Distributed Joins and Data Placement for Minimal Network Traffic

ORESTIS POLYCHRONIOU, WANGDA ZHANG, and KENNETH A. ROSS,
Columbia University, USA

Network communication is the slowest component of many operators in distributed parallel databases deployed for large-scale analytics. Whereas considerable work has focused on speeding up databases on modern hardware, communication reduction has received less attention. Existing parallel DBMSs rely on algorithms designed for disks with minor modifications for networks. A more complicated algorithm may burden the CPUs but could avoid redundant transfers of tuples across the network. We introduce track join, a new distributed join algorithm that minimizes network traffic by generating an optimal transfer schedule for each distinct join key. Track join extends the trade-off options between CPU and network. Track join explicitly detects and exploits locality, also allowing for advanced placement of tuples beyond hash partitioning on a single attribute. We propose a novel data placement algorithm based on track join that minimizes the total network cost of multiple joins across different dimensions in an analytical workload. Our evaluation shows that track join outperforms hash join on the most expensive queries of real workloads regarding both network traffic and execution time. Finally, we show that our data placement optimization approach is both robust and effective in minimizing the total network cost of joins in analytical workloads.

CCS Concepts: • **Information systems → Join algorithms**; **Relational parallel and distributed DBMSs**; *Data warehouses*;

Additional Key Words and Phrases: Distributed joins, network optimization, data placement

ACM Reference format:
Orestis Polychroniou, Wangda Zhang, and Kenneth A. Ross. 2018. Distributed Joins and Data Placement for Minimal Network Traffic. *ACM Trans. Database Syst.* 43, 3, Article 14 (November 2018), 45 pages.
https://doi.org/10.1145/3241039

1 INTRODUCTION

The processing power and storage capacity of a single server machine can be large enough to fit small to medium scale databases. Nowadays, servers with memory capacity of more than a terabyte are not uncommon. Packing a few multi-core CPUs on top of shared non-uniform access (NUMA) RAM provides substantial parallelism, where we can run database operations (i.e., selection scans, sort, join, and group-by) on RAM-resident data at rates of a few gigabytes per second [2, 3, 41, 42, 51, 60, 63].

This research was supported by National Science Foundation grants IIS-0915956, IIS-1422488, and a gift from Oracle Corporation. Work partly done when the first author was at Oracle Labs.
Authors' addresses: O. Polychroniou, W. Zhang, K. A. Ross, Columbia University, 500 West 120th Street, New York, NY 10027, USA; emails: {orestis, zwd, kar}@cs.columbia.edu.

0362-5915/2018/11-ART14 $15.00
https://doi.org/10.1145/3241039

Database research has also evolved to catch up to the hardware advances. Fundamental design rules of the past on how a DBMS should operate are now being revised due to their inability to scale and achieve good performance on modern hardware. Special purpose databases are now popular against the one-size-fits-all approach [56], while database accelerators for specific types of workloads [46] are the implicit manifestation of the same concept.

The advances in database design for storage and execution on modern hardware have not been met by similar advances in distributed parallel database design. When the most fundamental work on distributed and parallel databases was published [5, 19, 31], hardware advances considered fundamental today, such as multi-core parallelism in CPUs, had not yet occurred. The database literature includes techniques to speed up short-lived distributed transactions [56] by targeting distributed commit protocols. However, distributed transaction performance is dominated by network latency rather than throughput. Analytical queries where large-scale data exchange is inevitable, such as a distributed join between two large tables, remain less popular research topics or are left for data-centric generic distributed systems for batch-processing [11, 39].

The latest network technologies can be slow relative to in-memory processing. A 40 Gbps InfiniBand measured less than 3GB/s real data rate per node during hash partitioning. If done in RAM, then partitioning to a few thousand outputs runs close to the memory bandwidth [42, 51, 60]. For instance, a server with 4 8-core CPUs and 1,333MHz quad-channel DDR3 DRAM achieved a partition rate of 30–35GB/s, the memory-to-memory copy bandwidth, which is more than an order of magnitude higher than the 40 Gbps InfiniBand network. Recent work [41] achieved a hash join rate of 6.5GB/s of 32-bit key, 32-bit payload tuples on the same hardware. Such high-end hardware is common in marketed configurations for large-scale analytics.

Network optimization is important for both low-end and high-end hardware. In low-end platforms where the network is slow compared to in-memory processing, we expect the execution time to be dominated by the network transfers and any traffic reduction directly translates to faster execution. In high-end platforms, given that the network still cannot be as fast as the RAM bandwidth, end-to-end execution time is improved if the reduction in network traffic is comparable with the increase in CPU cycles.

To show how much time databases can spend on the network, we give an example of a real workload from a large commercial vendor, using a market-leading commercial DBMS. Using 8 machines connected through 40 Gbps InfiniBand (see Section 5 for more details), we found that the five most expensive queries spent ≈65–70% of their execution time transferring tuples over the network and account for 14.7% of the time required to execute the entire analytical workload that contains more than 1,500 queries. All five queries have a non-trivial query plan (4–6 joins), but spent 23%, 31%, 30%, 42%, and 43% of their total execution time on a single distributed hash join.

1.1 A New Join Algorithm for Minimal Network Traffic

A sophisticated DBMS should have available options to optimize the trade-off between network and CPU utilization. One solution would be to apply network optimization at a higher level treating the network as a less desired route for data transfers, without modifying the underlying algorithms. These approaches are common in generic distributed processing systems [11]. A second solution would be to employ data compression before sending data over the network. This solution is orthogonal to any algorithm but can consume a lot of CPU resources without always yielding substantial compression. A third solution is to create novel algorithms for database operator evaluation that minimize network communication by incurring local processing cost. This approach is orthogonal and compatible with compression and other higher level network transfer optimizations.

Grace hash join [12, 26] (we use the term *hash join* to refer to *Grace* hash join on network [12], rather than disk [26]) is the predominant method for executing distributed joins and uses hash partitioning to split the problem into shared-nothing sub-problems that can proceed locally on each node. Partitioning both tables is independent on the number of nodes but is far from network-optimal, because it transfers both tables over the network. Hash partitioning guarantees load balancing, but limits the probability that a hashed tuple will not be transferred over the network to $1/N$ on N nodes.

In this article, we first introduce track join, a novel algorithm for distributed joins that minimizes transfers of tuples across the network. The main idea of track join is to decide where to send rows on a key by key basis using information about where records of the given key are located. Track join has the following properties: it (i) is orthogonal to data-centric compression, (ii) can co-exist with semi-join optimizations, (iii) does not rely on favorable schema properties, such as foreign key joins, (iv) is compatible with both row-store and column-store organization, and (v) does not assume favorable pre-existing tuple placement. We implement track join to evaluate the most expensive join operator in the most expensive queries of real workloads. We found that track join reduces the network traffic significantly over known methods, even if pre-existing data locality is removed and all data remain compressed form throughout the join.

1.2 Optimizing the Data Placement for Analytical Workloads

To reduce the network traffic when executing distributed hash joins, existing systems co-partition frequently joined tables by their join attribute. This approach, however, is limited to one attribute per table, unless the tables are physically copied. Consequently, when a table is involved in multiple joins using different join attributes, the network cost of subsequent joins is unaffected. For example, in a workload with two joins $R \bowtie_A S$ and $R \bowtie_B T$, if we hash partition tables R and S on attribute A, then the network cost of $R \bowtie_A S$ is zero, but the network cost of $R \bowtie_B T$ is not reduced, unless attributes A and B on table R are highly correlated. Even when the two columns A and B are correlated, manual work is often required to identify such correlation and partition the data properly. Such problems are common for typical online analytical processing (OLAP) query workloads using the star schema, where joins occur between the attributes of a large fact table and multiple dimension tables and each join uses a different attribute (or set of attributes in case of composite keys) of the fact table.

The locality-sensitive property of track join gives us the opportunity of tuning the data placement to create locality on purpose. For a workload where distributed joins take a significant amount of time, track join makes it possible to optimize the network cost for each distinct key independently, providing more flexibility than explicitly co-partitioning by hashing specific join attributes. Here, we consider a typical data warehouse where the expensive joins occur between the fact table and multiple dimension tables, and study how to place the data in a way that reduces the cost of subsequent track joins with all dimension tables using different attributes per dimension.

Here, we propose a data placement optimization algorithm that reduces the network traffic of distributed joins in an OLAP workload, provided that we use track join instead of hash join. Our approach focuses on optimizing high cardinality keys over low cardinality keys, as well as more frequently selected keys over less frequently selected keys. Rather than blind horizontal partitioning, we place high cardinality, frequently selected keys together, localizing a large portion of the distributed query execution.

The data placement optimization is executed while queries are running on the cluster. The operation runs (mostly) separately from the nodes executing the queries to be transparent. We use statistics provided by queries, such as selectivity or join frequency. Once the optimized data

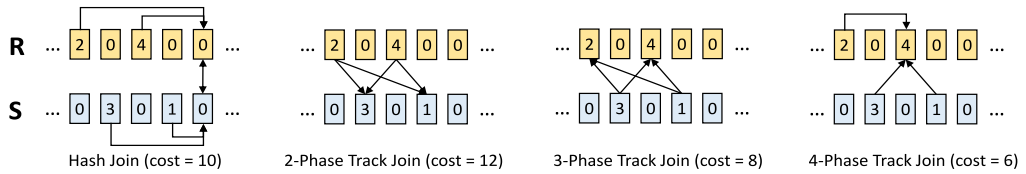


Fig. 1. Example of network transfers in hash join and track join per distinct join key. Vertically aligned R and S boxes are executed on the same node. The numbers in the boxes denote the network cost for transferring the tuples from that node.

placement has been adopted, we restart the process using more recent statistics to accomodate either a temporal shift in the tables accessed and joined, or to optimize the placement of new data inserted in the dataset.

The core idea consists of two steps. The first step clusters matching tuples to a subset of nodes, while maintaining load balancing. We create mappings from all distinct join keys to nodes and use it to place dimension table tuples. However, we still have multiple potential placements for fact table tuples, since join keys across different attributes can be mapped to different nodes. The second step finalizes the placement of fact table tuples, by picking from the potential mappings per join key combination the one that minimizes the total network cost of all distributed joins in the workload.

We evaluate our data placement algorithm using the TPC-H benchmark [9], including skew [7], and show that our approach significantly reduces the total network cost of joins across multiple dimensions.

1.3 Paper Organization

Section 2 describes the track join algorithm, presenting three variants starting from the simplest. In Section 3, we discuss query optimization costs for track join, tracking-aware hash joins, and semi-join filtering. We describe the data placement optimization in Section 4. Section 5 presents our experimental evaluation using both synthetic and real workloads. In Section 6, we discuss related work and conclude in Section 7.

2 TRACK JOIN

Formally, the problem under study is the general distributed equi-join. The input consists of tables R and S split arbitrarily across N nodes. Every node can send data to all other nodes and all links are considered to have the same network bandwidth.

The main idea of track join is to logically decompose the entire join into single key joins for each distinct join key and to schedule tuple transfers independently for each key. We present three versions of track join gradually evolving in complexity, as shown in Figure 1, along with the traditional hash join between tables R and S. In hash join, we assume the hash destination is the fifth node. For the purpose of transfer scheduling, all track join variants employ a tracking phase first to gather information on tuple distributions, which is not shown in the figure.

The first version of track join (2-phase track join) discovers network locations that have at least one matching tuple for each unique key. We then pick one side of the join and broadcast each tuple to all locations that have matching tuples of the other table for the same join key. We use the term *selective broadcast* for this process. In the 2-phase track join example in Figure 1, we selectively broadcast R to matching S tuple locations.

The second version of track join (3-phase track join) gathers not only a boolean indicator whether a node has tuples of a specific join key or not but also the total size of these tuples.

Using this information at runtime, we pick the cheapest side to be selectively broadcast, a decision taken independently for each distinct join key. In Figure 1, 3-phase track join chooses the opposite direction to 2-phase track join, since it is cheaper.

The third and full version of track join (4-phase track join) generates an optimal join schedule for each distinct key achieving minimum payload transfers, without hashing or broadcasting. The schedule migrates tuples from one table to potentially fewer nodes, before selectively broadcasting tuples from the other table. In the example above, the 4-phase track join first migrates R tuples to a single node and then selectively broadcasts S tuples, leading to minimal network cost.

