SPRINTER: A Fast *n*-ary Join Query Processing Method for Complex OLAP Queries

Yoon-Min Nam
DGIST, Republic of Korea
ronymin@dgist.ac.kr

Donghyoung Han
DGIST, Republic of Korea
icedrak@dgist.ac.kr

Min-Soo Kim*
KAIST, Republic of Korea
minsoo.k@gmail.com

ABSTRACT

The concept of OLAP query processing is now being widely adopted in various applications. The number of complex queries containing the joins between non-unique keys (called FK-FK joins) increases in those applications. However, the existing in-memory OLAP systems tend not to handle such complex queries efficiently since they generate a large amount of intermediate results or incur a huge amount of probe cost. In this paper, we propose an effective query planning method for complex OLAP queries. It generates a query plan containing *n*-ary join operators based on a cost model. The plan does not generate intermediate results for processing FK-FK joins and significantly reduces the probe cost. We also propose an efficient processing method for *n*-ary join operators. We implement the prototype system SPRINTER by integrating our proposed methods into an open-source in-memory OLAP system. Through experiments using the TPC-DS benchmark, we have shown that SPRINTER outperforms the state-of-the-art OLAP systems for complex queries.

CCS CONCEPTS

• **Information systems** → **Query optimization**; **Query planning**.

KEYWORDS

n-ary join operator, query planning, query optimization, FK-FK join, complex OLAP query processing, co-processing

ACM Reference Format:
Yoon-Min Nam, Donghyoung Han, and Min-Soo Kim. 2020. SPRINTER: A Fast *n*-ary Join Query Processing Method for Complex OLAP Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3380565

*A corresponding author

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380565

1 INTRODUCTION

There are a number of in-memory OLAP query processing systems for large-scale data analytics such as Quickstep [34], MemSQL [11], MonetDB [9], Hyrise [16], Oracle [23], DB2 [6], SQL Server [24], SAP HANA [41], Peloton [28], OmniSci [33], CoGaDB [10], Kinetica [21] and Ocelot [17]. These systems mainly focus on processing conventional star/snowflake join queries efficiently that contain a number of join operations between a primary key of a table and a foreign key of another table [45, 50]. One of the goals of query optimization in those systems is reducing the cost of processing intermediate results caused by the join operations in a query plan [40, 45]. To achieve the goal, the existing systems exploit various techniques such as pipelining of intermediate results [28, 50].

The concept of OLAP query processing is now being widely adopted in various applications including graph analytics [1, 13], artificial intelligence [22], and bioinformatics [20, 26]. As the applications become complex, the query workload in those applications tends to become more and more complex. In particular, the number of queries which contain join operations between a pair of foreign (or non-unique) keys rather than a conventional pair of primary and foreign keys increases in the applications. We denote such a join operation as a *FK-FK join* in this paper. A FK-FK join usually incurs a large number of join results due to duplicated join key values. If a query contains FK-FK join operation(s) between two *fact* tables, it becomes more difficult to process the query efficiently due to a huge amount of intermediate results. Such complex queries are commonly occurred on a database having a snowstorm schema which consists of multiple star or snowflake schemas [3]. For example, the TPC-DS benchmark uses a snowstorm schema, and 26 out of a total of 99 queries in the TPC-DS benchmark (i.e., 26.2% queries) contain one or more FK-FK join operations [30].

The existing OLAP systems tend not to handle the complex queries containing FK-FK join operations efficiently. They typically generate a left-deep join tree and process the plan in an operator-at-a-time manner, where the performance may

be significantly degraded, or processing itself may fail, due to too large amount of intermediate results to be processed by the next operator of a FK-FK join (or a non-key join) operator in a query plan [47]. The pipelining technique evaluates a series of multiple join operators in a join tree by looking up each tuple of a probe relation in a set of hash tables built from the remaining relations [27, 50]. It may be effective for a query of PK-FK joins, but may not be effective for a complex query containing FK-FK joins on fact tables due to the following two issues: a lot of key comparisons and a large amount of memory usage. First, a join key value of each tuple in a probe relation needs to be matched to all the duplicate key values in the hash tables. There may be a lot of matches due to the FK-FK join operations, which can largely degrade the query performance in proportion to the degree of duplication of join key values. Second, keeping a set of all hash tables built from non-probing relations in main memory requires a large amount of memory, in particular, when the number of fact tables increases. Due to the above two issues, the performance may be significantly degraded, or processing itself may fail when we use the pipelining technique for a query containing FK-FK joins.

In this paper, we propose an efficient query processing method called *SPRINTER* for complex OLAP queries containing FK-FK joins. The intuition behind SPRINTER is that a query plan of n-ary join operators can reduce the amount of intermediate results compared to the one of a series of binary join operators, and multi-way join processing [44] is not always, but can be very efficient for processing the query plan. Multi-way join processing has been used for distributed query processing [2, 13, 49] and graph pattern query processing [31, 44], but has almost not been used in in-memory OLAP systems since hash join usually outperforms sort-merge join in in-memory processing environments [5, 39], and OLAP queries are ad-hoc and acyclic different from graph pattern queries. We propose a query planning method that can generate a query plan containing n-ary join operators instead of a conventional plan having only a series of binary join operators.

In general, it is non-trivial to generate a query plan having n-ary join operators (called n-ary join tree) since the search space of query plans becomes larger compared to when considering only binary join operators. Our proposed query planning method searches a good query plan heuristically and recursively based on a cost model. We present the cost model used and a query optimization method that allows the query planning method to generate an n-ary join tree only when it is beneficial compared to the conventional binary left-deep join tree. Then, we explain an n-way join processing method for an n-ary join operator that is based on the worst-case optimal join algorithm. We implement all our methods into one of open-source modern in-memory OLAP processing systems, OmniSci [33], across all relevant layers and modules including the query plan generator and the physical join operator. Through extensive experiments using the TPC-DS benchmark, we have demonstrated that SPRINTER significantly outperforms the state-of-the-art OLAP query processing systems in terms of both processing speed and data size that can be processed without our of memory.

Our major contributions are summarized as follows:

- We propose a query planning method that can generate n-ary join trees for complex OLAP queries containing FK-FK joins.
- We propose a cost model and a query optimization method for n-way join trees.
- We present a n-way processing method for an n-ary join operator that is based on the worst-case optimal join algorithm.
- We implement a prototype system based on an open-source OLAP system across all relevant layers and modules including the query plan generator and the physical join operator.
- Throughout extensive experiments, we have demonstrated that SPRINTER significantly outperforms the state-of-the-art CPU-based and co-processing OLAP processing systems.

The rest of this paper is organized as follows. In Section 2.2, we present a worst-case optimal join algorithm used in SPRINTER. In Section 3, we describe a motivation example of this paper. In Section 4, we propose the query planning method for n-ary join trees. In Section 5, we present an n-ary join processing method and optimization techniques to improve query performance. We propose the cost model in Section 6. Section 7 presents the results of experimental evaluation. Finally, we discuss related work in Section 8 and conclude this paper in Section 9.

2 PRELIMINARIES

2.1 Sorting algorithms

In this section, we summarize the existing parallel sorting algorithms and techniques used in our SPRINTER in Table 1. Although the core contribution of this paper is the optimization of the queries containing FK-FK joins using n-ary join operators, sorting using GPU can further improve the query performance when the sizes of inputs of n-ary join operators become very large. Thus, SPRINTER uses different sorting algorithms and techniques depending on the cardinality of a table to be sorted, the number of sorting columns, and the capacity of GPU memory in case of GPU sorting.

The comparison-based algorithms in Table 1 mean that they require the comparison between key values for sorting (e.g., quick sort, merge sort, bubble sort), and the non-comparison based algorithms mean that they do not require such a comparison for sorting (e.g., radix sort). For a given input array of N tuple values, each of which consists of k columns, it is known that the limit of the speed of the comparison-based algorithm is $O(N \cdot logN)$, while that of the non-comparison based algorithms is $O(k \cdot N)$. Thus, the latter is usually faster than the former if k is small.

Table 1: Parallel sorting algorithms and techniques used in SPRINTER.

processor	comparison / non-comparison	algorithm	implementation
CPU	comparison	merge	Intel TBB[37]
	non-comparison	radix	Thrust[7]
GPU	comparison	merge	Thrust[7]
	non-comparison	radix	CUB[29]
CPU + GPU	comparison(CPU), non-comparison (GPU)	merge(CPU), radix(GPU)	heterogeneous sorting[15]

Most of GPU sorting algorithms only can sort the data that can fit in GPU memory. However, the input array to be sorted can be much larger than the capacity of GPU memory in OLAP systems. In that case, SPRINTER uses heterogeneous sorting [15] that exploits both CPU and GPU efficiently. Figure 1 shows the timeline of heterogeneous sorting for an input array X, where we assume that X is too large to fit in GPU memory, and so, we need to split it into six subarrays $\{X_1, \cdots, X_6\}$. Although heterogeneous sorting can use different algorithms on CPU and GPU theoretically, we use radix sort on GPU and merge sort on CPU since the combination usually shows the best performance. Heterogeneous sorting uses multiple GPU streams in order to hide the cost of data copy between main memory and GPU memory as much as possible by overlapping three kinds of low-level GPU operations, H2D copy, sort, and D2H copy. It performs merge sort for some sorted subarrays in main memory using CPU immediately whenever they are available. We assume that $\hat{X}$ is the global sorted array and initialized as an empty array. The red boxes in Figure 1 show such an immediate merge sort using CPU, where $merge(\{X_i, X_j\})$ performs merging a set of sorted chunks $\{X_i, X_j\}$ into $\hat{X}$.

