From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System

Shumo Chu, Magdalena Balazinska, Dan Suciu
Computer Science and Engineering, University of Washington
Seattle, Washington, USA
{chushumo, magda, suciu}@cs.washington.edu

ABSTRACT

Big data analytics often requires processing complex queries using massive parallelism, where the main performance metrics is the communication cost incurred during data reshuffling. In this paper, we describe a system that can compute efficiently complex join queries, including queries with cyclic joins, on a massively parallel architecture. We build on two independent lines of work for multi-join query evaluation: a communication-optimal algorithm for distributed evaluation, and a worst-case optimal algorithm for sequential evaluation. We evaluate these algorithms together, then describe novel, practical optimizations for both algorithms.

1. INTRODUCTION

Novel large-scale data analytics engines such as Shark-Spark [35], Dremel [19], F1 [30], Myria [15] and others [3, 32] use massive parallelism in order to support complex queries on large data sets. These engines are designed to evaluate queries in main memory, because they use sufficiently many servers to ensure that their data fits in main memory. For these engines, the traditional performance metrics consisting of the number of disk I/Os is replaced by a new bottleneck, which is the communication cost for reshuffling data during query execution. Each reshuffling step requires a repartition of the entire dataset or intermediate result, which can be expensive. Data shuffling can also create load imbalance (a.k.a skew) between operator partitions.

The workloads on traditional OLAP engines usually consist of star-joins with aggregates, where the fact table is significantly larger than the dimension tables. These queries are often optimized by partitioning the fact table and replicating the dimension tables on all workers. But new data analytics engines face new kinds of workloads, where multiple large tables are joined, or where the query graph has cycles. For example, Yaveroğlu et al. [37] have recently discovered that the structure of a complex network can be characterized by counting various patterns in the graph. Each pattern, called a graphlet, represents a small graph. It could be, for example, a triangle, or the complete graph K_5. The frequencies of graphlets in the network represent an important statistics for analyzing the network's structure. However, computing these patterns is computationally expensive. Most graphlets have cycles, and involve 5-10 self-joins on the network data.

In this paper, we describe a system that can compute efficiently complex join queries, including queries with cyclic joins, on a massively parallel architecture. We build on two lines of work that introduce a novel parallel [5, 8, 9], and a novel sequential [23, 33] algorithm respectively. While the former has been studied only theoretically, the latter is in use in the LogicBlox DBMS. Our first contribution is to empirically evaluate these two recent algorithms *together*, compared to the traditional ones, explaining when and why they are better. Then we describe two new key contributions that allow the parallel and the sequential algorithm to be deployed efficiently in parallel systems.

All traditional engines[1] compute conjunctive queries using a tree of join operators. It is well known that, if a query is cyclic, then query plans can be highly suboptimal, no matter what join order one chooses. If one computes the query $R(x, y) \bowtie S(y, z) \bowtie T(z, x)$, which lists all triangles, as a sequence of two join operators, then the size of the intermediate join is much larger than that of the final answer, because there are typically many more paths of length two than triangles. This was not considered to be a major issue in traditional engines, because cyclic queries were rare. But modern data analytics engines must support such queries frequently, and they require new approaches.

Recently, Ngo et al. [23] and Veldhuizen [33] have described novel sequential algorithms that compute a query with multiple joins in one shot, avoiding the computation of intermediate results. These are sequential algorithms, and their runtime has been proven to be worst-case optimal, meaning that it is bounded by the largest possible output that the query can produce for inputs of a given size. Both algorithms require the data to be preprocessed. For parallel computation, Afrati and Ullman [5] have described an algorithm that computes any multi-join query in a single communication round. Beame at al [8, 9] refined this algorithm, calling it HyperCube, and performed a theoretical analysis proving that it is optimal. However, the optimality criterion described by Beame et. al. is not practical, because it assumes that the available servers can be partitioned into sets with fractional number of servers.

We start by performing an empirical evaluation of the above new sequential and parallel algorithms and compare them to standard reshuffling and join computation methods. In experiments on the Twitter and Freebase datasets, we find that conjunctive queries with large intermediate results can execute up to 8x faster when using a HyperCube shuffle than a traditional one dimensional shuffle. They also transmit up to 98 percent less data. Furthermore, Tributary join, our implementation of LFTJ based on sorted relations, further cuts runtimes by up to 80 percent and CPU times by up to 71 per-

[1]With the exception of Eddies [7].

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2750545.

cent, when used in conjunction with a HyperCube shuffle. We then consider practical aspects of both algorithms. For the HyperCube algorithm, we design a new approach to optimize the number of server shares per variable, which always results in an integral number of shares, thus overcoming a key limitation of prior work [8, 9]. We demonstrate empirically its performance; for example we show that it cuts the maximum amount of data per worker in half compared with a naïve application of existing theoretical methods (Figure 11), and that its workload per server is never larger than 1.06 times the theoretically optimal load. For the sequential multi-join, we describe Tributary join, our implementation of the LFTJ's API [33]. LogicBlox' implementation of LFTJ stores each database relation in a B-tree. In our setting, data preprocessing is not possible, because the multi-join is performed after the reshuffling step; instead, Tributary join simply sorts the relations and operates on arrays instead of B-trees. We then describe a novel optimization method for choosing the variable order, and validate it empirically; for example, we show that our Tributary join optimization algorithm can cut runtimes by an order of magnitude compared with an unoptimized execution of the operator (Table 2).

To summarize, we make the following contributions:

1. We perform an empirical evaluation of the HyperCube shuffle and the Tributary join *together* (Sec. 3).
2. We present a practical algorithm to compute an optimal configuration for the HyperCube algorithm (Sec. 4).
3. We describe a new optimization algorithm for choosing the variable order of the Tributary join (Sec. 5).

2. BACKGROUND

In this section, we present the two theoretical building blocks behind our efficient join query evaluation approach: hypercube shuffle [5] and a new sequential multiway join operator that we call *Tributary join*, our implementation of the API in LFTJ [33].

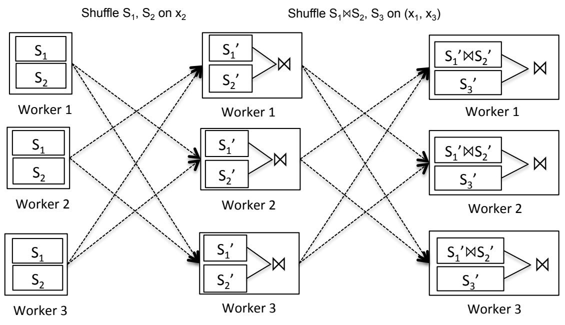
2.1 HyperCube Shuffle

We consider conjunctive queries, denoted with the following Datalog rule, where each $\overline{x}_i$ represents a set of variables.

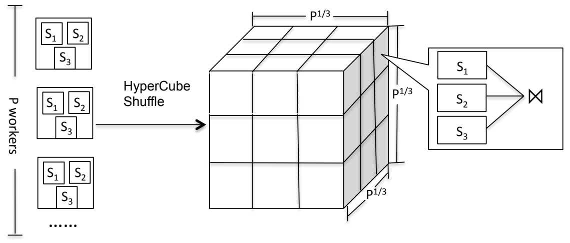
$$q(x_1, \ldots, x_k) = S_1(\overline{x}_1), \ldots, S_l(\overline{x}_l) \qquad (1)$$

Our goal is to compute the query on a distributed architecture, using p servers connected by a network. We assume that the data is initially partitioned uniformly on the p servers, for example using a hash function, or round robin, and we aim to balance the computation evenly among the servers. The traditional way to compute the query is to first construct a query plan, consisting of several joins, then evaluate the query by computing one join at a time, using a partitioned hash-based algorithm. This requires a number of communication rounds at least equal to the depth of the query tree (joins on different branches can be evaluated in parallel).

Afrati and Ullman [5] have described an algorithm to compute any full conjunctive query in a single communication round. While their description is for MapReduce, it applies equally well to our setting where we know the number of servers (simply identify one server with one reducer). Write p as a product of k factors, $p = p_1 \cdot p_2 \cdots p_k$, then organize the p servers in a hypercube with k dimensions, where the size of dimension i is p_i. In other words, each server can be uniquely identified by a point in the hypercube $[p_1] \times \ldots \times [p_k]$. The algorithm works as follows. During the first (and only) communication round, each server holding some fragment of S_j will send every tuple $S_j(\overline{x}_j), 1 \leq j \leq l$ as follows:



(a) Traditional parallel query plan



(b) HyperCube shuffle-based parallel query plan

Figure 1: Traditional and HyperCube shuffle-based parallel query plan for query $T(x_1, x_2, x_3) = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_1)$

1. For every $x_i \in Var(S_j)$, set the ith coordinate of the target server as $h_i(x_i)$ (h_i is a hash function chosen independently for x_i). $Var(S_j)$ represents the variables in $\overline{x}_j$.
2. If the coordinate in a dimension, let's say, mth, is undefined, then we do not set any constraints on the coordinate of mth dimension of target server.

In other words, the algorithm "knows" the coordinate $h_i(x_i)$ of the destination server, for all i such that $S_j(\overline{x}_j)$ contains the variable x_i; for all the other coordinates, it will simply replicate the data.

We use an example to show how this algorithm works. For query $T(x_1, x_2, x_3) = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_1)$, consider organizing the servers into a 3-D cube with dimension sizes $p_1 = p_2 = p_3 = p^{\frac{1}{3}}$. Each server is uniquely identified by three coordinates (i, j, k), where $1 \leq i, j, k \leq p^{\frac{1}{3}}$. Then a tuple $S_1(x_{1_a}, x_{2_a})$ from S_1 is sent to servers with coordinate $(h_1(x_{1_a}), h_2(x_{2_a}), \star)$. Similarly, a tuple $S_2(x_{2_b}, x_{3_b})$ from S_2 is sent to $(\star, h_2(x_{2_b}), h_3(x_{3_b}))$ and a tuple $S_3(x_{3_c}, x_{1_c})$ from S_3 is sent to $(h_1(x_{1_c}), \star, h_3(x_{3_c}))$. The load per server is $(|S_1| + |S_2| + |S_3|)/p^{2/3}$. Figure 1b shows the parallel query plan for this query using HyperCube shuffle.

The difficult question is how to determine the numbers $p_1, p_2, \ldots, p_k$, called *shares*, in order to optimize the load per server. Afrati and Ullman model this as a non-convex optimization problem, which is difficult to solve. Beame et al. [8, 9] model it as a linear optimization problem, and establish a tight connection between the optimal shares and a fractional edge packing of the query hypergraph. This problem is easily solvable but leads to a fractional solution for the shares. However, in practice, we want $p_1, \ldots, p_k$ to be integers. If we use the method presented in [9], and simply round the fractional share to integer, the workload may become sub-optimal. For example, for the previously mentioned query $T(x_1, x_2, x_3) = S_1(x_1, x_2), S_2(x_2, x_3), S_3(x_3, x_1)$, if $|S_1| = |S_2| = |S_3| = m$, the optimal sizes of dimensions will be $p_1 = p_2 = p_3 = p^{\frac{1}{3}}$. If $p = 63$, theoretically, the workload of each server is $3m/p^{2/3} \approx 0.19m$. But we cannot let to $p_1 = p_2 = p_3 = 63^{1/3}$ in real world. Rounding down to nearest integer (let $p_1 = p_2 = p_3 = 3$) will increase the workload of each server to $3m/9 \approx 0.33m$. The cost of rounding is non-negligible. On the other hand, the theoretical optimum does provide some interesting insights. For example, if $|S_1| \ll |S_2| = |S_3| = m$, then the optimal shares in [9] are $p_1 = p_2 = 1$, $p_3 = p$, which corresponds to hash-partitioning S_2, S_3 on the variable x_3 using p servers, and broadcasting S_1 to all servers.