Next, we describe the detailed track join algorithms using a pipelined approach. We assume three processes (R, S operating on tables R and S, respectively, and T that generates schedules) running simultaneously on every node. A non-pipelined approach is also possible (see Section 5).

2.1 2-Phase Track Join

The 2-phase track join is the simplest version of track join. The first phase is the tracking phase, and the second phase is the selective broadcast phase.

In the tracking phase, both R and S are projected to their join key and sent over the network. The destination is determined by hashing the key, in the same way that hash join sends tuples. Duplicates are redundant and are eliminated. At every destination node, a process T receives unique keys and stores them alongside the id of the source node.

In the second phase, we only transfer tuples from one table. Choosing whether to send R tuples to S tuple locations, or S tuples to R locations, has to be decided by the query optimizer before the query starts executing, similar to the traditional inner-outer relation distinction of hash join. Assume that we transfer R tuples. For a particular join key, its corresponding process T first sends messages to each location with matching R tuples. The message contains the key and a list of matching S tuples' locations. Given a list of tracked S locations, R tuples are selectively broadcast to only such locations, instead of all nodes in the network, and are joined locally by process S.

For clarity, we next present the pseudocode for processes R, S, and T separately. A distributed barrier is used to synchronize all processes, indicating the boundary between two phases. The algorithmic display convention used throughout this section intentionally shows types next to variables when their values are being assigned. Unless a new value assignment is explicitly written, the variable was previously assigned a value now used as an input parameter. The type $\text{key}_{R|S}$ refers to the join key type for both R and S, payload_R and payload_S refer to R and S tuples, respectively, excluding the join key.

Algorithm 1 describes process_R of 2-phase track join that selectively broadcasts tuples from table R. In the first phase, it distributes the distinct keys to process_Ts at destination nodes determined by hashing. In the second phase, it receives from process_T a list of matching S locations for each distinct key, and transfer the R tuples to these locations.

Algorithm 2 describes process_S during 2-phase track join. Note that the algorithm is similar to process_R in the first phase but is different in the second phase. When R tuples are transferred (selective broadcast of R to S), the process_R sends the payloads in the second phase, while the process_S receives the R tuples and joins them locally. The inverse would happen if S tuples were transferred.

Finally, Algorithm 3 presents process_T. In the tracking phase, it receives the join keys from both R and S and aggregates node locations for every key. In the second phase, it distributes the broadcast locations, so that R tuples are actually transferred by process_R and joined by process_S.

The tracking phase of 2-phase track join resembles a hash join on the join keys, after eliminating duplicates. The selective broadcast phase resembles a broadcast join, since only the tuples of one table are transferred over the network. However, compared to a broadcast join that sends the R

ALGORITHM 1: 2-Phase Track Join: process$_R$ − R to S

```
1  T_R ← {}
2  for all ⟨key_{R|S} k, payload_R p_R⟩ in table R do
3      if k not in T_R then
4          n ← hash(k) mod N
5          send k to process_T n
6      T_R ← T_R ∪ ⟨k, p_R⟩
7  barrier
8  while any process_T n_T sends do
9      for all ⟨key_{R|S} k, process_S n_S⟩ from n_T do
10         for all ⟨k, payload_R p_R⟩ in T_R do
11             send ⟨k, p_R⟩ to n_S
```

ALGORITHM 2: 2-Phase Track Join: process$_S$ − R to S

```
1  T_S ← {}
2  for all ⟨key_{R|S} k, payload_S p_S⟩ in table S do
3      if k not in T_S then
4          n ← hash(k) mod N
5          send k to process_T n
6      T_S ← T_S ∪ ⟨k, p_S⟩
7  barrier
8  while any process_R n_R sends do
9      for all ⟨key_{R|S} k, payload_R p_R⟩ from n_R do
10         for all ⟨k, payload_S p_S⟩ in T_S do
11             commit ⟨k, p_R, p_S⟩
```

ALGORITHM 3: 2-Phase Track Join: process$_T$ − R to S

```
1  T_{R|S} ← {}
2  while any process_R or any process_S n_{R|S} sends do
3      for all ⟨key_{R|S} k⟩ from n_{R|S} do
4          T_{R|S} ← T_{R|S} ∪ ⟨k, n_{R|S}⟩
5  barrier
6  for all distinct key_{R|S} k in T_{R|S} do
7      for all ⟨k, process_R n_R⟩ in T_{R|S} do
8          for all ⟨k, process_S n_S⟩ in T_{R|S} do
9              send ⟨k, n_S⟩ to n_R
```

tuples to all nodes, track join selectively broadcasts the R tuples to nodes that are known to contain matching S tuples. For instance, consider a join with two equally sized tables with unique join keys and high join selectivity. The total cost of track join would be the cost of the tracking phase plus the size of the smallest table $\min(|R|, |S|)$, taking into account the average payload size times the number of tuples. In the same scenario, a broadcast join would cost $(N - 1) \cdot \min(|R|, |S|)$, and a

hash join would cost $(N-1)/N \cdot (|R|+|S|)$. The $(N-1)/N$ factor accounts for the $1/N$ probability that a tuple is sent to the same node, thus does not incur network cost. Having entirely unique keys maximizes the cost of the tracking phase to $w_{key}/(w_{key}+w_{payload})$ times that of hash join, where w_{key} and $w_{payload}$ are the key and payload widths. As long as $w_{key} < w_{payload}$ and the key is smaller than half of the tuple, track join will transfer less data than hash join.

2.2 3-Phase Track Join

The 3-phase track join improves over 2-phase track join, by deciding whether to broadcast R tuples to the locations of S tuples, or vice versa. The decision is taken while executing the join, independently for each distinct join key. To decide which selective broadcast direction is cheaper, we need to know how many tuples will be transferred in each case. We gather this information during the tracking phase. Instead of only tracking nodes with at least one matching tuple, as done in 2-phase track join, here we also track the number of matches for each distinct join key per node. Thus, we know not only which nodes have matching tuples per join but also how many tuples. To generalize for tuples of variable length, we use the sum of the widths of the matching tuples per node, rather than a single count of matching tuples.

Algorithm 4 shown below, describes the processing of R tuples during 3-phase track join. The algorithms for processing R and S are identical in 3-phase track join. The additional step over 2-phase track join is the count aggregation of the join keys. The unique join keys and the aggragated counts are then hash partitioned over the network. Duplicate key elimination is implicit in the aggregation. Note that the first of the two barriers can be omitted here, since the first phase sends nothing over the network. Nevertheless, we include the barrier to clarify that the tracking phase will practically start only after the local count aggregation on each node is completed.

ALGORITHM 4: 3-Phase Track Join: process$_R$

```
1  T_R ← {}
2  for all ⟨ key_{R|S} k, payload_R p_R⟩ in table R do
3      T_R ← T_R ∪ ⟨k, p_R⟩
4  barrier
5  for all distinct key_{R|S} k in T_R do
6      c ← |k in T_R|
7      n ← hash(k) mod N
8      send ⟨k, c⟩ to process_T n
9  barrier
10 while any process_S or any process_T sends do
11     if source is process_T n_T then
12         for all ⟨ key_{R|S} k, process_S n_S⟩ from n_T do
13             for all ⟨k, payload_R p_R⟩ in T_R do
14                 send ⟨k, p_R⟩ to n_S
15     else if source is process_S n_S then
16         for all ⟨key_{R|S} k, payload_S p_S⟩ from n_S do
17             for all ⟨k, payload_R p_R⟩ in T_R do
18                 commit ⟨k, p_R, p_S⟩
```

ALGORITHM 5: 3-Phase Track Join: process$_S$

```
1 ... symmetric with process_R of 3-phase track join ...
```

The improvement of 3-phase over 2-phase track join is that bi-directional selective broadcast can distinguish cases in which moving S tuples would transfer fewer bytes than moving R tuples. Because selective broadcasts occur from both directions, the algorithm for processing R and S includes both cases. The decision whether to selectively broadcast $R \rightarrow S$ or $S \rightarrow R$ is taken independently per distinct join key, and we do not pick a single direction for the entire join. Note that 3-phase track join also simplifies query optimization, since the join now becomes symmetric for the two input relations.

ALGORITHM 6: 3-Phase Track Join: process$_T$

```
1  barrier
2  T_{R|S} ← {}
3  while any process_R or any process_S n_{R|S} sends do
4      for all ⟨key_{R|S} k, count c⟩ from n_{R|S} do
5          T_{R|S} ← T_{R|S} ∪ ⟨k, n_{R|S}, c⟩
6  barrier
7  for all distinct key_{R|S} k in T_{R|S} do
8      R, S ← {}, {}
9      for all ⟨k, process_R n_R, count c⟩ in T_{R|S} do
10         R ← R ∪ ⟨n_R, c · width_R⟩
11     for all ⟨k, process_S n_S, count c⟩ in T_{R|S} do
12         S ← S ∪ ⟨n_S, c · width_S⟩
13     c_RS ← broadcast R to S
14     c_SR ← broadcast S to R
15     if c_RS < c_SR then
16         for all ⟨k, process_R n_R⟩ in T_{R|S} do
17             for all ⟨k, process_S n_S⟩ in T_{R|S} do
18                 send ⟨k, n_S⟩ to n_R
19     else
20         for all ⟨k, process_S n_S⟩ in T_{R|S} do
21             for all ⟨k, process_R n_R⟩ in T_{R|S} do
22                 send ⟨k, n_R⟩ to n_S
```

Algorithm 6 shown above, receives the join keys, generates the transfer schedules, and drives the bi-directional selective broadcasts. The hash partitioning of keys also distributes the matching tuple counts used to pre-compute the selective broadcast cost. We compute the cost for both directions and pick the cheapest. Then the suitable key and node-id messages are relayed to source nodes to drive the selective broadcast.

The cost of a selective broadcast ($R \rightarrow S$) for a single join key is shown in Algorithm 7 below. The cost is computed at runtime and is used to determine the $R \rightarrow S$ or $S \rightarrow R$ selective broadcast direction for each distinct join key independenty. In contrast, 2-phase track join uses the same predetermined direction throughout the join. The cost is the same as a Cartesian product excluding all

nodes that have no tuples of the particular join key. We compute the total size of R tuples (r_{total}), the size of R tuples in nodes with matching S tuples (r_{local}), the number of nodes with matching R tuples (n_R), and the number of nodes with matching S tuples (n_S). All computed numbers refer to a single join key. The total cost for the join key is then computed by subtracting the local R tuples (r_{local}) from all R tuples (r_{total}). We also include the cost of sending location messages that drive the selective broadcast. The size of these messages (M here) is the length of the join key, the length of the node-id, and the length of the tuple count, although we discuss simple ways to reduce M (see Section 2.4).

ALGORITHM 7: Track Join: Broadcast R to S (for single join key) (used in process$_T$)

```
1  r_total ← Σ_i |R_i|                                   //the total size of matching R tuples
2  r_local ← Σ_i |R_i|   //the total size of matching R tuples in nodes with matching S tuples
   (local sends)
3  n_R ← 0                    //the number of nodes with matching R tuples (excluding self)
4  n_S ← 0                    //the number of nodes with matching S tuples (including self)
5  for all process_{R|S} i do
6      r_total ← r_total + R_i            //add the total number of matching R tuples in node i
7      if R_i ≠ 0 and i ≠ n_self then
8          n_R ← n_R + 1                                //count nodes with both R and S tuples
9      if S_i ≠ 0 then
10         r_local ← r_local + R_i n_S ← n_S + 1
11 return r_total · n_S − r_local + n_R · n_S · M          //M: the size of the schedule message
```

The complexity to compute the cost is $O(n)$, where n is the number of nodes with at least one matching tuple for the particular join key. Note that n_R and n_S are simply the number of R and S nodes with matching join keys, they do not induce a higher complexity. The total number of steps required is less or equal to the number of tuples in the join. Thus, the complexity remains in the worst case linear, which is also the lower bound for the join operator.

2.3 4-Phase Track Join

In the full version of track join, the join is logically decomposed into single key (Cartesian product) joins for each distinct join key. By minimizing the network cost of each Cartesian product, we reduce the network cost of the entire join. If we omit the cost of tracking, which cannot be optimized in a generic way, then 4-phase track join transfers the minimum amount of payload data possible for an early-materialized distributed join. An early-materialized join transfers the payloads while executing the join. In contrast, a late-materialized join can still execute the join using keys and record ids and use the record ids to transfer the payloads later.

