical enumeration are shaded. $(12, 3)$ denotes a logical alternative of $T_1T_2 \bowtie T_3$. Note that edges between a logical alternative and child equivalence classes are omitted in the figure, and the equivalence class with a base table only is also omitted for simplicity. Let's consider $T_1T_2T_3$ and $T_1T_2T_3T_4$ that exist in the initial query graph. All binary join enumerations are performed in these two equivalence classes, and new logical alternatives and equivalence classes are generated together. For example, during the logical enumeration of $T_1T_2T_3$, two logical join alternatives, $(13, 2)$ and $(1, 23)$, and two new equivalence classes, T_1T_3 and T_2T_3, are newly generated. Similarly, new join alternatives and new equivalence classes are generated for $T_1T_2T_3T_4$. As shown in the figure, all possible equivalence classes are generated only with the enumeration in the equivalence class that exists in the initial query graph. Therefore, additional logical enumerations in 3 new equivalences classes (i.e. $T_1T_2T_4$, $T_1T_3T_4$, and $T_2T_3T_4$) are skipped, and the logical alternatives that are crossed out in the figure are not enumerated accordingly. Note that equivalence classes that consist of two tables do not have logical alternatives in the SAP HANA optimizer anyway, since two children of the logical inner join operator are unordered, and the order is decided during physical enumeration.

The corresponding algorithm change shown in Algorithm 5 is simple, and it is a slight modification of Algorithm 1. Lines 9-12 in Algorithm 5 are newly added for reducing unnecessary join enumeration. After logical enumeration is finished, the child equivalence classes are marked as logical-enumeration-finished to skip unnecessary join enumeration if the current equivalence class consists of join operators only.

Our approach above still generates all necessary equivalence classes for group-by operator pushdown even though some binary join enumerations can be skipped. To prove this and show how effective the SAP HANA approach is, we first define three variables below for a join query of $((T_1 \bowtie T_2) \bowtie ...) \bowtie T_n$. For simple explanation, a left-deep join tree is assumed in the initial query graph but bushy join trees are also considered during logical enumeration, and completely-connected query graph is assumed.

$\mathbf{S_n}$ is a search space for the join query graph $((T_1 \bowtie T_2) \bowtie ...) \bowtie T_n$. $\mathbf{A_n}$ is the number of logical alternatives in the equivalence class $T_1T_2...T_n$. $\mathbf{J_n}$ is the number of logical join operators in the search space S_n. It is analogous to the size of MEMO table in Volcano/Cascade [17] and the number of join pairs of connected sub-graphs (#*ccp*) in DP-based enumeration [14, 16].

LEMMA 1. *In the SAP HANA optimizer approach, S_n contains all possible equivalence classes even though some binary join enumerations are skipped.*

PROOF. We define S_n as *root-complete* iff A_n contains all possible logical join alternatives. Each join alternative in A_n has k relations in the left child and $n-k$ relations in the right child for $1 \leq k \leq n-1$. Thus, once S_n is *root-complete*, $A_n = \sum_{k=1}^{n-1} \binom{n}{k}/2 = 2^{n-1} - 1$. Then, each equivalence class in S_n except a root equivalence class (i.e. $T_1T_2...T_n$) is referenced by one of logical alternatives in A_n. Note that the root equivalence class already exist in the initial query graph. That is, S_n contains all possible equivalence classes if S_n is *root-complete*. We prove by induction that S_k is *root-complete* for any $k >= 2$ as follows.

- For $k = 2$, S_2 is *root-complete* because T_1T_2 has only one possible join alternative, and it exists in the initial query graph.
- For $k > 2$, S_k is *root-complete* if S_{k-1} is *root-complete*. Since S_{k-1} is *root-complete*, $A_{k-1} = 2^{k-2}-1$. If we add T_k to either left or right side of each join alternative

in A_{k-1}, we can generate two new join alternatives including T_k. Thus, we generate $2*(2^{k-2}-1)$ new join alternatives as well as one join alternative that exists in the initial query graph. They are same as A_k.

