

# 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

**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], Hive [3], F1 [4], Myria [5], 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 [6] contain complex FK-FK joins, where multiple large tables are joined, or where the query graph has cycles, and has seen many applications, such as querying knowledge graph [7], finding triangle and other complex patterns in graphs [8], 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 [9], [10].

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 [11], 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 [12], [13]. In the join processing stage, HCubeJ uses an in-memory sequential algorithm Leapfrog [14] at each server to join the data received. It can be seen in Fig. 1(a) that the one-round multi-way join outperforms the multi-round binary join significantly, regarding the number of shuffled tuples, for complex join queries.

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. As shown in the first two bars for each of the two queries ( $Q_5$  and  $Q_6$  in Sec.VII-A) in Fig. 1(b), the communication cost can be small, but the computational cost can be high. 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. As shown in Fig. 1(b), by our approach, we can reduce the computation cost significantly with some additional overhead for communication and pre-computing cost. 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

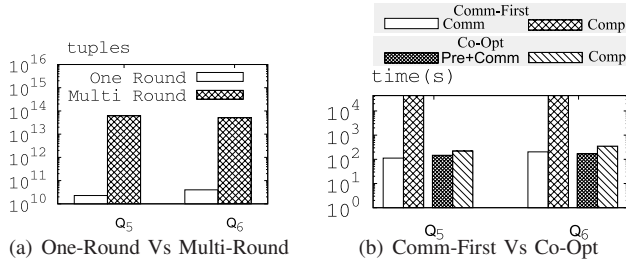


Fig. 1: Comparisons using two join queries,  $Q_5$  and  $Q_6$  (refer to Sec. VII-A), over the LJ dataset (refer to Table I). Here, “Comm” denotes communication cost, “Comp” denotes computation cost, “Pre+Comm” denotes pre-computing cost plus computation cost.

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. To find such an optimal query plan, we reduce the search space of query plans to filter ineffective query plans early, and propose a heuristic approach to explore cost-effective pre-computed partial results and join orders.
- We propose a simple yet effective distributed sampling process with a theoretic guarantee to provide accurate cardinality estimation for query optimization.
- We implement a prototype system ADJ and propose several implementation optimizations that significantly improve the performance of HCube, reduce the storage cost, and eliminate some redundant computation of HCubeJ.
- We conducted extensive performance studies, and confirm that our approach can be orders of magnitude faster than the previous approaches in terms of the total cost.

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, and discuss how to perform cardinality estimation via distributed sampling in Section IV. In Sec V, we discuss the implementation optimization of our prototype system. Section VI, we discuss the related work, and in Section VII we report our experimental studies. We conclude our work in Section VIII.

## II. PRELIMINARIES

A database  $D$  is a collection of relations. Here, a relation  $R$  with schema  $\{A_1, A_2, \dots, A_n\}$  is a set of tuples,  $(a_1, a_2, \dots, a_n)$ , where  $a_i$  is a value taken from the domain of an attribute  $A_i$ , denoted as  $\text{dom}(A_i)$ , for  $1 \leq i \leq n$ . Below, we use  $\text{attrs}(R)$  to denote the schema (the set of attributes) of  $R$ . A relation  $R$  with the schema of  $\text{attrs}(R)$  is a subset of the Cartesian product of  $\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$

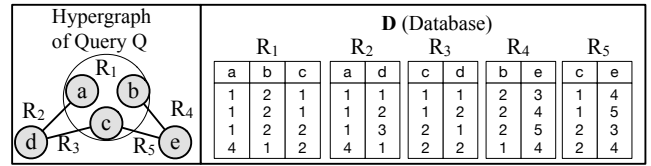


Fig. 2: The hypergraph of query  $Q$  (Eq (2)), and an example of database  $D$

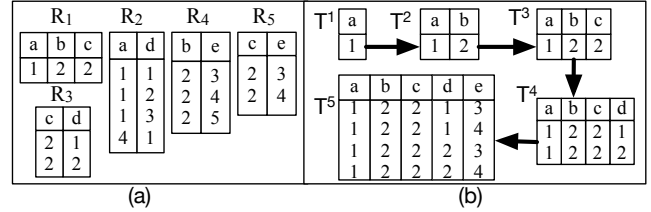


Fig. 3: (a) The tuples shuffled to server  $S_0$  with hypercube of coordinate  $(0, 0, 0, 0, 0)$ . (b) Leapfrog at the server  $S_0$  with hypercube of coordinate  $(0, 0, 0, 0, 0)$

for  $A_i \in \text{attrs}(R)$ . We focus on natural join queries (or simply join queries). A natural join query,  $Q$ , is defined over a set of  $m$  relations,  $\mathcal{R} = \{R_1, R_2, \dots, R_m\}$ , for  $m \geq 2$ , in the form of

$$Q(\text{attrs}(Q)) :- R_1(\text{attrs}(R_1)) \bowtie \dots \bowtie R_m(\text{attrs}(R_m)). \quad (1)$$

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

$$Q(a, b, c, d, e) :- R_1(a, b, c) \bowtie R_2(a, d) \bowtie R_3(c, d) \bowtie R_4(b, e) \bowtie R_5(c, e) \quad (2)$$

Its hypergraph representation  $H$  is shown in Fig. 2 together with the 5 relations. Here,  $V = \text{attrs}(Q) = \{a, b, c, d, e\}$ , and  $E = \{e_1, e_2, e_3, e_4, e_5\}$  for  $e_1 = \text{attrs}(R_1)$ ,  $e_2 = \text{attrs}(R_2)$ ,  $e_3 = \text{attrs}(R_3)$ ,  $e_4 = \text{attrs}(R_4)$ , and  $e_5 = \text{attrs}(R_5)$ . 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 [11] 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 [12], [13] and Leapfrog [14],

where HCube is a one-round communication optimal shuffling method that shuffles data to every server in the cluster, and Leapfrog is a fast in-memory sequential multi-way join algorithm to process the join query at each server over the data shuffled to it. For a join query  $Q$  over  $m$  relations,  $\mathcal{R} = \{R_1, R_2, \dots, R_m\}$ , HCube is proven in theory to be the optimal method in worst-case sense for transmitting the tuples to servers such that each server can evaluate the query on its own without further data exchange. Leapfrog [14] is proven in theory to be the optimal method in worst-case sense to evaluate a join query  $Q$ , while binary join could be sub-optimal. Also, Leapfrog is an iterator-based algorithm, which leaves little footprint in memory when processing the query.