4-phase track join extends 3-phase track join by using a more complicated scheduling algorithm that includes a migration phase. The additional phase ensures that when tuples from one table are selectively broadcast to matching tuple locations from the other table, we have already migrated tuples to a subset of nodes, minimizing the total network transfers for the particular join key. Note that track join does the scheduling of tuples transfers independently for each distinct join key, the finest granularity level.

As shown in Figure 1, the migration phase allows tuples from one table to be moved to fewer nodes before the selective broadcast from the other table. The schedules do not only decide which side to selectively broadcast but also which nodes will migrate migrate to other nodes, so that the

selective broadcast can skip nodes. Even if we end up migrating all tuples to a single node, track join still performs better than hash join, because the destination is determined by hashing but is the node with the most pre-existing matching tuples. Algorithm 8 describes the migration of R tuples. Processing S is identical. The migration of tuples during 4-phase track join is transient.

Note that 4-phase track join can choose to partially migrate tuples, as shown in Figure 1. In such a case, track join behaves quite differently from both broadcast join and hash join. Since the transfer schedule aims to minimize the network traffic, sending most tuples to the same nodes creating imbalance is theoretically possible. However, in practice, we did not encounter such significant imbalance even with the skewed version of TPC-H [7], because local key imbalance will even out over many keys. Nevertheless, in cases of extreme skew such as zipfian distributions, we can filter the heavy hitter keys and treat them differently. This technique has already been applied on top of hash join [48] and is compatible with track join as well.

ALGORITHM 8: 4-Phase Track Join: Process$_R$

```
... identical to the 1st phase of 3-phase track join ...
barrier
... identical to the 2nd phase of 3-phase track join ...
barrier
while any process_T or any process_R sends do
    if source is process_T n_T then
        for all ⟨key_{R|S} k, process_R n_R⟩ from n_T do
            for all ⟨k, payload_R p_R⟩ in T_R do
                send ⟨k, p_R⟩ to n_R
                T_R ← T_R \ ⟨k, p_R⟩
    else
        for all ⟨key_{R|S} k, payload_R p_R⟩ from n_R do
            T_R ← T_R ∪ ⟨k, p_R⟩
barrier
... identical to the 3rd phase of 3-phase track join ...
```

ALGORITHM 9: 4-Phase Track Join: Process$_S$

```
... symmetric with process_R of 4-phase track join ...
```

We now show the driving of the migration phase in Algorithm 10. During the migration phase, we send key and node-id messages to source nodes to drive the migration matching tuples. Note that we never send the same message to multiple destinations. Every node that receives a $\langle k, n_R \rangle$ message, sends all R tuples with key equal to k to the node n_R (and only that node). The same is true for S tuples.

We are still missing two pieces of the puzzle to complete 4-phase track join. First, we need to explain how to generate transfer schedules that include migration. Second, we have to prove that these transfers schedules are optimal for single key joins.

We formalize the problem of network traffic minimization for a single key (Cartesian product) join, assuming for simplicity (but without loss of generality) that the plan is known to every node and there is no need to send any location messages. Assume that x_{ij} is the binary decision of

ALGORITHM 10: 4-Phase Track Join: Process$_T$

```
1  barrier
2  ... identical to the 2nd phase of 3-phase track join ...
3  barrier
4  for all distinct key_{R|S} k in T_{R|S} do
5      R, S ← {}, {}
6      for all ⟨k, process_R n_R, count c⟩ in T_{R|S} do
7          R ← R ∪ ⟨n_R, c · width_R⟩
8      for all ⟨k, process_S n_S, count c⟩ in T_{R|S} do
9          S ← S ∪ ⟨n_S, c · width_S⟩
10     c_RS, S_migrate ← migrate S & broadcast R
11     c_SR, R_migrate ← migrate R & broadcast S
12     if c_RS < c_SR then
13         for all ⟨k, process_S n_S⟩ in S_migrate do
14             n'_S ← any process_S not in S_migrate
15             send ⟨k, n'_S⟩ to n_S
16             T_{R|S} ← T_{R|S} \ ⟨k, n_S, count c_src⟩
17             T_{R|S} ← T_{R|S} \ ⟨k, n'_S, count c_dst⟩
18             T_{R|S} ← T_{R|S} ∪ ⟨k, n'_S, c_src + c_dst⟩
19     else
20         for all ⟨k, process_R n_R⟩ in R_migrate do
21             n'_R ← any process_R not in R_migrate
22             send ⟨k, n'_R⟩ to n_R
23             T_{R|S} ← T_{R|S} \ ⟨k, n_R, count c_src⟩
24             T_{R|S} ← T_{R|S} \ ⟨k, n'_R, count c_dst⟩
25             T_{R|S} ← T_{R|S} ∪ ⟨k, n'_R, c_src + c_dst⟩
26 barrier
27 ... identical to the 3rd phase of 3-phase track join ...
```

sending R tuples from node i to node j, while y_{ij} is the binary decision of sending S tuples from node j to node i. $|R_i|$ is the total size of R tuples in node i and $|S_j|$ is the total size of S tuples in node j. Since each schedule is on a single key, we have to join every R_i data part with every S_j data part. There are three possible ways we can transfer the tuples to execute the join. One way is by sending R tuples from i to j, setting $x_{ij} = 1$. Another way is by sending S tuples from j to i, setting $y_{ij} = 1$. The last way is by sending both R_i and S_j to a common third node and join them there. In short, we need some node k, where $x_{ik} = 1$ and $y_{kj} = 1$. Local sends do not affect the network traffic, thus all x_{ii} and y_{jj} are excluded from the cost.

$$\text{minimize: } \sum_i \sum_{j \neq i} x_{ij} \cdot |R_i| + y_{ij} \cdot |S_j| \qquad \text{subject to: } \forall i, j \sum_k x_{ik} \cdot y_{kj} \geq 1$$

We solve the traffic minimization problem by allowing tuples to be migrated before the selective broadcasts. Before the $R \rightarrow S$ selective broadcast, we migrate S tuples, by testing how the cost of the selective broadcast is affected. Since the mechanism allows only S tuples to be transferred, we can easily decide whether the S tuples of some node should stay in place, or be transferred somewhere with more S tuples. Also, the destination can be any node that has S tuples (that will not be migrated), thus deciding the destination node is not an issue. The cost of the selective

ALGORITHM 11: Track Join: Migrate S and Broadcast R (for single join key) (used in process$_T$)

```
1  r_total, r_local, n_R, n_S ← 0                          //same as in Algorithm 7
2  c_max ← 0                          //the most matching tuples on a node with some S tuples
3  for all process_{R|S} i do
4      r_total ← r_total + R_i
5      if R_i ≠ 0 and i ≠ n_self then
6          n_R ← n_R + 1
7      if S_i ≠ 0 then
8          r_local ← r_local + R_i
9          n_S ← n_S + 1
10         if R_i + S_i > c_max then
11             c_max ← R_i + S_i
12             i_max ← i          //the node with the most matching tuples and some S tuples
13 c ← r_total · n_S − r_local + n_R · n_S · M
14 S_migrate ← {}                          //the set of nodes that will migrate S tuples
15 for all process_{R|S} i do
16     if S_i ≠ 0 and i ≠ i_max then
17         Δ ← R_i + S_i − r_total − n_R · M
18         if i ≠ i_self then
19             Δ ← Δ + M
20         if Δ < 0 then
21             c ← c + Δ
22             S_migrate ← S_migrate ∪ {i}
23 return c, S_migrate
```

broadcast is considered alongside the cost of migration. If migrating tuples from a node n reduces the selective broadcast cost but increases the total cost, then the migration from node n is rejected.

Algorithm 11 shown above, computes the optimal schedule of S tuple migration followed by the $R \rightarrow S$ selective broadcast. The key and node-id messages have size m. We first compute the cost of the selective broadcast $R \rightarrow S$. Then, we iterate over every S node with matching tuples and check whether migrating the S tuples would reduce the overall cost, since we will have to selective broadcast to one less node later. The decision can be taken greedily without considering other nodes. Since we consider each node with matching tuples once, the schedule generation algorithm is in the worst case linear to the number of matching tuples, not the number of nodes.

Theorem 2.1. *We can generate optimal selective broadcast $R \rightarrow S$ schedules for single key joins by migrating S tuples, in linear time to the number of R and S tuples.*

Proof. We want to compute a set of nodes $S_{migrate} \subseteq S$, where any node i in $S_{migrate}$ migrates local matching tuples. Observe that the migration destination does not affect the network cost, thus, can be any node in $S \setminus S_{migrate}$. Let m_i be the binary decision whether node i keeps all local matching S tuples ($m_i = 0 \Leftrightarrow i \in S \setminus S_{migrate}$) or migrates them ($m_i = 1 \Leftrightarrow i \in S_{migrate}$). The cost of migrating selective broadcast from R to S is

$$\left(\sum |R_i|\right) \cdot \left(1 - \sum m_i\right) - \left(\sum |R_i| \cdot m_i\right) + \left(\sum |S_i| \cdot m_i\right).$$

The $\sum |R_i|$ term is the total size of R tuples, the $\sum (1 - m_i)$ is the number of locations with S tuples, the $\sum |R_i| \cdot (1 - m_i)$ are the R tuples that were local during broadcast and $\sum |S_i| \cdot m_i$ is the cost of

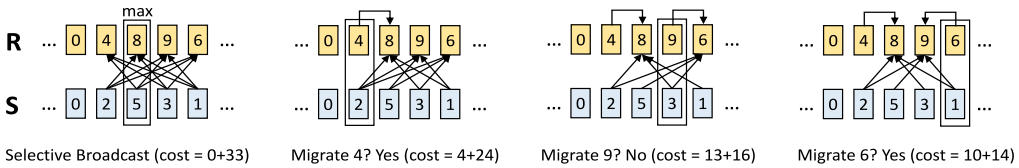


Fig. 2. Example of 4-phase track join optimal schedule generation per distinct join key.

migrating S tuples. Since $\sum |R_i| (\equiv R)$ and $\sum |S_i| (\equiv S)$ are independent of all m_i, the formula can be minimized by checking each m_i independently. Finally, since $S_{migrate}$ cannot contain all nodes, we never consider adding the node i with the largest $|R_i| + |S_i|$ (and $|S_i| > 0$) in $S_{migrate}$ ($m_i = 0$). □

Theorem 2.2. *The cheapest between the two optimized (using migrations) selective broadcast directions ($R \rightarrow S$ or $S \rightarrow R$) is (equivalent to) the schedule with minimum network traffic for single key joins.*

Proof. Given a computed $S_{migrate}$, migrating any R tuples never reduces the cost. Assume we migrate R tuples from node x to node y. If $y \in S \setminus S_{migrate}$, then the cost remains the same, as we moved the R tuples from node x once ($+|R_x|$), but now they are local to the S tuples at node y, and we can skip node y during the selective broadcast ($-|R_x|$). If $y \in S_{migrate}$, then we increase the cost of the migrating phase by $|R_x|$, but the cost of the selective broadcast remains the same. □

What the last proof means is that the migrate and broadcast mechanism is sufficient to generate optimal schedules for single key joins. There are no plans that would migrate data from both tables with a lower network cost. Similarly, there are no other tuple transferring mechanisms that can achieve a lower network cost, without taking into account the cost of the tracking phase. Thus, the mechanism of 4-phase track join is sufficient to minimize the network traffic at the finest granularity possible.

In Figure 2, we show an example of 4-phase track join schedule generation for a single key. We start from the cost of selective broadcast and check for each node whether migrating all matching tuples reduces the cost. As shown, nodes with no matching tuples per distinct join key are not considered at all.

The time to generate each schedule depends on the number of tuples per join key. The input is an array of triplets: key, node-id, and tuple size. If a node does not contain a key (tuple size of 0), then the triplet is not generated at all. Thus, scheduling is in the worst case linear to the number of input tuples from both tables.

Note that track join is a greedy algorithm, thus we use the term *minimal*, rather than *minimum*, to refer to the network traffic optimization. While we proved that the amount of payload data transferred is indeed minimum, we cannot always guarantee the same overall, since the tracking phase cannot be optimized. We cannot optimize the tracking phase, since we have yet no information about where matching tuples are located per join key. The tracking phase can be considered as a hash join using the unique join keys, and we have no information in order reroute any network transfers until the tracking phase is completed.