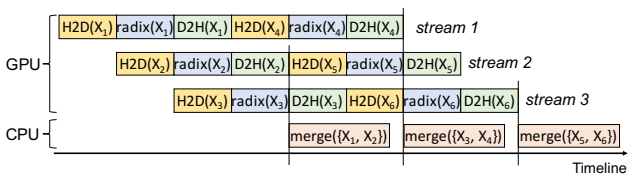


Figure 1: Timeline of heterogeneous sorting [15].

2.2 Worst-case optimal join algorithm

In this section, we describe the worst-case optimal join algorithm that our SPRINTER utilizes for processing of an n-ary join operator. The worst-case optimal join algorithms [13, 44] are mainly proposed to process a graph pattern query (e.g., triangle query) efficiently by avoiding the generation of intermediate results. Some algorithms [1, 31, 44] require preprocessing input relations and storing the result of preprocessing as a data structure like B+-tree before join, while the others [13, 47] just sort the relations and perform join over the sorted relations using binary search. We use the latter kind of algorithms, in particular, Tributary Join (TJ) [13].

Figure 2 illustrates the processing of TJ for a triangle query `Q(x, y, z) :- R(x,y), S(y,z), T(x,z)`, where R, S, and T are the edge relations. The worst-case optimal join algorithms usually use a fixed global ordering on all join variables. We assume the global variable order is $x \prec y \prec z$ in Figure 2. The TJ algorithm sorts R by (x, y) due to the order $x \prec y$, sorts S by (y, z) due to the order $y \prec z$, and sorts T by (x, z) due to the order $x \prec z$, for preprocessing. Then, TJ starts by scanning all relations on the first join variable x (i.e., R and T) and proceeds as merge join until finding a matching value for the variable, e.g., $x = 1$. Then, it simply computes the residual query $Q'(y, z) = R_{x=1}(y), S(y, z), T_{x=1}(z)$ recursively, where the residual relations $R_{x=1}$ and $T_{x=1}$ are actually subarrays of R and T having $x = 1$ obtained by adjusting the start and end points in R and T. During the recursive call, it scans all relations on the next join variable y until finding a matching value $y = 3$. Then, it proceeds recursively again by scanning over z and finally outputs $(1, 3, 4)$.

When seeking a specific value in an array or subarray (e.g., seeking $y = 3$ in S), TJ uses binary search, and so, the cost of a single seek is $O(logN)$. However, the dominating cost of TJ is the sorting cost of input relations [13]. If a query has L join variables, the number of possible global variable orders is $L!$. Although TJ is worst case optimal given any variable ordering, this "worst case" practically can be far from optimal because of different search cost that each variable order has, and so, TJ estimates the cost of join processing for each possible order and choose the best one.

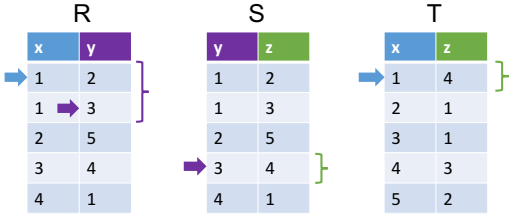


Figure 2: Example of Tributary join [13].

3 MOTIVATION

In this section, we present a motivating query that shows the drawbacks of the existing OLAP query processing systems. The query is the one on the TPC-DS benchmark database [30] and widely used for testing the query performance of the OLAP systems [18]. Although the query contains various operations including aggregation, we focus on join operations in this section. Figure 3 shows a join graph of the query, which contains three fact tables $\{SS, SR, CS\}$ of blue rectangles and three dimension tables $\{D, I, C\}$ of green rectangles. For simplicity, we use only the abbreviations of the relation names in this paper. In the figure, we describe join condition on each edge of the join graph. There are two FK-FK join operations of blue lines, and five PK-FK join operations of green lines.

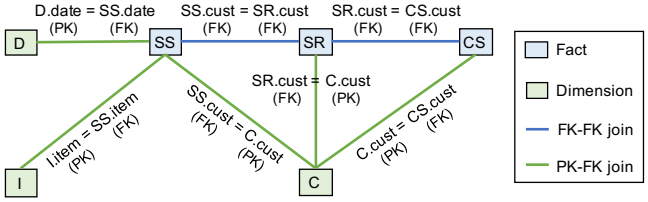


Figure 3: A join graph of the motivating query.

Figure 4 shows the query plans generated by System-X, OmniSci [33], and our SPRINTER for the motivating query and their performance. System-X is a full-featured state-of-the-art commercialized in-memory database system with a support of index-driven query execution and query optimization techniques such as bloom filter and adaptive join. It supports a number of query processing techniques such as index-driven query execution, bloom filter and adaptive join. In Figure 4(a), it generates a left-deep join tree for a query plan and executes the plan in an operator-at-a-time manner. $C1$, $C2$, and $C3$ are the same relation, i.e., a relation C, but dealt with as different relations during query processing, which is common in OLAP processing systems. In addition to System-X, a number of systems including MonetDB [9], CoGaDB [10], Kinetica [21] and Quickstep [34] generate almost the same query plan as in Figure 4(a) and execute it in an operator-at-a-time manner.

The plan in Figure 4(a) generates 2.23 billion (B) intermediate tuples before the last join as the left operand and takes 144 million (M) tuples of CS as the right operand. Then, it builds the hash table for the 144 M tuples and probes the 2.23 B intermediate tuples against the hash table. Because the join operation is FK-FK join, and there are many duplicate key values in the hash table, the number of times that the key values in the hash table are accessed during probing for join, which we call *probe cost*, is enormous, specifically 160.9 B times.

OmniSci [33] is an open-source co-processing database system, where co-processing means exploiting both CPUs and GPUs for query processing. In Figure 4(b), it generates a query plan of a left-deep join tree as in Figure 4(a), but executes the plan in a non-blocking and pipelined manner which does not generate and store intermediate results for join. In detail, it evaluates a series of all join operators in the join tree by probing each tuple of the relation SS against a total of seven hash tables built from the relations $\{SR, CS, I, D, C1, C2, C3\}$ sequentially. The first join between SS and SR in Figure 4(b) probes each tuple of SS against the hash table for SR of 29 M tuples, which probe cost becomes 4.1 B. Then, the second join probes each of the tuples passed the first join against the hash table for CS of 144 M tuples, which probe cost becomes 20.7 B. We can calculate the probe cost of each join similarly, and the total probe cost becomes 25.8 B.

SPRINTER is the prototype system that our proposed methods are integrated into OmniSci across all relevant layers and modules seamlessly. We choose OmniSci as a base system for SPRINTER since it is one of the state-of-the-art open-source modern database systems. Although we present SPRINTER using OmniSci as the base system, SPRINTER can use any database system as a base system in principle. In Figure 4(c), SPRINTER generates a query plan consisting of multiple binary join operators in white and a single n-ary join operator in red. It executes each of three join subtrees, i.e., $S_1 = \{SS, C1, D, I\}$, $S_2 = \{SR, C2\}$, and $S_3 = \{CS, C3\}$, in a pipelined manner like OmniSci, and then, executes the n-ary join operator in operator-at-a-time manner which will be described in Section 5. When we calculate the total probe cost for S_1, S_2, and S_3, it becomes just 758 M. In addition, when we calculate the total cost of processing the n-ary join among the results of three join subtrees, it becomes 312 M. Overall, the total processing cost of SPRINTER is about 1.07 B, which is much smaller than those of the other two systems.

Figure 4(d) shows the query performance of the above three systems for the TPC-DS SF=100 database. The performance results show that OmniSci improves the performance of System-X by eliminating the large intermediate results through pipelining, and SPRINTER improves the performance of OmniSci by splitting a single large join tree into multiple smaller join subtrees and performing n-ary join over the results of the join subtrees. We will explain a more exact cost model than the figure in Section 6.

4 QUERY PLANNING METHOD

In this section, we first present the query planning method for generating a query plan containing a single n-ary join operator in Section 4.1. Then, we generalize the method for more complex queries such that it can generate a query plan required to contain multiple n-ary join operators in Section 4.2.

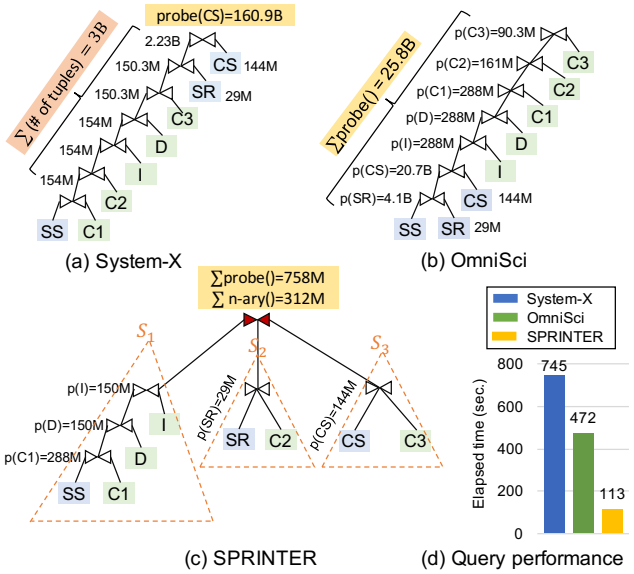


Figure 4: Query plans for the motivating query.

