Fast Distributed Complex Join Processing

Hao Zhang*, Miao Qiao†, Jeffrey Xu Yu*, Hong Cheng*
*The Chinese University of Hong Kong
{hzhang, yu, hcheng}@se.cuhk.edu.hk
†The University of Auckland
{miao.qiao}@auckland.ac.nz

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00205

***Abstract*—Big data analytics often requires processing complex join queries in parallel in distributed systems such as Hadoop, Spark, Flink. The previous works consider that the main bottleneck of processing complex join queries is the communication cost incurred by shuffling of intermediate results, and propose a way to cut down such shuffling cost to zero by a one-round multi-way join algorithm. The one-round multi-way join algorithm is built on a one-round communication optimal algorithm for data shuffling over servers and a worst-case optimal computation algorithm for sequential join evaluation on each server. The previous works focus on optimizing the communication bottleneck, while neglecting the fact that the query could be computationally intensive. With the communication cost being well optimized, the computation cost may become a bottleneck. To reduce the computation bottleneck, a way is to trade computation with communication via pre-computing some partial results, but it can make communication or pre-computing becomes the bottleneck. With one of the three costs being considered at a time, the combined lowest cost may not be achieved. Thus the question left unanswered is how much should be traded such that the combined cost of computation, communication, and pre-computing is minimal.**

In this work, we study the problem of co-optimize communication, pre-computing, and computation cost in one-round multi-way join evaluation. We propose a multi-way join approach ADJ (Adaptive Distributed Join) for complex join which finds one optimal query plan to process by exploring cost-effective partial results in terms of the trade-off between pre-computing, communication, and computation.We analyze the input relations for a given join query and find one optimal over a set of query plans in some specific form, with high-quality cost estimation by sampling. Our extensive experiments confirm that ADJ outperforms the existing multi-way join methods by up to orders of magnitude.

I. Introduction

Join query processing is one of the important issues in query processing, and join queries over relations based on the equality on the common attributes are commonly used in many real applications. Large-scale data analytics engines, such as Spark [1], Flink [2], etc, use massive parallelism in order to enable efficient query processing on large data sets. Recently, data analytics engines are used beyond traditional OLAP queries that usually consist of star-joins with aggregates. Such new kind of workloads [3] contain complex FK-FK joins, where multiple large tables are joined, or where the query graph has cycles, and has seen many applications, finding triangle and other complex patterns in graphs [4], analyzing local topology around each node in graphs, which serves as powerful discriminative features for statistical relational learning tasks for link prediction, relational classification and recommendation [5].

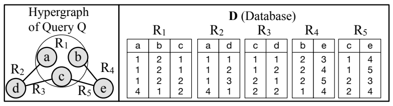
However, data analytics engines process complex joins by decomposing them into smaller join queries, and combining intermediate relations in multiple rounds, which suffers from expensive shuffling of intermediate results. To address such inefficiency, one-round multi-way join HCubeJ is proposed [6], which requires no shuffling after the initial data exchange. The one-round multi-way join processes a join query in two stages, namely, data shuffling and join processing. In the data shuffling stage, HCubeJ shuffles the input relations by an optimal one-round data shuffling method HCube [7], [8]. In the join processing stage, HCubeJ uses an in-memory sequential algorithm Leapfrog [9] at each server to join the data received.

However, the one-round multi-way join algorithm has a deficiency, since it puts communication cost at a higher priority to reduce than the computation cost by considering the communication cost as the dominating factor, which is not always true. The main reason is that the computation of complex multi-way join can be inherently difficult. We tested the communication-first strategy of HCubeJ in our prototype system using optimized HCube for data shuffling and Leapfrog for join processing. Overall, the performance may not be the best as expected.

In this paper, we study how to reduce the total cost by introducing pre-computed partial results with communication, computation, and pre-computing cost being considered at the same time. This problem is challenging since we may cause one cost larger when we reduce the other cost, and the search space of potential pre-computed partial results is huge. The main contributions are given as follows.

- We identify the performance issue of processing Q using HCubeJ due to the unbalance between computation and communication cost, and propose a simple mechanism to trade computation cost with communication and pre-computing cost such that the total cost is reduced for a multi-way join query Q.
- We study how to effectively find cost-effective pre-computed partial results from overwhelmingly large search space, and join them and the rest of relations in an optimal order.
- We implement a prototype system ADJ and conducted extensive performance studies, and confirm that our approach can be orders of magnitude faster over the previous approaches in terms of the total cost.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00205



D (Database)

R1			R2		R3		R4		R5	
a	b	c	a	d	c	d	b	e	c	e
1	2	1	1	1	1	1	2	3	1	4
1	2	1	1	2	1	2	2	4	1	5
1	2	2	1	3	2	1	2	5	2	3
4	1	2	4	1	2	2	1	4	2	4

Fig. 1: The hypergraph of query Q (Eq (2)), and database D

The paper is organized as follows. We give the preliminary of this work, and discuss HCube, Leapfrog algorithms, and the main issues we study in this work in Section II. We outline our approach in Section III. In section IV, we discuss the related work, and in Section V we report our experimental studies. We conclude our work in Section VI.