**LeapFrog Join** [14] is one of the state-of-the-art sequential join algorithms for a join query  $Q$  over  $m$  relations,  $\mathcal{R} = \{R_1, R_2, \dots, R_m\}$  (Eq. (1)). Let  $\text{attrs}(Q)$  be the schema of  $Q$  for  $n = |\text{attrs}(Q)|$ . Leapfrog is designed to evaluate  $Q$  based on the attribute order  $\text{ord}$  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  $\mathcal{R}_{i+1} = \{R \mid A_{i+1} \in R \text{ and } R \text{ is a relation appearing in } Q\}$  (line 4). To find all  $A_{i+1}$  values that can join the input  $i$ -tuple  $t^i$ , denoted as  $\text{val}(t^i \rightarrow A_{i+1})$ , (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  $A_S$  be all the attributes that appear in both  $\text{attrs}(R)$  and  $\text{attrs}(t^i)$ , 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  $A_S$ . 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  $\text{val}(t^i \rightarrow A_{i+1})$  is the intersection of  $T_{i+1}$  and  $T'_{i+1}$ . At line 6-7, for every value,  $v$ , in  $\text{val}(t^i \rightarrow A_{i+1})$ , 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  $i = |\text{attrs}(Q)|$ , 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.

*Example 1:* Fig. 3(b) shows the steps of Leapfrog on the server  $S_0$  with relations as shown in Fig. 3(a). The input for the initial Leapfrog call is with an empty tuple  $t^0$ . Assume the order among  $\text{attrs}(Q)$  (e.g.,  $\text{ord}$  is  $a \prec b \prec c \prec d \prec e$ ).

First, Leapfrog will project the values for the first attribute  $a$  by attempting to join with  $t^0$ . At the server  $S_0$ , both relations,  $R_1$  and  $R_2$ , have the attribute  $a$ . Since  $t^0$  is empty, it projects  $\{1\}$  from  $R_1$  and projects  $\{1, 4\}$  from  $R_2$ , the result of the intersection is  $\{1\}$  as shown in the relation  $T^1$ , whose schema is  $\{a\}$ , in Fig. 3(b).

Second, for the tuple  $t^1 = (1)$  in  $T^1$ , it calls Leapfrog in which the 2nd attribute  $b$  in order is considered. Note that

both relations,  $R_1$  and  $R_4$ , have the attribute  $b$ . By joining the tuples in  $R_1$  with  $t^1 = (1)$ , it projects the  $b$  attribute value,  $\{2\}$ , with  $R_1$  since the corresponding tuple  $t \in R_1$  can join with the input tuple  $t^1$  on the attribute  $a$ , and it projects the  $b$  attribute values,  $\{2\}$ , with  $R_4$ , since it does not have the attribute  $a$  to join with  $t^1$ . The intersection of  $b$  attribute values from the two relations is  $\{2\}$ , as shown in the relation  $T^2$  on the schema  $(a, b)$  in Fig. 3(b). The new  $t^2$  to be used in the next Leapfrog call becomes  $(1, 2)$  on the schema  $(a, b)$ .

Fig. 3(b) shows the results for  $T^1$  to  $T^5$  by Leapfrog at the server  $S_0$ . Here, the join result for the hypercube assigned is in  $T^5$ . It is worth noting that Leapfrog is implemented as a series of iterator to avoid the recursive function call, and every newly generated tuple  $t^{i+1} \in T^{i+1}$  is used immediately to generate tuples  $t^{i+2}$  without being stored in memory.

**HCube Shuffle** [12], [13] 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  $p = (p_1, p_2, \dots, p_n)$ , where  $p_i$  is the number of partitions for the attribute  $A_i$  under  $\text{ord}$ , and  $n = |\text{attrs}(Q)|$ , hypercubes of  $P = p_1 \times \dots \times p_n$  dimension are constructed. It is worth mentioning that  $P$  can be larger than  $N^*$ . Here, a hypercube is identified by an coordinate of  $C = (c_1, \dots, c_n)$  of  $[p_1] \times \dots \times [p_n]$ , 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.

*Example 2:* Consider the join query  $Q$  (Eq. (2)) and the 5 relations in Fig. 2. Here,  $\text{attrs}(Q) = \{a, b, c, d, e\}$ . Let  $P = N^* = 4$ , assume the order among the attributes of  $Q$  is  $\text{ord} = a \prec b \prec c \prec d \prec e$ . Suppose the vector  $p = (p_1, p_2, p_3, p_4, p_5) = (1, 2, 2, 1, 1)$  is obtained by the optimizer, where  $p_i$  denotes the number of partitions for the attribute  $A_i$ . For example,  $p_1$  is for the attribute  $a$  because  $a$  is the first attribute in  $\text{ord}$ . The hypercubes based on  $p$  are  $[1] \times [2] \times [2] \times [1] \times [1]$ . Note that  $[l]$  represents a range from 0 to  $l - 1$ . The 4 hypercubes to be assigned to the 4 servers,  $S_0, S_1, S_2$ , and  $S_3$  are hypercubes with coordinate  $C_0 = (0, 0, 0, 0, 0)$ ,  $C_1 = (0, 0, 1, 0, 0)$ ,  $C_2 = (0, 1, 0, 0, 0)$ , and  $C_3 = (0, 1, 1, 0, 0)$ . The tuples in any of the 5 relations will be sent to some hypercubes. Here, suppose that a hash function,  $h_i(\cdot)$ , is designed for the  $i$ -th attribute  $A_i$ , and the hash function is of  $h_i(x) = x \% p_i$  for this example. The first tuple,  $(1, 2, 1)$ , that appears at the top in the relation  $R_1(a, b, c)$ , will be sent to the servers with hypercubes with coordinate  $(0, 0, 0, \star, \star)$ ,

---

**Algorithm 1:** Leapfrog( $t^i, Q$ )

---

**Input:** an  $i$ -tuple  $t^i$ , the query  $Q$ **Output:** tuples of  $Q$  emitted through iterators

```
1 if  $i = |\text{attrs}(Q)|$  then
2    $\lfloor$  Emit( $t^i$ );
3 else
4   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;
5   find all  $A_{i+1}$  values that can join the input tuple  $t^i$ ,
   denoted as  $\text{val}(t^i \rightarrow A_{i+1})$ ;
6   for each attribute value  $v$  in  $\text{val}(t^i \rightarrow A_{i+1})$  do
7      $\lfloor$  Leapfrog( $t^i \parallel v, Q$ );
```

---

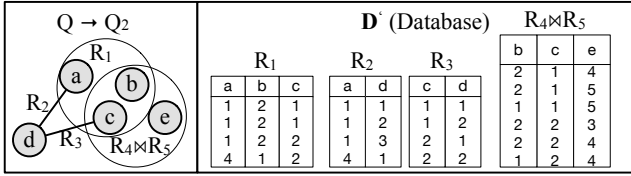


Fig. 4: A query candidate  $Q_i$  which gets the same result of  $Q$  in Fig. 2 by replacing  $R_4$  and  $R_5$  with  $R_4 \bowtie R_5$

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. The tuples of the 5 relations that are sent to the server  $S_0$  are shown in Fig. 3(a).