□

Figure 9 and the proof above do not explicitly mention a group-by operator, since original S_n is unchanged but expanded when a group-by operator is pushed down towards the illustrated search space. A new equivalence class including the group by operator is created on top of the existing equivalence classes for join operators. For example, when a group-by operator is pushed down towards $T_1T_2T_4$, a new $T_1T_2T_4G$ equivalence class is created by referencing the existing $T_1T_2T_4$.

When all possible join combinations are considered, J_n for a completely connected query graph is $(3^n - 2^{n+1}+1)/2$ as calculated in [17].[1] However, in our approach, it can be reduced to $2^{n+1}-3n-1$, which is beneficial as long as n is greater than or equal to four relations.

Lemma 2. *J_n for a completely connected query graph is $2^{n+1}-3n-1$ in the SAP HANA optimizer approach.*

Proof. During the enumeration for S_n after the enumeration for S_{n-1} is finished, the number of newly added logical join operators (i.e. $J_n - J_{n-1}$) equals to the summation of A_n and the number of newly generated equivalence classes as illustrated in Figure 9, since each newly created equivalence classes has a logical join operator. Since all of newly generated equivalence classes during the enumeration for S_n contain T_n, its number is the summation of $\binom{n-1}{k-1}$ where k is the number of tables involved in the equivalence class and can be varied from 2 to $n-1$. Therefore, J_n can be calculated as follows.

- $J_n - J_{n-1} = A_n + \sum_{k=2}^{n-1}\binom{n-1}{k-1}$ (for $n \geq 3$), $J_2 = 1$
- $J_n = \sum_{k=3}^{n}(2^k-3)+1 = 2^{n+1}-3n-1$

□

For the star-schema query, J_n is $(n-1)^2$ for n relations while $(n-1)*2^{n-2}$ join operators are needed in the original approach. The proof is skipped due to lack of space. The improvement is smaller compared to completely connected query because Cartesian product is already skipped in the SAP HANA optimizer regardless of this optimization. It means that the benefit of this optimization can be limited depending on the query shape and corresponding search space size. The different behavior depending on query shapes will be shown empirically in Section 7.

5.4.2 Avoid redundant m-way join unit generation

An m-way join unit corresponds to a join among m tables where the join order among the m tables is determined locally. Thus, the same m-way join unit can be created from different binary join trees. For example, the m-way join unit created from the binary join tree $(T_1 \bowtie T_2) \bowtie T_3$ is the same as the one created from $(T_1 \bowtie T_3) \bowtie T_2$.

Thanks to the enhancement in Section 5.4.1, logical enumeration for different binary join orders is skipped for newly created equivalence classes, and redundant m-way join units are not created accordingly. However, this enhancement is limited because different binary join order is still enumerated in the equivalence class that exists in the initial query graph such as $T_1T_2T_3$ and $T_1T_2T_3T_4$ in Figure 9. Note that we may have several redundant join units in an equivalence class.

To avoid generating redundant m-way join units, we use the concept of a *join unit set*, whose key consists of both the corresponding equivalence class and a list of tables involved. As shown in Figure 8(b), the equivalence class $T_1T_2T_3T_4G$ requires two join units because both of them are meaningful. The list of involved tables of the first join unit for γ_0 includes T_1, T_2, T_3 and T_4, while the other join unit for $\bowtie_3$ refers to TT_1, T_3, and T_4. This example explains why we maintain a list of involved tables for the key of join unit set as well as an equivalence class. Whenever trying to generate a new join unit for an equivalence class, we need to associate it with the corresponding join unit set. That is, unnecessary join unit creation is checked with the set before creating a new join unit.