II. Preliminaries

A database D is a collection of relations. Here, a relation R with schema $\{A_1, A_2, \cdots, A_n\}$ is a set of tuples, $(a_1, a_2, \cdots, a_n)$, where a_i is a value taken from the domain of an attribute A_i, denoted as $\mathsf{dom}(A_i)$, for $1 \leq i \leq n$.Below, we use $\mathsf{attrs}(R)$ to denote the schema (the set of attributes) of R. A relation R with the schema of $\mathsf{attrs}(R)$ is a subset of the Cartesian product of $\mathsf{dom}(A_1) \times \mathsf{dom}(A_2) \times \cdots \times \mathsf{dom}(A_n)$ for $A_i \in \mathsf{attrs}(R)$. We focus on natural join queries (or simply join queries). A natural join query, Q, is defined over a set of m relations, $\mathcal{R} = \{R_1, R_2, \cdots, R_m\}$, for $m \geq 2$, in the form of

$$Q(\mathsf{attrs}(Q)) \coloneq R_1(\mathsf{attrs}(R_1)) \bowtie \cdots \bowtie R_m(\mathsf{attrs}(R_m)). \tag{1}$$

Here, the schema of Q, denoted as $\mathsf{attrs}(Q)$, is the union of the schemas in $\mathcal{R}$ such as $\mathsf{attrs}(Q) = \cup_{R_i \in \mathcal{R}} \mathsf{attrs}(R_i)$. For simplicity, we assume there is an arbitrary order among the attributes of Q, denoted as ord, and A_i denotes the i-th attribute in ord. We also use $\mathcal{R}(Q)$ to denote the set of relations in Q. A resulting tuple of Q is a tuple, τ, if there exists a non-empty tuple t_i in R_i, for every $R_i \in \mathcal{R}$, such that the projection of τ on $\mathsf{attrs}(R_i)$ is equal to t_i (i.e., $\Pi_{\mathsf{attrs}(R_i)}\tau = t_i$). The result of a join Q is a relation that contains all such resulting tuples. A join query Q over m relations $\mathcal{R}$ can be represented as a hypergraph $H = (V, E)$, where V and E are the set of hypernodes and the set of hyperedges, respectively, for V to represent the attributes of $\mathsf{attrs}(Q)$ and for E to represent the m schemas. As an example, consider the following join query Q over five relations,

$$Q(a, b, c, d, e) \coloneq R_1(a, b, c) \bowtie R_2(a, d) \bowtie R_3(c, d) \bowtie R_4(b, e) \bowtie R_5(c, e) \tag{2}$$

Its hypergraph representation H is shown in Fig. 1 together with the 5 relations. Here, $V = \mathsf{attrs}(Q) = \{a, b, c, d, e\}$, and $E = \{e_1, e_2, e_3, e_4, e_5\}$ for $e_1 = \mathsf{attrs}(R_1)$, $e_2 = \mathsf{attrs}(R_2)$, $e_3 = \mathsf{attrs}(R_3)$, $e_4 = \mathsf{attrs}(R_4)$, and $e_5 = \mathsf{attrs}(R_5)$. In the following, we also use $V(H)$ and $E(H)$ to denote the set of hypernodes and the set of hyperedges for a hypergraph H.

A. Leapfrog *and* HCube *Join Algorithms*

We discuss HCubeJ [6] to compute join queries in a distributed system over a cluster of servers, where the database D is maintained at the servers disjointly. HCubeJ is built on two algorithms, namely, HCube [7], [8] and Leapfrog [9], where HCube is a one-round communication optimal shuffling method that shuffles data to every server in the cluster, and Leapfrog is an worst-case optimal fast in-memory sequential multi-way join algorithm to process the join query at each server over the data shuffled to it.

LeapFrog Join [9] is one of the state-of-the-art sequential join algorithms for a join query Q over m relations, $\mathcal{R} = \{R_1, R_2, \cdots, R_m\}$ (Eq. (1)). Leapfrog is designed to evaluate Q based on the attribute order ord using iterators. Let t^i be an i-tuple that has i attributes of A_1 to A_i. The Leapfrog algorithm is to find the t^{i+1} tuples by joining the tuple t^i with an additional A_{i+1} value recursively until it finds all n attribute values for Q.