4.1 Query of a single *n*-ary join operator

For query planning, we consider a *join graph* from a given query Q, which is defined as in Definition 4.1.

Definition 4.1. **(Join graph)** A join graph $G = (V, E, f(v \in V), g(e \in E), h(e \in E))$ of a query Q is an undirected multi-graph. A vertex $v \in V$ indicates a relation to be joined in Q, and an edge $e = (X, Y) \in E$ a join operation between two end tables X and Y, especially between $X[i]$ and $Y[j]$, where $i \in X$, and $j \in Y$. There are three labeling functions $f(v)$, $g(e)$, and $h(e)$. The function $f(v)$ returns the type of the relation v, i.e., fact or dimension, the function $g(e)$ the type of the join operation e, i.e., PK-FK or FK-FK, and the function $h(e)$ the equi-join predicates of e, i.e., $h(e) = (X[i], Y[j])$.

In this section, we consider the case where a join graph G contains only a single subgraph of vertices connected through the edges of FK-FK join operations. We denote such a subgraph as core subgraph and define it as in Definition 4.2.

Definition 4.2. **(Core subgraph)** A core subgraph of a join graph G, i.e., $core = (V_c, E_c) \subseteq G$, is a connected component in which any two vertices X and Y s.t. $X \in V_c$ and $Y \in V_c$ connected through a set of edges $E_{X,Y} \subseteq E_c$ s.t. $g(e \in E_{X,Y})$ = FK-FK, $f(X) = fact$ and $f(Y) = fact$.

In Figure 3, a subgraph $\{SS, SR, CS\}$ is a core subgraph since all vertices are fact tables, and a pair of vertices $SS - SR$ is connected via a FK-FK edge, and the other pair of vertices $SR - CS$ is also connected via a FK-FK edge. The subgraph $\{SS, SR, CS\}$ is the maximal subgraph connected through FK-FK edges in the join graph in Figure 3. If a pair of vertices $SS{-}SR$ has two edges $\{e_1, e_2\}$ s.t., $g(e_1) = FK{-}FK$ and $g(e_2) = PK - FK$, only e_1 belongs to the core subgraph $\{SS, SR, CS\}$.

A system can find a core subgraph in most cases based on metadata and statistics such as referential constraints, table cardinality and the number of distinct values of a column. In case that a system has limited knowledge about them, the techniques for finding them including automatic foreign-key detection [12, 48] will be helpful to find a core subgraph.

If a join graph G has only a single core subgraph *core*, then we can decompose G into the core subgraph *core* and a set of non-core subgraphs which are disjoint with each other in terms of edges. Here, any two subgraphs of either core or non-core can have one or more common vertices between both subgraphs. Figure 5 shows one of possible decompositions for the join graph in Figure 3, where there are a single core subgraph $core = \{e_4, e_5\}$ and three non-core subgraphs $G_1 = \{e_3, e_6, e_7\}$, $G_2 = \{e_2\}$, and $G_3 = \{e_1\}$. In general, there are a lot of possible decompositions for a join graph. We denote a set of possible decompositions as $\{D_i\}$. For example, we can decompose the join graph into a single core subgraph $core = \{e_4, e_5\}$ and a single non-core subgraph $G_1 = \{e_1, e_2, e_3, e_6, e_7\}$. Then, we can represent the above two decompositions as $D_1 = \{core, G_1, G_2, G_3\}$ and $D_2 = \{core, G_1\}$.

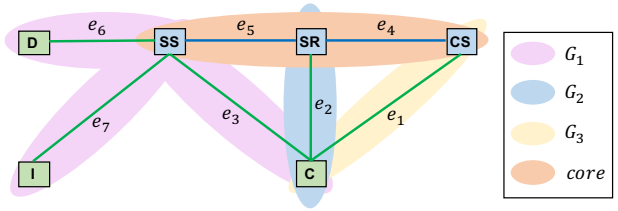


Figure 5: A decomposition of the motivating query.

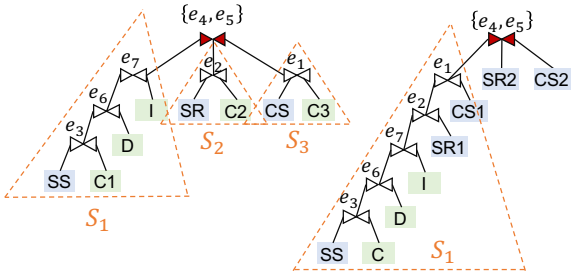
Algorithm 1 presents the basic query planning method for a query containing only a single core subgraph. The intuition behind this method is making a core subgraph a root node of a query plan and making non-core subgraphs the children of the root node in the query plan. It finds possible decomposition $\{D_i\}$ from a join graph G (Line 2), generates a query plan P_i from each decomposition D_i (Lines 3-9), and then picks the best plan P^* having the minimum cost among them (Line 11). We will present the cost model for a plan in Section 6. When making a join subtree S_j from a non-core subgraph G_j (Line 6), there are a lot of possible join subtrees in general. We however only consider generating a left-deep binary join subtree for a non-core subgraph in this paper, which can significantly reduce the search space of query plans. Making a core subgraph a root of a query plan is also a heuristic approach to reduce the search space of query plans. The intuition behind the heuristic approach is to reduce the amount of intermediate results generated from the join operations in non-core subgraphs by processing FK-FK joins on fact tables at the last step in the query plan.

Algorithm 1: QueryPlanning

Input: join graph G
Output: query plan P^*

```
1  core ← find a core subgraph in G;
2  {D_i} ← find possible decompositions in G;
3  foreach D_i ∈ {D_i} do
4      P_i ← make a query plan tree having core as a root;
5      foreach G_j ∈ {D_i} do
6          S_j ← make a join subtree from G_j;
7          make S_j a child of P_i;
8      end
9      ordering {S_j};
10 end
11 P* ← pick the best plan among {P_i};
12 Return P*;
```

Figure 6 shows the two possible query plan trees generated from the above two decompositions $D_1 = \{core, G_1, G_2, G_3\}$ and $D_2 = \{core, G_1\}$. Each binary join operator in Figure 6 corresponds to a green edge in Figure 5. In contrast, each root n-ary join operator in Figure 6 corresponds to a set of blue edges $\{e_4, e_5\}$ in Figure 5. We note that a relation shared among core or non-core subgraphs appears multiple times in a query plan. For example, C appears three times in Figure 6(a) since it is shared among three subgraphs $\{G_1, G_2, G_3\}$ in Figure 5. When comparing two query plans in Figure 6, we can say the plan P_1 usually has a lower cost than the plan P_2 since P_1 scans a dimension table C two more times, whereas P_2 scans two fact tables SR and CS one more time.



(a) Query plan P_1 (b) Query plan P_2

Figure 6: Query plans for the motivating query.

4.2 Query of multiple n-ary join operators

In this section, we explain the query planning method when there are multiple core subgraphs in a query. We denote the number of core subgraphs as N_{core}. A naive query planning method for such a query is to regard a specific core in a join graph as the only core in the join graph and applies the method explained in Section 4.1 to the join graph. A better planning method would be the generalization of Algorithm 1 by applying Algorithm 1 to all non-core subgraphs recursively. For the generalized version, we only need to modify Line 6 in Algorithm 1. In detail, we call the below Algorithm 2 at Line 6 in Algorithm 1 by considering G_j as the input of Algorithm 2 and considering the output P^* of Algorithm 2 as S_j. In Algorithm 2, Line 1 generates a conventional query plan P_{old} (e.g., left-deep join tree) for G_j, which is done by the query planning method of the base system (e.g., OmniSci for SPRINTER). If G_j has no core subgraph, then Algorithm 2 returns the conventional query plan as output. If G_j has one or more core subgraphs, Line 3 generates a query plan P_{new} having a n-ary join operator as a root node for G_j, which is done by Algorithm 1. Then, Line 4 estimates the cost of P_{old} and the cost P_{new}, which will be explained in detail in Section 6. After comparing both costs, only the plan having the lower cost is returned as output.

Algorithm 2: QueryOptimization

Input: join graph G
Output: query plan P^*

```
1  P_old ← QueryPlanning(G) of the base system;
2  if G contains at least one core subgraph then
3      P_new ← QueryPlanning(G);          // Algorithm 1
4      if cost(P_new) < cost(P_old) then
5          P* ← P_new; // plan having n-ary join op.
6      else
7          P* ← P_old; // conventional plan(e.g. left-deep)
8  else
9      P* ← P_old;
10 Return P*;
```

4.3 Search space

We can calculate the number of possible query plans for a query in our approach. In general, there are $N_{core}(G)$ core subgraphs in a join graph G, and the number of decompositions depends on which core subgraph is selected as a root node. We denote the number of decompositions when having a core subgraph C_i as a root node as $N_{dcmp}(C_i)$. If Line 6 in Algorithm 1 always uses the conventional plan for a subgraph, each decomposition would generate a single query plan. Thus, we can calculate the number of possible plans having a single n-ary join operator (as a root node) generated by the naive method as in Eq. 1.

$$N_{naive}(G) = \sum_{i=1}^{N_{core}(G)} N_{dcmp}(C_i) \quad (1)$$

If Line 6 in Algorithm 1 uses Algorithm 2, each decomposition could generate multiple query plans. For a specific decomposition D_j $(1 \le j \le N_{dcmp}(C_i))$, we assume there are $N_{subg}(D_j)$ non-core subgraphs. For example, in Figure 6, $N_{subg}(D_1) = 3$, and $N_{subg}(D_2) = 1$. For each subgraph G_k, there exists $N_{recur}(G_k)$ query plans. Thus, we can calculate the number of possible plans generated by the generalized

method, which can have up to $N_{core}(G)$ n-ary join operators, as in Eq. 2.

$$N_{recur}(G) = \sum_{i=1}^{N_{core}(G)} N_{dcmp}(C_i) \sum_{j=1}^{N_{dcmp}(C_i)} \prod_{k=1}^{N_{subg}(D_j)} N_{recur}(G_k) \quad (2)$$

In Figure 6(a), it is possible to make 3! = 6 query plans depending on the order of join subtrees. In Eq. 2, we do not take into account the order among join subtrees because a total processing time of the join subtrees are the same, regardless of the order. We will explain it in detail in Section 5. Thus, we can use any order for join subtrees at Line 9 in Algorithm 1.