Using multiple threads is essential to achieve good performance on all CPU intensive tasks, even when the complexity is linear. The optimization becomes more important if the data remain main-memory resident throughout the join. Track join allows all in-process operations across keys to be parallelized across threads freely, since all local operations combine tuples with the same join key only. Recent work showed that dynamic thread sharing [28] is essential to achieve good performance when multiple operators are pipelined. Such techniques are usable in track join.

Finally, while the algorithmic descriptions of track join in this section implement track join by pipelining different operators, track join can also be implemented by fully separating the steps. Each step would materialize the results in memory to be used by the following step, similar to how in-memory column-oriented query engines operate. In such an implementation, there is no need to separate processes R, S, and T, as they will run in successive steps and not in parallel. Note that each step can still be implemented using multiple threads internally. We will use such an implementation in the experimental evaluation of track join (see Section 5).

2.4 Traffic Compression

Track join, as described so far, makes absolutely no effort to reduce the data footprint by compressing the data before sending them over the network. Modern analytical database systems employ distinct column value dictionaries [46, 61]. The compression process can occur offline, and data can be processed in compressed form throughout the join. In this section, we briefly discuss some techniques that can further reduce network traffic on top of track join.

A simple compression technique is delta encoding [29, 55]. With sufficient spare CPU cycles, we can sort all columns before sending them over the network. Track join imposes no specific message ordering, besides the barriers between phases. The highest compression rate is likely achieved by sorting.

A second technique is to perform partitioning at the source to create common prefixes. For instance, we can radix partition the first b_p out of b bits and pack $(b-b_p)$-bit suffix with a common prefix. We can tune the compression rate by employing more partition passes to create wider prefixes. Each pass has been shown to run close to the memory-to-memory copy bandwidth [42, 51, 60]. If the column values are dictionary-encoded throughout the join, then the number of distinct values per common prefix is maximized.

A similar optimization we can use for track join is to group the keys by node-ids. We can avoid sending node-ids in messages with key and node-id pairs used to drive the migration and the selective broadcast, by partitioning the pairs by node-id and sending all keys with the same node-id destination together alongside a single node-id label.

3 QUERY OPTIMIZATION FOR TRACK JOIN

The formal model of track join is used by the query optimizer to decide whether to use track join, hash join, or broadcast join. We prove track join superior to hash join, even after making the latter tracking-aware using globally unique record identifiers (rids). Finally, we study how track join interacts with Bloom filters in semi-joins.

3.1 Network Cost Model

In this section, we assume a uniform distribution of tuples across nodes. This is the worst case for track join, since it precludes locality. Track join optimization is not limited to one specific distribution, as we will explain shortly in this section.

The network cost of the standard hash join with early materialized payloads is

$$\frac{N-1}{N} \cdot (t_R \cdot (w_k + w_R) + t_S \cdot (w_k + w_S)),$$

where t_R and t_S are the tuple counts of tables R and S, w_k is the total width of the join key columns used in conjunctive equality conditions, while w_R and w_S are the total width of relevant payload columns projected for the later steps of the query.

To define the network cost of 2-phase track join, we first determine the cost of tracking. The number of nodes that contain matches for each key is upper bounded by N and is t/d, in the worst

case when equal keys are randomly distributed across nodes, where t is the number of tuples and d is the number of distinct values. We use the terms $n_R \equiv \min(N, t_R/d_R)$ and $n_S \equiv \min(N, t_S/d_S)$ for these commonly reused quantities. We also use $\mathcal{N} = \frac{N-1}{N}$ to exclude keys that are already in their hash destination.

We define the input selectivity (s_R and s_S) as the percentage of tuples of one table that have matches in the other table, after applying all other selective predicates. The number of distinct nodes with matching payloads also includes the input selectivity factor. We assume that the selective predicates do not use the key column and the number of distinct keys is unaffected. Again, we synopsize these commonly reused quantities using the terms $m_R \equiv \min(N, (t_R \cdot s_R)/d_R)$ and $m_S \equiv \min(N, (t_S \cdot s_S)/d_S)$. A message of size $w_l \equiv \log(N)$ bits is used to encode a node location. We also assume that destination node locations for a particular key are aggregated into an array before their transmission over the network.

Using the terms described above, the cost of 2-phase track join is

$$\begin{aligned} &\mathcal{N} \cdot (d_R \cdot n_R + d_S \cdot n_S) \cdot w_k && \text{(track R \& S keys)} \\ &+d_R \cdot m_R \cdot (w_k + w_l \cdot m_S) && \text{(transfer S locations)} \\ &+t_R \cdot s_R \cdot m_S \cdot (w_k + w_R) && \text{(transfer R} \rightarrow \text{S tuples).} \end{aligned}$$

For 3-phase track join, we transfer counts during tracking alongside the keys. The $t/(d \cdot s)$ fraction gives the average repetition of keys on each node if the distribution and node placement is assumed to be uniform random. We use this metric to estimate how many bits to use for representing counters (or tuple widths). We refer to the counter lengths using $c_R \equiv \log(t_S/(d_S \cdot n_S))$ and $c_S \equiv \log(t_S/(d_S \cdot n_S))$. Even if some keys occur frequently enough to exceed the maximum count, we can still aggregate at the destination before the schedule generation. The cost of 3-phase track join is

$$\begin{aligned} &\mathcal{N} \cdot d_R \cdot n_R \cdot (w_k + c_R) + d_S \cdot n_S \cdot (w_k + c_S) && \text{(tracking)} \\ &+d_{R_1} \cdot m_{R_1} \cdot (w_k + w_l \cdot m_{S_1}) + t_{R_1} \cdot s_{R_1} \cdot m_{S_1} \cdot (w_k + w_{R_1}) && (R_1 \rightarrow S_1) \\ &+d_{S_2} \cdot m_{S_2} \cdot (w_k + w_l \cdot m_{R_2}) + t_{S_2} \cdot s_{S_2} \cdot m_{R_2} \cdot (w_k + w_{S_2}) && (S_2 \rightarrow R_2). \end{aligned}$$

In this formula, we assume the tuples are split into two separate classes R_1, S_1 and R_2, S_2. The tuples of R_1 are defined based on whether the $R \rightarrow S$ direction is less expensive than the $S \rightarrow R$ direction at distinct join key granularity.

If the query optimizer only needs to decide which version of track join to use rather than to estimate the actual cost, then the task is simplified singificantly. 2-phase track join suffices when both tables have almost entirely unique keys, or one table has many more repeats per value than the number of nodes, which is the case where a simple broadcast join would be used. 3-phase track join is preferred when we cannot estimate the cardinality of the relations and would avoid the inner–outer table distinction. If the number of output tuiples exceeds the number of input tuples, such as a *m-n* joins, then 4-phase track join should be used to generate the optimal transfer schedules. The same is true regardless of the input, if significant pre-existing locality is detected in the workload or if locality is generated artificially prior to executing the joins.

If the query optimizer must estimate the join cost rather than a relative comparison of available algorithms, then we can estimate the network cost of track join by splitting the input in classes with different degrees of correlation. For example, the R_1 and R_2 classes used in the cost of 3-phase track join represent the percentages of tuples of R that are joined using the $R \rightarrow S$ (for R_1), or the $S \rightarrow R$ (for R_2) selective broadcast direction. To populate the classes, we can use *correlated sampling* [64], a technique that preserves the join relationships of tuples. Correleated sampling works independently of the distribution, and can be generated off-line. The sample is augmented

with initial placements of tuples. Besides computing the exact track join cost, we incrementally classify the distinct join keys to correlation classes based on estimated network traffic per schedule. The cost of 4-phase track join is shown below, using two classes for 3-phase track join (R_1, S_1, R_2, S_2) and one for hash join (R_3, S_3):

$$\begin{aligned}
&\mathcal{N} \cdot d_R \cdot n_R \cdot (w_k + c_R) + d_S \cdot n_S \cdot (w_k + c_S) && \text{(tracking)}\\
+&d_{R_1} \cdot m_{R_1} \cdot (w_k + w_l \cdot m_{S_1}) + t_{R_1} \cdot s_{R_1} \cdot m_{S_1} \cdot (w_k + w_{R_1}) && (R_1 \rightarrow S_1)\\
+&d_{S_2} \cdot m_{R_2} \cdot (w_k + w_l \cdot m_{S_2}) + t_{S_2} \cdot s_{S_2} \cdot m_{R_2} \cdot (w_k + w_{S_2}) && (S_2 \rightarrow R_2)\\
+&d_{R_3} \cdot n_{R_3} \cdot (w_k + w_l) + t_{R_3} \cdot s_{R_3} \cdot (w_k + w_{R_3}) && (R_3 \rightarrow \text{h(k)})\\
+&d_{S_3} \cdot n_{S_3} \cdot (w_k + w_l) + t_{S_3} \cdot s_{S_3} \cdot (w_k + w_{S_3}) && (S_3 \rightarrow \text{h(k)}).
\end{aligned}$$

The three correlation classes that combine hash join with 3-phase track join combine broadcasting as well as partitioning to a common destination. If higher estimation accuracy is required, then we can create more classes as step functions between the above. Each class is a step function between broadcast join and hash join. By sampling join keys and gathering tracking info, we can classify a percentage of tuples to a middle point between broadcast join and hash join. Each point assumes that the transfer schedules for these groups of keys will have similar cost. The granularity and cost of the analysis depends on the number of cases we are willing to discern. In cases with few nodes, the number of possible asymmetric schedules is not too high, and we can compute a histogram for the percentage of tuples that will execute each schedule.

Track join invests significant time working with keys compared to early materialized hash join. By computing costs using the above models, the optimizer can choose the best join algorithm, which, in general, depends on the relative key/payload widths, tuple counts, and join selectivity. For instance, when the payloads are small compared with keys, track join could perform worse than hash join in the absence of locality. Here, we present a more concrete example showing a class of workloads where track join is strictly better than hash join.

Example 3.1. Consider a join of R and S where there is a one-to-one match for each tuple in each table, and both tables have cardinality $t = t_R = t_S = d_R = d_S$. As the number of nodes scales, $\lim_{N\rightarrow\infty} \mathcal{N} = 1$. The cost of hash join is $t \cdot (w_R + w_S + 2w_k)$. The cost of track join is $2t \cdot w_k + t(w_k + w_l) + t(\min(w_R, w_S) + w_k)$. Assume $w_R = 2w_S$ for this example. Then the ratio of the track join cost to the hash join cost is $(w_S + 4w_k + w_l)/(3w_S + 2w_k)$, so that track join would be a factor of almost 3 better when $w_S >> w_k, w_l$.

3.2 Tracking-Aware Hash Join

Late materialization is a popular approach for designing main memory databases, because it allows for operators to be performed on the keys alone. The payloads are only accessed when needed using rids. The technique is typically used in column stores [46, 55], where each column is stored as a fixed width array and any column value can be accessed as an array slot. Late materialization can also be viewed as a query optimizer decision of whether to defer payload accesses or to process payloads alongside the keys. Interleaving late materialized operators to exploit low selectivities is out of the scope of this article. However, since late materialization can reduce the amount of payload traffic significantly if the following operators have low selectivity, we study its relation to track join.

In the simple late materialized hash join, join keys are hashed, rids are implicitly generated (0 to N), and payloads are fetched afterwards. The network cost is

$$\mathcal{N} \cdot (t_R + t_S) \cdot w_k + t_{RS} \cdot (w_R + w_S + \log t_R + \log t_S),$$

where t_RS is the number of joined tuples. The join (output) selectivity is also defined as $s_{RS} = t_{RS}/(t_R + t_S)$. Note that s_{RS} can be arbitrarily higher than 1 in *m-n* joins. The $\log t_R$ and $\log t_S$ correspond to rids.

When distributed, rids contain a local rid and a node identifier. To use the implicit tracking information carried in an rid would be to migrate the result to the tuple location, instead of fetching the tuples where the rid pair is. For instance, assume a join of R and S, where fetching S payloads is costlier than R. Using the R rid, we migrate the R rid where the R tuple resides. The S tuple is sent to the location of the R tuple, after a request with the S rid and the R destination.