One advantage of the HyperCube algorithm is that it is more resilient to data skew than a binary join. To see this, consider a standard, hash-partitioned join computing $S_1(x, y), S_2(y, z)$ on p servers. To avoid skew, all nodes y must have degree $\leq m/p$ in both S_1 and S_2: otherwise, if a node y has a larger degree[2], then all tuples having value y will be hashed to the same server, and its load will exceed m/p. In contrast, the HyperCube algorithm can tolerate degrees up to $m/p^{1/3}$, because every value x, y, or z is hashed into only $p^{1/3}$ buckets [9].

2.2 Tributary Join

The HyperCube algorithm only delivers the data to the right servers in one communication round; after that, the servers still need to compute the entire query locally, on a fragment of the database. To take advantage of the HyperCube communication method we need to couple it with an evaluation algorithm that can compute the entire query more efficiently than one join at a time; such algorithms have been discussed recently in the literature.

The first worst case optimal, multiway join algorithms for sequential query computation were the NPRR algorithm [24] and the Leapfrog Triejoin (LFTJ) [33]; a concise, unified presentation is given in [25, (Algorithm 3)]. We implemented our own version of this family of algorithms, by following the API of LFTJ; we call our algorithm Tributary Join (TJ).

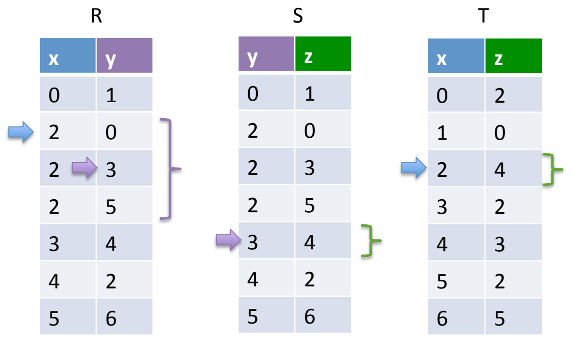
LFTJ was introduced by LogicBlox, and their implementation assumes that each relation is preprocessed and stored as a B-tree. In our setting preprocessing is not possible, because the relation fragments are available only after reshuffling. Instead, in TJ, we sort the relations, then implement the API over the sorted relations; notice that sorting on the fly is cheaper than computing a B-tree on the fly. We explain TJ next.

We present the Tributary join algorithm through the example shown in Figure 2, illustrating the query `Q(x,y,z):-R(x,y), S(y,z), T(z,x)`. Just as LFTJ, the Tributary join is based on sort-merge join. Both algorithms fix a global ordering on all join variables: $A_1 \prec A_2 \prec \ldots \prec A_k$, and assume each relation is sorted lexicographically according to this attribute order. In the example, the global variable order is $x \prec y \prec z$. Thus, R is sorted on x, y, S on y, z and T on x, z.

Given the sorted input, Tributary join starts by scanning all relations on the first join variable, x in the example. Here, the algorithm proceeds just as merge join. It scans the input until it finds matching values in all relations that contain the variable. In the example, the first such value is $x = 2$.

Once such a value $x = v$ is found, the algorithm simply computes, recursively, the residual query $Q' = Q[v/x]$. To see the recursion in our example, notice that the two relations that contain x are R and T, hence the "residual" relations are $R_{x=2}$ and $T_{x=2}$. The algorithm computes the residual query $Q'(y, z) =$

[2]Some parallel hash join algorithms detect the heavy hitters and treat them specially, to avoid skew.



x	y
0	1
2	0
2	3
2	5
3	4
4	2
5	6

y	z
0	1
2	0
2	3
2	5
3	4
4	2
5	6

x	z
0	2
1	0
2	4
3	2
4	3
5	2
6	5

Figure 2: Tributary join example

$R_{x=2}(y), S(y, z), T_{x=2}(z)$. Notice that the residual relations are continuous sub-arrays of the larger arrays, so, for the recursive call the algorithm simply adjusts the start and endpoints in these arrays. During the recursive call it scans the next join variable, y, until it finds the first common value in $R_{x=2}(y)$ and $S(y)$, which is $y = 3$, then proceeds recursively again, by scanning over z, to find a common value in $S_{y=3}(z)$ and $T_{x=2}(z)$; upon finding the value 4, it outputs $(2, 3, 4)$, and advances z, etc.

We briefly compare a B-tree v.s. array-based implementation of the LFTJ API. The main API function is `seek(v)`, which fetches the next value `v'` of the current attribute A_i s.t. `v'` $>$ `v`: in a B-tree this can be computed in amortized time $O(1)$, while our implementation uses a binary search on the remaining part of the array at a cost per operation of $O(\log n)$. Thus, TJ is at most a factor $\log n$ slower than LFTJ, and, in particular, it is also worst case optimal (up to $\log n$). In practice, the dominating cost of TJ is given by the sorting phase (which, as explained, is unavoidable), hence our choice to use a sorted array instead of a B-tree, because sorting is cheaper than computing a B-tree.

Both NPRR and LFTJ fix a variable order. While worst case optimality holds for any variable order, in practice the runtime can be improved significantly by optimizing this order. This optimization problem may sound superficially related to join ordering, but it is in fact quite different; for example all variable orders are linear, while join trees may be bushy. Some variable orders correspond to join trees, for example, considering the query `A(x,y,z,p):-R(x,y),S(y,z),T(z,u), K(u,v)`, the variable order $y \prec z \prec u$ corresponds to $((R \bowtie S) \bowtie T) \bowtie K$, while $u \prec z \prec y$ corresponds to $((K \bowtie T) \bowtie S) \bowtie R$, but the variable orders $z \prec y \prec u$ and $y \prec u \prec z$ do not correspond naturally to a join plan. We discuss choosing the variable order in Section 5.

3. HYPERCUBE SHUFFLE AND TRIBUTARY JOIN IN PRACTICE

In this section, we empirically study the combination of HyperCube and Tributary join algorithms. We ask three questions: *When is HyperCube Shuffle beneficial compared to traditional one-dimensional hash-based data shuffling? When is the Tributary multiway-join algorithm preferable to a left-deep tree of binary joins? When the HyperCube shuffle and Tributary join are used together, how much does each one impact performance?*

We run all experiments on Myria [15], an open-source, state-of-the-art parallel data management system. All experiments are in a shared-nothing cluster with 16 physical machines, each machine has 4 Intel Xeon CPU E5-2430L 2.00GHz processors (6 cores per processor), 64GB DDR3 RAM and 4 7200rpm hard drives. These machines are connected by 10Gbps ethernet. We deploy 4 workers on each machine. Each worker has its own data storage (a Postgres database on a separate disk) and runs in a JVM. In the experiments, all the input relations are horizontally partitioned across the 64 workers using round-robin partitioning.

We compare empirically the following three shuffle algorithms.

(1) *Regular shuffle* hash partitions a relation on a single attribute. We use it to hash partition relations on their join attributes. The regular shuffle thus requires the use of binary joins, except in the special case of queries where all joins use the same join attributes. In the latter case, a multiway join operator can be used.

(2) *HyperCube shuffle*, as described in Section 2.1, organizes all relations following a conceptual hypercube formed by the workers. It shuffles all relations in one step with replication.

(3) *Broadcast*, keeps the largest relation in place and broadcasts *all* the other relations to all workers.

We compare two join algorithms.

(1) *Binary symmetric hash join*, which creates a hash table for each of its two inputs. When data arrives on an input, the join inserts it into a hash table and probes the other hash table for matches. Our implementation pulls data from the inputs in a round-robin fashion. If one input does not have any data, the join pulls the other input.

(2) *Tributary join*, described in Section 2.2.

We compare the performance of each type of shuffle with each type of join for a total of six configurations. Tributary join with regular shuffle becomes a binary Tributary join, which is a merge-join. This is not what Tributary join is designed for, but we include the result for completeness.

We study the impact of the shuffle and join algorithms on the query wall-clock time but also on the overall resource utilization, which includes the CPU time and network I/O, and overall load balance between workers.

We evaluate four queries on two different datasets. Q1 and Q2 is on a subset of Twitter social network data. The relation is 2 column table, each tuple of which represents a follower-followee relationship. This relation has $1,114,289$ tuples in total. Q3 and Q4 is on Freebase knowledge base, which is originally represented in <*subject, predicate, object*> triples. We partition the Freebase knowledge base to multiple relations according to the predicate. Table 1 [3] shows the relations that we use in Q3 and Q4.

Relation	schema	number of tuples
ObjectName	(object_id, name)	59,324,337
ActorPerform	(actor_id, perform_id)	1,100,844
PerformFilm	(perform_id, film_id)	1,094,294

Table 1: Relations from Freebase

3.1 Query 1 (Q1) on Twitter

The first query lists all directed triangles in the Twitter dataset. Because the query joins three copies of the Twitter dataset, we use subscripts to distinguish them in the text. We use datalog notation for queries.

```
Twitter(x,y,z) :-
  Twitter_R(x,y), Twitter_S(y,z), Twitter_T(z,x)
```

This query performs two joins. It first joins the Twitter dataset with itself to find all follower-followee pairs that are two hops away from each other. The second join narrows down which of these sets of vertices also forms a triangle. Observe that the output from the first join is much larger than either the input or output of this query. A query with a large intermediate result is precisely the type of query that should benefit from the HyperCube shuffle and Tributary join combination. Figures 3 shows the performance of this query in all six configurations of shuffles and joins. Figure 3a shows that, as expected, the runtime for this query is lowest for the HyperCube shuffle and Tributary join configuration. Three factors contribute to this low runtime. Compared to the other configurations, less data is shuffled during query processing (lower total network IO), the data is distributed more evenly (less skew), and fewer intermediate tuples are produced during query processing (lower CPU cost). We consider each of these factors in turn.

[3]We pushed selection down, thus selections like ObjectName(actor_id, "Joe Pesci") etc. can be considered as only containing very few tuples.

shuffle	tuples sent	producer skew	consumer skew
R(x, y) ->h(y)	1,114,289	1	1.35
S(y, z) ->h(y)	1,114,289	1	1.72
RS(x, y, z) ->h(z)	50,862,578	20.8	1
T(z, x) ->h(z)	1,114,289	1	1.01
Total	54,205,445	N.A.	N.A.

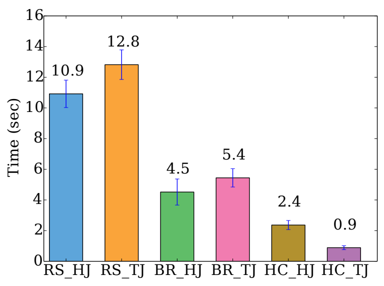
Table 2: Load balance with regular shuffles in query Q1

shuffles	tuples sent	producer skew	consumer skew
HCS R(x, y)	4,457,156	1	1.05
HCS S(y, z)	4,457,156	1	1.05
HCS T(z, x)	4,457,156	1	1.05
Total	13,371,468	N.A.	N.A.