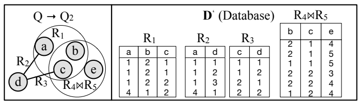
The Leapfrog algorithm is illustrated in Algorithm 1 for a given-input i-tuple t^i. The initial call of Leapfrog is with an empty input tuple t^0. Below, we explain the algorithm assuming that the input is a non-empty i-tuple, t^i, for $i > 1$. Let $\mathcal{R}_{i+1}$ be the set of relations R in Q if R contains the (i+1)-th attribute A_{i+1} in order such as $\mathcal{R}_{i+1} = \{R \mid A_{i+1} \in R$ and R is a relation appearing in $Q\}$ (line 4). To find all A_{i+1} values that can join the input i-tuple t^i, denoted as $\mathsf{val}(t^i \rightarrow A_{i+1})$, (line 5), it is done as follows. Here, for simplicity and without loss of generality, we assume $\mathcal{R}_{i+1} = \{R, R'\}$. First, for R, let As be all the attributes that appear in both $\mathsf{attrs}(R)$ and $\mathsf{attrs}(t^i)$, it projects the A_{i+1} attribute value from every tuple $t \in R$ that can join with the i-tuple on all the attributes As. Let T_{i+1} be a relation containing all A_{i+1} values found. Second, for R', repeat the same, and let T'_{i+1} be a relation containing all A_{i+1} values found. The result of $\mathsf{val}(t^i \rightarrow A_{i+1})$ is the intersection of T_{i+1} and T'_{i+1}. At line 6-7, for every value, v, in $\mathsf{val}(t^i \rightarrow A_{i+1})$, it calls Leapfrog recursively with an (i+1)-tuple, $t^{i+1} = t^i \| v$, by concatenating t^i and v. At line 1-2, If $i = |\mathsf{attrs}(Q)|$, the tuple t^i is emitted through the iterator. It is important to note that the main cost of Leapfrog is the cost of the intersections.

HCube Shuffle [7], [8] is one of the state-of-the-art communication methods to evaluate a join query Q in a distributed system by shuffling data in one-round. The main idea is to divide the output of a join query Q into hypercubes with coordinates, and assign one or more hypercubes to one of the N^* servers to process by shuffling the tuples, whose hash values partially matches the coordinate of the given hypercube, to the server. Given a vector $p = (p_1, p_2, \cdots, p_n)$, where p_i is the number of partitions for the attribute A_i under ord, and $n = |\mathsf{attrs}(Q)|$, hypercubes of $P = p_1 \times \cdots \times p_n$ dimension are constructed. It is worth mentioning that P can be larger than N^*. Here, a hypercube is identified by an coordinate of $C = (c_1, ..., c_n)$ of $[p_1] \times \cdots \times [p_n]$, where $[l]$ represents the range from 0 to $l - 1$. Each machine can be assigned one or more hypercubes. HCube distribute tuples of each relation to machines via shuffling by hashing. For example, let's assume $p = (1, 2, 2, 1, 1)$, which specifies four hypercubes with coordinates $(0, 0, 0, 0, 0)$, $(0, 1, 0, 0, 0)$, $(0, 0, 1, 0, 0)$, $(0, 1, 1, 0, 0)$. The first tuple, (1, 2,1), that appears at the top in the relation $R_1(a, b, c)$, will be shuffling to the servers that are assigned hypercube with coordinate $(0, 0, 0, \star, \star)$, since $h_a(1) = 0$, $h_b(2) = 0$, $h_c(2) = 0$, where h_{A_i} means the hash function h_i for attribute A_i, and $\star$ means any integer.

After HCube completes its shuffling by hashing, each server

Algorithm 1: Leapfrog(t^i, Q)

```
Input: an $i$-tuple $t^i$, the query $Q$
Output: tuples of $Q$ emitted through iterators
if $i = |\mathsf{attrs}(Q)|$ then
    Emit($t^i$);
else
    let $\mathcal{R}_{i+1}$ be the set of relations $R$ in $Q$ if $R$ contains the ($i$+1)-th attribute $A_{i+1}$ in order;
    find all $A_{i+1}$ values that can join the input tuple $t^i$, denoted as $\mathsf{val}(t^i \rightarrow A_{i+1})$;
    for each attribute value $v$ in $\mathsf{val}(t^i \rightarrow A_{i+1})$ do
        Leapfrog($t^i \| v$, $Q$);
```



R1

a	b	c
1	2	1
1	2	1
1	2	2
4	1	2

R2

a	d
1	1
1	2
1	3
4	1

R3

c	d
1	1
1	2
2	1
2	2

R4⋈R5

b	c	e
2	1	4
2	1	5
1	1	5
2	2	3
2	2	4
1	2	4

Fig. 2: A query candidate Q_i which gets the same result of Q in Fig. 1 by replacing R_4 and R_5 with $R_4 \bowtie R_5$

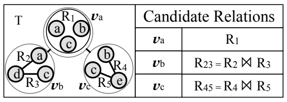
can compute the data assigned to it using an in-memory multi-way join algorithm independently, i.e., Leapfrog, and the union of the results by the servers is the answer for the join query Q.

III. Adaptive Multi-way Join

In this paper, we study how to minimize the total cost of both communication cost and computation cost together with some additional pre-computing cost. To achieve it, we need a mechanism that allows us to balance the total costs with the condition that the mechanism is cost-effective to achieve the goal of minimization of the total costs.