Assuming that no later operation precedes payload fetching, we can further elaborate the improvement by redoing the join on the final destination. In the above example, we send one local id to the R node and the local id to the S node, alongside the node id of the R node. Sending the node id can be avoided as in track join. The payload brought from the S node will be coupled with the join key. The R rid will access the tuple and rejoin it with the incoming tuple from the S node. The network cost is

$$\mathcal{N} \cdot (t_R + t_S) \cdot w_k + t_{RS} \cdot (\min(w_R, w_S) + w_k + \log t_R + \log t_S).$$

We prove that this approach is less effective than track join. Initially, it transfers the key column without any duplicate elimination or reordering to allow implicit rid generation. In contrast, track join transfers either the distinct keys of each node only, or an aggregation of keys with short counts. Afterwards, track join performs optimal transfers overall. The rid-based hash join transfers payloads from the shorter side to all locations of the larger side where there is a match, the same schedule as 2-phase track join. Instead, track join resends keys, which, unless uncompressed, have shorter representation than rids. Thus, the simplest 2-phase track join transfers less data over then network than the tracking-aware hash join using network-global rids.

The extra cost of transferring rids is non-trivial in most cases. As shown in our experiments, real workloads may use less than 8 bytes of payload data while globally unique rids must be at least 4 bytes. A pair of rids from both input tables may be wider than the payloads with smaller size, making late materialization less effective for distributed query execution compared to local in-memory query execution.

3.3 Semi-Join Filtering

When join operations are coupled with selections, we can prune tuples both individually per table and across tables. To that end, databases use semi-join [4, 35] implemented using Bloom filters [6], which are optimized toward network traffic. In our analysis, we assume a two-way semi-join as in Reference [58].

When a false positive occurs, hash join transfers the whole tuple in vain. In track join, the matches are determined when the keys are tracked. All join keys with no matches are discarded up front using the key projection; no location or tuples are transferred thereafter. Whereas late materialized hash join reduces the penalty of filter errors by using key and rid pairs, track join sends less than the key column alone.

Assuming the length per qualifying tuple in the filter to be w_{bf}, the cost of early materialized hash join that uses Bloom filters as an initial step is

$$\begin{aligned} &(t_R \cdot s_R + t_S \cdot s_S) \cdot N \cdot w_{bf} && \text{(broadcast filters)} \\ +&\mathcal{N} \cdot t_R \cdot (s_R + e) \cdot (w_k + w_R) && \text{(transfer S tuples)} \\ +&\mathcal{N} \cdot t_S \cdot (s_S + e) \cdot (w_k + w_S) && \text{(transfer R tuples),} \end{aligned}$$

where e is the relative error of the Bloom filters.

The cost of late materialized hash join that uses Bloom filters as an initial step is

$$\begin{aligned}
&(t_R \cdot s_R + t_S \cdot s_S) \cdot N \cdot w_{bf} && \text{(broadcast filters)}\\
+&\mathcal{N} \cdot t_R \cdot (s_R + e) \cdot (w_k + \log t_R) && \text{(transfer S pairs)}\\
+&\mathcal{N} \cdot t_S \cdot (s_S + e) \cdot (w_k + \log t_S) && \text{(transfer R pairs)}\\
+&t_{RS} \cdot (w_R + w_S + \log t_R + \log t_S) && \text{(fetch payloads),}
\end{aligned}$$

where t_{RS} includes the standard output selectivity factor.

The cost of 2-phase track join that uses Bloom filters as an initial step is

$$\begin{aligned}
&(t_R \cdot s_R + t_S \cdot s_S) \cdot N \cdot w_{bf} && \text{(broadcast filters)}\\
+&\mathcal{N} \cdot d_R \cdot (s_R + e) \cdot me_R \cdot w_k && \text{(track filtered R)}\\
+&\mathcal{N} \cdot d_S \cdot (s_S + e) \cdot me_S \cdot w_k && \text{(track filtered S)}\\
+&d_R \cdot s_R \cdot m_S \cdot w_k && \text{(transfer S locations)}\\
+&t_R \cdot s_R \cdot m_S \cdot (w_k + w_R) && \text{(transfer R tuples),}
\end{aligned}$$

where $me_R \equiv \min(N, t_R \cdot (s_R + e)/d_R)$ and similarly for me_S.

The cost of broadcasting the filters may exceed the cost of sending a few payload columns for a reasonable number of nodes N. Track join implicitly executes perfect semi-join filtering during the tracking phase. Thus, with track join, we are more likely to not need the initial Bloom filter step compared to both early materialized hash join, which sends all key and payload columns during hash partitioning, and late materialized hash join, which transfers only keys and rids initially.

The advantage of the tracking phase in track join over using Bloom filters is that the tracking phase not only prunes unmatched tuples but also determines the best node locations to transfer matching tuples in an optimal schedule with minimal network cost, whereas using two-way Bloom filters followed by a hash join may transfer matching tuples to a blindly chosen node location without any collocating tuples, incurring higher network cost. In particular, if all keys participate in the join, then there is no Bloom filter reduction at all. In contrast, track join can still save network transmission, because it generally avoids transferring tuples with larger payloads, as shown in Example 3.1.

4 OPTIMIZING THE DATA PLACEMENT FOR ANALYTICAL WORKLOADS

Identifying and naturally taking advantage of the pre-existing data locality is one of the advantages of track join. However, data locality may not be naturally present in every workload. Nevertheless, if we intentionally place tuples with the same join key to fewer nodes, we can expect track join to take advantage of our deliberate placement. Since track join computes an optimal schedule for each distinct join key, we have more flexibility on how to place the data, in contrast to hash co-partitioning the tables based on their join attribute. Thus, for multiple joins with conflicting data placement requirements, track join introduces significant opportunities for us to tune the data placement and minimize the network traffic costs of joins for the entire workload.

4.1 Formulation and Setting

The typical relational OLAP queries denormalize a subset of the database by joining the fact table with a subset of the dimension tables, and then summarize via group-by aggregations. The dominant cost of these workloads comes from joining the large fact table and the multiple dimension tables on different join attributes. If the workload consists of two tables only, then we can simply distribute both tables across the nodes by hashing the join key attributes. Whenever we need to perform the join, any matching tuples will always be collocated on the same node and the join can

proceed locally in a shared-nothing fashion. However, this approach is not applicable if there are multiple joins on different attributes. While we can eliminate the network cost of one distributed join, the remaining joins will have to redistribute the tuples over the network.

Most commercial distributed database systems support hashing based placement. Some database tuning advisors already exist to propose the best candidate [36, 47] attribute to use for the placement. However, joining with any other dimension table will still incur the redistribution network cost. Instead, if track join is used in place of distributed hash join, then there are some placements that do not have zero network cost for any specific join, but still minimize the total network cost.

Formally, we assume our dataset contains a fact table F and δ dimension tables D_1, D_2, $D_3, \ldots, D_\delta$. The tuples of F, denoted by $f(k_1, k_2, \ldots, k_\delta, \ldots)$, have join attributes (or join keys) k_j used to join with the dimension tables and some extra payload attributes. The tuples of each D_j have the same k_j join attribute and some payload attributes. Note that k_j can be a composite key. We do not consider replicated tables and assume that every table is unique. In a workload, each dimension table D_j is joined with the fact table F on attribute k_j, with a weight θ_j. The weight θ_j is the product of the frequency of the join $F \bowtie_{k_j} D_j$ in the workload, with the average selectivity applied to D_j before joining with F. Naturally, some joins are executed more frequently than others. Also, joins are often executed after selective conditions have been applied to the base tables. For each join, we use the weight to incorporate both the frequency information and the selectivity of selections on the dimension tables before the join.

The goal is to place the tuples across the network nodes in such a way, so that, if we use track join to execute the distributed joins $F \bowtie_{k_j} D_j$ in the workload, the network cost of executing the distributed joins of the entire workload will be minimized:

$$\text{minimize: } \sum network\_cost(F \bowtie_{k_j} D_j) \cdot \theta_j.$$

Track join provides a cost model that minimizes the network cost of the cartesian product per distinct join key. The minimization of the overall network cost can be formulated by combining the δ track joins into a single optimization problem. In a single dimension, we presented an optimal linear-time solution that minimizes the network traffic. Across multiple dimensions, however, the problem requires a quadratic objective function with too many variables, making the holistic approach of δ-dimension co-optimization impractical. Instead, we use the network cost of each track join individually and, assuming a weight for each join, we minimize the total network cost of all distributed joins in the workload (see Section 4.7.1 for non-distributed joins). An important statistic, not visible in the formula above, is the average width of the tuples participating in the join. Each join in the workload will project a different set of columns depending on the query. The width of the tuple from each table participating in join, is used by track join to determine the network cost (Algorithm 11).

The data placement optimization focuses on a single fact table and multiple dimension tables in a star schema, which is typical for OLAP workloads. In cases where the data warehouse is organized in a snowflake schema rather than a star schema, we can optimize the data placement for the central star sub-schema that includes many large tables. Reducing the cost of joining such large tables is crucial to achieve low overall cost. As we demonstrate in Section 5.3, simply co-partitioning the fact table with first-level dimension tables, or more advanced *reference partitioning* [16] where chains of tables linked by foreign keys are co-partitioned, will incur much higher total cost than optimizing the data placement for the central star schema.

The data placement algorithm proceeds in four steps. In the first step (Section 4.2), we group the entire fact table using all join attributes $(k_1, k_2, \ldots, k_\delta)$, count the number of tuples per group, and send the aggregates to a special node, the *optimization node*, separate from the N cluster nodes. The optimization node then sorts the join key combinations by frequency. In the second

phase (Section 4.3), we place fact table tuples with the matching k_j values together. In the third step (Section 4.4), we refine the mapping based on the total cost of all joins, using the key-by-key examination of track join. In the final step (Section 4.5), we distribute the mapping from the optimization node to the N cluster nodes and relocate the data.

The process described above is executed while other queries are running. Most steps take place on the separate optimization node that must have enough memory to store all distinct combinations of join attributes of the fact table tuples. To be effective, the optimization must have a global view of all the join key combinations. Local optimizations on individual nodes diminish the network cost benefits. Nevertheless, we consider the data placement optimization as an offline process that runs on the optimization node while queries are running in the N cluster nodes. The process is repeated periodically, every time supplemented with updated statistics used to compute the weights θ_j and the payloads of each join. In Section 4.7.2, we explain in more detail what statistics have to be maintained and how they are used in the data placement optimization.

4.2 Step 1: Accumulate the Join Key Combinations

The first step gathers information from the cluster nodes and prepares for computing the optimized placement. In this step, we group the fact table tuples using all join attributes and compute a count for each group. This operation happens in all compute nodes in parallel. The resulting key combinations and counts are sent to the optimization node. The optimization node re-aggregates the tuples resulting in a table of $(k_1, k_2, \ldots, k_\delta)$ tuples with a count c per tuple. The optimization node then sorts the key combinations in decreasing order of c. Prioritizing key combinations with high counts allows for more effective clustering as discussed in Section 4.3. If the frequency counts are almost the same (for example, all key combinations are unique), then the sorting step can be omitted completely.

While the optimization node re-aggregates the counts per join key combination of each compute node, we also maintain a set of nodes that have matching tuples for each combination. We do not need to store the exact number of tuples per node; only the nodes that have one more such combinations. When the placement is decided later, we will use this information to distribute the placement information for each join key combination back to the subset of cluster nodes in order for the data to be relocated.

4.3 Step 2: Mapping by Clustering

This step analyzes the key combinations obtained in the previous step and tries to cluster correlated fact table tuples together, if they have the same value in some join attribute. If the fact table tuples with the same join key value are collocated in a few node locations, rather than scattered across many nodes, then track join can exploit such locality to reduce network traffic. If these tuples are also collocated with the matching dimension table tuples, then the single key joins can be executed entirely at local nodes.

For the output of this step, we create several mappings that maintain the placement information for the fact and dimension tables. For the fact table F, this step generates a mapping P that maps each join key combination $(k_1, k_2, \ldots, k_\delta)$ to some node location $n \in [0, N)$. For every dimension table D_j, this step generates a mapping M_j that maps each distinct join key value k_j to some node location n. These mappings are later used to relocate the data. In addition, as a preparation for the next step where the mapping P is refined using the cost model of track join, we also need to know the counts for each distinct join key at every node location. For each dimension table D_j, we use an array C_j to store the counts, tracking the number of tuples we will place on each of the N cluster nodes following the mappings.