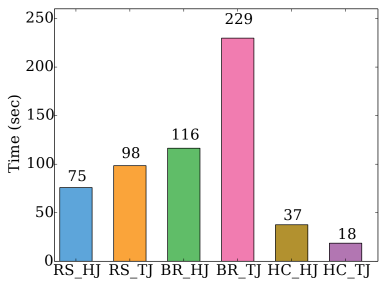
Table 3: Load balance with HyperCube shuffles in query Q1

Data shuffling network IO: A key benefit of the HyperCube shuffle is that intermediate join results need not be shuffled at the expense of replicating the input data. Overall, when the size of intermediate join results is large, the HyperCube shuffle reduces the amount of data shuffled. Figure 3c shows the total amount of data shuffled. While regular shuffle sends 54 million tuples over the network, HyperCube shuffle sends only approximately 13 million tuples, four times less. The regular shuffle does a hash partition of $Twitter_R$ and $Twitter_S$ on y first. The total amount of tuples that is sent in this step is only the number of tuples in $Twiter_R$ and $Twitter_S$. However, after the first join, the intermediate result becomes large (more than 50 million tuples). This intermediate result needs to be re-shuffled. Using HyperCube shuffle, since we are using a 64 workers configuration, we let the workers form a $4 \times 4 \times 4$ cube. Each relation ($Twitter_R, Twitter_S$ and $Twitter_T$) is replicated 4 times. Broadcast is the least efficient for this query. It shuffles 143 million tuples because it replicates the Twitter dataset twice ($Twitter_S$ and $Twitter_T$) on all workers. Since this query is a self-join, a further optimization could be only broadcast Twitter data once. But this will not affect the overall performance result, according to Table 5, the cost of the join dominates the overall runtime.

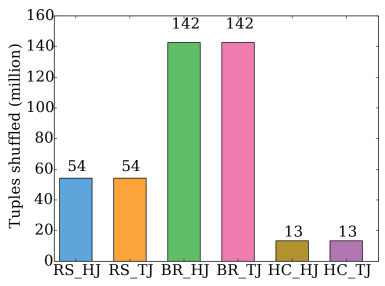
Data shuffling load balance: Interestingly, even though broadcast shuffle has the highest network IO cost (Figure 3c) and total CPU cost (Figure 3b), the query runtime (Figure 3a) is much lower than with a regular shuffle. The reason is a better overall load balance or conversely much less skew in data distribution. There is no skew in broadcast since it keeps $Twitter_R$ partitioned across workers and replicates the entire $Twitter_S$ and $Twitter_T$ to each worker. As we explained, the HyperCube shuffle is more tolerant to data skew, because it partitions every value into 4 buckets instead of 64. The skew in regular shuffle becomes a serious bottleneck. As shown in Table 2, when shuffling $Twitter_R$ and $Twitter_S$, we need to send a tuple to a consumer worker based on the hash value



(a) Wall clock time



(b) Total CPU time



(c) Number of tuples shuffled

Figure 3: Triangle query (Q1)

shuffle	tuples sent	producer skew	consumer skew
Broadcast S(y, z)	71,314,496	1	1
Broadcast T(z, x)	71,314,496	1	1
Total	142,628,992	N.A.	N.A.

Table 4: Load balance with broadcast shuffles in query Q1

operator(s)	total time	contribution of local join
BR_TJ: TJ(R, S, T)	1m 22s	19%
BR_TJ: all sorts	5m 11s	73%
BR_HJ: Join R and S	1m 52s	39%
BR_HJ: Join RS and T	2m 38s	54%

Table 5: Operator time in local join in query Q1

of y. As a social network, the degrees of twitter nodes follows a Power-Law distribution [12], since we are only hashing on a single column, the skew factor (ratio between the maximum load and the average load) computed at the consumer are 1.35 and 1.72 respectively. When we join the shuffled $Twitter_R$ and $Twitter_S$ locally, the skew becomes much worse and reaches 20.8 because the skew factors are "multiplied". Skew affects the wall clock time significantly when using the regular shuffle. Figures 3a and 3b show that regular shuffle (with either type of join) takes more wall clock time but less CPU time compared with broadcast (using the same join algorithm). This shows the impact of skew.

CPU cost of data joining: Comparing HyperCube shuffle with pipelined hash join (HC_HJ) and HyperCube shuffle with Tributary join (HC_TJ), we can observe that HC_TJ is more efficient both in wall clock time and CPU time. The reason is twofold: With HC_HJ, although the shuffle cost of the intermediate result ($Twitter_R$ joins $Twitter_S$) is avoided, the system still needs to generate the intermediate results in the pipeline and join it with $Twitter_T$. So for the query, using Tributary join also improves the efficiency significantly by reducing the CPU cost of the join operation.

Comparing broadcast shuffle with pipelined hash join (BR_HJ) and broadcast shuffle with Tributary join (BR_TJ), we observe that, interestingly, BR_HJ performs better than BR_TJ. To know why this happens, we take a closer look at the time spent on the local join phase in Table 5. We observe that with BR_TJ, the join itself only takes 19% of the time, the bottleneck is sorting. In BR_TJ, we need to sort $Twitter_R/64$, the entire $Twitter_S$ and the entire $Twitter_T$, which is expensive. Recall that with HC_TJ, we only need to sort $Twitter_R/16$, $Twitter_S/16$ and $Twitter_T/16$.

Summary: Experiments with Q1, a query that has a large intermediate join result, show that the HyperCube shuffle outperforms the two other types of shuffling because it yields the lowest communication cost while also achieving a good load balance. Tributary join outperforms a tree of binary joins because it avoids generating a huge number of intermediate tuples. However, Tributary join needs to be coupled with HyperCube shuffle since it requires all the input relations shuffled and naive broadcast causes large relations to be sorted in this join.

3.2 Query 2 (Q2) on Twitter

The second query lists all cliques with 4 vertices in the Twitter dataset:

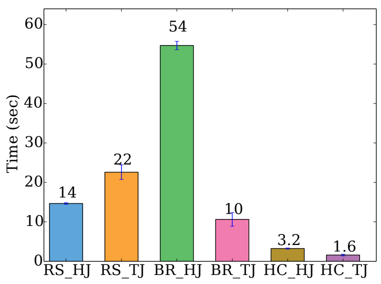
```
Twitter(x,y,z,p):-
  Twitter_R(x,y), Twitter_S(y,z), Twitter_T(z,p),
  Twitter_P(p,x), Twitter_K(x,z), Twitter_L(y,p)
```

This query is a 6-way self-join of the Twitter dataset. Intuitively, this query can be seen as first computing triangles xyz just as Q1 and then finding a vertex p that connects to all three vertices of the triangle to create the clique. This query has thus the same key property as Q1: A left deep tree of joins will have large intermediate results but a small final result. In Q2, two joins produce more data than they consume. The other joins, reduce the amount of data. This query demonstrates the performance of various shuffle and join configurations on a more complex query.

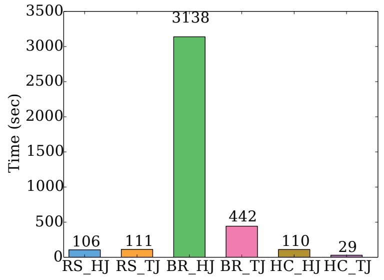
Figures 4 shows the performance of this query in all six configurations of shuffles and joins. Figure 4a shows that the HyperCube shuffle and Tributary join configuration with $2\times4\times2\times4$ dimension sizes has, again, the lowest runtime. The same three factors impact performance: amount of data shuffled, load balancing of shuffles and the CPU cost.

Data shuffling network IO: The total amount of data shuffled, shown in Figure 4c, follows the same trends as in Q1. The HyperCube shuffle is most efficient because it does not shuffle any intermediate results. Broadcast is most expensive because it replicates the large input relations multiple times. In this experiment, we also observe the same load balance trends as in Q1 but omit the results due to space constraints.

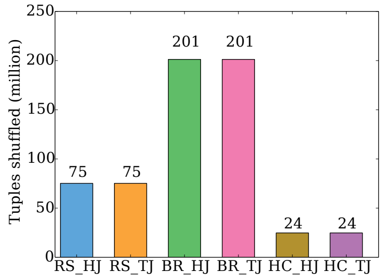
CPU cost of data joining: Figure 4b shows that although the CPU time in different shuffle and join configurations follows the same trend as in Q1, the CPU time of broadcast shuffle with hash join (BR_HJ) is as high as $30x$ that of the regular shuffle with hash join (RS_HJ). In Q1, the CPU time of broadcast shuffle with hash join is less than $2x$ that of RS_HJ configuration's. In Q2, the join operators in BR_HJ contribute 2,084 seconds total CPU time, while the join operators in RS_HJ contribute only 156 seconds. The reason is that every join in the BR_HJ plan has at least one input relation that is at least $64x$ larger than the same relation in RS_HJ, since the latter partitions all relations across the 64 workers used in the experiments. This large difference in CPU time causes the



(a) Wall clock time



(b) Total CPU time



(c) Number of tuples shuffled

Figure 4: Clique query (Q2)

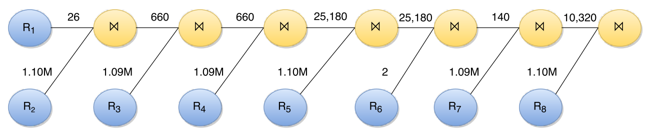


Figure 5: Query plan for Freebase query 1 (Q3) using regular shuffle, with numbers of tuples shuffled.

runtime of BR_HJ to be larger than the runtime of RS_HJ (Figure 4a). This trend is different than in Q1, although BR_HJ still has the advantage of better load balance between workers.

When comparing the same shuffle algorithm but different joins (HC_HJ v.s. HC_TJ and BR_HJ v.s. BR_TJ), we observe that the configurations with Tributary Join have both lower runtime and lower CPU time. Comparing with the result of Q1, we can see BR_TJ is better than BR_HJ in Q2 while it was the opposite in Q1. The reason is that the size of intermediate result in the local hash join pipe line of Q2 is much larger than Q1, the cost saving on TJ dominates the disadvantage of sorting broadcasted relations.

Summary: Experiments with Q2 show that in a 6-ways join query plan with large intermediate results, HC_TJ is the best shuffle and join configuration in terms of query runtime, total CPU time across all workers, and total data shuffled. One interesting observation is, in both Q1 and Q2, RS_HJ and BR_HJ is trading off data load balance in shuffled data for total CPU time. In Q1, BR_HJ configuration has lower runtime compared with RS_HJ because the load balancing factor dominates. In Q2, the gap in CPU time increases dramatically, thus RS_HJ configuration has a lower total runtime.

3.3 Query 3 (Q3) on Freebase

This query is an example knowledge extraction query in the Freebase knowledge base. It asks for the set of all cast members from films starring both Joe Pesci and Robert de Niro. This is the first example query (query 1) provided with the Freebase database.

```
CastMember(cast):-
  ObjectName(a1, "Joe Pesci"), ActorPerform(a1, p1),
  PerformFilm(p1, film), ObjectName(a2, "Robert De Niro"),
  ActorPerform(a2, p2), PerformFilm(p2, film),
  PerformFilm(p, film), ActorPerform(p, cast)
```

This query is an acyclic query with 7 joins. Figure 5 shows a left deep query plan for the query.

The edges show the cardinality of each input and intermediate relation. The query selects "Joe Pesci" and "Robert De Niro" from the $ObjectName$ relation and joins their actor ids through the $ActorPerform$ and $PerformFilm$ relations to find the films where both actors played. The query then finds the cast members in these films using two more joins.