We discuss our main idea using an example. Consider a join query as $Q = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5$ (refer to Eq. (2) for the details) over the database D shown in Fig. 1. Let it be executed by HCubeJ, where HCube shuffles the database D, and Leapfrog is deployed on each server to compute the data shuffled to it. Assume, the system finds out that the time spent on HCube for shuffling tuples is relatively small, while a considerable amount of time is spent on Leapfrog on each server. Furthermore, suppose the system finds out that the computation cost of Leapfrog can be reduced for the same query Q if $R_4 \bowtie R_5$ has already been joined as one relation instead. In other words, let $Q_2 = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_{45}$ where $R_{45} = R_4 \bowtie R_5$, instead of executing Q directly, it is to pre-compute R_{45} first, then execute Q_2. Though it would be more expensive to do the pre-computing and shuffle the tuples of $\mathcal{R}(Q_2)$, which is shown in Fig. 2 (18 integers in R_{45}, 16 integers in R_4 and R_5 in total), it is still preferable to execute the new query Q_2 instead of Q to trade the communication cost and pre-computing cost for computation cost, which is the bottleneck. The message by this example is: there is a way to reduce the computation cost at the expense of increased communication cost with some pre-computing cost, and it is possible to minimize the total cost by balancing the computation cost, communication cost, and pre-computing cost.

We give our problem statement below based on the idea presented in the example. Consider a join query $Q = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m$ (refer to Eq. (1)). Let $\mathcal{Q}$ be a collection of query candidates such as $\mathcal{Q} = \{Q_1, Q_2, \cdots Q_{|\mathcal{Q}|}\}$, where $Q_i = R'_1 \bowtie R'_2 \bowtie \cdots \bowtie R'_l$. Here, Q_i is equivalent to Q such that Q_i and Q return same results, $\mathsf{attrs}(Q_i) = \mathsf{attrs}(Q)$, $l \leq m$, and a relation R'_j in $\mathcal{R}(Q_i)$ is either a relation R_k in $\mathcal{R}(Q)$ or a relation by joining some relations in $\mathcal{R}(Q)$. Let a query plan be a pair (Q_i, ord) that consists of a query candidate $Q_i \in \mathcal{Q}$, which specifies how to pre-compute relations, and an attribute order ord for attributes of Q_i, which specifies how to join the relations of new query Q_i using Leapfrog. The problem is to find a query plan such that the total cost for communication, pre-computing, and computation is minimized. This problem is challenging due to the huge search space. For example, there exists 2^m possible combinations of joins to construct a single relation R'_j in total, where m is the number of relations in Q, and $n!$ possibilities to order the attributes of Q_i.



Candidate Relations	
v_a	R1
v_b	R23 = R2 ⋈ R3
v_c	R45 = R4 ⋈ R5

Fig. 3: Hypertree $\mathcal{T}$ and candidate relations

In this paper, we propose a prototype system (ADJ) that explores cost-effective query plans from a reduced search space. The workflow of our system (ADJ) is as follows. First, we shrink the search space according to an optimal hypertree $\mathcal{T}$ constructed for query Q such that search space of candidate relations and attribute order ord are reduced based on $\mathcal{T}$. Then, we explore cost-effective query plans derived from the $\mathcal{T}$ by considering the cost-effectiveness of trading the computation with communication and pre-computing of each pre-computed candidate relations with the cost model. The cardinality estimation is done via a distributed sampler. Given an optimal query plan (Q_i, ord), first, for each relation $R'_j \in Q_i$ that needs to be joined, we pre-compute and store it. After every R'_j is computed, we execute $Q_i = R'_1 \bowtie R'_2 \bowtie \cdots \bowtie R'_l$.

Next, in Sec III-A, we explain how to reduce the search space. Then in Sec III-B we show how to explore cost-effective query plans based on hypertree $\mathcal{T}$. The cost model and how to estimate the cardinality via distributed sampling is discussed in the full paper [10].

A. The Reduced Search Space

To reduce the search space for selecting an optimal query plan from the collection of query candidates $Q_i \in \mathcal{Q}$ and possible attribute orders, we only consider a limited number of joins such that a join (e.g., $R_4 \bowtie R_5$) is as small as possible and could lower join cost of Q. More specifically, we find query candidates that are almost acyclic queries and can be easily transformed from Q. Our intuition is that the computation cost of evaluating an acyclic query is usually significantly smaller than that of evaluating an equivalent cyclic query. Thus an almost acyclic query Q_i could be easier to evaluate than Q.

This is done as follows. First, we represent a given join query Q using its hypergraph representation, $H = (V, E)$. Second, for the hypergraph H, we find a hypertree representation, $\mathcal{T} = (V, E)$, where $V(\mathcal{T})$ is a set of hypernodes and

$E(\mathcal{T})$ is a set of hyperedges. Recall that, in the hypergraph H, a hypernode represents an attribute, and a hyperedge represents a relation schema. The corresponding hypertree $\mathcal{T}$ represents the same information. (1) A hypernode in $V(\mathcal{T})$ represents a subset of hyperedges (e.g., relation schemas) in $E(H)$, and it also corresponds to a potential pre-computed relation, which can be computed by joining the corresponding relations of the relation schemas it contains. (2) Hyperedges $E(\mathcal{T})$ of $\mathcal{T}$ is constructed such that the hypernodes in $\mathcal{T}$ that contains a common attribute A, must be connected in the hypertree $\mathcal{T}$.