After HCube completes its shuffling by hashing, each server 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  $Q$ .

**Remark.** Given the two main costs, namely, communication cost (shuffling cost) and computation cost, HCubeJ is designed to put the communication cost at a higher priority and minimizes the communication cost first by optimizing  $p$ . There is no concern from HCube on the computation cost of Leapfrog, which does its best to process the query  $Q$  over the data shuffled to it.

However, the query  $Q$  could be inherently computationally difficult, and the communication cost may not be the dominating factor in distributed join processing as shown in Fig. 1(b). A key question we ask is which cost it should minimize. There are several options, (1) the communication cost, (2) the computation cost, and (3) the both. HCubeJ takes the first option. However, It is highly likely that the minimization of communication cost leads to high computation cost. In this paper, we study how to optimize query  $Q$  by converting it into an equivalent query  $Q_i$  with potential higher communication cost and lower computation cost with some additional pre-computing cost such that the total cost is minimal.

### 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. 2. 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. 4 (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  $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$  (refer to Eq. (1)). Let  $\mathcal{Q}$  be a collection of query candidates such as  $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_{|\mathcal{Q}|}\}$ , where  $Q_i = R'_1 \bowtie R'_2 \bowtie \dots \bowtie R'_l$ . Here,  $Q_i$  is equivalent to  $Q$  such that  $Q_i$  and  $Q$  return same results,  $\text{attrs}(Q_i) = \text{attrs}(Q)$ ,  $l \leq m$ , 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  $(Q_i, \text{ord})$  that consists of a query candidate  $Q_i \in \mathcal{Q}$ , which specifies how to pre-compute relations, and an attribute order  $\text{ord}$  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  $Q_i$ .

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  $\text{ord}$  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

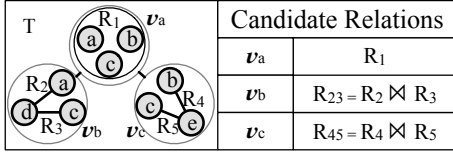


Fig. 5: Hypertree  $\mathcal{T}$  and candidate relations

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  $(Q_i, \text{ord})$ , 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  $Q_i = R'_1 \bowtie R'_2 \bowtie \dots \bowtie R'_l$ . As shown in Fig. 1(b), our approach can significantly reduce the total cost.

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  $\mathcal{T}$ . How to estimate the cardinality via distributed sampling is shown in Sec IV.

#### 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) [15]. 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) [16]. Such a hypertree  $\mathcal{T}$  found by GHD satisfies that  $\max_{v \in V(\mathcal{T})} |R_{max}|^{\text{fhw}}$  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  $|R_{max}|^{\text{fhw}}$  for the chosen  $\mathcal{T}$  and it is the lowest one among all possible  $\mathcal{T}$ .

*Example 3:* Consider the join query  $Q = R_1(a, b, c) \bowtie R_2(a, d) \bowtie R_3(c, d) \bowtie R_4(b, e) \bowtie R_5(c, e)$  (Eq. (2)). Its hypergraph is shown in Fig. 2, and its hypertree  $\mathcal{T}$  is shown the leftmost in Fig. 5. For the hypertree  $\mathcal{T}$ , its hypernodes are  $v_a, v_b, v_c$ , where  $v_a, v_b$ , and  $v_c$ , 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. The hyperedges  $\{(v_a, v_b), (v_b, v_c)\}$  ensure 1)  $\mathcal{T}$  is a hypertree 2) For any attribute  $A \in \{a, b, c, d, e\}$ , e.g.,  $a$ , the hypernodes that contains it, e.g.,  $v_a, v_b$ , are connected.

As shown in Example 3, the hypertree  $\mathcal{T}$  found from the hypergraph representation for a given join query,  $Q = R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$ , has two implications regarding the reduced search space to find the optimal  $Q_i = R'_1 \bowtie R'_2 \bowtie \dots \bowtie R'_l$ , 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. 5 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,  $R_1(a, b, c)$  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  $R_2(a, d) \bowtie R_3(c, d)$ , to pre-compute the join of  $R_4(b, e) \bowtie R_5(c, e)$ , 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  $n = |\text{attrs}(Q)|$ , 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 [11]. For hypertree  $\mathcal{T}$  shown in the leftmost of Fig. 5, 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 constraint by each other, while attributes between two hypernodes are loosely constraint, thus when following a traversal order, the attributes of  $A_1, \dots, A_{n-1}$  are more likely to be tightly constraint, which results in less intermediate tuples  $t^1, \dots, t^{n-1}$  of  $T^1, \dots, T^{n-1}$  respectively during Leapfrog. An experimental study in Sec. VII 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  $cost_E^i(C, O)$  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 [11], 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. 6. 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  $cost_M(R_v) + cost_C(C \cup R_v) + cost_E^{n^*}(C \cup R_v, O)$  and  $cost_C(C) + cost_E^{n^*}(C, O)$  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 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

---

### Algorithm 2: Optimizer( $Q, D$ )

---

**Input:** Query  $Q$

**Output:** The optimal query plan ( $Q_i, \text{ord}$ )

```

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    $C^* = C, O^* = O, cost = \text{inf}, v^* = \text{null}, i = n^*$ 
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        $cost'' =$ 
10         $cost_M(R_v) + cost_C(C') + cost_E^i(C', O')$ 
11       if  $cost' < cost$  then
12          $C^* = C, O^* = O', cost = cost',$ 
13          $v^* = v$ 
14       else if  $cost'' < cost$  then
15          $C^* = C', O^* = O', cost = cost'',$ 
16          $v^* = v$ 
17    $i = i - 1, V.remove(v^*), C = C^*, O = O^*$ 
18 convert  $C, O.reverse()$  to  $Q_i, \text{ord}$ 
19 return ( $Q_i, \text{ord}$ );
```

---

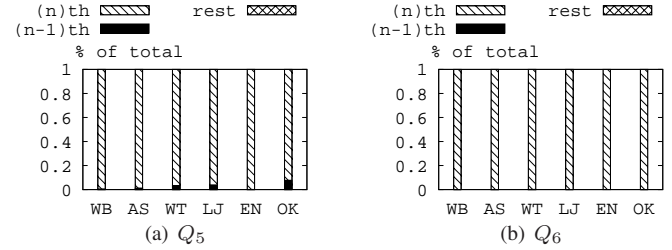


Fig. 6: Percentages of intermediate tuples to extends during traversing  $n$ -th node,  $(n-1)$ -th node, and the rest of the node using two join queries,  $Q_5$  and  $Q_6$  (same as Fig. 1).

order, which is described in the last section, is considered. In lines 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  $cost_E$ .