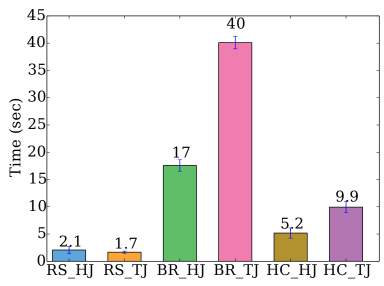
Figures 6 shows the performance of this query in different configurations of shuffles and joins. Interestingly, as Figure 6a shows, for this query the regular shuffle and Tributary join configuration has the lowest runtime followed closely by the regular shuffle with hash join. We observe that these two configurations both shuffle less data and have lower total CPU times than the other configurations.

Data shuffling network IO: We examine the sizes of the shuffled relations for the three types of shuffle algorithms. Figure 6c shows that the regular shuffle transmits the least amount of data across the network (7.18 million tuples) among all shuffle algorithms. This is different from Q1 and Q2. The query plan for the regular shuffle and hash join configuration is shown in Figure 5. As the figure shows, the first join reduces the amount of data in the pipeline dramatically. For subsequent joins, the amount of data in the pipeline remains much smaller than the large input relations. HyperCube shuffle needs to shuffle more data since it replicates the base data when populating the hypercube. We see this result even though we use a hypercube optimizer (described in Section 4) to find the configuration that shuffles the least amount of data among all possible hypercube configurations. HyperCube shuffle forms a hypercube that has 6 dimensions using 64 servers. HyperCube shuffle transmits 105.4 million tuples. The broadcast shuffle transmits 7 input relations to all 64 workers (it partitions the largest 8th relation across the workers), thus it needs to shuffle 351.01 million tuples.

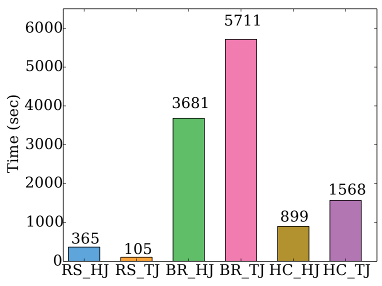
Data shuffling load balance: The data shuffling skew is not a factor that influences the query runtime in this query. Although we still can observe data shuffling skew with the regular shuffle, this skew only happens when shuffling relatively small intermediate results as shown in Figure 5. Considering that these small imbalanced relations are then joined locally with large, balanced relations, the effects of skew on runtime are negligible. The data shuffling in HyperCube shuffle and broadcast are well balanced.

CPU cost of data joining: The CPU cost of this query is mostly affected by the choice of shuffling strategy. The total CPU time with the regular shuffle is lowest, which is not surprising, as we have analyzed before, the amount of data that is joined using regular shuffle is much less than with HyperCube shuffle. For the same reason, the configurations using HyperCube shuffle have lower CPU times than those using broadcast.

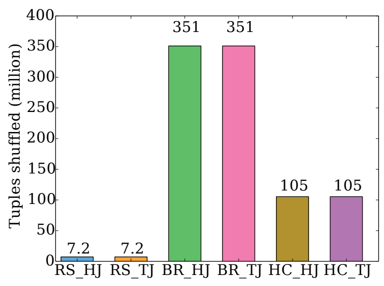
Comparing the join algorithms, we observe that RS_TJ uses less CPU time than RS_HJ. In contrast, HC_HJ uses less CPU time than HC_TJ. Finally, BR_HJ uses less CPU time than BR_TJ. The key factor that determines which of the two join algorithms is most efficient (TJ or HJ) is the amount of data that is being sorted. With RS, when using TJ, only $1/64$ of the base data needs to be sorted.



(a) Wall clock time



(b) Total CPU time



(c) Number of tuples shuffled

Figure 6: Freebase query 1 (Q3)

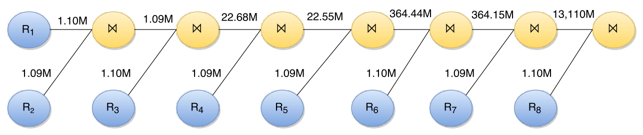
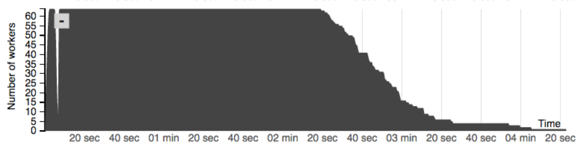
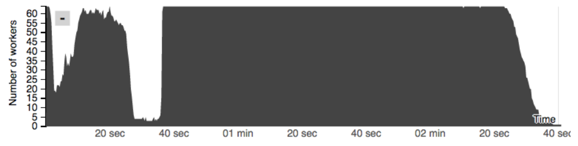


Figure 7: Query plan for Freebase query 2 (Q4) using regular shuffles, with numbers of tuples shuffled.



(a) HC_TJ



(b) BR_TJ

Figure 8: Worker utilization

With HC, when using TJ and our HC configuration algorithm, $1/4$ of the original data needs to be sorted. In BR, the entire original data needs to be sorted. Sorting a large amount of data is less efficient than executing a tree of hash-join operators, but when a small amount of data is sorted, a single TJ outperforms a tree of HJ operators.

Summary: Experiments with Q3 show that, in an acyclic query with small intermediate results, regular shuffle outperforms the other two shuffle strategies. It saves both network IO and CPU costs.

3.4 Query 4 (Q4) on Freebase

The next query finds all pairs of actors or actresses who have co-starred in at least two different movies. This is the second example query provided with the Freebase database:

```
ActorPairs(a1, a2):-
  ActorPerform(a1, p1), PerformFilm(p1, f1),
  PerformFilm(p2, f1), ActorPerform(a2, p2),
  ActorPerform(a2, p3), PerformFilm(p3, f2),
  PerformFilm(f2, p4), ActorPerform(p4, a1), f1>f2.
```

This query is cyclic and contains eight joins. Figure 7 shows the query plan for Q4 when using a regular shuffle. As the query plan shows, in contrast to Q3 and similar to Q1 and Q2, this query has large intermediate relations.

Figures 9 shows the performance of this query in different configurations of shuffles and joins. Figure 9a shows that HC_TJ and BR_TJ outperform other configurations. Due to the large amount of intermediate result, RS_HJ is the least efficient configuration in terms of runtime and RS_TJ fails because it runs out of memory.

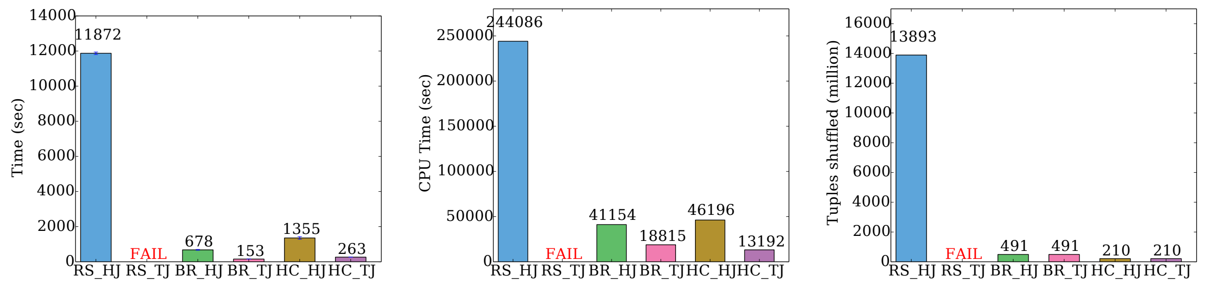
Data shuffling network IO: Figure 6c shows the total amount of data shuffled in this query. The regular shuffle transmits significantly larger amounts of data (13,893 million tuples), compared with broadcast (491 million tuples) and HyperCube shuffle (210 million tuples). If we examine the regular shuffle in detail (Figure 7), the sizes of intermediate results keep increasing with each additional join. Before the last join, the intermediate result contains 13,100 million tuples. HyperCube shuffle shows its advantage in keeping a much lower communication cost. However, we need to construct an 8-D cube out of 64 workers, the replication factor is high. As a result, the total amount of data shuffled is only a little less than half that of the broadcast shuffle. The savings that come from using HyperCube shuffle compared with broadcast are less significant than for queries with a smaller number of joins (e.g. Q1).

Data shuffling skew: The regular shuffle algorithm has the worst data shuffling skew among all shuffle algorithms. The heaviest skew (skew factor = 9.31) happens when shuffling the intermediate result to the last join, which is also the largest relations shuffled. Skew contributes to the slow runtimes of configurations using regular shuffle. The skew of HyperCube shuffle and broadcast are both negligible.

CPU cost of data joining: Comparing configurations using the same join operator but different shuffle algorithms, clearly regular shuffle is the worst shuffle algorithm for this join in terms of CPU time because of the large volume of intermediate results. Comparing HC_HJ and BR_HJ, BR_HJ uses less total CPU time and leads to a lower total runtime. A very interesting comparison is between HC_TJ and BR_TJ. Although HC_TJ uses less CPU time than BR_TJ, HC_TJ has a large runtime. We profile the worker utilization during the query execution (Figure 8), we can see there are long tail workers in HC_TJ. Although hypercube shuffle shuffles the input relations without much skew, the differences in computation time is still visible.

Comparing configurations using the same shuffle algorithm but different join operators (HC_HJ v.s. HC_TJ and BR_HJ v.s. BR_TJ), due to the large amount of intermediate result generated by pipelined hash join, Tributary join is much more efficient in both total CPU time and runtime.

Summary: Experiments with Q4 show that, when there is large intermediate results, Tributary join is the choice of join operator. HyperCube shuffle will have the advantage of cheaper communication cost. Although if the replication factor is high, broadcast



(a) Wall clock time (b) Total CPU time (c) Number of tuples shuffled

Figure 9: Freebase query 2 (Q4)

Query	# Tables	# Join Variables	Cyclic	Input size (millions)	RS size (millions)	HC size (millions)	RS Skew (max)	Time(RS_HJ)/Time(HC_TJ)	Config. with lowest runtime
Q1	3	3	Y	3.3	54	13	20	12	HC_TJ
Q7	4	2	N	0.31	0.24	0.24	2.6	1.3	HC_TJ
Q5	4	4	Y	4.4	1841	36	29	12	HC_TJ
Q6	5	4	Y	5.5	74	17	29	13	HC_TJ
Q2	6	4	Y	6.6	75	25	16	9.2	HC_TJ
Q8	6	6	Y	2.4	54	60	3.5	0.44	RS_HJ
Q3	8	7	N	6.6	7	106	2.8	0.21	RS_TJ
Q4	8	8	Y	8.8	13893	210	9.3	45	BR_TJ

Table 6: Summary of extended evaluation. Queries grouped by the best query plan and listed by increasing number of joined tables. RS size is the total number of tuples shuffled when using a regular shuffle. HC size is the same measurement for the HyperCube shuffle.

may have a comparable or even better runtime by having less skew in computation cost among each worker.

3.5 Additional Queries

We evaluate four additional queries using the Twitter and Freebase datasets to better cover the space of plans with different numbers of joins, input sizes, and intermediate result sizes. We present the detailed queries and associated graphs in Appendix A. Table 6 summarizes the experimental results across all eight queries. The table lists the queries by increasing number of tables joined.