There are many possible hypertrees for a given hypergraph, we use the one whose maximal size of the pre-computed relation of each hypernode is minimal. This requirement ensures that for any subset of hypernodes $V'(\mathcal{T}) \subseteq V(\mathcal{T})$ to be pre-computed, the resulting relations do not incur too much pre-computing and communication overhead in later join query Q_i. We find such a hypertree $\mathcal{T}$ using GHD (Generalized HyperTree Decomposition) [11]. To bound the maximum size of the pre-computed relation of each hypernode in the worst-case sense, in theory, we can select the one with minimal fhw (fractional hypertree width) [12]. Such a hypertree $\mathcal{T}$ found by GHD satisfies that $\max_{v \in V(\mathcal{T})} |R_{max}|^{\mathsf{fhw}}$ is the lowest among all hypertrees, where $|R_{max}| = \max_{R \in \mathcal{R}(Q)} |R|$. In other words, the size of every pre-computed relation of each hypernode is upper bounded by $|R_{max}|^{\mathsf{fhw}}$ for the chosen $\mathcal{T}$ and it is the lowest one among all possible $\mathcal{T}$.

The hypertree $\mathcal{T}$ found from the hypergraph representation for a given join query, $Q = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_m$, has two implications regarding the reduced search space to find the optimal $Q_i = R'_1 \bowtie R'_2 \bowtie \cdots \bowtie R'_l$, namely, the number of joins and the attribute order.

Reducing Numbers of Candidate Relations. Instead of finding any possible joins to replace a single relation R'_j in Q_i, we only consider the joins represented as hypernodes in the hypertree $\mathcal{T}$. By pre-computing such joins, query Q_i is almost acyclic. Consider the hypertree, $\mathcal{T}$, as shown the leftmost in Fig. 3 for $Q = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5$. The hypertree $\mathcal{T}$ has three hypernodes that represent $R_1(a, b, c)$, $R_2(a, d) \bowtie R_3(c, d)$, and $R_4(b, e) \bowtie R_5(c, e)$, respectively. Here, R_1 is a relation appearing in Q, and there is no need to join. For the other two hypernodes, there are only 4 choices, namely, not to pre-compute joins, to pre-compute the join of $R_2 \bowtie R_3$, to pre-compute the join of $R_4 \bowtie R_5$, to pre-compute both joins. In other words, by the hypertree, $\mathcal{T}$, for this example, we only need to consider 4 possible query candidates, which decides whether R_{23} and R_{45} should be pre-computed. The search space of query candidates is significantly reduced to $2^{|V(\mathcal{T})|}$.

Reducing Choice of Attribute Orders. Leapfrog needs to determine the optimal attribute order to expand from i-tuple to (i+1)-tuple. For a query Q with n attributes for $n = |\mathsf{attrs}(Q)|$, there are $n!$ possible attribute orders to consider for any query Q_i in $\mathcal{Q}$, which incurs high selection cost. With the hypertree $\mathcal{T}$, it can reduce the search space to determine an attribute order following a traversal order ($\prec$) of the hypernodes of the hypertree, $\mathcal{T}$. Consider any hypernodes, u and v, in $\mathcal{T}$, where u appears before v (e.g, $u \prec v$) by the traversal order. First, an attribute that appears in u will appear before any attribute in v that does not appear in u. Second, the attributes in a hypernode v can vary if they do not appear in u, and can be determined via [6]. For hypertree $\mathcal{T}$ shown in the leftmost of Fig. 3, let's assume the traversal order among the hypernodes are $v_a \prec v_b \prec v_c$. A valid attribute order is $a \prec b \prec c \prec d \prec e$, and an invalid attribute order is $a \prec b \prec e \prec d \prec c$. The rationale behind such reduction is that the attributes inside a hypernode are tightly constrained by each other, while attributes between two hypernodes are loosely constrained, thus when following a traversal order, the attributes of $A_1, ..., A_{n-1}$ are more likely to be tightly constrained, which results in less intermediate tuples $t^1, ..., t^{n-1}$ of $T^1, ..., T^{n-1}$ respectively during Leapfrog. An experimental study in Sec. V confirms such intuition. By adopting such order, the search space of attribute order is reduced from $O(n!)$ to $O(|V(\mathcal{T})|!)$, where $|V(\mathcal{T})| < n$.

B. Finding The Plan

In this section, we discuss how to find a good plan from the reduced search space.

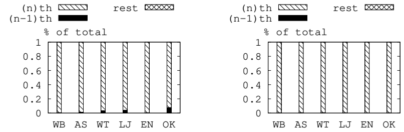
The Optimizer. Let $n^* = |V(\mathcal{T})|$, a naive approach finds the optimal plan by considering every combination of query candidates that form from candidate relations and every traversal orders, which are $O(2^{n^*} \times n^*!)$ plans in total. It is worth mentioning that calculating the cost for each plan could be costly as well. Thus finding plans by such a naive approach is not feasible.