**Computing the Cost.** Next, we discuss how to compute pre-computing cost  $cost_M$ , communication cost  $cost_C$ , and computation cost of  $i$ -th step in Leapfrog  $cost_E$ . We focus on computing  $cost_C$  and  $cost_E$ , as  $cost_M$  is just a combination of  $cost_M$  and  $cost_E$ .

$cost_C(C)$  measures the communication cost of shuffling relations of  $R_v \in C$  and remaining relations of  $u \in V(\mathcal{T})$  that are not pre-computed  $R_u$  in terms of seconds needed to

transmit them across servers. Let us denote such collection of relations by  $\mathcal{R}_C$ . Recall that HCube has a parameter  $p$ , which determines the numbers of partitions on attribute  $A \in \text{attrs}(Q)$  and is related to how tuples are shuffled to each servers. Given a  $p$ , for each relation  $R \in \mathcal{R}_C$ , each tuple  $t \in R$  will be sent to  $\text{dup}(R, p) = \prod_{A \in \text{attrs}(Q) \setminus \text{attrs}(R)} p_A$  servers following the rules of HCube, where  $p_A$  denotes numbers of partitions on attribute  $A$ . And, we can represent  $\text{cost}_C(C)$  as  $\frac{\sum_{R \in \mathcal{R}_C} |R| \times \text{dup}(R, p)}{\alpha}$ , where  $\alpha$  is the number of tuples transmitted per seconds. Here,  $p$  is a parameter of HCube and it needs to be optimized to minimize  $\text{cost}_C(C)$  under the constraints 1) numbers of partition for each attribute should  $\geq 1$ ; 2) on average, the total amount of data a server received should be less than memory size  $M$  of the server, which translates to  $M - \sum_{R \in Q_i} |R| \times \text{frac}(R, p) \geq 0$ . Here,  $\text{frac}(R, p)$  denotes the average percentage of  $R$  will be sent to a server, which is  $\frac{1}{\prod_{A \in \text{attrs}(R)} p_A}$ . The optimization program is as follows:

$$\begin{aligned} & \text{minimize } \text{cost}_C(C) \\ & \text{s.t. } \quad \quad \quad p - \mathbf{1} \geq \mathbf{0} \\ & \quad \quad \quad M - \sum_{R \in Q_i} \text{size}(R) \times \text{frac}(R, p) \geq 0 \end{aligned} \quad (3)$$

By solving above optimization program, we can obtain  $\text{cost}_C(C) = \sum_{R \in \mathcal{R}_C} |R| \times \text{dup}(R, p)$ .

$\text{cost}_E^i(C, O)$  measures the computation cost of steps that extends attributes of  $i$ -th traversed nodes in Leapfrog. Recall that Leapfrog gradually extends  $i$ -tuple  $t^i \in T^i$  to  $(i+1)$ -th tuples  $t^{i+1}$ ,  $(i+2)$ -th tuples, ...,  $n$ -th tuples. As single node  $v \in V(\mathcal{T})$  might contains several attributes, and extending one node  $v$  might corresponds to extending several attributes, for simplicity, we use  $T^{v_i}$  to denote the tuples of partial binding of attributes are from  $v_1, v_2, \dots, v_i$ , where  $v_i$  is the  $i$ -th traversed node. Thus, we can represent the cost of extending attributes of  $i$ -th traversed nodes in Leapfrog,  $\text{cost}_E^i(C, O)$ , as  $\frac{|T^{v_{i-1}}|}{\beta^i \times N^*}$ , where  $|T^{v_{i-1}}|$  is the numbers of partial bindings whose attributes are from  $v_1, \dots, v_{i-1}$ ,  $\beta^i$  is numbers of partial bindings extended per seconds per server, and  $N^*$  is the number of servers. Notice that  $\beta^i$  can be significantly higher if  $v_i$  is pre-computed.

$\text{cost}_M$  measures the pre-computing cost of  $R_v$ . Let  $\lambda(v)$  be the relations of a node  $v$  in  $V(\mathcal{T})$ ,  $\text{cost}_M$  consists of the communication cost of shuffling  $\lambda(v)$  and computation cost of  $\bowtie \lambda(v)$ , which can be computed using above methods for computing  $\text{cost}_C(C)$  and  $\text{cost}_E^i(C, O)$ .

In the above calculation,  $\alpha$  can be regarded as a constant that measures the communication performance of the cluster. More specifically, we can measure it by randomly generating tuples of size  $k$ , which is to be shuffled to random servers in the cluster, and recording the time  $t$  to shuffling  $k$  tuples to their destination, where  $\alpha = \frac{k}{t}$ .  $\beta$  can be estimated by sampling some partial bindings, extending them, and taking the average of their extending time. More specifically, if  $v_i$  is pre-computed, the main cost of extending a partial binding is querying the trie for candidate values, thus  $\beta^i$  is a constant that can be pre-measured as  $\frac{k}{t}$  by recording the time  $t$  to perform  $k$

query on a trie of size  $|R_{v_i}|$ . It is worth noting that we can pre-measure  $\beta^i$  on trie of various sizes. If  $v_i$  is not pre-computed, we set  $\beta^i$  by reusing statistics gathered during sampling, which is to be explained in the next section. More specifically, let the total numbers of extension performed during sampling be  $k$  and aggregated extension time be  $t$ , we set  $\beta^i = \frac{k}{t}$ .

#### IV. ESTIMATING CARDINALITY VIA DISTRIBUTED SAMPLING

In this section, we discuss how we perform cardinality estimation via distributed sampling and why we choose sampling-based approaches to estimate cardinality.

**Why Sampling.** An accurate cardinality estimation is crucial for the optimizer to choose a good query plan [17]. Currently, there are two styles to do cardinality estimation: 1) sketch-based approaches 2) sampling-based approaches.

Theoretical [18] as well as empirical [17] work has shown that existing sketches-based approaches, which utilize fixed-size, per-attribute summary statistics (histograms) with strong assumptions (uniformity, independence, inclusion, ad hoc constants) to estimate cardinalities, often return estimations with large errors, especially on complex joins with more than 2 relations. Such error has been shown to lead to sub-optimal plans that are up to  $10^2$  slower than optimal plans in empirical study work [17]. For sampling-based approaches, promising result is shown in [19] that sampling-based approaches could produce estimations that are orders of magnitude more accurate than sketch-based approaches in a reasonable time by performing a sequence of index join with samples. In summary, sketch-based approaches often incur less overhead than sampling-based approaches when performing estimations, but sampling-based approaches usually return estimations with much fewer errors.

As our work targets complex join, which usually is long-running tasks and the additional cost brought by sampling is negligible compared to its benefits in reducing queries' running time, we choose to estimate cardinality via sampling.