5.4.3 Reduce local optimization of m-way join unit

The cost estimation for an m-way join unit can be expensive, since local join order optimization explained in Section 5.3 needs to be performed from scratch. Once a join unit is expanded by combining a join operator, a local join order such as fact-table decision or determining the order of semi-join reduction, needs to be re-calculated without relying on the local join orders of the smaller join units. To reduce the expensive local join optimization of an m-way join unit, the cost estimation is deferred until truly necessary.

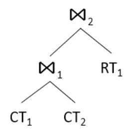
The cost is meaningful only if the equivalence class consists of at least one group-by operator to make a cost-based decision in the event that a group-by operator pushdown or an eager aggregation is better than the original plan. For example, in $T_1T_2T_3T_4G$ of Figure 8(b), two join units fall into the case. However, if the equivalence class consists of join operators only, the cost-based comparison is not needed thanks to *larger join unit heuristic*, and the cost estimation for the corresponding join unit can be skipped. T_1T_2, T_3T_4, and $T_1T_2T_3T_4$ fall into the category.

6. MIXED JOIN BETWEEN COLUMN TABLE AND ROW TABLE

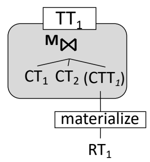
As previously mentioned, SAP HANA supports row tables as well as column tables, and a mixed join between a row table and a column table is also supported [11]. Therefore, the optimizer needs to consider additional physical operators to the scenario of the column table alone, which is the main assumption in Section 5.

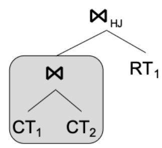
In SAP HANA, two different types of mixed join algorithms are considered as described in Figure 10. Figure 10(b) and Figure 10(c) are two different physical operators enumerated for the given logical alternative in Figure 10(a) - a join between the two column tables CT_1, CT_2, and a join with the row table RT_1. In Figure 10(b), the contents of RT_1 are materialized in a temporary column table, and a dictionary is built over it on-the-fly. Then, one of m-way join algorithms is executed together with CT_1 and CT_2. This kind of physical operator is better when the size of the row table is relatively smaller than other column tables. Conversely, in Figure 10(c), the result of the join between CT_1

[1] The equation in [17] is divided by 2 because two children of logical join operator are unordered in SAP HANA.



(a) Logical alternative.





(b) m-way join execution. (c) Hash join execution.

Figure 10: Comparison of mixed join executions.

and CT_2 is materialized in a temporary row table, and a binary join algorithm such as hash join is executed.

Figure 10(c) illustrates a single hash join algorithm, but different types of binary join algorithms such as index join and merge join can be enumerated as well with separate physical operators. Here, a cost-based decision is made among all available physical operators. These steps are additionally handled in Algorithm 2 when a mixed join is involved. For a mixed join, it is still important to enumerate all available binary join orders for binary join algorithms. Therefore, reducing unnecessary join enumeration is not applicable to the mixed join.

7. EXPERIMENTS

7.1 Setup

This section compares three enumeration variants, ***BJ***, ***NM***, and ***EM***. *BJ* refers to a normal transformation-based enumeration for binary join algorithms without considering m-way joins. *NM* refers to a naive integration of m-way joins in the optimizer. *EM* strengthens *NM* by applying the enhancements mentioned in Section 5.4.

We use two sets of queries, TPC-H SF100 queries and customer queries from a S/4HANA customer database. The former queries are prefixed with T, and the latter are prefixed with C. We show the results for eight TPC-H queries and four customer queries to save space. Table 1 shows how many tables and relational operators are in each initial query plan. For instance, C2 has 22 tables, 16 joins, 17 group-by's and three union-all's. All tables used in the measurement are column tables, which are the main focus of the paper.