We propose an approach to find good plans by exploring effective candidate relations in terms of trading the computation with communication. Recall that, pre-computing candidate relations could reduce the computation cost but increase the communication cost, and bring additional pre-computing cost. By finding the candidate relations that have a large positive utility in terms of reducing computation cost, we can effectively trade the computation cost with communication cost.

Let C be the set of candidate relations to pre-compute, O be the traversal orders, $cost_M(C)$, $cost_C(C)$, and $cost^i_E(C, O)$ be the cost of pre-computing cost, communication cost, and the computation cost of steps that extends to attributes of i-th traversed nodes in Leapfrog. It is worth noting that in complex join, the last few steps of Leapfrog usually dominate the entire computation cost due to a large number of partial bindings to extend [6], and reducing such cost by pre-computing R_v usually has maximum benefits in terms of reducing computation cost. An example is also shown in Fig. 4. Assuming we have an empty C and empty O. For each candidate relations R_v, where $v \in V(\mathcal{T})$, we try to explore its maximum utility by setting last traversed node of O to v. Then we compare the cost of pre-computing R_v and not pre-computing R_v, which are $cost_M(R_v) + cost_C(C \cup R_v) + cost^{n*}_E(C \cup R_v, O)$ and $cost_C(C) + cost^{n*}_E(C, O)$ respectively, with the cost of current optimal candidate relation R_{v*} in terms of cost. We only consider computation cost last steps of Leapfrog, as it usually

Algorithm 2: Optimizer(Q, D)

```
   Input: Query $Q$
   Output: The optimal query plan ($Q_i$, ord)
1  find optimal hypertree $\mathcal{T}$ for $Q$
2  let $C = \emptyset$, $O = \emptyset$, $V = V(\mathcal{T})$
3  while $V \neq \emptyset$ do
4      $C^* = C, O^* = O, cost = \inf, v^* = null, i = n^*$
5      for $v \in V$ do
6          if any two nodes in $V \setminus v$ are connected then
7              $O' = O.add(v), C' = C \cup R_v$
8              $cost' = cost_C(C) + cost_E^i(C, O')$
9              $cost'' = cost_M(R_v) + cost_C(C') + cost_E^i(C', O')$
10             if $cost' < cost$ then
11                 $C^* = C, O^* = O', cost = cost', v^* = v$
12             else if $cost'' < cost$ then
13                 $C^* = C', O^* = O', cost = cost'', v^* = v$
14     $i = i - 1, V.remove(v^*), C = C^*, O = O^*$
15 convert $C$, $O.reverse()$ to $Q_i$, ord
16 return ($Q_i$, ord);
```



(a) Q_5 (b) Q_6

Fig. 4: Percentages of intermediate tuples to extends during traversing $n - th$ node, $(n-1) - th$ node, and the rest of the node using two join queries, Q_5 and Q_6.

dominates the entire computation cost. After that, we can proceed to the next round of selecting R_u from the remaining candidate relations in a similar fashion and determining which node u the $(n-1)$-th traversed node and whether R_u should be pre-computed.

The detailed procedure is described in Alg. 2. Here, in lines 3-14, we gradually determine all candidate relations and the traversal order in reverse order. In lines 5-13, we find the next candidate relations. The if condition in line 6 is used to ensure that only O that could be extended to valid traversal order, which is described in the last section, is considered. In line 7-13, we compare the cost of pre-computing R_v and not pre-computing R_v with the cost of current optimal candidate relation R_{v^*}. Notice that, in i-th iteration, we only need to compute the $cost_E^i(C', O')$, as the computation cost of $cost_E^{i'}(C', O')$ is the same for all candidates relations for $i' > i$.

Lemma 1: Cost of Alg. 2 is $O(\frac{1}{2}(2n^*)(2n^*-1)L)$, and L is a large constant factor that is related to the cost of estimating the $cost_M$, $cost_C$, and $cost_E$.

IV. Related work

Our work is related to previous works on distributed multi-way join. Traditional multi-way join in the distributed platform such as Spark [1], consists of a sequence of distributed binary joins, such as distributed sort-merge join. They suffer from high communication cost for shuffling intermediate results when processing complex join queries. Such heavy communication cost can be reduced by one round multi-way join method HCube [7], which avoid shuffling of intermediate results. The combination of HCube and Leapfrog forms the HCubeJ [6], which processes the complex join queries effectively. However, when communication cost has been well optimized, the computation cost becomes the new bottleneck.

Dataset	WB	AS	WT	LJ	EN	OK
$\lvert R\rvert$ ($\times 10^6$)	13.2	22.1	50.9	69.4	183.9	234.4
Size (MB)	101.5	169.3	388.2	529.2	1370.0	1788.1

TABLE I: Datasets.