Algorithm 12 describes how we generate the mappings for clustered tuples. Initially, the mappings P and M_j are empty, and the count arrays are cleared with zeros, $C_j[k_j][n] \leftarrow 0$ for every distinct k_j and $n \in [0, N)$. We scan through the join key combinations, which are sorted by frequency, and map each combination to a node. For each join key k_j in the key combination, we search the mapping M_j. If k_j already exists in M_j and maps to some node n, then we add n to the set of possible node locations (i.e., *locations*). If there is no possible locations for the key combination, then we pick any node at random. Otherwise, we pick one of the nodes found in the *locations* at random. We then set the mappings to n for evey k_j that has not been seen before, so that key combinations $(k'_1, k'_2, \ldots, k_j, \ldots)$ that appear later can be potentially placed at the same node with the current key combination, thereby collocating tuples with join key k_j. This mapping creates a cascading effect that clusters joining tuples to the same node. Key combinations processed earlier are likely to collocate with multiple k_j values, since the k_j values have not been mapped before. Key combinations processed later have a larger chance that all k_j has been mapped, thus such tuples can only collocate with tuples from one of the dimensions. Since we iterate through the key combinations in decreasing order of their counts, we essentially cluster more fact table tuples together. For each k_j, the count at the chosen node location is also updated with the count of the current key combination c.

ALGORITHM 12: Mapping by Clustering

```
1  for key combination (k_1, k_2, ..., k_δ) and its count c do
2      locations ← ∅
3      for 0 ≤ j < δ do
4          if k_j ∈ M_j then
5              locations ← locations ∪ {n}
6      if locations ≠ ∅ then
7          n ← any ∈ locations
8      else
9          n ← any ∈ [0, N)
10     P[(k_1, k_2, ..., k_δ)] ← n
11     for 0 ≤ j < δ do
12         if k_j ∉ M_j then
13             M_j[k_j] ← n
14         C_j[k_j][n] ← C_j[k_j][n] + c
```

The time complexity of this step is $O(d_F \cdot \delta)$, where d_F is the number of distinct join key combinations that we examine. The number of possible mappings per join key combination cannot exceed either the number of dimensions δ or the number of nodes N. The algorithm is non-parallel, since the join key mappings M_j are updated per join key combination and must remain globally unique. Parallelizing this step requires pre-partitioning the workload on the join key combinations, but it diminishes the benefits.

For each dimension table, this step generates a M_j mapping from a key value k_j to a node location. These mappings do not change after this step and will guide the relocation of dimension table tuples.

4.3.1 Encoding Join. Note that the mapping M_j can also be implicitly used as a hash function for executing a hash join. The size of this hash function representation is proportional to the

number of distinct join keys. To avoid storing this representation repeatedly on every node, we can use surrogate keys that encode the node location information, in place of the original keys. Consider an example where 64 nodes are used in the system, and each key takes up 32 bits. Then, we can use 6 bits in the surrogate key to represent its location (i.e., the node where the record should be after hashing) and use the remaining bits as a counter to distinguish the keys at the same location. Using this encoding, we can derive the hashed location of a key directly from its value, without consulting the actual hash function mappings. Thus, the repetitive space overhead for hash function representations is eliminated.

To execute a join under this encoding scheme, we would simply decode the join key value of a fact table tuple, and send the tuple to the decoded location if it is not the local node. The dimension table tuples are already placed at the mapped locations, so they can be joined directly. We call this approach *encoding join*. Since encoding join is aware of matching tuple locations by decoding the join keys, it is similar to a 2-phase track join, which obtains the location information via a tracking phase. But encoding join does not have the other benefits of 3-phase and 4-phase track joins. In our experiments, we show that without using track joins, encoding join can also achieve much lower cost than the hash join with a standard hash function.

4.4 Step 3: Mapping Refinement

In this step, we refine the placement P of key combinations created in the previous step. Each combination has multiple potential mappings to nodes. In the previous step, we chose randomly. Track join gives us a cost model for the network cost of each cartesian product in each join. We can compute the total track join cost for all potential mappings and pick the placement with the minimum total cost for a key combination.

The refinement algorithm is shown in Algorithm 13. For each key combination, we reconsider the possible node locations $M_j[k_j]$ that collocates the fact table tuples with dimension tuples from at least one dimension. For every such location, we compute the total cost of the δ single key joins on $k_1, k_2, \ldots, k_\delta$, and choose the node location with the lowest cost as the final mapping for the key combination. For a particular join key k_j, the function `network_cost` computes the minimal network cost of the cartesian product join on this key.

To compute the minimum cost, `network_cost` needs the placement information and tuple widths. The placement of the dimension table tuple is given by the mapping M_j, and the placement of fact table tuples across the N nodes is given by the count array C_j. Note that the cost of a single key join on k_j involves not only the key combination under consideration, but all fact table tuples with matching k_j. Since we have accumulated the counts at the previous step, we can now simply adjust the counts between new and old node locations to obtain the updated count distribution for a single join key. The w_{F_j} and w_{D_j} parameter is the average width of fact table F or dimension table D_j tuples, as projected from each table before the $F \bowtie_{k_j} D_j$ join. Note that each query will project a different subset of attributes. Using such information, the function `network_cost` computes the minimum cost of a single key join using Algorithm 11 and its symmetric version that swaps the tables for migration and broadcasting.

The time complexity of the `network_cost` function is linear to the number of nodes where we will place matching k_j tuples. Nodes with zero such tuples are completely omitted as in track join. The number of possible mappings per join key combination cannot exceed either the number of dimensions δ or the number of nodes N. The total time complexity is $O(d_F \cdot \min(N, \delta) \cdot \delta \cdot \min(N, t_F/d_{D_j}))$, where d_F is the number of distinct key combinations, t_F is the number of tuples in the fact table F, and d_{D_j} is the average number of distinct join key values of a dimension table.

ALGORITHM 13: Mapping Refinement

```
1  for key combination (k_1, k_2, ..., k_δ) and its count c do
2      locations ← ∅
3      for 0 ≤ j < δ do
4          locations ← locations ∪ {M_j[k_j]}
5      if |locations| > 1 then
6          cost_min ← +∞
7          n_old ← P[(k_1, k_2, ..., k_δ)]
8          for n_new ∈ locations do
9              cost ← 0
10             for 0 ≤ j < δ do
11                 C_j[k_j][n_old] ← C_j[k_j][n_old] − c
12                 C_j[k_j][n_new] ← C_j[k_j][n_new] + c
13                 cost ← cost + network_cost(F ⋈_{k_j} D_j, M_j[k_j], C_j[k_j], w_{F_j}, w_{D_j}) · θ_j
14                 C_j[k_j][n_new] ← C_j[k_j][n_new] − c
15                 C_j[k_j][n_old] ← C_j[k_j][n_old] + c
16             if cost < cost_min then
17                 cost_min ← cost
18                 n_min ← n_new
19         for column 0 ≤ j < δ do
20             C_j[k_j][n_old] ← C_j[k_j][n_old] − c
21             C_j[k_j][n_min] ← C_j[k_j][n_min] + c
22         P[(k_1, k_2, ..., k_δ)] ← n_min
```

4.5 Step 4: Relocating the Data

The last step of data placement is to relocate the data across the N compute nodes in a transparent way to query execution. The output of the previous phase is a placement for all key fact table join key combinations as well as for dimension tuples. During step 1 (Section 4.2), each cluster node sent key combinations and counts to the optimization node and the optimization node stored the nodes with matching fact table tuples.

To avoid sending the key combinations as well as the node destination, we could use the same order at which the optimization node received the data. This operation is mainly sequential, and we can even use secondary storage beyond main memory in the optimization node. The same is true for the N cluster nodes, which can sequentially store the key combinations in an array and receive the new node placement from the optimization node in the same order without sending any key combinations.

To relocate the dimension tables, we can send the k_j keys in any order and receive their placement from the optimization node in the same order using M_j. We can also use the $\mathsf{hash}(k_j, N)$ node for the matching, similarly to track join. For the fact table, we can track the placement of each join key combination using the $\mathsf{hash}(k_1, k_2, \ldots, k_\delta, N)$ node that will relay the information of the destination node back to the source nodes.

Relocating the data needs to be transparent to query execution. After a new data placement is determined, we have to shuffle (copy) the data across all compute nodes but can pause the process while expensive queries are running. Once the shuffling is completed, we can execute all new incoming queries using the new data placement. Once all queries using the old placement

Table 1. Data Placement of Tuples (Denoted by their Join Keys) after Step 2

Table	Key columns	Node 0	Node 1
F	A, B	(a_1, b_1), (a_3, b_3), (a_3, b_2)	(a_2, b_2), (a_2, b_2), (a_2, b_2), (a_1, b_2)
D_1	A	a_1, a_3	a_2
D_2	B	b_1, b_3	b_2

Table 2. Network Cost Per Data Placement of Key Combination (a_1, b_2) During Step 3

Key	Table	(a_1, b_2) placed on node 0			(a_1, b_2) placed on node 1		
		Node 0	Node 1	Cost	Node 0	Node 1	Cost
a_1	F	(a_1, b_1), (a_1, b_2)	–	0	(a_1, b_1)	(a_1, b_2)	1
	D_1	a_1	–		a_1	–	
b_2	F	(a_1, b_2), (a_3, b_2)	$3 * (a_2, b_2)$	1	(a_3, b_2)	(a_1, b_2), $3 * (a_2, b_2)$	1
	D_2	–	b_2		–	b_2	

are completed, we can safely delete it and rerun the data placement optimization including more recent statistics and new inserts.

4.6 Example

We now show a small example to illustrate how the two basic steps of the algorithm work. Assume a scenario with 3 tables F, D_1, and D_2, joined on attributes A ($F \bowtie_A D_1$) and B ($F \bowtie_B D_2$). Dimension table D_1 has 3 tuples with join keys: a_1, a_2, a_3. Dimension table D_2 also has 3 tuples with join keys: b_1, b_2, b_3. The fact table F consists of 7 tuples with the following 5 combinations of join keys: (a_1, b_1), (a_2, b_2), (a_3, b_3), (a_1, b_2), (a_3, b_2), 3 tuples with (a_2, b_2) join keys, and 4 distinct tuples with the other join keys.

We first group the fact table tuples by join attributes and accumulate their counts (Section 4.2). We find the count for (a_2, b_2) is 3 while other counts are 1. We therefore sort the key combinations by their cadinality and will place (a_2, b_2) tuples first. Assuming the order of other key combinations remains unchanged, we can use the this order to place them across nodes (Section 4.3). The mappings M_A and M_B are initially empty and thus the key combinations (a_2, b_2), (a_1, b_1), (a_3, b_3) will each be placed randomly. Let us assume for this example that we have two nodes, and pairs (a_1, b_1) and (a_3, b_3) are placed on the node 0, while (a_2, b_2) tuples are placed on node 1. The mappings are now $M_A[a_1] = M_A[a_3] = M_B[b_1] = M_B[b_3] = 0$ and $M_A[a_2] = M_B[b_2] = 1$.

Given the mappings M_A and M_B, both (a_1, b_2) and (a_3, b_2) can be placed in either node. Thus, during step 2, they will be placed randomly. Let us assume that (a_3, b_2) is placed on node 0 and (a_1, b_2) is placed on node 1. The placement is shown in Table 1.

The key combinations (a_1, b_1), (a_2, b_2), and (a_3, b_3) have only one possible placement given the values in mappings M_A and M_B. The (a_1, b_2) and (a_3, b_2) key combinations however, can be placed on either node. We examine the network cost for placing (a_1, b_2) on both nodes in Table 2. We show that if (a_1, b_2) is moved from node 1 to node 0, we reduce the network cost of the two track joins. We assume for simplicity that all attributes have size 1. The cost of the tracking phase and sending node-id messages are excluded. When (a_1, b_2) is on node 1, the track join on key a_1 will need to transfer the dimension a_1 tuple from node 0 to node 1, and the track join on key b_2 will transfer the b_2 tuple from node 1 to node 0 to join with (a_3, b_2), incurring a total network cost of 2. After moving (a_1, b_2) to node 0, the single key join for a_1 becomes local, so track join will save the

cost of broadcasting the a_1 dimension tuple. We then examine (a_3, b_2) and find that the network cost is the same for both nodes, so we do not change its placement.