**Estimating Cardinality Via Sampling.** Given a query  $Q$ , whose result is  $T$ , we want to estimate  $|T|$ . Let  $T_{A=a}$  be result tuples in  $T$  whose value on attribute  $A$  is  $a$ , we can express  $T$  as follows.

$$|T| = \sum_{a \in \text{val}(A)} |T_{A=a}| = |\text{val}(A)| \times \overline{|T_{A=a}|} \quad (4)$$

where  $\text{val}(A)$  is the collection of values of  $A$  in  $T$ , and  $\overline{|T_{A=a}|} = \frac{\sum_{a \in \text{val}(A)} |T_{A=a}|}{|\text{val}(A)|}$ . Suppose  $|\text{val}(A)|$  is known, then we need to estimate  $\overline{|T_{A=a}|}$  to obtain an estimation of  $|T|$ . To estimate  $\overline{|T_{A=a}|}$ , let  $a$  be a randomly selected value from  $\text{val}(A)$ . Let  $X$  be the random variable that is  $|T_{A=a}|$ , and  $\mu = \mathbf{E}[X] = \overline{|T_{A=a}|}$ .

Suppose we wish to estimate  $\mu$ . We simply choose  $k$  independent values  $a_1, a_2, \dots, a_k$  from  $\text{val}(A)$  with associated random variables  $X_1, X_2, \dots, X_k$ . Define  $\bar{X} = \frac{1}{k} \sum_{i < k} X_i$  as our estimate. The generalized Chernoff-Hoeffding bounds [20] give guarantees on  $\bar{X}$ , as follows.

*Lemma 2:* Let  $X_1, X_2, \dots, X_k$  be independent random variables with  $X_i \in [0, b]$ , where  $b$  is the maximum values  $X_i$  can take. Define  $\bar{X} = \frac{1}{k} \sum_{i < k} X_i$ . Let  $\mu = \mathbf{E}[X]$ . Then for  $p \in [0, 1]$ , we have

$$PR\{|\bar{X} - \mu| \geq pb\} \leq 2exp(-2kp^2)$$

Hence, if we set  $k = \lceil -0.5p^{-2} \ln(2/\delta) \rceil$ , then  $PR\{|\bar{X} - \mu| > pb\} < \delta$ . In other words, for  $k$  samples, with confidence at least  $1 - \delta$ , the error rate, which measures the deviation of  $E[\bar{X}]$  in terms of  $b$  is at most  $p$ .

In practice, we can easily obtain  $val(A)$  by performing intersections over relations of  $Q$  that contains  $A$  in their schemas, which is  $\bigcap_{R \in Q \wedge A \in \text{attrs}(R)} \Pi_A R$ . We can obtain  $|T_{A=a}|$  for any  $a$  chosen from  $A$  by performing an Leapfrog starting from  $A$  with attribute on  $A$  being fixed as  $a$ , which obtains  $T_{A=a}$ .

**Distributed Sampling.** A naive approach parallelize the sampling process described above by utilizing HCube directly. More specifically, it first shuffling the relations of  $Q$  into servers using HCube such that each server can perform the sampling on its own based on tuples on it, then on each server, the sampling process described in the above paragraph is performed. However, such naive approaches would shuffle many unnecessary tuples during HCube, as only a small fraction of  $val(A)$ , and performing Leapfrog for them probably will not involve all tuples of every relation in  $Q$ .

We can reduce such costs by reducing the database first before all relations in it are shuffled by HCube. First, we find all relations  $\mathcal{R}$  in a database whose schema contains  $A$ , and compute a projected relation for each of them  $\Pi_A R, R \in \mathcal{R}$ . Then, for all  $R \in \mathcal{R}$ , we shuffle their  $\Pi_A R$  such that we can compute the intersection of them and obtain  $val(A)$ . Then, from  $val(A)$ , we randomly select some samples  $S'$ . Next, we reduce the original database by performing semi-join between  $S'$  and  $R \in \mathcal{R}$  to filter unpromising tuples. Finally, we shuffle the reduced database instead of the original database, and perform sampling on it.

## V. IMPLEMENTATION

We implemented a prototype system in Spark, which is the de-facto platform to perform large scale analytic tasks.

**Optimizing HCube.** Previously, HCube is implemented as a sequence of `map` and `reduce` stage [12], where `map` stage marks the destination coordinate for each tuple and `reduce` stage shuffles each tuple to their corresponding servers. Such implementation suffers from significant performance loss due to overwhelming amount of tuples being shuffled. To reduce the cost of HCube, the key is to reduce the cost of expensive shuffling. A solution is to pull the tuples in blocks from remote machines directly instead of shuffling tuples one by one, which bypass shuffling process. The new HCube proceed in two steps:

- Group all tuples from the same relation and with the same hash values under the HCube’s hash function into a block and tagged that block with that has values.

- For each server, it pulls the entire block of each relation whose hash values “fits” its own coordinate in blocks from remote machines.

We next use an example to better illustrate the idea.

*Example 4:* Let’s take query in Fig. 2 whose share  $p = (1, 2, 2, 1, 1)$ , which result in four servers with coordinate  $(0, 0, 0, 0, 0)$ ,  $(0, 1, 0, 0, 0)$ ,  $(0, 0, 1, 0, 0)$ ,  $(0, 1, 1, 0, 0)$ . For the relation  $R_3$  with schema  $R_3(c, d)$ , its tuples will be split into two blocks, where  $(1, 1)$ ,  $(1, 2)$  will be in block  $B(0, 0)$ , and  $(2, 1)$ ,  $2, 2$  will be in block  $B(1, 0)$  as their hash value for  $c, d$  is 1, 0 and 0, 0 respectively. And, the servers with coordinate  $(0, 0, 0, 0, 0)$ ,  $(0, 1, 0, 0, 0)$  will pull block  $B(0, 0)$  as their coordinate on  $c$  and  $d$  is 0, 0. Similarly, servers with coordinate  $(0, 0, 1, 0, 0)$ ,  $(0, 1, 1, 0, 0)$  will pull block  $B(1, 0)$ .

A further benefit that this new HCube implementation has is that it allows us to do some preprocessing works on a block level. More specifically, we can reduce the cost of constructing the trie of local database in each machine by pre-build the trie for each block of every relation.

## VI. RELATED WORK

Our work is related to previous works from three areas: multi-way join on a single machine, distributed multi-way join, and cardinality estimation.

**Multi-Way Join on a Single Machine.** Optimizing the computation cost of a multi-way join has been studied for decades. Traditional multi-way join [21] is based on relational algebra (RA) — an RA expression of a multi-way join represents a sequence of binary joins, i.e., sort-merge join. The recently emerged AGM bound [22], [23] on the worst-case output size of a multi-way join provides a standard to evaluate the computation efficiency of a join algorithm. In the worst-case, using traditional binary joins is suboptimal while worst-case optimal join algorithms such as NPRR [24], Generic Join [25], Leapfrog[14] are optimal. To improve the efficiency of worst-case optimal join algorithm for general case rather than worst-case, EmptyHeaded [26] combines binary join and worst-case optimal join via tree decomposition [15], [16], and Yannakakis algorithm [27], which improves the computation efficiency at a great cost of memory consumption. To overcome the memory issue of the EmptyHeaded, CacheTrieJoin [28] is proposed, which incorporates multi-level cache into Leapfrog. However, it is difficult to set the size of the cache for each level and the total amount of the cache.