As the table shows, all six cyclic queries have large intermediate result sizes (shown by a large number of tuples shuffled by the regular shuffle). Except for Q8, the regular shuffle has high skew (9.3 or higher) and, therefore, for all queries (except Q8), HC_TJ outperforms the regular shuffle. Query Q4 is interesting because here the Broadcast plan outperforms HC_TJ.[4] The reason is that the query has 8 join variables, requiring an 8-dimensional hypercube, leading to significant data replication. The broadcast plan shuffles more than twice as much data, but it has a somewhat lower skew, as we showed earlier in Figure 8, and is thus the fastest plan for this query. The only cyclic query where regular shuffle has little skew is Q8; here HC_TJ uses a 6-dimensional hypercube, and therefore reshuffles a relatively large amount of data (60M tuples, for an input of 2.4M), which is comparable with the amount of intermediate results of the regular shuffle (54M). All this causes regular shuffle to be twice as fast as HC_TJ.

The regular shuffle plans for acyclic queries Q3 and Q7 produce intermediate relations of size comparable to that of the input. The behavior of HC on the two queries differs. For Q3, it uses a 6-dimensional hypercube, resulting in significant data reshuffling overhead, and the traditional, regular shuffle plan is faster. Q7 requires only a 2-dimensional hypercube (a square). In fact, this query joins three large relations on a single join attribute and a small relation on a different attribute. The optimal configuration of shares is 1×64, which causes the small relation to be broadcast and the three large relations to be hash-partitioned on their join attribute. This plan enables the HyperCube shuffle to avoid any overhead and outperform the traditional plan thanks to a more even load distribution. We present more details in the Appendix.

[4]Broadcast was not the best plan for any other query, and for that reason we omit it from the table.

Summary: Our evaluation comparing the new combination of HyperCube and Tributary join (HC_TJ) with traditional hash join (RS_HJ) and broadcast join shows that there is no overall best query plan. Large intermediate results or significant data skew lead HC_TJ to outperform the others, sometimes by large amounts. Moreover, HC_TJ can automatically detect when to use broadcast. However, for complex queries HC_TJ uses a high-dimensional cube, which increases the amount of data shuffled, canceling the advantage gained by not having to compute intermediate results. Similarly, when the intermediate results are small, and there is no significant skew, the traditional hash join leads to fastest query runtimes.

We observe that the HC_TJ outperforms BR_TJ in almost every query with the exception of Q4. As we discussed in Section 3.1 (*Q1*), since HC reduces the size of data shuffled to workers, it improves the sorting time of TJ, which is the bottleneck in many queries.

3.6 Comparison with Semijoin Plans

The purpose of a semijoin reduction is to remove from input tables the "dangling" tuples, which do not contribute to any answer in the output. Only acyclic queries admit full semijoin reductions [36], meaning that with a fixed number of semijoins one can completely remove all dangling tuples from all tables; for a cyclic query, there is no bound on the number of semijoin reductions. We implement the distributed semijoin reduction from recent work [4] (details in the Appendix) and evaluate it on the acyclic queries in our workload, which are Q3 and Q7.

For Q3, the semijoin plan shuffles 2.29 million tuples from the projected tables and 6.57 million tuples from the input tables compared with 7.18 million tuples for the regular shuffle. In this query, the use of the semijoin thus provides little benefit, which is reflected in the runtime. This plan takes 4.127 seconds, which is slower than RS_HJ (the best plan for this query). The semijoin is slower than the regular shuffle because the plan has a longer pipeline: 2.5x more operators than RS_HJ. Similar to the regular shuffle, the semijoin plan outperforms the HyperCube shuffle on this query.

For Q7, the semijoin plan shuffles 0.14 million tuples from the projected input tables (containing only the distinct values from the joined columns) and 0.24 million tuples from the input tables. The HyperCube shuffle transmits 0.24 million tuples over the network. For this query, the semijoin thus only adds overhead. As a result, the runtime is $1.427sec$, the second slowest among all configurations.

Thus, in our workload, the standard semijoin reduction did not improve the runtime: the extra cost of additional rounds of communication canceled all savings from removing the dangling tuples. We note that, unlike traditional distributed join processing $R \bowtie_{A=B} S$, where R is stored in one server and S on a second server and the semijoin reduction sends only the attribute $S.B$ to the first server, in our setting every relation is distributed, and to perform the semijoin one needs to re-shuffle R in addition to re-shuffling $S.B$: the cost of the semijoin is higher, and it requires more complex queries, or more dangling tuples to be beneficial.

The exact values reported in the above experiments are tied to the DBMS that we used in the evaluation. Other engines may differ in their efficiency to shuffle data or execute operators. The relative differences in the values across algorithms, however, are likely to remain similar. Other engines could include yet another set of parallel join evaluation algorithms. The conclusions that we draw are limited to the algorithms that we tested.

3.7 Scalability

The analysis in the previous section shows that, for queries with large intermediate results but a small final result (Q1, Q2, and Q4) the combination of HyperCube Shuffle and Tributary Join outperforms traditional plans that use either a Regular Shuffle or a Broadcast together with a tree of binary join operators. For Q4, HyperCube shuffle and broadcast achieve comparable performance.

However, there is a second critical aspect of performance in shared-nothing parallel systems: scalability. How well does the HyperCube Shuffle and Tributary Join combination scale compared with traditional plans?

In this section, we study the scalability of the HyperCube Shuffle and Tributary Join combination compared with a traditional plan that uses a left-deep tree of binary join operators and a regular shuffle. We use **Query 1** (triangle query), a 3-way cyclic join to show the scalability of combining HyperCube Shuffle and Tributary Join.

In this experiment, we vary the number of workers that process the query from two to 64. We run the query using either HyperCube Shuffle with a Tributary join or a traditional plan using regular shuffles and hash-based joins. Figure 10 shows the results.

Figure 10a shows the wall clock time of Query 1 using hash join with regular shuffle and using Tributary join with HyperCube shuffle. Tributary join with HyperCube shuffle scales well while hash join with regular shuffle scales poorly beyond four workers for this query. The main reason for the improved scalability is better load balance. As discussed in Section 3.1, the regular shuffle experiences a significant degree of skew.

It is not obvious that the HyperCube shuffle should scale well because of the way it replicates data: the larger the cluster, the larger the replication factor the more data must be shuffled and processed by the Tributary Join. Figure 10b and Figure 10c show the detailed resource utilization for the query. We plot the ratio of the total number of tuples shuffled compared with a 2-worker configuration. Indeed, the total amount of shuffled data grows almost linearly with the cluster size. In spite of this growth, however, the total runtime decreases as shown in Figure 10. To see why, Figure 10c shows the total wall clock time per worker for both the data sorting time and the actual join time. As the figure shows, the time per worker drops significantly with cluster size because each worker processes less data, even though the workers as a whole process more data.

4. HYPERCUBE OPTIMIZATION

In this section we present practical optimizations of the HyperCube Shuffle. Recall that the algorithm factorizes the number of servers into a product of shares, $p = p_1 p_2 p_3 \cdots$, and the theoretically optimal solutions lead to fractional shares and, as we have seen, rounding down leads to significant performance loss, because it leaves many servers unused. For example, consider the 4-clique query in Sec. 3.2, and suppose we have $p = 15$ servers: the theoretically optimal shares are $p_1 = p_2 = p_3 = p_4 = 15^{1/4} \approx 1.96$: rounding down leads to $p_1 = p_2 = p_3 = p_4 = 1$, which means that we use only one server, and have no parallelism at all.

Problem statement: The HyperCube shuffle distributes the input data for a join query across the machines in the cluster. We choose the optimization objective to be *minimizing the maximum amount of data (measured as the number of tuples) assigned to a single worker*. Since the HyperCube shuffle enables a single round of communication during query evaluation, the runtime of a query is determined by the runtime of the slowest worker, which can be approximated by the amount of data processed by that worker.

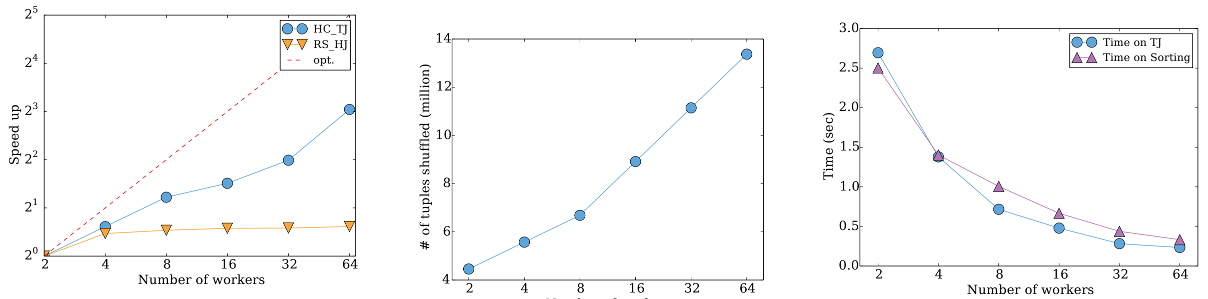
We discuss here four approaches. The first is simply based on rounding down:

Naïve Algorithm 1: Rounding down: We use the linear programming problem described by Beame et al. [9], which give the optimal, fractional shares; then round down each share. As we have discussed, this rounding down approach may be highly non-optimal, especially when p is small or the number of joins is large.

The next two approaches are based on virtual servers, which we call *cells*. We will use N to denote the number of physical machines, and $M \geq N$ the number of cells; when they are the same, then we denote $p = M = N$. The HyperCube algorithm will be designed for M cells (virtual servers), i.e. the shares satisfy $M = M_1 M_2 M_3 \cdots$ However, there are only N physical servers, so we have to define a many-to-one mapping from M to N. Two important problems arise: (1) how do we choose the number of cells M? Intuitively, a larger value for M will reduce the effect of rounding error. But increasing M leads to more data reshuffling: for example, the data reshuffled by the triangle query with shares $M^{1/3} \cdot M^{1/3} \cdot M^{1/3}$ is $(|S_1| + |S_2| + |S_3|)M^{1/3}$ (Sec. 3), hence more data is reshuffled as we increase M. (2) How do we map cells to physical servers? Some data items are sent to multiple cells, and therefore we should aim to place those cells in the same physical server. For example, in the triangle query $S_1(x, y)$ is sent to all cells $(i, j, 1), (i, j, 2), \ldots$, where i, j are the hash values of x and y. We want to map these cells to a set of physical servers of smallest possible size: if they were mapped to the same physical server then we avoid data replication.

Naïve Algorithm 2: Many cells per worker with random allocation: To increase the degree of parallelism and to improve load balance, an alternate approach is to have M ($M \gg N$) virtual HyperCube cells; by increasing the number of cells, we reduce the loss due to rounding down. Then we assign these cells randomly to N workers. This approach contains two steps:

1. Compute a (possibly) non-integer solution using the LP-based approach [8] using M and Q as inputs. Then round the solution down. This will generate a HyperCube configuration with M_1 cells and $M_1 \leq M$ but $M_1 \gg N$.



(a) Query runtime speed up for different cluster sizes (compared with the query runtime with two workers). "opt" plots optimal, linear speed up. (b) Total number of shuffled tuples for different cluster sizes using HyperCube shuffle. (c) Per-worker Tributary join time and per-worker sort time for different cluster sizes.

Figure 10: Scalability of HyperCube Shuffle and Tributary Join combined

2. Assign M_1 cells randomly to the N physical servers.

This approach will use all N workers. However, a serious problem is that the data replication grows significantly, which conflicts with the goal of using HyperCube shuffle to minimize the amount of communicated data. We present an example in Appendix B to illustrate why random allocation increases data replication dramatically.