The final placement of fact table F tuples will be that node 0 has all tuples except for the 3 (a_2, b_2) tuples, which are placed on node 1. Note that if the cardinality of (a_2, b_2) tuples is different (e.g., 1), then for this minimal example there would appear to be skew in the distribution of records to nodes. We expect such imbalances to even out across many keys (see Section 4.7.3 below).

4.7 Other Considerations

4.7.1 Small Tables. The data placement optimization only needs to include tables that are too large and too costly to broadcast. Very small tables that can be broadcast across all N nodes should not be included in the optimization, since the joins can be executed locally. The join attributes from the fact table that are relevant to broadcast tables should be removed during the projection of the fact table tuples in the first step. Determining which tables are to be broadcast and which are to be distributed across the network in commercial data warehouses is commonly chosen by the user alongside the schema. In our approach, instead of using a hash partitioning scheme on a specific attribute to distribute the large tables, we use a global optimization that periodically relocates these large tables to reduce the network traffic of joins.

4.7.2 Statistics. The statistics used during data placement optimization are threefold. First, we maintain the number of times each join was executed in the workload so far and compute the frequency of each join between the fact table and any large dimension table. Second, we maintain the number of tuples from each dimension table D_j that are participating in the join $F \bowtie D_j$ and normalize to the average selectivity. The weight θ_j for join $F \bowtie D_j$ is then the product of these two factors. The selectivity of the fact table is harder to take into account, since the selective condition might be on the join key or another independent column. Nevertheless, since the fact table is used in all joins that we optimize the placement for, we can ignore the selectivity of the fact table. Finally, we need to take into account the average width of tuples from F and D_j, as projected from the two tables before the $F \bowtie D_j$ join. The tuple width is then used to compute the network cost of track join. Every time we repeat the process, we take advantage of more recent statistics, as well as optimize the placement of newly inserted tuples.

4.7.3 Load Balancing. While steps 2 and 3 make no effort to guarantee load balancing across N nodes in the output data placement, we found it to be sufficient for common data warehousing workloads such as TPC-H, due to the randomness in choosing node locations for a particular key. We can guarantee load balancing by keeping N counters $(c_1, c_2, \ldots, c_N)$ with the total size of tuples placed on each node (both fact and dimension tuples) and instead of choosing random nodes, choose the node with the minimum total size of tuples so far. We can also ensure that if a node reaches a certain threshold, we do not allow any tuples to be placed. Nevertheless, in our experiments, we found that even under skew, the dataset is relocated and load balancing is maintained.

4.7.4 Incremental Updates. Inserting new tuples after the workload has been optimized is not a problem. We can place the new tuples in any node of the cluster. The next iteration of the optimization will include the new nodes in the optimization. We do, however, need a way to locate tuples if we need to perform updates, since neither the fact table nor the dimension tables are partitioned by hashing a specific set of attributes. The solution to this problem is to insert new tuples by hashing and maintain a hash table of mapping through the N cluster nodes after every relocation of data. The location of each fact table tuple will be stored in the $\mathsf{hash}(k_1, k_2, \ldots, k_\delta, N)$ node and the location of each dimension table (D_j) tuple will be stored in the $\mathsf{hash}(k_j)$ node. We

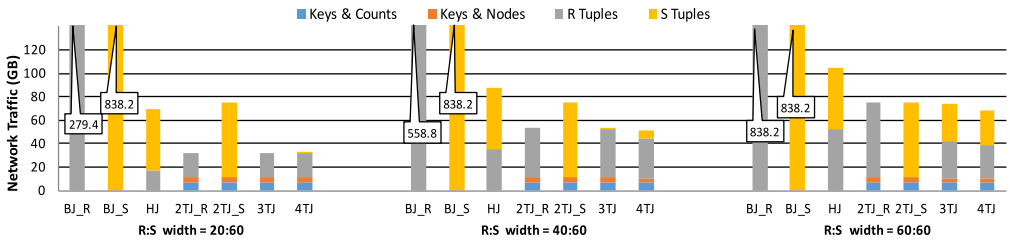


Fig. 3. Synthetic 10^9 unique R tuples $\bowtie$ 10^9 unique S tuples (1×1 key matches).

will need to send two messages instead of one for each update, one to send the update to the node that stores the location and from there to the actual location. If the data warehouse is only updated in large batches, then the cost of this process is similar to that of a regular track join that batches the tracking of the location of each tuple in larger operations.

5 EXPERIMENTAL EVALUATION

Our evaluation is split into three parts. First, we simulate distributed joins to measure network traffic across encoding schemes (Section 5.1). Second, we implement and measure track join against hash join execution times (Section 5.2). Third, we evaluate the data placement strategies based on simulated distributed joins (Section 5.3).

5.1 Track Join Simulations

In the simulations shown in this section, we measure the total network traffic. We assume 16 network nodes and show 7 cases of distributed join algorithms: broadcast join $R \rightarrow S$ (BJ_R) and $S \rightarrow R$ (BJ_S), hash join (HJ), 2-phase track join $R \rightarrow S$ (2TJ_R) and $S \rightarrow R$ (2TJ_S), 3-phase track join (3TJ), and 4-phase track join (4TJ).

In the three experiments shown in Figure 3, the width of R tuples varies between 20 and 60 bytes, while the S tuples are fixed at 60 bytes. The key width (included in the tuple width) is always 4 bytes. The tables have equal cardinality (10^9) and have (both the same) unique keys. Thus, track join selectively broadcasts tuples from the table with smaller payloads to the one matching tuple from the table with larger payloads and 2-phase track join is sufficient to minimize the network traffic. The distribution of both R and S keys is uniform random as is the placement across nodes.

In the general case, hash join has $1/N^k$ probability in order for all k matching tuples to be on the same node that matches the hash destination of the join key. For track join, this probability is $1/N^{k-1}$, because the collocation can occur in any node. When both tables have almost unique keys (Figure 3), the network cost gap between hash join and track join is maximized, since $k \approx 2$ is the lower bound for equi-joins.

In the three experiments shown in Figure 4, we vary the tuple width ratio R and S as in Figure 3, but we assume a foreign-key join where R has 10^8 unique tuples and S has 10^9 tuples. The distribution of both R and S keys is uniform random as is the placement across nodes. The match rate is 100%, thus each unique tuple in R is repeated 10 times in S. The cost of broadcasting R is competitive and is actually better than hash join when the tuple width ratio of R to S is 1/3. Again, the network cost is minimized using 2-phase track join that broadcasts the R tuples to all locations with matching S tuples.

In Figure 5, we show how locality can affect track join performance. We assume R to have 200 million 30-byte tuples with unique keys and S to have 1 billion 60-byte tuples. Each distinct join key in S is repeated 5 times but key locations follow the patterns that are specified below each chart. The pattern `5,0,0,...` denotes that all 5 key repeats are on a single node. The pattern

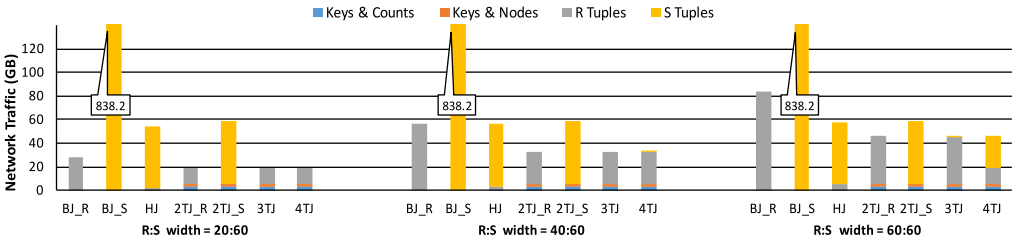


Fig. 4. Synthetic $2 \cdot 10^8$ unique R tuples $\bowtie$ 10^9 non-unique S tuples (1×5 key matches).

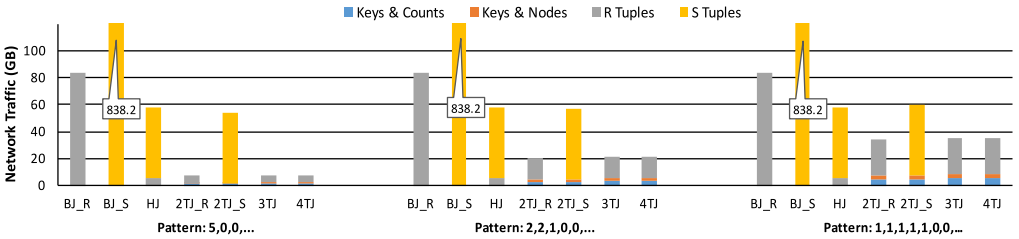


Fig. 5. Synthetic $4 \cdot 10^7$ unique R tuples $\bowtie$ $2 \cdot 10^8$ S tuples (1×5 key matches).

`2,2,1,0,0,...` represents 2 nodes with 2 repeats of the same key and 1 node with 1 repeat. The pattern `1,1,1,1,1,0,0,...` denotes that all repeats are on different nodes. Naturally, the nodes for placing the keys to satisfy each pattern, are chosen randomly per distinct join key. The distribution of R and S keys and the placement of R tuples is uniform random. The distinct locations of S tuples are also uniform random placing the specified number of tuples per distinct location as discussed above. Overall, these placements are artificially generated to represent different degrees of locality.

As shown in the left part of the result, when all repeats are collocated on the same node, regardless of which node that may be, track join will only transfer matching keys to a single location. When the tuple repeats follow the `2,2,1,0,0,...` pattern, still better than uniform placement, the traffic is still reduced compared to hash join. Note that random placement collocates repeats to some degree. Note that in the `1,1,1,1,1,0,0...` pattern, track join broadcasts each unique R tuple to the locations of the S tuples, since each R tuple is 30 bytes and each S tuple is 60 bytes, thus transferring $5 \cdot 30 = 150$ bytes in the worst case excluding the cost of tracking. Hash join transfers all tuples transferring $30 + 5 \cdot 60 = 330$ bytes in the worst case.

We now do the same experiment using small tables. Each table has 40 million tuples with unique keys and repeats each key 5 times. Since each key has 5 repeats on each table, we generate 25 output tuples per distinct join key. The datasets were generated to have the same number of output tuples (1 billion). In Figures 6 and 7, we use 200 million tuples with 40 million distinct join keys for both tables, thus each distinct join key produces 25 output tuples. The tuple widths are again 30 bytes for R and 60 bytes for S. Note that in Figure 6 with no collocation, 3TJ chooses the best direction dynamically given the tuple widths but 4TJ determines more efficient schedules. This happens because the sets of locations per table are determined independently and can have overlaps.

The difference between Figures 6 and 7 is whether the tuples across tables are collocated. In the first, the `5,0,0,...` configuration denotes that all repeats from R are on a single node and that all repeats from S are also on a (different) single node. We term this case, shown on Figure 6, *intra-table* collocation. When same key tuples from across tables are collocated, locality is further

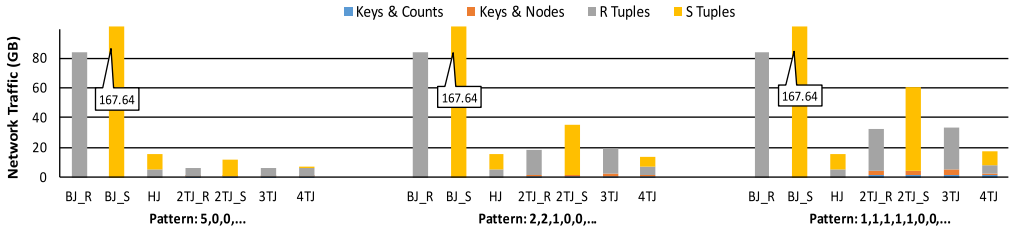


Fig. 6. Synthetic $2 \cdot 10^8$ R tuples $\bowtie$ $2 \cdot 10^8$ S tuples with $4 \cdot 10^7$ unique keys (5×5 key matches intra-table collocated).