5 n-ARY JOIN PROCESSING METHOD

In this section, we present the n-ary join processing method of SPRINTER. A naive method for processing an n-ary join operator having a set of child join subtrees $\{S_1, \cdots, S_n\}$ would be performing the following six steps.

- Step 1: Evaluate $\{S_1, \cdots, S_n\}$ one by one.
- Step 2: Calculate the necessary statistics for the results of $\{S_1, \cdots, S_n\}$.
- Step 3: Estimate the cost of each global order of join variables.
- Step 4: Choose the best global variable order.
- Step 5: Sort the results of $\{S_1, \cdots, S_n\}$ by the global order.
- Step 6: Merge join on the n sorted relations.

For an n-ary join operator, the number of possible orderings of the children becomes $n!$ in principle. We assume that the children of an n-ary join operator are evaluated from left to right. For the naive method, the elapsed time to process an n-ary join operator becomes the same regardless of the processing order of the children. Figure 7(a) shows an example timeline to evaluate three join subtrees $\{S_1, S_2, S_3\}$ according to the above six steps. In the case where the n-ary join operator has relations instead of join subtrees as its children, e.g., $SR2$ and $CS2$ in Figure 6(b), we regard the relations as S_2 and S_3, respectively. We denote the elapsed time of evaluating S_i as $T_{eval}(S_i)$ and the elapsed time of sorting its result as $T_{sort}(S_i)$. We assume that $T_{eval}(S_i) = T_{sort}(S_i)\,(1 \le i \le 3)$ for simplicity and $T_{eval}(S_i) < T_{eval}(S_j)\,(i < j)$. Steps 2-4 are done at the time t_1 in Figure 7(a), which determines a certain global order among join columns for applying the TJ algorithm. Step 5 sorts each result according to the global order, and Step 6 is done at the time t_2 by using the TJ algorithm described in Section 2.2. We can know that the total elapsed time would not be changed in Figure 7(a) even if the evaluation order of $\{S_1, S_2, S_3\}$ is changed, or even if $T_{eval}(S_i) > T_{sort}(S_i)$, or $T_{eval}(S_i) < T_{sort}(S_i)$.

In the naive method, $T_{eval}(S_i)$ and $T_{sort}(S_i)$ cannot overlap each other due to a kind of synchronization barrier at the time t_1, and thus, it is hard to reduce the total elapsed time much even if we exploit GPU for the sorting step. Thus, we use a modified method for processing an n-ary join operator, which consists of the following four steps.

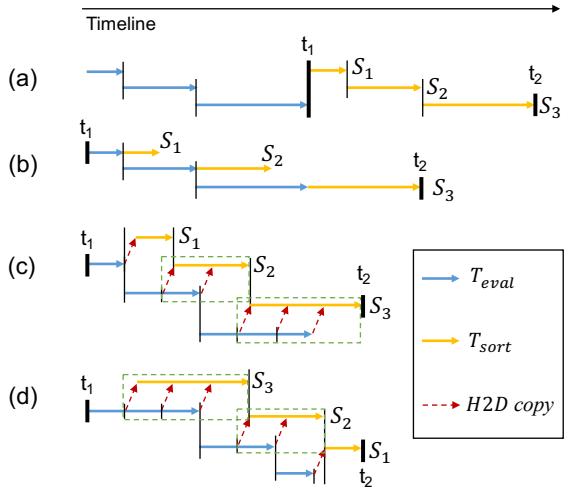


Figure 7: Timelines of the evaluation of three join subtrees $\{S_1, S_2, S_3\}$ in Figure 6(a).

- Step 1: Estimate the cost of each global order of join variables.
- Step 2: Choose the best global variable order.
- Step 3: Overlap evaluating $\{S_1, \cdots, S_n\}$ with sorting their results.
- Step 4: Merge join on the n sorted relations.

Figure 7(b) shows the timeline of the modified method. In Steps 1-2, it first determines the global variable order without calculating the statistics for the results of $\{S_1, \cdots, S_n\}$ (at the time t_1). We will explain how to determine the global order for SPRINTER in detail in Section 5.1. Then, in Step 3, it overlaps evaluating S_i using CPU and sorting the result of S_j using GPU ($i \neq j$). In this method, the total elapsed time is still not changed in Figure 7(b) regardless of the processing order of $\{S_1, S_2, S_3\}$, and also, not changed regardless of whether $T_{eval}(S_i) > T_{sort}(S_i)$, or $T_{eval}(S_i) < T_{sort}(S_i)$.

We can further reduce the total elapsed time than in Figure 7(b). SPRINTER can exploit the pipelining technique for evaluating each child join subtree since its base system, i.e., OmniSci, is the one based on the pipelining technique. Since SPRINTER can use both CPU and GPU simultaneously, and at the same time, use the pipelining technique, evaluating S_i and sorting the result of S_j can overlap with each other even if $i = j$. Figure 7(c) shows the timeline when both use CPU and GPU simultaneously and enables the pipelining technique. We assume that the sizes of the intermediate results of S_1, S_2 and S_3 are one, two and three times larger than the size of data that can be processed in GPU at a time, respectively. In the figure, the red dotted lines mean host (i.e., main

memory) to device (i.e., GPU memory) copying, denoted as H2D copy. For S_2 and S_3, the partial intermediate result is collected through the pipelining technique and chunk-copied to GPU memory for being sorted.

Even though we use CPU and GPU simultaneously and also use the pipelining technique, the total elapsed time is not changed depending on the processing order of the children. We regard $T_{eval}(S_i) + T_{sort}(S_i)$ as the cost of the child S_i. Figure 7(c) shows the strategy of evaluating a Lower Cost child node First (LCF), while Figure 7(d) the strategy of evaluating a Higher Cost child node First (HCF). We can see both strategies are completed at the same time. This tendency is maintained regardless of whether $T_{eval}(S_i) > T_{sort}(S_i)$, or $T_{eval}(S_i) < T_{sort}(S_i)$. Therefore, we use any fixed strategy (e.g., HCF) for ordering the children of an n-ary join operator.

5.1 Determination of Global Order

In general, it is a very challenging problem to calculate the cost of each global join variable order precisely within a reasonably short time. The existing worst-case optimal join algorithms also use heuristic algorithms practically to determine the global order for graph pattern queries [1, 13, 44]. We also present a heuristic algorithm in this paper that can determine a reasonably good global variable order quickly for OLAP queries, which we will further improve in future work.

Algorithm 3: Selecting Global Variable Order

Input: core subgraph *core*
Output: global variable order $\hat{W}$

```
1  W ← join variables (columns) in core;
2  list ← ∅;
3  foreach w ∈ W do
4      E_w ← a set of edges (join conditions) in core using w;
5      U_w ← a set of relations having w;
6      C_w ← Σ_{R∈U_w} ||R||; // ||·||: cardinality function
7      list ← ⟨w, |E_w|, C_w⟩;        // |·|: length function
8  end
9  Ŵ ← sort list by the descending order of (|E_w|, C_w);
10 Return Ŵ;
```

Algorithm 3 shows our heuristic algorithm. Given a core subgraph, it finds all the join variables and prepares a triplet $\langle w, |E_w|, C_w\rangle$ for each join variable, where w is a join variable, $|E_w|$ the number of join conditions for w, C_w the sum of cardinalities of the relations having w. For example, Figure 8 shows a core subgraph of TPC-DS Q17, which has three join variables, *item*, *cust*, and *ticket*. For *item*, $|E_{item}| = 2$, and $C_{item} = ||CS|| + ||SR|| + ||SS||$. Then, the algorithm sorts the list of triplets by the descending order of $(|E_w|, C_w)$. Here, if a join variable w_1 may tie with another join variable w_2 in terms of $(|E_w|, C_w)$, either one can precede the other in the global variable order. For example, both $item \prec cust \prec ticket$ and $cust \prec item \prec ticket$ are allowed. Intuitively, the algorithm chooses a join variable having more join conditions and potentially more tuples to be processed as higher priority to reduce the total amount of binary search on sorted relations. We will show the impact of this approach in Section 7.3.

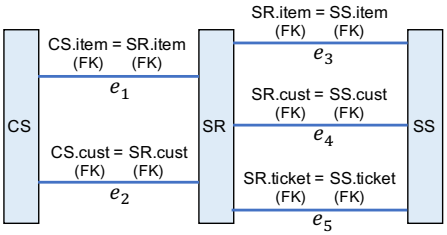


Figure 8: A core subgraph of TPC-DS Q17.