Naïve Algorithm 3: Many cells per worker with an optimal allocation: This approach improves the problem of random cell allocation above. As above, the approach starts by solving the linear program on a large number of virtual cells and rounding down the solution to integer hypercube dimension sizes. As second step, this approach computes an optimal cell allocation to minimize workload assigned to each server.

To compute the allocation, we formulate the HyperCube cell allocation problem as a combinatorial optimization problem. We use the state of the art answer set solver [1] with symmetric breaking optimization. However, for the set of queries and parallel database configurations in our experiments ($N = 64$, $M \gg N$), the time to compute the optimal allocation is much beyond the query running time. For example, for $N = 64$ and $M = 100$, the optimal cell allocation for $Q1$ takes longer than 24 hours to compute on a laptop using Intel I5 CPU.

Finally, our fourth approach is the following.

A Practical and Efficient Solution. Our fourth approach is a simple yet efficient solution to the hypercube configuration problem. Algorithm 1 shows the details. The key idea is to keep one cell per worker as in prior work [8], but identify the hypercube configuration that uses as many physical workers as possible. The approach is simple: the algorithm performs a breadth-first-search to enumerate all the integral HyperCube configurations with a number of workers ($nw(c)$) less than the number of physical machines N. For each configuration c, we compute its maximum workload, $workload(c)$ (*i.e.*, total amount of allocated data) for a single worker. The algorithm chooses the HyperCube configuration with the minimal workload. Note that the optimal configuration may not necessarily use all N physical machines. For example, considering the 4-clique query in Sec. 3.2 on $N = 15$ physical machines, the algorithms searches configurations $p_1 \times p_2 \times p_3 \times p_4 \leq 15$: it will consider $1 \times 1 \times 5 \times 3$ and $5 \times 1 \times 3 \times 1$, etc, but also $2 \times 2 \times 3 \times 1$ which uses only 12 machines, etc.

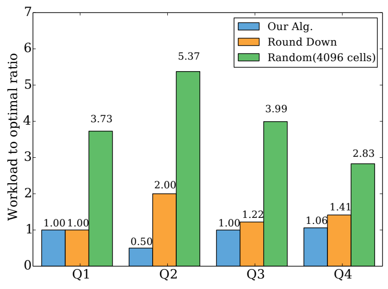
In this algorithm, if more than one configuration yields the minimal workload, we choose the configuration with more even dimension sizes (e.g. $2 \times 2 \times 2 \times 2$ is better than $1 \times 4 \times 1 \times 4$). That is to reduce possible skew during HyperCube shuffle. For example, assuming both x and y in relation `A(x,y)` are join attributes, the algorithm selects $d_x = 2, d_y = 2$ rather than $d_x = 1, d_y = 4$ (let d_x be the dimension size which corresponds x). Although `A` will be partitioned into 4 partitions in both configurations, in configuration $d_x = 1, d_y = 4$, `A` is shuffled based on only hash valued of x. While in configuration $d_x = 2, d_y = 2$, `A` is partitioned based on both x and y, which is more resilient to possible skew in either attribute value.

Algorithm 1 HyperCube Configuration Algorithm

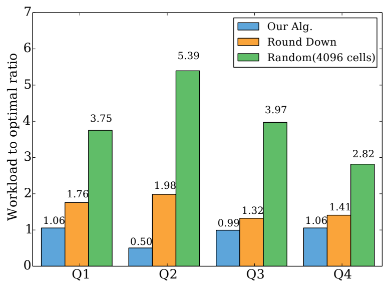
```
Input: N                                        ▷ Number of workers
 1: $wl \leftarrow \infty$                                   ▷ best workload
 2: $C \leftarrow null$                    ▷ configuration of best workload
 3: for all integral HC configuration $c$ s.t. $nw(c) \leq N$ do
 4:     if $workload(c) < wl$ then
 5:         $wl \leftarrow workload(c)$
 6:         $C \leftarrow c$
 7:     else if $workload(c)$ == $wl$ and $max(dim_c)$ < $max(dim_C)$ then then
 8:         $wl \leftarrow workload(c)$
 9:         $C \leftarrow c$
10:     end if
11: end for
```

Evaluation: We evaluate the effectiveness of all the above hypercube configuration algorithms using the queries from Section 3. For each query, we first compute the optimal workload using the linear programming solver GLPK [2] and the problem formulation proposed in prior work [8]. Then we compare this optimal workload to that produced by each of the above algorithms ($workload/opt.$). We compare our algorithm to the single cell per worker algorithm with rounding down (Round Down) and the many cells per worker algorithm with random allocation.

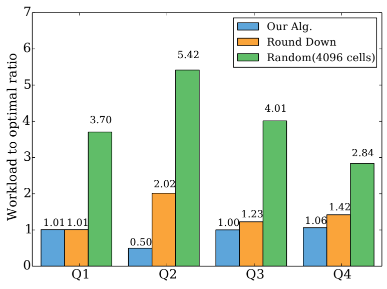
Figure 11 shows the workload to optimality ratio of the three hypercube configuration algorithms for $N = 64, 63$, and 65. We observe that for all three cluster sizes, our algorithm performs better than the other two approaches. For $Q2$, when $N = 63$ and $N = 65$, our algorithm achieves a workload ratio below 1. Our solution thus outperforms even the non-integer solution from prior work [8]. The reason for this performance result is that the non-integer solution computed by the LP solver is only optimal within a constant factor. In contrast, our approach does not have this limitation. The round down approach performs well only if the LP solver outputs an integer solution (e.g. $Q1, N = 64$). The random allocation approach has the worst performance due to the high data replication rate.



(a) N=64



(b) N=63



(c) N=65

Figure 11: Compare hypercube configuration algorithms

Interestingly, even though our approach performs an exhaustive search over all possible hypercube configurations, the algorithm runtime is low. For a cluster with 64 workers and queries with even large numbers of joins (Q1 through Q4), the algorithm computes the hypercube configuration in under 100msec, which makes it practical.

5. ATTRIBUTES ORDER OPTIMIZATION

Tributary join, like the Leapfrog Triejoin whose API it implements, requires the optimizer to chose a global order of all attributes that participate in the join. Prior work on Leapfrog Triejoin [33] did not address this optimization problem.

We introduce a cost model for the Tributary join algorithm to estimate the cost of that operator for a given variable order. We validate experimentally the effectiveness of this cost model in selecting good variable orders.

Optimizing the variable order is important because, although Tributary join is worse case optimal given any variable ordering, in practice, this "worse case" can be far from optimal.

5.1 Tributary Join Cost Model

We want to estimate the cost of Tributary join using different variable orderings, so that the optimizer can choose an optimal variable order for a given query plan. Consider the following query:

$$Q(\overline{x}) : -R_1(\overline{x}_1), \dots, R_k(\overline{x}_k)$$

In Tributary join, a attribute global order is defined as a function $\phi, J \rightarrow \mathbf{Z}^+$, where J is the set of joined attributes. For example $\phi(A_i) = 1$ means A_i is the first attribute in the global variable order.

We assume that the following commonly used statistics are available: the cardinality of each relation that participates in each join ($|R_1|, \dots, |R_k|$), the number of unique values of each variable in each relation ($V(R_i, x_i)$) and the number of unique "prefix" values ($V(R_i, (\overline{p}))$ in each relation. The prefix values for relation R_i are the values of the prefix of join attributes that appear in R_i as per the global variable order. For example, assuming that $\bar{(x)}_i = (A_{i,1}, A_{i,2}, \dots, A_{i,j})$ is following the global variable order, the prefixes are $(A_{i,1})$, $(A_{i,1}, A_{i,2})$ until $(A_{i,1}, \dots, A_{i,j-1})$.

In the Tributary join algorithm, the most expensive step is the binary search to find the matching value for the next join variable at the beginning of each recursive merge join. In Example 2, the algorithm performs a binary search to find $S(y) = 3$, then $T(z) = 4$, then $S(y) = 5$, etc. Each binary search takes $O(\log(N))$ time. We want to estimate the cost of Tributary join by estimating the number of binary searches during the multiway join operation.

We observe that Tributary join works on one joined variable in each step. If there is a value appearing in this variable in the relations involved in this step, then the join will move on to the next variable. The size of the set intersection of the active domains of the variables involved in each step decides on how many times the binary search will be performed in that step and how many times the join will move to the next variable. For example, in Figure 2, $|R(2, y) \cap S(y)| = 2$.

$$S_{step=1} = \min_{\varphi(1) \in R_j} (V(R_j, (\varphi(1)))$$

φ is the reverse function of ϕ. $\varphi(1)$ represents the first variable in global variable order. Hence, for step one, the cost model computes the minimum number of distinct values in the first variable across all relations that have the variable.

As the Tributary join moves to the next variable, it only searches intersection within the scope defined by a certain prefix. For example, in query:

$$Q(x_1, x_2, x_3) : -R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4) \quad (2)$$

Assuming the global variable order is $x_2 \prec x_3$, the Tributary join tries to find the first value of x_2 that appears in both R_1 and R_2. Upon finding that value, for example, v_0, the Tributary join will move on to the next variable, x_3, in $R_2(x_2 = v_0)$ and R_3. We estimate the number of unique values in the "residual" relation as:

$$V_{step=i}(R_j) = \frac{V(R_j, \overline{p}_{i,j})}{V(R_j, \overline{p}_{i-1,j})}$$

$\overline{p}_{i,j}$ is the prefix variable vector of R_j at ith joined variable. For example, when Tributary join is joining the 2nd variable x_3 of query showed in Equation 2, we have $\overline{p}_{2,2} = (x_2, x_3)$ and $\overline{p}_{1,2} = (x_2)$. Thus, the number of unique value in residual relation size of R_2 can be estimated as $V(R_2, (x_2, x_3))/V(R_2, (x_2))$.

We still estimate the size of the intersection as the size of the smallest number of unique values in all involved (residual) relations.

$$S_{step=i, i>1} = \min_{\varphi(i) \in R_j} (\frac{V(R_j, \overline{p}_{i,j})}{V(R_j, (\overline{p}_{i-1,j})}) \quad (3)$$

Combining the estimate of the set intersection and the estimate of the residual relation size, we can estimate the cost of Tributary join using the estimate of the number of binary searches performed. We obtain the following recursive cost function.

$$Cost_{step \geq i}(Q) = S_{step=i} + S_{step=i} \times S_{step=i+1} \quad (4)$$

5.2 Evaluating the cost model

In general, accurate estimation of the cost of query plans with multiple joins is a hard problem. Ioannidis and Christodoulakis [16] show that the error in the estimation of result sizes grows exponentially with the arity of the join. In this section,

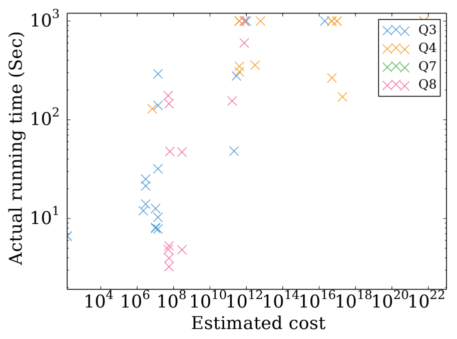


Figure 12: Estimated cost and actual query runtime

query	average runtime in seconds (random)	runtime in seconds (best order from cost model)
Q3	155.22	12.62
Q4	864.75	129.35
Q7	0.072	0.060
Q8	26.39	0.23

Table 7: Query runtime with random attribute order and best order from cost model

we evaluate whether our simple cost model suffices to identify good variable orders.