7.2 Results

Figure 11 reports the overall optimization times of *NM* and *EM*, relative to that of *BJ*. *NM* shows the worst optimization time for all queries, since the m-way join algorithms are naively considered in addition to the binary join enumeration of *BJ*. However, *EM* shows significant reduction of query optimization time compared to *NM* for all queries. Especially for T2, C3, and C4, more than 90% of the query optimization time of *NM* is reduced in *EM*. In the other seven queries, more than 60% of the optimization time is reduced in *EM*. This demonstrates that our enhancements are very effective. Two TPC-H queries, T11 and T20, show smaller reductions in the optimization time, at 30% and 44%, respectively. The reason reduction ratios are smaller

Table 1: Query characteristics: number of logical operators.

Query	Table	Join	Group-by	Union
T2	9	8	1	0
T5	6	5	1	0
T7	6	5	1	0
T8	8	7	1	0
T9	6	5	1	0
T11	6	5	2	0
T20	5	4	2	0
T21	5	4	2	0
C1	13	10	4	2
C2	22	16	17	3
C3	11	10	1	0
C4	18	17	3	0

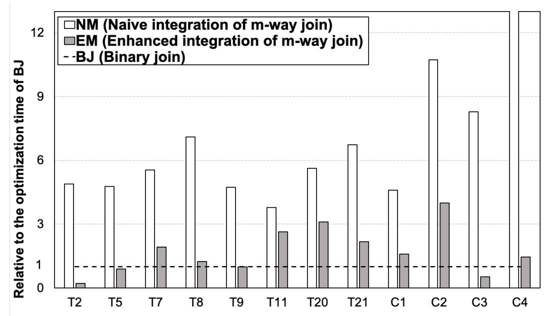


Figure 11: Query optimization times.

in these queries will be explained with the analysis of other metrics. Compared to *BJ*, eight of 12 queries are slower in *EM*, since *EM* still requires some portions of binary join enumeration.

Although there is significant improvement of query optimization time, Figure 12 shows that both *NM* and *EM* configurations show similar query execution times for all queries. This demonstrates that the optimization time improvement of *EM* does not sacrifice the query plan quality. The comparison of *BJ* with *EM* (or *NM*) shows how much faster m-way join algorithms are compared to binary join algorithms for column tables in SAP HANA. Additionally, another baseline that enumerates m-way join algorithms with the group-by pushdown heuristic is compared. This baseline is slower than *EM* in most queries and demonstrates why the cost-based enumeration of group-by operators is important. It is even slower than *BJ* in three queries, since *BJ* considers group-by operators in cost-based enumeration.

To show the impact of each enhancement for efficient m-way join enumeration, additional metrics are analyzed. In Figure 13, we compare the number of logical join operators (J_n) in the search space to show the effect of unnecessary join enumeration reduction, as explained in Section 5.4.1. The results of *BJ* and *NM* are the same for all queries since no enhancement is applied to *NM*, and m-way join is considered during physical enumeration only. However, J_n is reduced in *EM* for all queries except for T11. T2, C3, and C4 have the biggest optimization time reduction; more than 90% of J_n is also reduced, and the significant reduction of J_n contributes to the overall optimization time reduction for these queries. The reduction ratio of J_n generally gets higher as the number of joins increases since the corresponding search space gets bigger. The reason these three queries show significant improvement is that they have many joins.

However, there are other factors that affect J_n. Depending on the query shape, much of the search space is not sub-

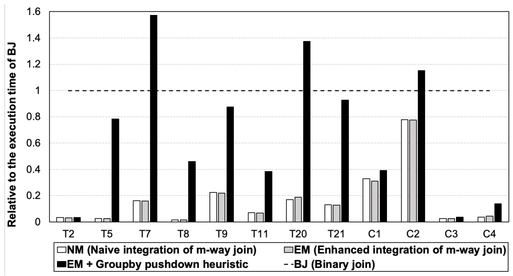


Figure 12: Query execution times.