5.2 Strategy for Sorting

In this section, we present our strategy of selecting a specific sorting algorithm for the result of each child join subtree. Figure 9 shows the strategy considering the following three factors: availability of GPU, the size of data to be sorted, and the number of sorting columns. First, if GPU is not available, SPRINTER uses CPU sorting, in particular, a non-comparison based algorithm if the number of sorting columns is only one, but otherwise a comparison-based algorithm, as explained in Section 2.1. Second, if the data to be sorted can fit in GPU memory, SPRINTER uses a non-comparison based or comparison-based GPU sorting algorithm depending on the number of sorting columns. Third, if the data to be sorted cannot fit in GPU memory, we need to select a sorting algorithm carefully due to the issue of lack of good implementations for GPU sorting. To the best of our knowledge, the implementation of a comparison-based algorithm for GPU [7] is only slightly faster than that for CPU [37].

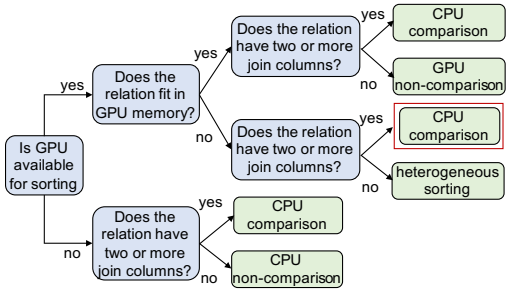


Figure 9: Strategy for selecting sort algorithms.

If a relation to be sorted has multiple sorting columns and is larger than GPU memory, a typical method would split the relation into subarrays, sort each subarray using the comparison-based algorithm for GPU, and merge the subarrays using CPU. However, the cost of merging sorted

subarrays can be quite large since the column values in each sorted subarray are duplicated and not ordered in terms of a specific sorting column as the index of the sorting column increases (e.g., 3rd, 4th). The combination of sorting using GPU and merging using CPU is actually slower than sorting using CPU in many cases, and thus, we use CPU sorting for the case in red-lined box, which can be changed if the performance of the comparison-based GPU sorting algorithm is improved later. If a relation has only a single sorting column, we just use heterogeneous sorting in Section 2.1 since there is a very fast implementation of a non-comparison based algorithm for GPU, and the cost of merging subarrays is not so large. We will show experimental evaluation of each case in Figure 9 in Section 7.

Since SPRINTER uses the columnar layout like other modern OLAP systems, it maintains both a set of join column arrays and a tuple ID vector (tidVec) for each S_i ($1 \le i \le n$), where the former is used for sorting each S_i and joining $\{S_i\}$, and the latter used for tuple reconstruction. The maximum number of tuples that can be sorted in a single GPU at once depends on the implementation of sort algorithm used. For example, when we sort the result of S_i having a single 4-byte join column and 4-byte tuple ID with NVIDIA GTX 1080 ti of 11 GB memory, we can sort about 1.4 and 0.7 billion tuples using in-place and out-of-place sort, respectively.

5.3 Merge Join of Sorted Relations

For an n-ary join operator having n join subtrees $\{S_1, \cdots, S_n\}$, since we have n sorted results $\{\hat{S_1}, \cdots, \hat{S_n}\}$ by sorting the result of each join subtree, we can easily perform merge join using the TJ algorithm. Figure 10(a) shows an example of merge join for the ternary join operator of the core subgraph in Figure 8. For simplicity, we denote the columns *item*, *cust* and *ticket* as i, c and t, respectively. We assume the global order is $i \prec c \prec t$. SPRINTER scans $\{\hat{CS}, \hat{SR}, \hat{SS}\}$ on the first join variable i and assume the current pointer is $i = 2$ (blue arrow). Then, it performs a residual query $Q'(c,t) = \hat{CS}_{i=2}(c), \hat{SR}_{i=2}(c,t), \hat{SS}_{i=2}(c,t)$ recursively on the second join variable c, which can find at least one matching, i.e., $c = 4$ (green arrow). Thus, it performs a narrower residual query $Q''(t) = \hat{CS}_{i=2,c=4}(\cdot), \hat{SR}_{i=2,c=4}(t), \hat{SS}_{i=2,c=4}(t)$ recursively. The residual query $Q''(t)$ finds two matching tuples $\{(2,4,1), (2,4,1)\}$. Similarly, it can find four more matching tuples of $(2,4,1)$ by moving the green pointer in $\hat{CS}$. In this way, we can process a core subgraph having many FK-FK joins without generating a large amount of intermediate results.

One may think about another merge join method that sorts CS, SR, and SS only by i and so reduces the sorting cost. It however may significantly increase the merge join cost and so degrade the overall performance. Figure 10(b) shows such an example where the values in column c are not ordered, and we assume CS and SR are the same with Figure 10(a). When performing the residual query $Q'(c,t) = \hat{CS}_{i=2}(c), \hat{SR}_{i=2}(c,t), \hat{SS}_{i=2}(c,t)$, we have to scan the residual relation $\hat{SS}_{i=2}$ on the column c sequentially instead of doing binary search. Thus, it is important to sort all the relations in a core subgraph according to all join variables.

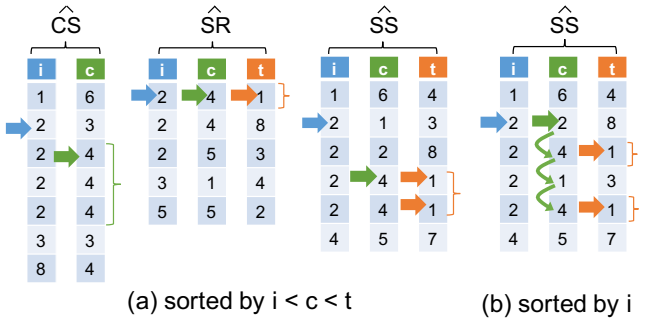


Figure 10: Example of merge join for the core subgraph in Figure 8.

Table 2: Summary of symbols.

symbol	description
p_i	filtering predicates on a relation R_i
F_i	result relation after applying p_i to R_i
f_i	selectivity of p_i ($0 \le f_i \le 1$, 0 means no tuple pass, 1 all tuples pass)
dup_i	average number of elements having the same key in a hash table for F_i
$u(w)$	number of relations related to a join variable w
N^w_{min}	$min\{\|\|S_{(1)}\|\|, \cdots, \|\|S_{(u(w))}\|\|\}$
N^w_{max}	$max\{\|\|S_{(1)}\|\|, \cdots, \|\|S_{(u(w))}\|\|\}$

6 COST MODEL FOR QUERY OPTIMIZATION

A query processing using n-ary join operators may not always achieve better performance than the conventional query processing methods using only binary join operators. Thus, we make SPRINTER generate a query plan containing n-ary join operators only when it is beneficial in terms of the cost model. Given a query Q, Algorithm 2 considers the original plan P_{old} and the new plan P_{new}. Thus, we establish both cost models $cost(P_{new})$ and $cost(P_{old})$ in order to determine whether $cost(P_{new}) < cost(P_{old})$. In particular, we establish the cost model for $cost(P_{old})$ with respect to the base system (i.e., OmniSci). We consider the number of tuples (or elements in a hash table) accessed during query processing as the measure of the cost. Table 2 summarizes the symbols used in this section.

6.1 Cost model of the base system

We consider that a query plan P_{old} consists of $M-1$ binary join operators for M relations. Each input relation R_i may have its own set of filtering predicates p_i, and we consider

the result relation F_i after applying the predicates to the relation ($1 \leq i \leq M$). We consider each binary join operator of either PK-FK join or FK-FK join is evaluated using main-memory hash join algorithm, which is common not only in OmniSci [33] but also in other in-memory query processing systems [5, 8, 19, 28, 50]. We assume that query processing probes each tuple of the left-most relation, i.e., F_1 against the set of hash tables built from the remaining relations $\{F_i|2 \leq i \leq M\}$ in a pipelined manner. Eq. 3 shows the cost function for P_{old}, where $build(F_i)$ is the cost function for building the hash tables (Eq. 4), and $probe(F_1)$ the cost function for probing the tuples of F_1 (Eq. 5).

$$cost(P_{old}) = \sum_{i=2}^{M} build(F_i) + probe(F_1) \tag{3}$$

$$build(F_i) = \kappa \cdot ||F_i|| \tag{4}$$

$$probe(F_1) = \sum_{i=2}^{M}(||F_1|| \cdot \prod_{j=3}^{i} f_{j-1} \cdot dup_i) \tag{5}$$

In Eq. 4, κ is a constant value indicating the number of full scans on the relation F_i for building the corresponding hash table. We use $\kappa = 2$ since our base system uses a common technique of a prefix sum over a histogram on a join key [19] which requires two full scans on a relation. The value κ depends on the base system used. In Eq. 5, we consider that the term $\prod f_{j-1}$ becomes 1 when $j > i$. For example, the cost of probing of F_1 against F_2 (i.e., $i = 2$) becomes $||F_1|| \times dup_2$ since the term $\prod_{j=3}^{i} f_{j-1} = 1$. However, the term becomes itself when $j \leq i$ (e.g., $\prod_{j=3}^{i=4} f_2$). Without loss of generality, we can assume that each tuple of F_1 is compared with dup_2 elements in the hash table for F_2 regardless of the type of the hash table (e.g., open addressing, separate chaining). After probing against F_2, we can assume that a total of $||F_1|| \cdot f_2$ tuples survive and are probed against F_3. That is, each of $||F_1|| \cdot f_2$ tuples is compared with dup_3 elements on average in the hash table for F_3. After probing against F_3, a total of $||F_1|| \cdot f_2 \cdot f_3$ tuples survive, and each tuple is compared with dup_4 elements on average in the hash table for F_4. In this way, we can aggregate the number of elements accessed in the hash tables on the pipeline as in Eq. 5.