We evaluate the effectiveness of the cost model by executing a query using different variable orders. Since an n-way multiway join can have $n!$ different variable orders, we randomly select 20 variable orders from **Q3**, **Q4**, **Q7** and **Q8** [5] separately. We terminate the query if the query takes more than $1,000$ seconds. Figure 12 shows the scatter plot of the real query runtimes against the estimated runtimes for all queries. Although the estimates are not perfect, the estimated cost and the actual runtime are positively correlated (The correlation coefficients are 0.658 for Q3, 0.216 for Q4, 1 for Q7, and 0.932 for Q8). Table 7 shows the Tributary join runtime (on a single machine, using pre-shuffled data) when using either the variable order with the lowest estimated cost or a random variable order. Choosing a good variable order using our cost model improves the query runtime by a factor of up to $10x$ in this experiment.

6. RELATED WORK

Sequential multiway join algorithms The sequential evaluation of multiway joins has extensively been studied in the database community. Most prior work focused on the optimization of query trees consisting of two-way joins. Schneider and DeWitt [29] studied the tradeoffs between using left-deep, right-deep and bushy query trees to organizing the joins. They also discussed the strategies to decluster a query tree given memory limitations. Some recent developments, mostly from the theoretical community, have lead to an entirely new approach to processing multi-join queries, initiated by Aterias, Grohe and Marx (hence AGM) [6] who proved a tight bound on the maximum result size of a full conjunctive query. Ngo et al. [23, 22] described a *worst-case optimal* algorithm (called NPRR) whose runtime is guaranteed to be bounded by the AGM bound. Veldhuizen [33] described another worst-case optimal (up to a log factor) algorithm called Leapfrog Triejoin (LFTJ), which had been developed and deployed by LogicBlox. A good, unified overview of these two algorithms can be found in [25]. We used an implementation of LFTJ in our evaluation and developed a cost model to optimize its join attribute order.

[5]We only examine 2 attribute orders in **Q7** since it only has two join attributes.

Efficient join algorithms in parallel systems Lu et al. [18] studied the parallel execution of multiway joins in a shared-memory, multi-processor system. Recently, more work has focused on shared-nothing parallel systems. Zhang et al. [38] studied how to efficiently decompose multiway Theta-join into multiple MapReduce jobs. Elsedy et al. [11] proposed an adaptive repartition algorithm to automatically balance workload among shared-nothing computing nodes when evaluating binary joins. Bruno et al. [10] evaluated different join algorithms with skew handling techniques in the Microsoft SCOPE system. Vemuri et al. [34] studied efficient join and aggregation algorithms in MapReduce under the assumption that the data is partitioned into user defined data units. Polychroniou et al. [27] proposed track join, which tries to make the best tradeoff between CPU cost and network cost. Zhou et al. [39] proposed a cache-conscious MapReduce star join algorithm, which utilizes local memory more efficiently. Phan et al. [26] proposed a filtering optimization using Bloom filter to avoid communicating unnecessary data for joins in MapReduce.

Parallel multiway join algorithms and data partitioning. Ganguli, Silberschatz, and Tsur described a parallel datalog query engine that pioneered the idea of using redundancy in order to reduce the communication cost [13, Sec.7]. and was generalized by Afrati and Ullman [5] to an algorithm for computing any conjunctive query in a single MapReduce step. Beame, Koutris, and Suciu [17, 8, 9] introduced a new formal model for parallel computation where the number of servers is given explicitly, and analyzed the communication cost for single-round, and multi-round query evaluation algorithms. Afrati et al. [4] developed parallel semi-join algorithms to explore tradeoffs between the number of rounds of communication and the communication/computation cost, which are generalizations of Yannakais semijoins [36]. In this paper, we address the HyperCube size-rounding problem to use it in practice and evaluate its effectiveness in a wider range of queries, which are not necessarily full conjunctive queries, and in conjunction with the Tributary join algorithm.

Communication cost is usually affected by the data partition layout. Stöhr et al. [31] studied the allocation of data warehousing data based on the star schema and assumed that bitmap indices were used. Rao et al. [28] and Nehme et al. [21] studied the optimal partitioning design for expected workloads in parallel database systems.

7. CONCLUSION

In this paper, we showed how to convert recent theoretical algorithms for conjunctive query into a practical query evaluation approach on a shared-nothing execution engines. We made three contributions: We studied when HyperCube shuffle [5, 14, 8] together with Tributary multiway join algorithm (an implementation of the Leapfrog Triejoin API [33]) outperform traditional query plans. Second, we developed an algorithm to compute the optimal hypercube configuration for a given query and a cluster of arbitrary size. Finally, we developed a cost model for the new Tributary join algorithm that enables the optimization of that operator.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF grant IIS-1247469. We thank D. Moritz for his query-profiling visualization tool [20], D. Halperin, J. Wang, and S. Xu for discussions, and the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] Clasp. `http://potassco.sourceforge.net/`, 2014.

[2] Glpk. `https://www.gnu.org/software/glpk/`, 2014.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[4] F. N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[5] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[6] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748, 2008.

[7] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 261–272, 2000.

[8] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284, 2013.

[9] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 212–223, 2014.

[10] N. Bruno, Y. Kwon, and M. Wu. Advanced join strategies for large-scale distributed computation. *PVLDB*, 7(13):1484–1495, 2014.

[11] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.

[12] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[13] S. Ganguly, A. Silberschatz, and S. Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992.

[14] S. Ganguly, A. Silberschatz, and S. Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992.

[15] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the myria big data management service. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 881–884, 2014.

[16] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991.*, pages 268–277, 1991.

[17] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 223–234, 2011.

[18] H. Lu, M. Shan, and K. Tan. Optimization of multi-way join queries for parallel execution. In *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings.*, pages 549–560, 1991.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[20] D. Moritz, D. Halperin, B. Howe, and J. Heer. Perfopticon: Visual query analysis for distributed databases. In *Computer Graphics Forum (EuroVis), Cagliari, Italy*, volume 34, 2015.

[21] R. V. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1137–1148, 2011.

[22] H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra. Beyond worst-case analysis for joins with minesweeper. In *PODS*, pages 234–245, 2014.

[23] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[24] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48, 2012.

[25] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[26] T. Phan, L. d'Orazio, and P. Rigaux. Toward intersection filter-based optimization for joins in mapreduce. In *2nd International Workshop on Cloud Intelligence (colocated with VLDB 2013), Cloud-I '13, Riva del Garda, Trento, Italy, August 26, 2013*, page 2, 2013.

[27] O. Polychroniou, R. Sen, and K. A. Ross. Track join: distributed joins with minimal network traffic. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1483–1494, 2014.

[28] J. Rao, C. Zhang, N. Megiddo, and G. M. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 558–569, 2002.

[29] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*, pages 469–480, 1990.

[30] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[31] T. Stöhr, H. Märtens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 273–284, 2000.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a

petabyte scale data warehouse using hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005, 2010.

[33] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In N. Schweikardt, V. Christophides, and V. Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, pages 96–106, 2014.

[34] S. Vemuri, M. Varshney, K. Puttaswamy, and R. Liu. Execution primitives for scalable joins and aggregations in map reduce. *PVLDB*, 7(13):1462–1473, 2014.

[35] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 13–24. ACM, 2013.

[36] M. Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94, 1981.

[37] â. N. Yaveroğlu, N. Malod-Dognin, D. Davis, Z. Levnajic, V. Janjic, R. Karapandza, A. Stojmirovic, and N. Pržulj. Revealing the hidden language of complex networks. *Scientific Reports*, 4, 2014.

[38] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.

[39] G. Zhou, Y. Zhu, and G. Wang. Cache conscious star-join in mapreduce environments. In *2nd International Workshop on Cloud Intelligence (colocated with VLDB 2013), Cloud-I '13, Riva del Garda, Trento, Italy, August 26, 2013*, page 1, 2013.

APPENDIX

A. ADDITONAL QUERIES EVALUATED

Query 5: Twitter Rectangle

The fifth query lists all directed rectangles in the Twitter data that we used in Section 3.1.

```
Rectangle(x, y, z, p):-
    Twitter_R(x, y), Twitter_S(y, z),
    Twitter_T(z, p), Twitter_K(p, x)
```

This query is a 4-way self-join of the Twitter dataset; it can be thought of as an intermediate between Query 1 and Query 2. Figures 13 shows the performance of this query in all six configurations. Figure 13a shows that the configuration using HyperCube shuffle and Tributary join leads to lowest runtime.

Data shuffling network IO: The total amount of data shuffled, shown in Figure 13c does not follow exact the trend in Q1 and Q2. HyperCube shuffle is the most efficient since it does not shuffle any intermediate results. Regular shuffle becomes the most expensive shuffle algorithm since the intermediate result is huge. All the 2-hops (the intermediate result of the first join) and 3-hops (the intermediate result of the second join) need to be shuffled.

CPU cost of data joining: The CPU cost of this query is affected by both shuffle algorithm and local join algorithm. Comparing shuffle algorithms while fixing join algorithm, the configurations using HyperCube shuffle have better CPU cost than these using regular shuffle and broadcast. Comparing join algorithms while fixing shuffle algorithm, we observe that Tributary join is always better than hash join since the hash join still suffers from the large intermediate result, even pipelined in memory.

Summary: In this 4-way self join on Twitter data, HC_TJ is still the most efficient configuration both in wall clock time and CPU cost. An interesting fact observed here is that, as the size of the intermediate result goes to even larger (compared with Q1), broadcast is shown to be better than regular shuffle.

Query 6: Twitter Two Rings

The sixth query lists 4-vertice subgraph patterns which consist of two back to back triangles.

```
Two_Rings(x,y,z,p):-
    Twitter_R(x,y),Twitter_S(y,z),Twitter_T(z,p),
    Twitter_P(p,x),Twitter_K(x,z)
```

This query is a 5-way twitter self join. It can be thought as Q5 with an extra join to constrain the size of the final output. However, a better join order for Q6 would be firstly constructing a triangle and do the other two joins afterward. Since we assume there is a state of the art optimizer, we use the later join order in all the pipelined configurations.

Figures 14 shows the performance of this query in different configurations of shuffles and joins. The configuration using HyperCube shuffle and Tributary join has lowest wall clock time, as shown in Figure 14a.

Summary The wall clock time, CPU cost and shuffle size of this query exhibit a similar trend as in Q2. Interestingly, comparing Q1, Q2, Q5 and Q6, which are 3 to 6 way self joins on Twitter, we can observe:

- HC_TJ shows best performance over all these queries.
- As size of the intermediate result of pipelined join differs, the relative advantage of regular shuffle and broadcast changes. If the size of the intermediate result of pipelined join is extremely large (e.g. Q5), broadcast could be better than regular shuffle.
- Using HyperCube shuffle, Tributary join is always better than pipelined join in these queries. However, using Broadcast, Tributary join can be slightly worse than hash join due to the sorting cost on relatively large input relations.