**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 [12], [13], which avoid shuffling of intermediate results. The combination of HCube and Leapfrog forms the HCubeJ [11], which processes the complex join queries effectively. However, when communication cost has been well optimized, the computation cost becomes the new bottleneck. Also, simply combining HCube



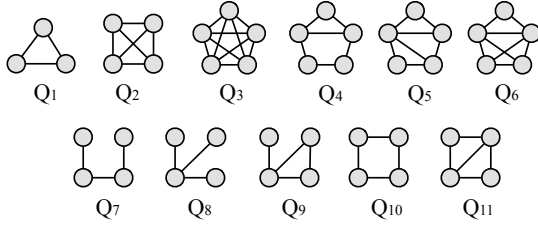


Fig. 7: Queries

Dataset	WB	AS	WT	LJ	EN	OK
$ R  (\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.

and optimized version of Leapfrog, such as CachedTrieJoin, helps little, as it prioritizes the memory usage for HCube over memory usage for CacheTrieJoin. Compared to previous work, we try to co-optimize pre-computing, communication, and computation cost via introducing effective partial results.

**Cardinality Estimation.** The estimation of cardinality methods can be roughly classified into two classes: 1) sketches based, which use statistics of the database to estimate the cardinality of the query, see [17] as an entry, 2) sampling-based, which estimates the cardinality by sampling over the database according to query, see [29], [19] as an entry. It has been shown that the estimation of the sketch-based method could be orders of magnitude deviate from the ground truth [19], [17] on complex join.

## VII. EXPERIMENTS

### A. Setup

**Queries.** We study complex join queries used in the previous work [11], [28], [8]. The queries used are for subgraph queries with nodes in the range of 3-5 nodes. The queries studied are shown in Fig. 7. We report the experimental studies for the representative queries from  $Q_1$  to  $Q_6$ , which are not easy to compute. We omit the results for  $Q_7$  to  $Q_{11}$ , as they can be computed fast, and the performance of these queries are very similar among the approaches being tested.

$$\begin{aligned}
 Q_1 &:- R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(a, c) \\
 Q_2 &:- R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(c, d) \bowtie R_4(d, a) \bowtie R_5(a, c) \\
 &\quad \bowtie R_6(b, d) \\
 Q_3 &:- R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(c, d) \bowtie R_4(d, e) \bowtie R_5(e, a) \\
 &\quad \bowtie R_6(b, d) \bowtie R_7(b, e) \bowtie R_8(c, a) \bowtie R_9(c, e) \\
 &\quad \bowtie R_{10}(a, d) \\
 Q_4 &:- R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(c, d) \bowtie R_4(d, e) \bowtie R_5(e, a) \\
 &\quad \bowtie R_6(b, e) \\
 Q_5 &:- R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(c, d) \bowtie R_4(d, e) \bowtie R_5(e, a) \\
 &\quad \bowtie R_6(b, e) \bowtie R_7(b, d) \\
 Q_6 &:- R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(c, d) \bowtie R_4(d, e) \bowtie R_5(e, a) \\
 &\quad \bowtie R_6(b, e) \bowtie R_7(b, d) \bowtie R_8(c, e)
 \end{aligned}$$

**Datasets.** Following [11], [28], 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. WB (web-BerkStan) is a web graph of Berkeley and Stanford. AS (as-Skitter) is an internet topology graph, from traceroutes run daily in 2005. WT (wiki-Talk) is a Wikipedia talk (communication) network. LJ (com-LiveJournal) is a LiveJournal online social network. EN (en-wiki2013) represents a snapshot of the English part of Wikipedia as of late February 2013. OK (com-Orkut) is an Orkut online social network. Their statistical information is listed in Table I. EN can be downloaded from the link <sup>1</sup>, while the rest of the graphs can be downloaded from SNAP<sup>2</sup>.

**Competing Methods.** We compare ADJ with four state-of-the-art multi-way join methods in the distributed environment.

- SparkSQL [1]: The state-of-the-art multi-round multi-way join framework on Spark, which performs multi-way join based on decomposing the query into smaller join queries, and combining intermediate relations in a pairwise way.
- HCubeJ [11]: The state-of-the-art one-round multi-way join framework that utilizes a one-round shuffling method HCube and the worst-case optimal join Leapfrog
- HCubeJ + Cache [28]: The state-of-the-art one-round multi-way join framework that utilizes a one-round shuffling method HCube and adopt an optimized Leapfrog with cache[28].
- BigJoin [8]: The state-of-the-art multi-round distributed multi-round multi-way join framework, which parallelizes Leapfrog.

**Evaluation Metrics.** 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 due to insufficient memory, the figure will show a space instead of a bar in the corresponding location of the figure. If an approach failed in completing the test-case within 12 hours, we show a bar reaching the frame-top.

**Parameter Setting.** We set  $\alpha$  of ADJ by pre-measuring the communication performance of the cluster based on Sec. III-B. We set the numbers of samples to be  $10^5$  as it achieves a balance between accuracy and cost based on our experiments. We set  $\beta$  based on Sec. III-B for each test-case by reusing statistics during sampling of each test-case. For competing methods, we use their default settings.

**Distributed Settings.** All experiments are conducted on a cluster of a master server and 7 slave servers ( $2 \times$  Intel Xeon E5-2680 v4, 176 gigabytes of memory, interconnected via 10 gigabytes Ethernet). All methods are deployed on Spark 2.2.0. For Spark, we create 28 workers from 7 slave servers, where each worker is assigned 7 cores and 28 gigabytes of memory. Each core of the worker can be assigned a hypercube in HCube.

<sup>1</sup><http://law.di.unimi.it/webdata/enwiki-2013/>

<sup>2</sup><https://snap.stanford.edu/data/index.html>

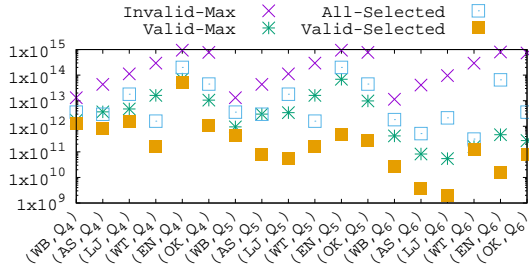


Fig. 8: Effectiveness of attribute order pruning.