6.2 Cost model of SPRINTER

In general, a query plan P_{new} consists of n-ary and binary join operators for M relations. We assume there is at least one n-ary join operator in P_{new} and do not need to generate P_{new} otherwise. In particular, P_{new} has an n-ary join operator as a root according to the query planning method in Section 4. We denote the n-ary join operator as O and assume that O has the join subtrees as its children $\{S_i|1 \leq i \leq n\}$. We note that, if a join subtree S_i also has one or more n-ary join operator, then it also has the operator as a root due to the query planning method in Section 4. Thus, we can define the cost function for P_{new} recursively as in Eq. 6, where $cost(S_i)$ becomes Eq. 3 if S_i has no n-ary join operator and becomes Eq. 6 otherwise.

$$cost(P_{new}) = \sum_{i=1}^{n} cost(S_i) + nary(O) \tag{6}$$

Eq. 7 represents the cost of n-ary join processing, which consists of the sorting cost for the results of n join subtrees and the join cost using the TJ algorithm. SPRINTER uses different sorting algorithm according to the sorting inputs, i.e., the number of sorting columns and the size of a table, and we omit the analysis of the costs of those algorithms since they are already well-known in literature.

$$nary(O) = (\sum_{i=1}^{n} sort(S_i)) + TJ(O) \tag{7}$$

If $w_1 \prec w_2 \prec \cdots \prec w_L$ is determined as a global variable order for the n-ary join operator O, Eq. 8 shows the cost of the TJ algorithm for merging n sorted relations $\{\hat{S}_i|1 \leq i \leq n\}$. We denote the number of relations related to a specific join variable w as $u(w)$. For example, $u(item) = 2$, and $u(ticket) = 1$ in Figure 8. We can consider the relation having the minimum cardinality among all the relations $\{S_{(1)}, \cdots, S_{(u(w))}\}$ related to the join variable w and denote its cardinality as N^w_{min}. Likewise, we can consider the relation having the maximum cardinality for w and denote its cardinality as N^w_{max}. Then, the cost of the TJ algorithm for a single join variable w becomes Eq. 9, which is also summarized in [44].

$$TJ(O) = \sum_{w=1}^{L} search(O, w) \tag{8}$$

In Eq. 9, the term $(1 + log(N^w_{max}/N^w_{min})$ indicates the amortized cost of binary searches for the next key value. Since TJ searches relations according to the global variable order consecutively, the total cost of TJ becomes Eq. 8.

$$search(O, w) = u(w) \cdot N^w_{min} \cdot (1 + log(N^w_{max}/N^w_{min})) \tag{9}$$

7 EXPERIMENTAL EVALUATION

In this section, we present experimental results in two categories. First, we compare SPRINTER with the existing OLAP query processing systems in terms of the elapsed times for complex OLAP queries in the TPC-DS benchmark. Second, we show the characteristics of SPRINTER. In detail, we empirically validate the strategy of selecting a sorting algorithm described in Section 5, the cost model proposed in Section 6, and the performance of sort algorithms.

7.1 Experimental Setup

Queries and datasets: There are a total of 26 TPC-DS queries having at least one FK-FK join in the TPC-DS benchmark [30]. Among the queries, we found that only a total of eleven queries can be evaluated commonly in all the OLAP systems

compared, and the remaining queries are not supported by at least one of OLAP systems with parsing errors. Thus, we use the eleven queries to compare the systems. We note that SPRINTER shows the same performance with the base system (e.g., OmniSci) for the queries having no FK-FK joins (e.g., 99 - 26 = 73 queries for TPC-DS) since SPRINTER uses the same query plans with the base system for such queries.

To evaluate the characteristics of SPRINTER, we usually use synthetic queries which are generated by modifying the motivating query in Section 3 and evaluate them on the TPC-DS database. For datasets, we use the TPC-DS database from SF=100 (100 GB) to SF=400 (400 GB), which are the sizes widely used in the previous studies [25, 46].

Environments: We conduct all the experiments on a single machine equipped with two Intel Xeon 10-core CPUs, 512 GB main memory, and a single NVIDIA GTX 1080 Ti GPU of 11 GB device memory. The operating system used is CentOS 7.5.

Systems compared: The OLAP systems compared with SPRINTER are classified into two types: CPU-based systems (e.g., System-X) and co-processing systems (e.g., OmniSci). All the systems are based on the columnar storage layout. We note that every system is set up to use both main memory and GPU device memory (only for co-processing systems) as much as possible. Table 3 summarizes the features of the systems used in the experiments.

Table 3: Summarization of the systems compared.

<table>
<tr><th></th><th>System type</th><th>Query evaluation model</th><th>Plan shape</th></tr>
<tr><td>System-X</td><td rowspan="4">CPU</td><td rowspan="3">operator-at-a-time</td><td rowspan="3">left-deep</td></tr>
<tr><td>MonetDB</td></tr>
<tr><td>Actian Vector</td><td rowspan="2">pipelined</td></tr>
<tr><td>System-Y</td><td>binary bushy</td></tr>
<tr><td>System-Z</td><td rowspan="4">Co-processing (CPU+GPU)</td><td rowspan="3">operator-at-a-time</td><td rowspan="3">left-deep</td></tr>
<tr><td>CoGaDB</td></tr>
<tr><td>OmniSci</td><td>pipelined</td></tr>
<tr><td>SPRINTER</td><td>operator-at-a-time + pipelined</td><td>n-ary bushy</td></tr>
</table>

For the CPU-based systems, System-Y is one of the state-of-the-art commercialized OLAP database systems with a support of index-driven query execution and query optimization techniques such as bloom filter for hash join and cost-based query planner, which is similar to System-X. It however generates a query plan of binary bushy tree, which is one of major difference from System-X. In addition, System-Y processes a query plan in pipelined manner and supports code-generation for query execution. We use the latest releases of System-X and System-Y for experiments. We also compare SPRINTER with the well-known vectorized engines, MonetDB [9] (v11.31.13) and Actian Vector [51] (v5.1), where the former uses the operator-at-a-time model, but the latter the pipelined model for query evaluation. For the co-processing systems, System-Z is the state-of-the-art commercialized OLAP database system exploiting GPU, and we use its latest release for experiments. We also use two open-source co-processing systems for evaluation, OmniSci [33] (v4.5.0) and CoGaDB [10] (v0.4.2). We denote the version of SPRINTER using only CPU as SPRINTER(C) and the one of SPRINTER using both CPU and GPU as SPRINTER(G). Since SPRINTER executes an n-ary join operator in an operator-at-a-time manner although it executes each join subtree in a pipelined manner, we describe its query evaluation model as a combination of operator-at-a-time and pipelined.

7.2 Comparison of performance

Figure 11 shows the comparison results of SPRINTER(C) and SPRINTER(G) with the existing OLAP systems described in Section 7.1. In the figure, O.O.M. means a failure of query evaluation due to out-of-memory. T.O. means a timeout (exceed 1,800 seconds). M.E. means a failure of query evaluation due to main-memory related errors such as segmentation fault and bad allocation. Y-axis is log-scale in the figure. We use a dataset of SF=100 since a bigger dataset incurs a lot of O.O.M. in many systems. For each system and each query, we run the query five times to warm up the system and report the best elapsed time.

Comparison with CPU-based systems: Figure 11(a) shows the comparison results with the CPU-based systems. We first note that only SPRINTER executes all eleven queries successfully, but other systems fail in at least one query due to O.O.M. from a large amount of intermediate results or T.O. from a huge amount of probe cost, explained in Section 3. In addition, both SPRINTER(C) and SPRINTER(G) outperforms all the systems compared for the most of queries tested, which is due to their different query planning and different join processing. For the queries commonly executed by all the systems compared, i.e., Q37, Q64 and Q95, SPRINTER achieves the best performance among the systems. In particular, for Q64, SPRINTER(G) improves the performance compared with System-Y, System-X, MonetDB and Actian Vector by 6.6, 7.4, 20.1, and 5.1 times, respectively.

The performance gap between SPRINTER(C) and SPRINTER(G) is not large since the data size is relatively small (SF=100). We note that the current SPRINTER does not use advanced query optimization techniques that System-X and System-Y use, since its base system, OmniSci, does not support them yet. Thus, the performance of SPRINTER can be further improved by applying the optimization techniques to OmniSci or SPRINTER.