Query 7: Freebase Query

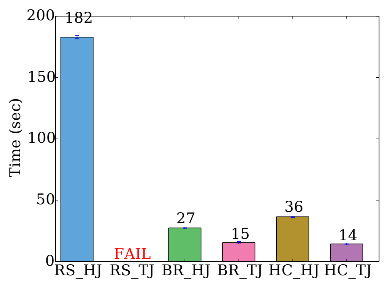
The seventh query is an example of Freebase knowledge exploration query. It finds all actors who win Oscar Award in 90s. Table 8 shows the sizes of relations joined in Q7.

```
OscarWinners(a):-
    ObjectName(aw, "The Academy Awards"),
    HonorAward(h, aw),
    HonorActor(h, a),
    HonorYear(h, y),
    y>=1990 AND y<2000
```

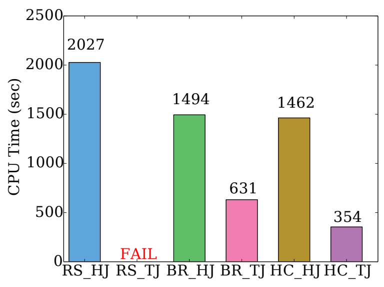
Relation	Cardinality
$\sigma_{n=\text{``}TheAcademyAwards''}(ObjectName)$ (R_1)	1
$HonorAward$ (R_2)	93,468
$HonorActor$ (R_3)	126,238
$\sigma_{1990\leq year<2000}(HonorYear)$ (R_4)	17,681

Table 8: Relations joined in Q7

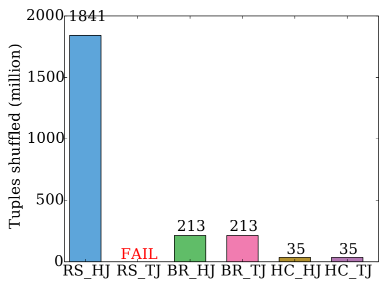
This query is an acyclic 4-way join. It can be viewed as a star join using three relations and another join on one branch of the



(a) Wall clock time

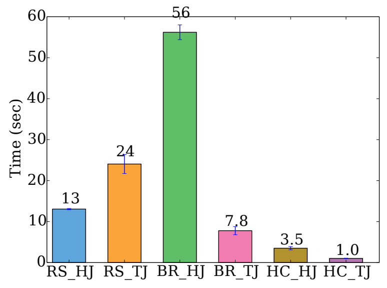


(b) Total CPU time

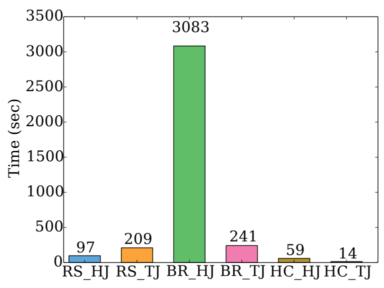


(c) Number of tuples shuffled

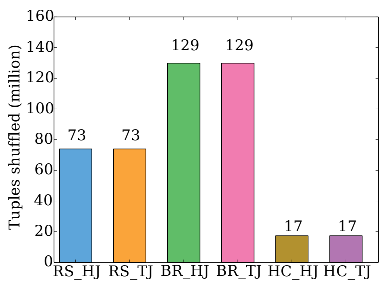
Figure 13: Twitter Rectangle (Q5)



(a) Wall clock time



(b) Total CPU time



(c) Number of tuples shuffled

Figure 14: Twitter Two Rings (Q6)

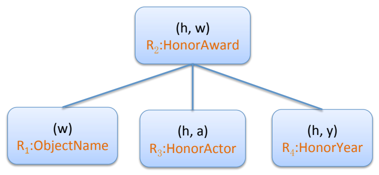


Figure 16: A generalized hypertree decomposition (GHD) of Q7

star. Figures 15 shows the performance of this query in different configurations of shuffles and joins. HC_TJ configuration has the lowest runtime.

Data shuffling network IO & load balance Broadcast shuffles significant larger amount of data than regular shuffle and HyperCube shuffle. Regular shuffle and HyperCube shuffle both shuffle slightly more tuples than the input data. The size of the intermediate result is very small in regular shuffle. In HyperCube shuffle, since the the sizes of input relations are unbalanced, the HyperCube size is 1×64, which means the very small relation `ObjectName(aw, "The Academy Awards")` is broadcasted and the other tree relatively large relations is shuffled without duplication. This shows how HyperCube shuffle adapts to skewed input sizes.

Although the amount of data being shuffled (thus being joined in workers) is close for HyperCube shuffle and regular shuffle. The maximum destination skew (defined by max/average number of tuples received) of regular shuffle is 1.7. The same metric of HyperCube shuffle is 1.15. This explains why HC_TJ has the best runtime.

Query 8: Freebase Query

The eighth query is another Freebase knowledge exploration query, which finds the pairs of actor and director appearing in two films. As reported in Section 3.3, `ActorPerform` and `PerformFilm` have 1.1 million and 1.09 million tuples respectively. `DirectorFilm` has 0.19 million tuples.

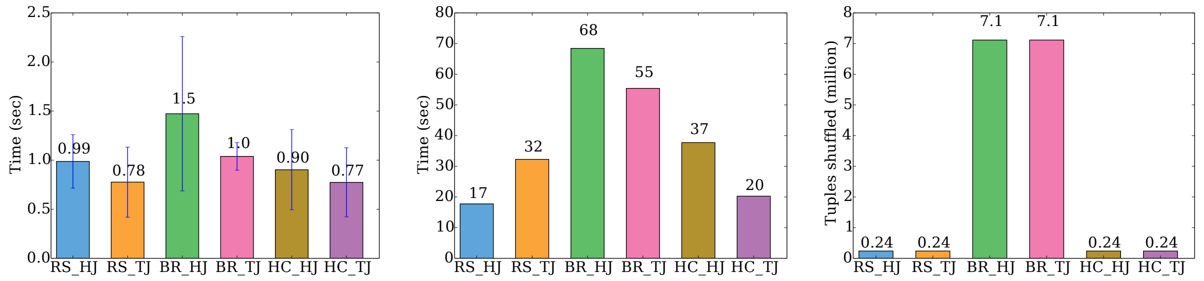
```
ActorDirector(a, d):-
    ActorPerform(a, p1), ActorPerform(a, p2),
    PerformFilm(p1, f1), PerformFilm(p2, f2),
    DirectorFilm(d, f1), DirectorFilm(d, f2)
```

This query is a 6-way cyclic join query. Figures 17 shows the performance of this query in different configurations. RS_HJ has lowest runtime among all the configurations. In general, this query shows a similar trend in runtime, data shuffling and CPU cost with Q3.

Semijoin Reduction Algorithm Implementation

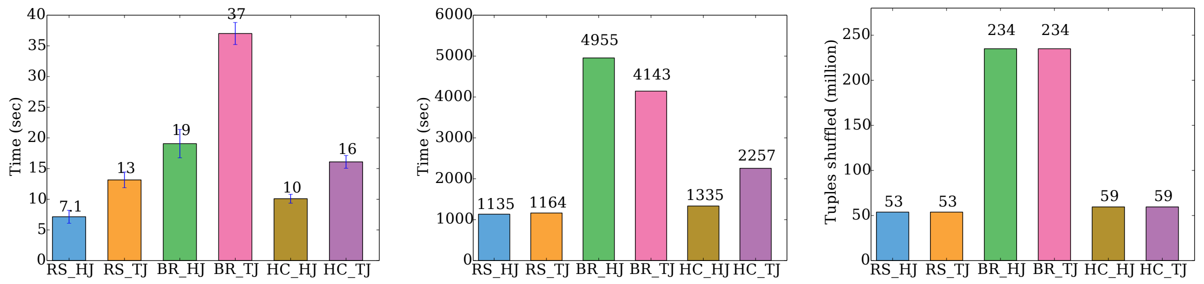
For the semijoin experiments, we implement the distributed version of Yannakakis' semijoin reduction algorithm [36] as described in the GYM paper by Afrati et al. [4]. The algorithm has three steps. The first two steps reduce the dangling tuples using semijoins. The last step joins the reduced tables. Taking **Query 7** as an example, we construct a generalized hypertree decomposition (GHD) of the query as shown in Figure 16. The semijoin can be done in the following steps (using shorter names $R_1, R_2, \ldots$ for the relations, see Figure 16):

1. **Bottom-up semijoins.** Repace R_2 with $R_2' = ((R_2 \ltimes R_1) \ltimes R_3) \ltimes R_4$.
2. **Top-down semijoins.** Repace R_1, R_3 and R_4 with $R_i' = R_i \ltimes R_2'(i = 1, 3, 4)$.



(a) Wall clock time (b) Total CPU time (c) Number of tuples shuffled

Figure 15: Freebase Query 3 (Q7)



(a) Wall clock time (b) Total CPU time (c) Number of tuples shuffled

Figure 17: Freebase Query 4 (Q8)

3. **Final joins.** Join the reduced relations $R'_1, \dots R'_4$, $R'_1 \bowtie R'_2 \bowtie R'_3 \bowtie R'_4$.

Since we do not assume that any relation is partitioned on the join attribute, we evalute $R \ltimes S$ in parallel as follows:

1. **Local preprocessing.** Get S^A by projecting S on joined keys and perform duplicate elimination.
2. **Shuffle.** Shuffle R and S^A on their join attributes.[6]
3. **Local join.** Join shuffled R and S^A.

B. AN EXAMPLE OF RANDOM HYPERCUBE CELL ALLOCATION

Example 1. Consider query `A(x, y, z, p) :- R(x, y), S(y, z), T(z, p)`. Each tuple from S will be sent to the server responsible for cell $(h(y), h(z))$. However, each tuple from R will be sent to *all* servers assigned any of the cells in the column $(h(y), *)$. Tuples from T will be sent to all servers assigned cells in the row $(*, h(z))$. Figure 18 shows a random HyperCube cell allocation with 16 HyperCube cells and 4 physical servers. As the figure shows, each physical server covers a large portion of both $h(y)$ and $h(z)$. For example, server 1 covers 7 out of 8 parts of $h(y)$ and 7 out of 8 parts of $h(z)$. Thus, although $1/4$ of S needs to be shuffled to server 1, $7/8$ of R and $7/8$ of T need to be shuffled to server 1. Similarly, $7/8$ of R, $1/4$ of S, $7/8$ of T need to be shuffled to server 2 and server 3. Entire R, $1/4$ of S and entire T need to be shuffled to server 4.

[6]If one relation is much smaller than the other ($< 1/P$, P is the number of workers), broadcasting the smaller relation is cheaper. However, for semijoins on intermediate results, it is difficult to automatically decide whether to broadcast or hash partition.

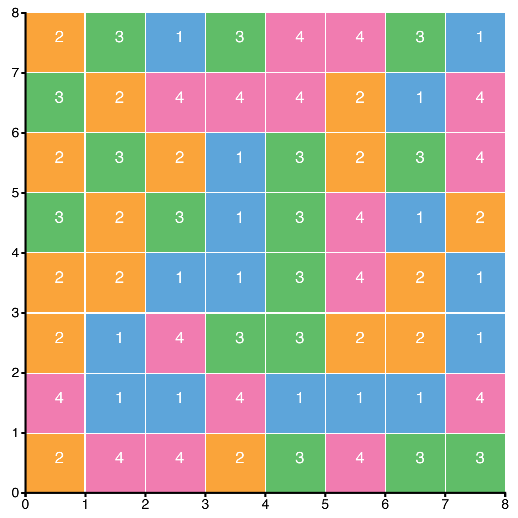


Figure 18: Example: random HyperCube cell allocation