### B. 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 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. 8, where Invalid-Max denotes the attribute order that results in the maximum number of intermediate tuples among all invalid orders. Valid-Max denotes the attribute order that results in the maximum number of intermediate tuples among any valid attribute orders. All-Selected denotes the attribute order selected by HCubeJ [11], which select the attribute order from all attribute order. Valid-Selected denotes the attribute orders selected by ADJ. It can be seen that in terms of the maximum number of intermediate tuples produced, valid attribute orders perform better than invalid attribute orders across all test-case. Also, we can see that selecting the attribute order from only valid attribute orders can produce a better attribute order than considering all attribute orders. This experiment confirms that the effectiveness of our heuristic in selecting good attribute orders and pruning non-effective attribute orders.

**Effectiveness of Optimizations on HCube.** In this test, we compare the effectiveness of the techniques proposed for optimizing the performance of HCube. We denote the original HCube implementation by Push, our optimized HCube implementation by Pull, and our optimized HCube implementation with tries pre-constructed by Merge. We run test-cases that consist of all datasets and query  $Q_2$ , and compare the communication cost and cost, where the results are shown in Fig. 9. In terms of communication cost, Pull and Merge outperform Push by up to two orders of magnitude. And, Merge outperforms Pull, as the block that contains one trie, which can be implemented using three arrays, are easier to serialize and deserialize than the block that contains many tuples. In terms of computation cost, Push and Pull are similar, and Merge outperforms the other two methods by up to an order of magnitude as tries has already been pre-constructed before HCube. This experiment shows that our proposed techniques for HCube can significantly reduce communication and some computation cost.

**Cost and Accuracy of Sampling Process.** In this test, we show that a relatively small amount of samples is enough for

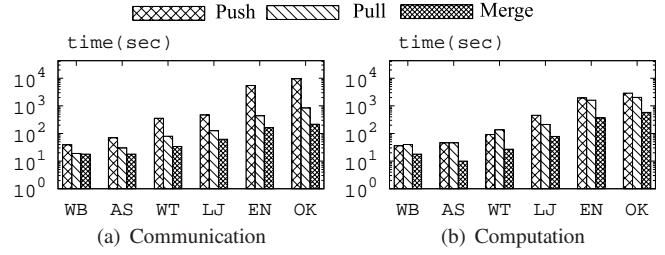


Fig. 9: Comparison of different implementation of HCube.

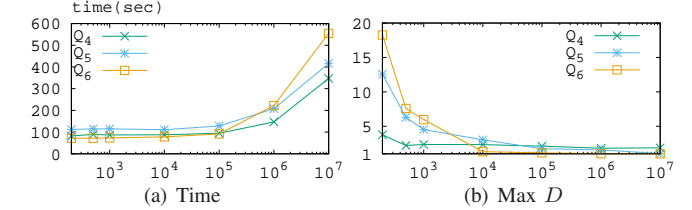


Fig. 10: Cost and accuracy of Sampling Process

an accurate estimation of cardinality. For an query  $Q$  whose result is  $T$ , let the real cardinality of  $T$  be  $|T|$  and the estimated one be  $|\tilde{T}|$ . Let  $D = \frac{\max(|\tilde{T}|, |T|)}{\min(|\tilde{T}|, |T|)}$  be an indicator that measures their relative difference, which means the close  $D$  is to 1, the better. We conduct experiments on test-cases that consist of dataset LJ and query  $Q_4, Q_5, Q_6$ . For each test-case, we vary the numbers of samples from  $2 * 10^2$  to  $10^7$  and plot the maximum relative difference of all estimated cardinality and the aggregated sampling time. The results are shown in Fig. 10. We can see that after the sampling budget is increased beyond  $10^4$ , the maximum relative difference converges to 1, which indicates there is almost no difference between the estimated value and real value. In terms of sampling cost, before  $10^6$  sampling budget, the cost stays almost the same. This experiment confirms the efficiency and accuracy of our sampling-based cardinality estimation approach.

**The Cost and Effectiveness of Co-optimization.** In this test, we show that co-optimization can effectively trading 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 datasets AS, LJ, OK 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-Table IV. 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. Also, there are test-cases such as  $(OK, Q_6)$ , whose Communication cost decreases as well. The reason is that introducing pre-computed relation increases the size of the input database, but also changes the query itself and alters share  $p$  of HCube, which could result in smaller Communication cost. From Table II-Table IV, it also can be seen that although Optimization cost of Co – Optimization strategy is consistently larger than Optimization cost of Communication – First Optimization strategy, it is still small compared to the total cost. This experiment confirms the

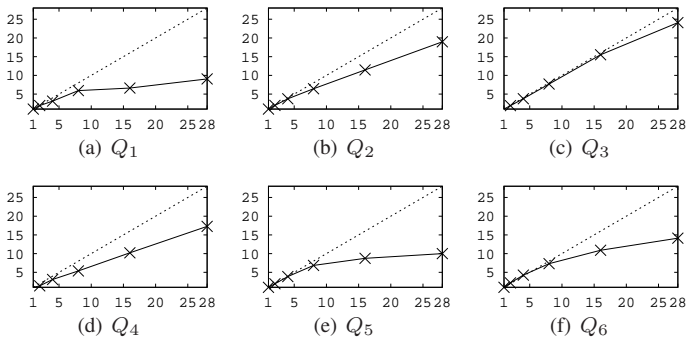


Fig. 11: Speed-up factor of ADJ under different workers under 1 to 28 workers .

effectiveness of Co – Optimization strategy and relatively low query Optimization cost of Co – Optimization strategy.

**Scalability.** In Fig 11, 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  $Q_2, Q_3, Q_4, Q_6$ . 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. For query  $Q_5$ , its limited scalability is due to the skewness, where the “last straggler” effect plays a bigger role in determining the elapsed time.

### C. Comparison with Other Join Approaches

In this section, we compare ADJ against state-of-the-art methods.

**Varying Dataset.** In this test, we compare each method on test-cases where the queries are fixed to  $Q_1, Q_2, Q_3$ . The results are shown in Fig. 12 (a)-(c). It can be seen that multi-round methods SparkSQL and BigJoin fail on many of the queries due to overwhelming intermediate results, while one-round methods successfully tackle most of the queries as the shuffling of intermediate results are avoided. Also, BigJoin is better than SparkSQL as the worst-case optimal join Leapfrog it parallelizes generates less intermediate tuples. Also, it can be seen that with the increase the input database size, HCubeJ, HCubeJ + Cache, spent more portion of time on HCube, and on test-case (LJ,  $Q_3$ ), (EN,  $Q_3$ ), (OK,  $Q_3$ ), they have a difficult time shuffling the tuples using original HCube implementation. In comparison, ADJ can successfully process all test-cases and spent significantly less time when shuffling the relations on test-cases that involve complex queries such as  $Q_3$  or large dataset EN, OK.

**Varying Query.** 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. 12 (d)-(e). 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. It can be seen that in almost all test-case ADJ can effectively balance the computation cost and communication cost by adopting a co-optimization strategy.