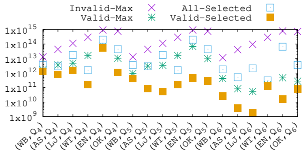


Fig. 5: Effectiveness of attribute order pruning.

Compare to previous work, we trying to co-optimize pre-computing, communication, and computation cost via introducing effective partial results.

V. Experiments

Queries. We study complex join queries used in the previous work [6], [13], [4]. The queries used are for subgraph queries with nodes in the range of 3-5 nodes. We report the experimental studies for the representative queries from Q_1 to Q_6, which are not easy to compute.

$$Q_1 \coloneq R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(a,c)$$
$$Q_2 \coloneq R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,d) \bowtie R_4(d,a) \bowtie R_5(a,c) \bowtie R_6(b,d)$$
$$Q_3 \coloneq R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,d) \bowtie R_4(d,e) \bowtie R_5(e,a) \bowtie R_6(b,d) \bowtie R_7(b,e) \bowtie R_8(c,a) \bowtie R_9(c,e) \bowtie R_{10}(a,d)$$
$$Q_4 \coloneq R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,d) \bowtie R_4(d,e) \bowtie R_5(e,a) \bowtie R_6(b,e)$$
$$Q_5 \coloneq R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,d) \bowtie R_4(d,e) \bowtie R_5(e,a) \bowtie R_6(b,e) \bowtie R_7(b,d)$$
$$Q_6 \coloneq R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,d) \bowtie R_4(d,e) \bowtie R_5(e,a) \bowtie R_6(b,e) \bowtie R_7(b,d) \bowtie R_8(c,e)$$

Datasets. Following [6], [13], we construct the database using the real large graph, where each graph is regarded as a relation with two attributes. The statistic of the graphs is shown in Table I. For each "test-case" that consists of a database and a query, the database is constructed by allocating each relation of the query with a copy of the graph. We select 6 commonly used graphs from various domains[1].

Competing Methods. We compare ADJ with four state-of-the-art multi-way join methods in the distributed environment, which are SparkSQL [1], HCubeJ [6], HCubeJ + Cache [13], and BigJoin [4].

Settings. All experiments are conducted on a cluster of a coordinator server and 7 follower servers[2]. All methods are deployed on Spark 2.2.0[3] except BigJoin. We used wall clock time to measure the cost of an algorithm with the time of starting up the system and loading the database into memory excluded. If an approach failed in a test-case, the figure will show an empty space, and if an approach timeout (12 hrs), we show a bar reaching the frame-top.

A. The Performance of ADJ

In this section, we investigate the performance of ADJ.

Effectiveness of Attribute Order Pruning. In this test, we compare the number of intermediate tuples generated during

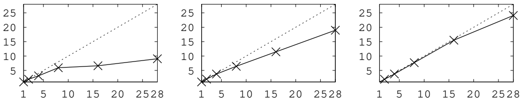
[1] EN can be downloaded from the link http://law.di.unimi.it/webdata/enwiki-2013/, while the rest of the graphs can be downloaded from SNAPhttps://snap.stanford.edu/data/index.html

[2] Each machine is equipped with 2 × Intel Xeon E5-2680 v4, 176 gigabytes of memory, interconnected via 10 gigabytes Ethernet

[3] For Spark, we create 28 workers from 7 slave servers, where each worker is assigned 7 cores and 28 gigabytes of memory.

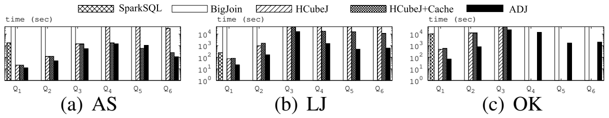
	Co − Optimization(sec)					Communication − First Optimization(sec)			
	Opt	Pre − Comp	Comm	Comp	Total	Opt	Comm	Comp	Total
Q_4	106	22	132	1282	1542	8	62	> 43200	> 43200
Q_5	132	44	103	222	501	9	112	> 43200	> 43200
Q_6	105	22	147	350	624	12	204	> 43200	> 43200

TABLE II: The comparison between co-optimization and communication-first optimization strategy in LJ dataset



(a) Q_1 (b) Q_2 (c) Q_3

Fig. 6: Speed-up factor of ADJ by varying number of workers.



(a) AS (b) LJ (c) OK

Fig. 7: Comparison of methods by varying datasets or queries

Leapfrog under valid attribute order and invalid attribute order on test-cases using $Q_4 - Q_6$ over all datasets. We omit $Q_1 - Q_3$, as their intermediate tuples are constant under any attribute order. The results are shown in Fig. 5[4] It can be seen that valid attribute orders produce fewer intermediate tuples than invalid attribute orders across all test-case, and attribute orders selected from valid attribute orders are better than attribute orders selected from all attribute orders.