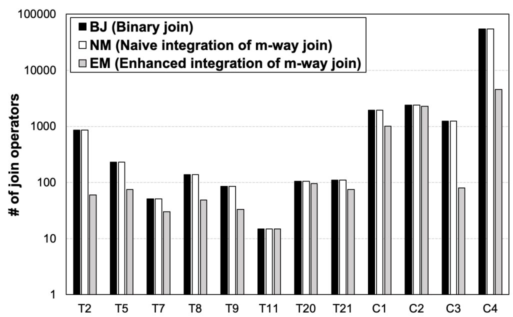


Figure 13: Total number of logical join operators (J_n).

ject to enumeration to avoid Cartesian product and invalid transformation, which limits J_n reduction. For example, T8 has a larger number of joins than T5 and T9, but the reduction ratio of these three queries are similar (61%-68%), since their original search space sizes in *BJ* and *NM* are not so different. Another example is a join operator whose join condition refers to the aggregate value of a child group-by operator. Then, the join operator cannot be reordered with a group-by operator. T11 has no improvement for this reason, while T5, T7, and T9, each of which has the same number of join operators as T11, show 41%-68% reduction. T11 consists of two sub-queries, and each of them has joins with three tables followed by a group-by operator. Two sub-queries are joined by comparison of the aggregate value of each sub-query. Therefore, there is no chance that join reordering can happen across these two sub-queries. Join reordering in each sub-query is still possible, but the reduction of unnecessary join enumeration is not applicable, since it has three tables only. Four or more tables are required for the enhancement to be applicable. For the same reason, the improvement is limited in T20 and C2, which shows 9% and 6% of reduction, respectively.

Figure 14 compares the number of m-way join units that exist in the search spaces for *NM* and *EM*. It is not applicable to *BJ* where m-way join algorithms are not considered. As shown in the figure, *EM* significantly reduces m-way join unit creations for all queries. It shows 29%-94% of reductions in all queries (69% on average). There are two reasons for this. First, the number of logical join operators is significantly reduced, as shown in Figure 13, so the corresponding physical enumeration, which corresponds to m-way join unit creations, is also reduced. The second reason is that redundant join unit creation is avoided by maintaining the *join unit set*, as explained in Section 5.4.2. That's why the reduction ratio of a join unit is usually higher than that of J_n. T11 shows a 29% reduction in join unit creations, even though it has no reduction in J_n.

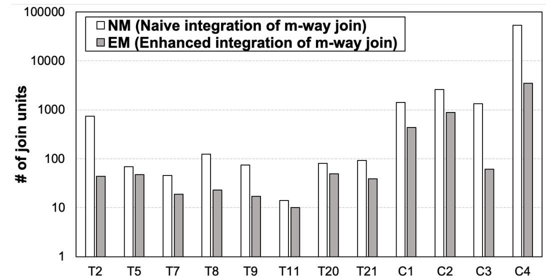


Figure 14: Total number of m-way join units.

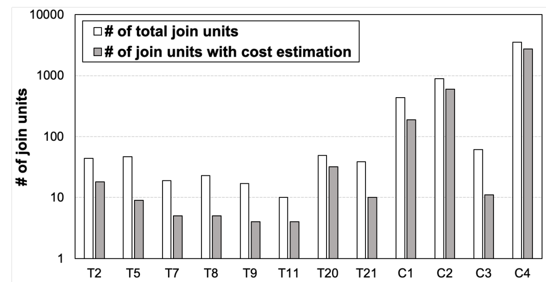


Figure 15: Effect of local optimization reduction.

Figure 15 compares the total number of m-way join units with the number of join units where the cost estimation and corresponding local optimization are performed. It demonstrates that local optimization reduction, explained in Section 5.4.3, is beneficial for all queries. (21%-82%)

8. CONCLUSION

This paper has presented a multi-way join aware query optimizer in SAP HANA. We first introduced a new concept of m-way *join unit*. We then provided a mechanism to enlarge small join units into larger join units in a bottom-up manner. We next provided detailed algorithms on how we implement the join unit in the SAP HANA query optimizer. We then provided a series of optimizer enhancements by exploiting m-way join unit characteristics.