## VIII. 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] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, “Hive: a warehousing solution over a map-reduce framework,” *Proc. of VLDB’09*, vol. 2, no. 2, pp. 1626–1629, 2009.
- [4] J. Shute, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, H. Apte, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, and D. Menestrina, “F1: a distributed SQL database that scales,” *PVLDB*, vol. 6, no. 11, pp. 1068–1079, 2013.
- [5] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, *et al.*, “The myria big data management and analytics system and cloud services.,” in *CIDR’17*, 2017.
- [6] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2055–2070, 2020.
- [7] B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z. M. Ozsoyoglu, “A complete translation from sparql into efficient sql,” in *Proc. of IDEAS’09*, pp. 31–42, 2009.
- [8] 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.

	Co – Optimization(sec)					Communication – First Optimization(sec)			
	Optimization	Pre – Computing	Communication	Computation	Total	Optimization	Communication	Computation	Total
$Q_4$	107	12	66	1276	1461	3	21	> 43200	> 43200
$Q_5$	90	24	50	907	1071	4	36	> 43200	> 43200
$Q_6$	63	12	19	18	112	4	47	30426	30477

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

	Co – Optimization(sec)					Communication – First Optimization(sec)			
	Optimization	Pre – Computing	Communication	Computation	Total	Optimization	Communication	Computation	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 III: The comparison between co-optimization and communication-first optimization strategy in LJ dataset

	Co – Optimization(sec)					Communication – First Optimization(sec)			
	Optimization	Pre – Computing	Communication	Computation	Total	Optimization	Communication	Computation	Total
$Q_4$	218	71	712	13214	14215	37	1050	> 43200	> 43200
$Q_5$	265	142	422	877	1706	46	1566	> 43200	> 43200
$Q_6$	278	71	1189	516	2054	42	2067	> 43200	> 43200

TABLE IV: The comparison between co-optimization and communication-first optimization strategy in OK dataset

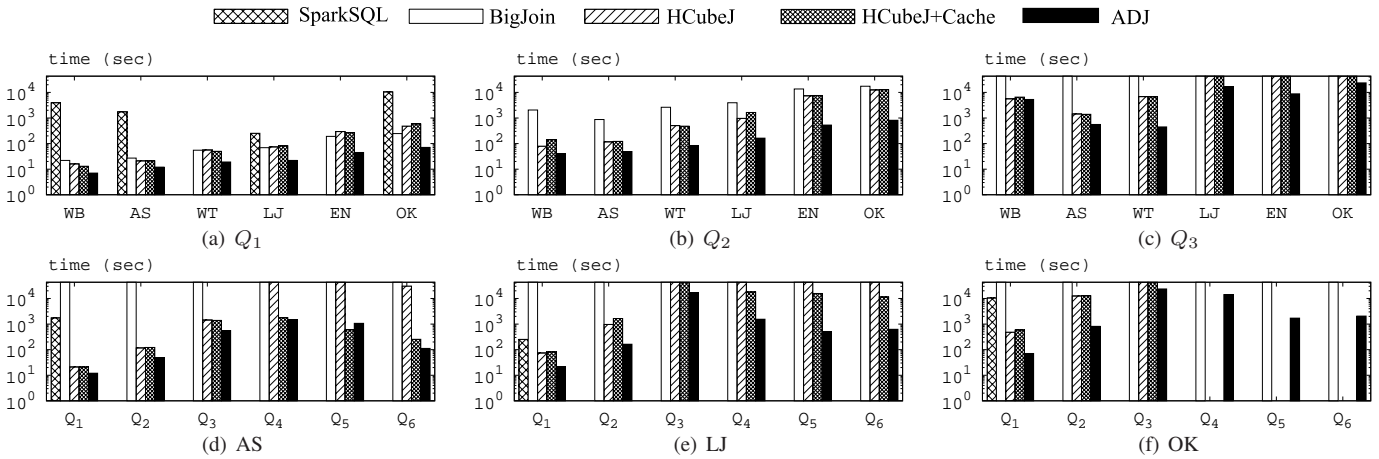


Fig. 12: Comparison of methods by varying datasets or queries

- N. N. Liu, L. He, and M. Zhao, "Social temporal collaborative ranking for context aware movie recommendation," *ACM TIST*, vol. 4, no. 1, pp. 1–26, 2013.
- 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.
- 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.
- F. N. Afrati and J. D. Ullman, "Optimizing Multiway Joins in a Map-Reduce Environment," *TKDE*, vol. 23, no. 9, 2011.
- P. Beame, P. Koutris, and D. Suciu, "Communication steps for parallel query processing," in *Proc. of SIGMOD'13*, pp. 273–284, 2013.
- T. L. Veldhuizen, "Leapfrog triejoin: A simple, worst-case optimal join algorithm," *arXiv preprint arXiv:1210.0481*, 2012.
- 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.
- G. Gottlob, G. Greco, N. Leone, and F. Scarcello, "Hypertree Decompositions: Questions and Answers," in *Proc. of PODS'16*, pp. 57–74, 2016.
- V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?," *PVLDB*, vol. 9, no. 3, pp. 204–215, 2015.
- Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *Proc. of SIGMOD'1991*, pp. 268–277, 1991.
- V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling," in *CIDR'17*, 2017.
- W. Hoeffding, "Probability inequalities for sums of bounded random variables," in *The Collected Works of Wassily Hoeffding*, pp. 409–426, 1994.
- P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," in *Proc. of SIGMOD'1979*, pp. 23–34, 1979.
- A. Atserias, M. Grohe, and D. Marx, "Size Bounds and Query Plans for Relational Joins," in *Proc. of FOCS'08*, pp. 739–748, 2008.
- M. Grohe and D. Marx, "Constraint Solving via Fractional Edge Covers," *ACM Trans. Algorithms*, vol. 11, no. 1, pp. 4:1–4:20, 2014.
- H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case Optimal Join Algorithms: [Extended Abstract]," in *Proc. of PODS'12*, pp. 37–48, 2012.
- H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: new developments in the theory of join algorithms," *ACM SIGMOD Record*, vol. 42, no. 4, pp. 5–16, 2014.
- C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "EmptyHeaded: A Relational Engine for Graph Processing," *ACM Trans. Database Syst.*, vol. 42, no. 4, pp. 20:1–20:44, 2017.
- M. Yannakakis, "Algorithms for Acyclic Database Schemes," in *Proc. of VLDB'1981*, pp. 82–94, 1981.
- O. Kalinsky, Y. Etsion, and B. Kimelfeld, "Flexible caching in trie joins," *arXiv preprint arXiv:1602.08721*, 2016.
- Y. Chen and K. Yi, "Two-Level Sampling for Join Size Estimation," in *Proc. of SIGMOD'17*, pp. 759–774, 2017.