The Cost and Effectiveness of Co-optimization. In this test, we show that co-optimization can effectively trade the computation with communication with a low query optimization cost, which includes the cost of sampling. We conduct experiment on test-cases that consist of dataset LJ and queries Q_4, Q_5, Q_6, and measures the cost of Optimization, Pre − Computing, Communication, Computation and Total. The results are shown in Table II. From them, we can see that on almost all test-cases, when Co − Optimization strategy is used, with a mildly increased Pre − Computing and Communication cost, the Computation cost is drastically reduced.

Scalability. In Fig 6, we show the speedup of our system when varying the number of workers of Spark from 1 to 28 on test-cases that consist of LJ, and all queries. It can be seen that our system has a near-linear speed up on query Q_2, Q_3. For query Q_1, the scalability is limited as it is a rather simple query, and the overhead of the systems gradually becomes the dominating cost.

B. Comparison with Other Join Approaches

In this section, we compare ADJ against state-of-the-art methods. In this test, we compare each method on test-cases where the datasets are fixed to AS, LJ, OK. The results are shown in Fig. 7 (a)-(c). For SparkSQL, it can only handle Q_1 and failed on all other queries due to overwhelming intermediate results. And, BigJoin can only handle Q_1 and Q_2. For $Q_1 - Q_3$, HCubeJ and HCubeJ + Cache performs similarly, and ADJ has a large lead due to the optimized HCube. For $Q_4 - Q_6$, HCubeJ + Cache performs better than HCubeJ, and HCubeJ + Cache has similar performance to ADJ on dataset AS as AS is relatively small and there is abundant remaining memory on each server to use for caching. On LJ dataset, HCubeJ + Cache is significantly outperformed by ADJ, as HCubeJ + Cache is a method that prioritizes communication cost over computation cost, and uses up all memory for shuffling and storing the tuples during HCube, which leaves little memory for caching. On OK dataset, both HCubeJ and HCubeJ +Cache failed, as the original HCube implementation shuffles too many tuples, which causes memory-overflow.

VI. Conclusion

This paper studies the problem of co-optimize communication and computation cost in a one-round multi-way join evaluation and proposes a prototype system ADJ for processing complex join queries. To find an effective query plan in a huge search space in terms of total cost, this paper study how to restrict the search space based on an optimal hypertree $\mathcal{T}$ and how to explore cost-effective query plans based on hypertree $\mathcal{T}$. Extensive experiments have shown the effectiveness of various optimization proposed in ADJ. We shall explore co-optimize computation, pre-computing, and communication for a query that consists of selection, projection, and join.

Acknowledgement

This work is supported by the Research Grants Council of Hong Kong, China under No. 14203618, No. 14202919 and No. 14205520, No. 14205617, No. 14205618, and NSFC Grant No. U1936205.

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, “Spark SQL: Relational Data Processing in Spark,” in *Proc. of SIGMOD’15*, pp. 1383–1394, 2015.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *IEEE TCDE*, vol. 36, no. 4, 2015.

[3] Y.-M. N. Nam, D. H. Han, and M.-S. K. Kim, “Sprinter: A fast n-ary join query processing method for complex olap queries,” in *Proc. of SIGMOD’20*, pp. 2055–2070, 2020.

[4] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar, “Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-memory Dataflows,” *PVLDB*, vol. 11, no. 6, pp. 691–704, 2018.

[5] R. A. Rossi, L. K. McDowell, D. W. Aha, and J. Neville, “Transforming graph data for statistical relational learning,” *Journal of Artificial Intelligence Research*, vol. 45, pp. 363–441, 2012.

[6] S. Chu, M. Balazinska, and D. Suciu, “From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System,” in *Proc. of SIGMOD’15*, pp. 63–78, 2015.

[7] F. N. Afrati and J. D. Ullman, “Optimizing Multiway Joins in a Map-Reduce Environment,” *TKDE*, vol. 23, no. 9, 2011.

[8] P. Beame, P. Koutris, and D. Suciu, “Communication steps for parallel query processing,” in *Proc. of SIGMOD’13*, pp. 273–284, 2013.

[9] T. L. Veldhuizen, “Leapfrog triejoin: A simple, worst-case optimal join algorithm,” *arXiv preprint arXiv:1210.0481*, 2012.

[10] H. Zhang, M. Qiao, J. X. Yu, and H. Cheng, “Fast distributed complex join processing,” 2021.

[11] G. Gottlob, N. Leone, and F. Scarcello, “Hypertree Decompositions and Tractable Queries,” *Journal of Computer and System Sciences*, vol. 64, no. 3, pp. 579–627, 2002.

[12] G. Gottlob, G. Greco, N. Leone, and F. Scarcello, “Hypertree Decompositions: Questions and Answers,” in *Proc. of PODS’16*, pp. 57–74, 2016.

[13] O. Kalinsky, Y. Etsion, and B. Kimelfeld, “Flexible caching in trie joins,” *arXiv preprint arXiv:1602.08721*, 2016.

[4] Invalid-Max, Valid-Max denote attribute orders that result in the maximum number of intermediate tuples among all invalid/valid orders respectively. And All-Selected, Valid-Selected denote attribute order selected by HCubeJ and ADJ respectively.