Comparison with co-processing systems: Figure 11(b) shows the comparison results with the co-processing systems. SPRINTER outperforms all the co-processing systems

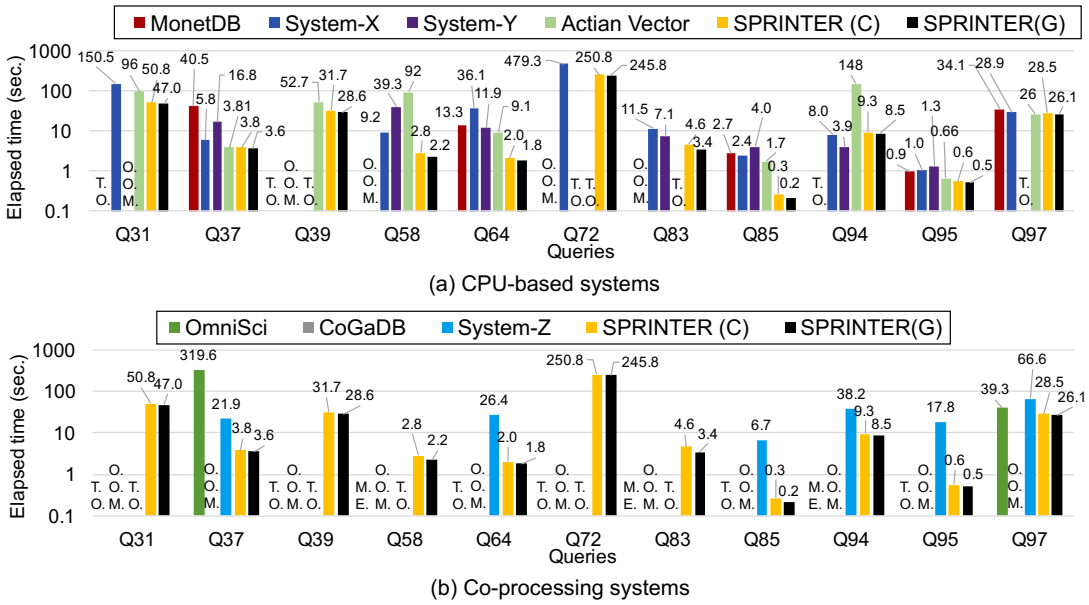


Figure 11: Performance comparison using TPC-DS benchmark queries (SF=100).

compared for all queries tested without any failure. Comparing with CPU-based systems, the existing co-processing systems have much more failures and usually worse performance when processing the same query. It is because the co-processing systems usually do not use advanced query optimization techniques and are not mature enough to process complex queries efficiently in terms of exploiting GPU. For example, OmniSci outperforms System-X in Figure 4 since the motivating query has high f_i values (i.e., closed to 1) for its relations. However, the TPC-DS queries in Figure 11 usually have low f_i values (i.e., closed to 0), and so, System-X equipped with a support of index-driven query execution and query optimization techniques outperforms OmniSci.

OmniSci can execute only two queries, Q37 and Q97, although SPRINTER using OmniSci as the base system can execute all eleven queries. OmniSci and CoGaDB first try to execute a given query using GPU. However, if the attempt fails, OmniSci and CoGaDB execute the query using CPU and main memory. This two-step approach increases the elapsed times when the data required to execute the query cannot fit in GPU memory. In addition, OmniSci executes the query in an pipelined manner, while CoGaDB in an operator-at-a-time manner. Thus, CoGaDB tends to fail due to O.O.M. from large intermediate results and lack of query optimization techniques. For OmniSci, even when executing the query using main memory, it tends to estimate the amount of memory required incorrectly if the query becomes more complex, and so, the left-deep join tree becomes deeper. Thus, OmniSci tends to fail due to M.E. from incorrect memory allocation or T.O. from a huge amount of probe cost for FK-FK joins between fact tables. In contrast, although SPRINTER is based on OmniSci, it has no failure and significantly improves the query performance. The left-deep join subtrees in a query plan generated by SPRINTER is much smaller than the join tree by OmniSci, and at the same time, there is almost no fact table used for building a hash table, as shown in Section 3. Such small and simple join subtrees can be sufficiently evaluated by OmniSci without failure of M.E. In addition, the probe cost of SPRINTER is much smaller than that of OmniSci due to n-ary join processing, and so, there is no failure of T.O. For Q37 and Q97, SPRINTER(G) improves the performance compared with OmniSci by 88.8 and 1.5 times, respectively.

Different from OmniSci and CoGaDB, System-Z uses both main memory and GPU memory for query processing from the beginning and supports more query optimization techniques. Thus, it can execute six queries among eleven ones without O.O.M. However, since it executes a query in an operator-at-a-time manner and still does not support many query optimization techniques of the CPU-based systems, it tends to fail in many queries due to T.O. or show worse performance than SPRINTER as well as the CPU-based systems.

7.3 Characteristics of SPRINTER

Validation of cost model: We validate the cost model in Section 6 empirically. In Algorithm 2, we determine whether to use a conventional plan of left-deep join tree (denoted as left-deep) or a more general plan having n-ary join operations (denoted as n-ary bushy) based on the cost model. Thus, it is important that the cost model coincides with the actual performance as much as possible.

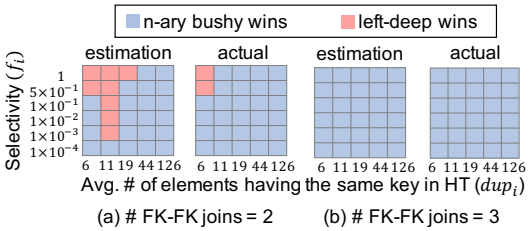


Figure 12: Validation of the cost model.

Figure 12 shows the estimation based on the cost model and the actual performance result in a query space. For the query space, we modify the motivating query in Figure 3 in terms of (1) the number of FK-FK joins, (2) filtering selectivity (i.e., f_i), and (3) the average number of elements having the same key in a hash table (i.e., dup_i). The number of FK-FK joins in the motivating query is two (Figure 12(a)), and the number becomes three by connecting (joining) a new fact table CR with CS and C via the $cust$ column (Figure 12(b)). We vary the selectivity f_i on Y-axis between 0.0001 and 1.0 by adjusting f_i for C. We also vary dup_i on X-axis by replacing C with different dimension tables such as CD, CA, I, and HD via different join columns. For example, if we use CD instead of C and join CD with fact tables via the $cdemo$ column, $\{dup_i\}$ of fact tables approximately become six ($dup_i = 6$). In this case, we vary the selectivity f_i on Y-axis by adjusting f_i for CD. Similarly, we set $dup_i = 11$ using C, $dup_i = 19$ using CA, $dup_i = 44$ using I, and $dup_i = 126$ using HD. Figure 12 shows the results, where red color cells indicate a left-deep plan achieves better performance than an n-ary bushy plan, and blue color cells indicate the opposite situation. In the figure, n-ary bushy plans are better than left-deep plans in terms of both estimation and actual result in most of the cells in the query space. We note that the estimation usually coincides with the actual result (90% match).

Validation of the heuristic algorithm for global variable order: We validate the effectiveness of the heuristic algorithm in Algorithm 3 using three TPC-DS queries, Q17, Q25 and Q29 (SF=100). In the left table in Figure 13, E_w and C_w are the statistics used in Algorithm 3, and $Rank$ means the rank when sorting the variables by the descending order of $(|E_w|, C_w)$. The core subgraphs of all three queries tested, Q17, Q25 and Q29, consist of three join variables, C, I, and T.

There exist 3! = 6 orders for the three variables, and the right figure in Figure 13 shows the query processing time of six different orders for the three queries. According to our heuristic algorithm, $I \prec C \prec T$ and $C \prec I \prec T$ should show the best performance, while $T \prec I \prec C$ and $T \prec C \prec I$ the worst performance. Such predicted results coincide with the actual results in the figure. That means our heuristic algorithm for selecting a good global variable order is simple, but effective.

w	$\lvert E_w \rvert$	C_w	$Rank$
C	71.3	2	1
I	71.3	2	1
T	42.3	1	3

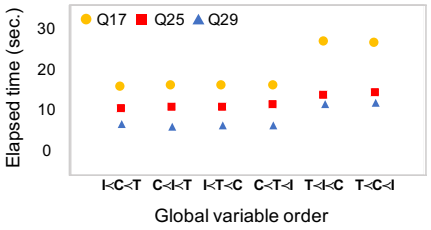


Figure 13: Results of different global variable orders.

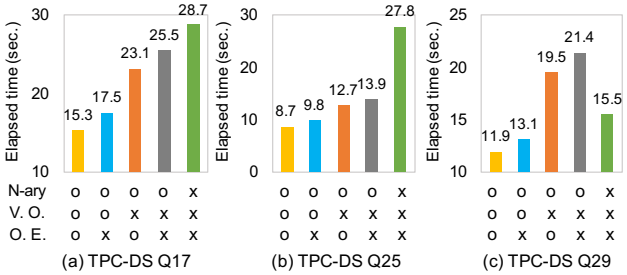


Figure 14: Results of performance breakdown.

Performance breakdown: Figure 14 shows the results of micro evaluation of SPRINTER using three TPC-DS queries Q17, Q25, and Q29. There are five possible configurations based on three major techniques proposed: n-ary join processing (shortly, N-ary), selecting a global variable order in Algorithm 3 (shortly, V.O.) and overlapping T_{eval} with T_{sort} (shortly, O.E.). In Figure 14, no V.O. option means selecting a global variable order randomly without using Algorithm 3. No O.E. option means evaluating join subtrees and then sorting their results as in Figure 7. In the figure, the fifth bar, i.e., no N-ary, no V.O., and no O.E., shows the performance of the base system, i.e., OmniSci.

In the figure, the N-ary option improves the query performance in Q17 and Q25. The reason why the fourth bar (i.e., N-ary, but no V.O and no O.E) shows worse performance than the fifth bar (i.e., base system) in Q29 is due to a poor global variable order randomly selected. The result means that not only n-ary query processing but also selecting a good global variable order is important in terms of performance. Between the second (i.e., V.O, but no O.E) and third (i.e., O.E, but no V.O) bars, the second bar improves the performance more significantly than the third bar, against the fourth bar (i.e., no O.E and no V.O). The result means selecting a good global variable order is more important than overlapping T_{eval} with T_{sort}. Overall, only N-ary and V.O options without O.E could outperform the base system for all queries tested. The O.E option just further improves the performance.