Our empirical results with TPC-H and customer workloads show that our m-way join aware optimizer often finds significantly faster execution plans containing m-way joins with a marginal overhead, compared to the typical binary join optimizer. Our framework is general enough to accommodate other m-way join algorithms such as leapfrog triejoin for graph queries. Overall, we believe we have provided comprehensive insight with a framework for future research.

9. ACKNOWLEDGMENTS

We thank all anonymous reviewers for valuable comments. The continuous efforts of HANA Optimizer Team to improve the product are also appreciated. Additionally, we thank two alumni - Sangyong Hwang, who built a foundation of the P*TIME/HANA query optimizer, and Sangil Song, who contributed to this paper at the beginning. The paper review and valuable comments of Guido Moerkotte are appreciated as well. Lastly, we thank Donghun Lee for coordinating research projects.

Wook-Shin Han was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2017R1A2B3007116).

10. REFERENCES

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM TODS*, 42(4):1–44, 2017.

[2] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *ACM SIGMOD conference*, pages 283–296, 2009.

[3] S. K. Cha and C. Song. P*TIME: Highly scalable OLTP DBMS for managing update-intensive stream workload. In *VLDB conference*, pages 1033–1044, 2004.

[4] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB conference*, pages 354–366, 1994.

[5] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: Data management for modern business applications. *ACM SIGMOD Record*, 40(4):45–51, 2012.

[6] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[7] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[8] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *IEEE ICDE conference*, pages 209–218, 1993.

[9] G. Hill and T. Peh. Fast algorithms for computing semijoin reduction sequences. U.S. Patent 8271478, 2012.

[10] G. Hill and A. Ross. Reducing outer joins. *The VLDB Journal*, 18(3):599–610, 2009.

[11] C. Jeong, S. Hwang, S. K. Cha, and S. H. Wi. Processing database queries using format conversion. U.S. Patent 8880508, 2014.

[12] J. Lee, Y. S. Kwon, F. Färber, M. Muehle, C. Lee, C. Bensberg, J. Lee, A. H. Lee, and W. Lehner. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *IEEE ICDE conference*, pages 1165–1173, 2013.

[13] G. Moerkotte, P. Fender, and M. Eich. On the correct and complete enumeration of the core search space. In *ACM SIGMOD conference*, pages 493–504, 2013.

[14] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB conference*, pages 930–941, 2006.

[15] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *Journal of the ACM*, 65(3):16, 2018.

[16] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB conference*, pages 314–325, 1990.

[17] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The complexity of transformation-based join enumeration. In *VLDB conference*, pages 306–315, 1997.

[18] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD conference*, pages 249–260, 2000.

[19] `https://www.sap.com/products/s4hana-erp.html`.

[20] V. Sikka, F. Färber, A. Goel, and W. Lehner. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. *PVLDB*, 6(11):1184–1185, 2013.

[21] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *ACM SIGMOD conference*, pages 731–742, 2012.

[22] `https://en.wikipedia.org/wiki/Snowflake_schema`.

[23] O. Steinau and J. Hartmann. Method for calculating distributed joins in main memory with minimal communication overhead. U.S. Patent 8 046 377, 2011.

[24] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. Integrating semi-join-reducers into state-of-the-art query processors. In *IEEE ICDE conference*, pages 575–584, 2001.

[25] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[26] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

[27] C. Weyerhaeuser, T. Mindnich, F. Faerber, and W. Lehner. Exploiting graphic card processor technology to accelerate data mining queries in SAP NetWeaver BIA. In *IEEE International Conference on Data Mining (ICDM)*, pages 506–515, 2008.

[28] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[29] W. P. Yan and P. A. Larson. Performing group-by before join. In *IEEE ICDE conference*, pages 89–100, 1994.

[30] W. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *VLDB conference*, pages 345–357, 1995.