Sorting performance: We validate the strategy of selecting join algorithms empirically. Figure 15 shows the performance evaluation of sorting algorithms in Table 1. When sorting a single column in Figure 15(a), CPU-radix is faster than CPU-merge, and GPU-radix is faster than GPU-merge,

as shown in Figure 9. When sorting multiple columns in Figure 15(b), GPU-merge is only slightly faster than CPU-merge, as explained in Section 5.2. When sorting a single-column relation larger than GPU memory in Figure 15(c), heterogeneous sorting is much faster than CPU-radix. Overall, the experimental result verifies the strategy in Figure 9.

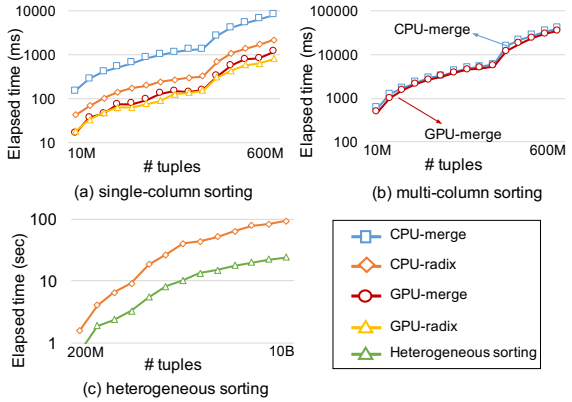


Figure 15: Validation of strategy for sorting.

8 RELATED WORK

Multi-way join processing: Recently, the worst-case optimal (WCO) join algorithms are proposed to evaluate multi-way join occurred in a graph pattern query [31, 44]. They provide theoretically a tighter bound in terms of the worst-case size of multi-way join results by performing a kind of set-intersection among multiple sorted relations. Some of them [1, 31, 44] require to build a large-scale index for a global variable order before join. They are useful for a graph pattern query having a few global variable orders on a small number of relations. However, it is challenging to apply them to ad-hoc OLAP queries since each OLAP query has many possible global variable orders on a large number of relations.

A few studies [2, 13, 49] discuss multi-way join processing on distributed environments. Their optimization goal is minimizing the communication cost by repartitioning multiple join relations together, instead of shuffling a pair of input relations of a join operation at a time.

Query planning for WCO join algorithms: DunceCap [36, 43] focuses on generating a hypertree query plan for WCO join algorithms. It exploits the connection between the minimum hypertree width of an input query and the size of query result. Here, the width means how far a given query is from acyclic query [32] or the degree of cyclicity of the corresponding hypergraph [38]. A plan tree having the minimum width guarantees the minimum worst-case output size, and thus, its query performance is proportional to the width [4], which is called AGM-bound.

DunceCap is useful for cyclic queries (e.g., triangle query) on graph dataset since it exploits a hypergraph capturing the cyclicity existing in a query. It is however not applicable for OLAP queries since they are usually acyclic. For instance, the motivating query in Section 3 has no cycle in the corresponding hypergraph since $C1$, $C2$, and $C3$ are regarded as different relations in hypergraph. There is no cyclic query in the TPC-DS benchmark. For such acyclic queries, the concept of hypertree width cannot provide an insight to determine the best query plan. Moreover, filter predicates on large fact tables may significantly reduce the size of join result and thus can become far from the worst-case bound.

Co-processing approach for OLAP query: Recently, the co-processing approach for OLAP query processing has been actively studied in database community [14, 15, 17, 35, 42]. They can be categorized into two groups: (1) accelerating database kernels [15, 42] and (2) end-to-end query evaluation engines [14, 17, 35]. The major issue of the co-processing approach is the GPU memory limitation, which may incur high I/O overhead due to frequent data transfer between main memory and device memory [14] or cause a failure at runtime due to out of memory.

9 CONCLUSIONS

In this paper, we have proposed a fast n-ary join query processing method for complex OLAP queries having FK-FK joins. It generates a query plan containing n-ary join operators, if it is better than the conventional left-deep binary join tree based on our cost model. The plan can significantly reduce the probe cost by placing a FK-FK join on fact tables into an n-ary join operator. We also have proposed an efficient n-ary join processing method which is based on the TJ algorithm and heuristic algorithm selecting a good global variable order. We have implemented the prototype system SPRINTER that our proposed methods are integrated into the open-source in-memory OLAP system, OmniSci, across all relevant layers and modules. Through experiments using the TPC-DS benchmark, we have shown that SPRINTER outperforms the state-of-the-art OLAP systems even without using GPU sorting, although its base system OmniSci achieves the second worst performance among them.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. 2018R1A5A1060031), Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2017R1E1A1A01077630), and Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2019-0-01267, GPU-based Ultrafast Multi-type Graph Database Engine SW).

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2016. Emptyheaded: A relational engine for graph processing. In *SIGMOD*.

[2] Foto N Afrati and Jeffrey D Ullman. 2011. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 23, 9 (2011), 1282–1298.

[3] Rafi Ahmed, Rajkumar Sen, Meikel Poess, and Sunil Chakkappen. 2014. Of snowstorms and bushy trees. In *VLDB*.

[4] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size bounds and query plans for relational joins. In *FoCS*.

[5] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. [n.d.]. Multi-core, main-memory joins: Sort vs. hash revisited.

[6] Ronald Barber, Guy Lohman, Vijayshankar Raman, Richard Sidle, Sam Lightstone, and Berni Schiefer. 2015. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *ICDE*.

[7] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier.

[8] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*.

[9] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[10] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *SIGMOD*.

[11] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. In *VLDB*.

[12] Zhimin Chen, Vivek Narasayya, and Surajit Chaudhuri. 2014. Fast foreign-key detection in Microsoft SQL server PowerPivot for Excel. In *PVLDB*.

[13] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*.

[14] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *SIGMOD*.

[15] Michael Gowanlock and Ben Karsin. 2018. Sorting Large Datasets with Heterogeneous CPU/GPU Architectures. In *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[16] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: a main memory hybrid storage engine. In *VLDB*.

[17] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. In *VLDB*.

[18] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*.

[19] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. In *VLDB*.

[20] Hyerin Kim, NaNa Kang, Kang-Wook Chon, Seonho Kim, NaHye Lee, JaeHyung Koo, and Min-Soo Kim. 2015. MRPrimer: a MapReduce-based method for the thorough design of valid and ranked primers for PCR. *Nucleic acids research* 43, 20 (2015), e130–e130.

[21] Kinetica. 2019. Kinetica: Active Analytics Platform for the Extreme Data Economy. https://www.kinetica.com.

[22] Arun Kumar, Jeffrey Naughton, Jignesh M Patel, and Xiaojin Zhu. 2016. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD*.

[23] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *ICDE*.

[24] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. In *VLDB*.

[25] Per-Åke Larson, Cipri Clinciu, Eric N Hanson, Artem Oks, Susan L Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL server column store indexes. In *SIGMOD*.

[26] Thomas J Lee, Yannick Pouliot, Valerie Wagner, Priyanka Gupta, David WJ Stringer-Calvert, Jessica D Tenenbaum, and Peter D Karp. 2006. BioWarehouse: a bioinformatics database warehouse toolkit. *BMC bioinformatics* 7, 1 (2006), 170.

[27] Feilong Liu and Spyros Blanas. 2015. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*.

[28] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. In *VLDB*.

[29] D Merrill. 2015. CUB v1. 5.3: CUDA Unbound, a library of warp-wide, blockwide, and device-wide GPU parallel primitives. *NVIDIA Research* (2015).

[30] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS.

[31] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms. In *PODS*.

[32] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *ACM SIGMOD Rec.* 42, 4 (2014), 5–16.

[33] OmniSci. 2019. A github repository of OmniSci database. https://github.com/omnisci/omniscidb.

[34] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A data platform based on the scaling-up approach. In *VLDB*.

[35] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *SIGMOD*.

[36] Adam Perelman and Christopher Ré. 2015. DunceCap: Compiling worst-case optimal query plans. In *SIGMOD*.

[37] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[38] Francesco Scarcello. 2005. Query answering exploiting structural properties. *ACM SIGMOD Rec.* Vol. 34, No. 3, pp. 91–99 (2005).

[39] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*.

[40] Anil Shanbhag and S Sudarshan. 2014. Optimizing join enumeration in transformation-based query optimizers. In *VLDB*.

[41] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. 2013. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. In *VLDB*.

[42] Elias Stehle and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on gpus. In *SIGMOD*.

[43] Susan Tu and Christopher Ré. 2015. Duncecap: Query plans using generalized hypertree decompositions. In *SIGMOD*.

[44] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*.

[45] Andreas Weininger. 2002. Efficient execution of joins in a star schema. In *SIGMOD*.
[46] Petrie Wong, Zhian He, and Eric Lo. 2013. Parallel analytics as a service. In *SIGMOD*.
[47] Konstantinos Xirogiannopoulos and Amol Deshpande. 2019. Memory-Efficient Group-by Aggregates over Multi-Way Joins. In *arXiv preprint arXiv:1906.05745*.
[48] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. 2010. On Multi-Column Foreign Key Discovery. In *PVLDB*.
[49] Xiaofei Zhang, Lei Chen, and Min Wang. 2012. Efficient multi-way theta-join processing using mapreduce. In *VLDB*.
[50] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M Patel. 2017. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. In *VLDB*.
[51] Marcin Zukowski, Mark Van de Wiel, and Peter A Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS.. In *ICDE*.