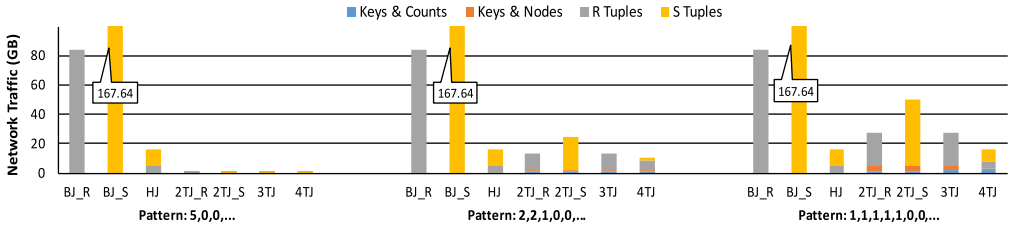


Fig. 7. Synthetic $2 \cdot 10^8$ R tuples $\bowtie$ $2 \cdot 10^8$ S tuples with $4 \cdot 10^7$ unique keys (5×5 matches inter-table collocated & intra-table collocated).

increased. We term this case, shown on Figure 7, *inter-table* collocation. When all key repeats are collocated, track join transfers no payloads. The messages used during the tracking phase to track the tuple locations can only be affected by the same case of locality as hash join.

We further explain the difference between *intra-table* and *inter-table* collocation with an example. Assuming the `2,2,1,0,0...` distribution with intra-table collocation, we pick three distinct random nodes for table R to place 2, 2, and 1 tuple, respectively, and three distinct random nodes for table S to place the S tuples. The selection of nodes is independent per table. If we also have *inter-table* collocation, then we pick a single set of three distinct random nodes for both R and S.

Figures 4, 5, 6, and 7 show the weakness of 2-phase and 3-phase track join to handle all possible equi-joins. If both tables have values with repeating keys shuffled randomly with no repeats in the same node, then 4-phase track join is similar to hash join. However, track join sends far less data when the matching tuples are collocated.

Figures 3 and 4 showed that 2-phase track join is sufficient for N–1 joins without collocation. To avoid sending N tuples of one table to 1 tuple of the other, we can pick the right side to selectively broadcast in 2-phase track join via the optimizer or use 3-phase track join. Figures 5, 6, and 7 show specific cases of collocation and M–N join cases. In such cases, 4-phase track join still outperforms hash join by being able to detect and take advantage of collocation as well as ensure that M–N joins do not behave like a broadcast join.

These cases also highlight another advantage of 4-phase track join if used for joining intermediate results. Because 4-phase track join always adapts either to distinct join cases (albeit being a little less efficient than 2-phase track join), or to M–N joins, which can be very costly when they produce intermediate results if the optimizer makes an erroneous approximation of the cost. At the same time, if intermediate results generate locality, 4-phase track join can take advantage of it dynamically.

The above experiments use synthetic data and a fixed number of nodes. Next, we compare the network traffic of broadcast join, hash join, and 4-phase track join on tables from the TPC-H

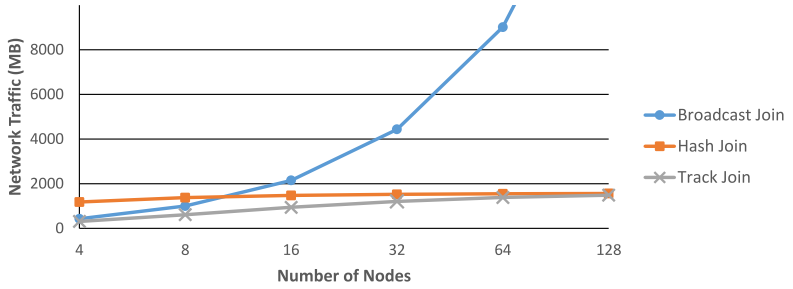


Fig. 8. Network cost of joining TPC-H tables with varying number of nodes.

benchmark [9] with varying number of nodes. Figure 8 shows the network costs of these join methods for a foreign key join between the CUSTOMER and ORDERS tables on the 4-byte CUSTKEY column, with scale factor = 10. The number of tuples in ORDERS is ten times as many as the number of tuples in CUSTOMER. We randomly place all the tuples across the nodes, and use the same 96-byte payload for both tables. As shown in Figure 8, the cost of broadcast join scales linearly with the number of nodes, while the other join methods do not. As a result, broadcast join performs much worse than hash join when the number of nodes is large (more than 10 nodes). Track join behaves like a broadcast join on a small number of nodes, because of a lower broadcast cost, and hash join incurs high cost, because it ignores the fact that many data are already collocated on the same node (so there is no need to hash them to another node again). Track join behaves like a hash join on a large number of nodes. This result shows that track join automatically adapts to the number of nodes to exploit data locality for lower network cost. Note that track join is not outperformed by hash join as the node number increases, although they seem to converge in this particular experimental setting. In general, the costs of hash join and track join depend on the relative payload size, number of joining tuples, and number of distinct keys. For uniformly distibuted data as in this experiment, the models in Section 3.1 accurately reflect the network costs and can be used to determine the better join algorithm. Example 3.1 shows a situation where track join would outperform hash join by a large margin even as the number of nodes increases.

In the rest of our experimental evaluation for both simulation and the implementation of track join (Sections 5.1 and 5.2), we will use queries from real analytical workloads. We did extensive profiling among real commercial workloads with one sophisticated commercial DBMS. The commercial hardware where we did the profiling is an 8-node cluster connected by 40Gbps InfiniBand. Each node has 2X Intel Xeon E5-2690 CPUs at 2.9GHz, 256GB of quad-channel DDR3 RAM at 1600MHz, 1.6TB flash storage with 7GB/s bandwidth, and 4X 300GB hard disks at 10,000RPM.

In the next experiment, we use the slowest query (Q1) from a real workload (X). All query plans were generated by the same commercial DBMS. Query Q1 includes a hash join that we profiled to take 23% of its total execution time. Details about the columns of two inputs participating in the slowest join are shown in Table 3. They are both intermediate relations and consist of 769,845,120 and 790,963,741 tuples, respectively. The join output has 730,073,001 tuples. Overall, query (Q1) executes 7 joins, after applying selective conditions on 4 relations, and then performs a single final aggregation.

All columns of the Q1 join are of number type and would use variable byte encoding by default, if left uncompressed. Sending uncompressed data over the network is expensive. Compression helps all join algorithms reduce their network traffic, since the column values are compressed using a global dictionary and use the minimum number of bits. Specifically for track join, we send keys over the network more often than we do payloads. Also, the tracking phase cannot be

Table 3. Column Details of R and S Tables Joined in Q1

R column	Cardinality	Bits	*S* column	Cardinality	Bits
J.ID (key)	769,785,856	30	J.ID (key)	788,463,616	30
O.U.AMT	26,308,608	25	T.ID	53	6
C.ID	359	9	S.B.ID	95	7
T.B.C.ID	233,040	18	T.ID	53	6
			J.T.AMT	9,824,256	24
			T.C.ID	297,952	19
			S.C.AMT	11,278,336	24
			M.U.AMT	54,407,160	26
	769,845,120			790,963,741	

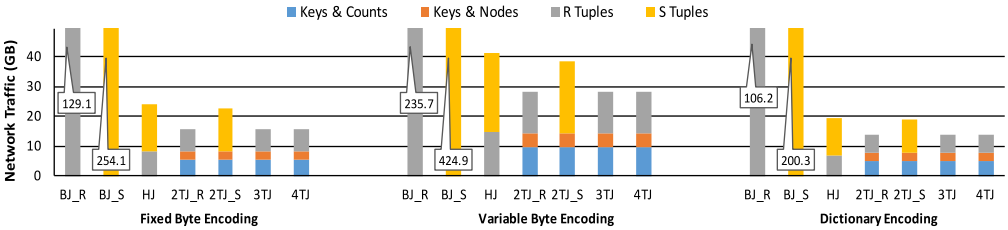


Fig. 9. Slowest join of slowest query (Q1) of workload X (original data placement).

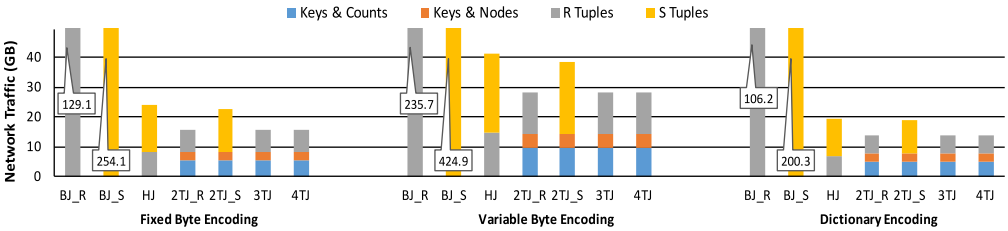


Fig. 10. Slowest join of slowest query (Q1) of workload X (shuffled data placement).

optimized, since the key locations have not yet been tracked. Thus, for track join, compressing the key columns will have a greater impact in reducing the network traffic compared to compressing the payloads. Note that compression is orthogonal to hash join or track join and the following figures show that both track join and hash join reduce the network traffic when the base data is compressed but track join still outperforms hash join.

In Figure 9, we show the network traffic of the slowest join in Q1. We use three different encoding schemes, fixed byte (1, 2, or 4 byte) encoding, fixed bit dictionary encoding, and variable byte encoding. The variable byte encoding is base-100 and stores two decimal digits per byte. The input of Q1 exhibits pre-existing locality, since broadcasting R transferred $\approx 1/3$ of R tuples compared to hash join. We would expect both hash join and track join to transfer the whole R table over the network. We shuffle the placement of the tuples across the cluster to remove any data locality and repeat the experiments in Figure 10. As expected, the network traffic increases, as we now have to transfer most R payloads over the network. Nevertheless, track join still outperforms hash join.

In the X workload, the key columns have less than 1 billion distinct values, and thus, fit in a 32-bit integer if dictionary encoded. However, the actual values do not fit in the 32-bit range. In some

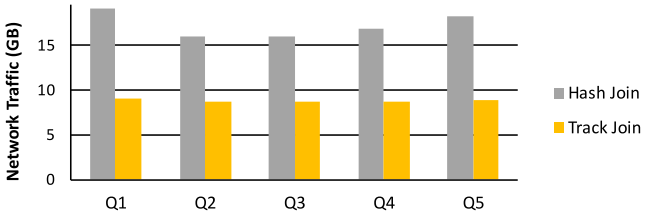


Fig. 11. Common slowest join in slowest queries 1–5 of X (dictionary compressed).

cases, we can effectively reduce the data width using simple compression schemes [29] that allow fast decompression and do not require accessing large dictionaries that are unlikely to remain CPU-cache-resident. However, dictionaries reduce the width of data types and, if sorted, can perform equality and ordering operations with the dictionary index. Decoding arbitrary n-bit dictionary indexes can be implemented efficiently [29, 43, 61]. However, in many operations, including non-selective equi-joins, dictionary accesses are redundant. In our results for compressed workload X, we omit dictionary dereference, since the join can use the encoded values of the join keys, given that the join key columns are encoded using the same dictionary.

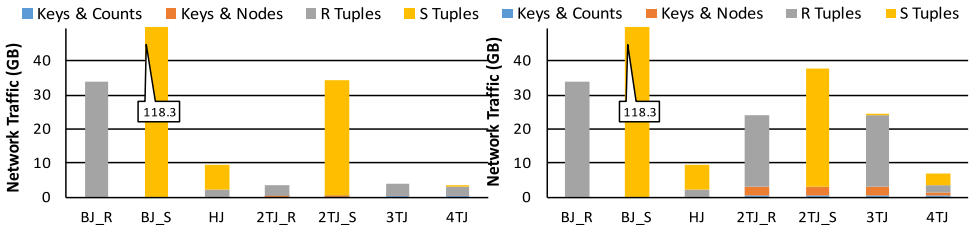
Compared to the synthetic experiments, the results from the X workload exhibit a much higher ratio of network traffic spent in the tracking phase and for control messages. This difference is due to the size of payloads, which is much lower here. We used larger payloads in the synthetic experiments to cover both cases, since workload X already covers small payloads.

In all experiments, we omit dictionary dereference traffic. Dictionaries are also assumed to be compressed before the join, using the minimum number of bits required to encode the distinct values of the intermediate relation, even if some join keys have been filtered before the join. This is the optimal encoding scheme in cases where most keys are unique and there is specific value ordering. If the dictionaries are not compressed after every selection, which is the expected scenario, then we would still rely on the original dictionary that encodes all distinct values of original tables. Encoding to the optimal compression before every join to achieve the minimal network traffic is possible, but the total gain be outweighed by the total time needed to re-compress.

Workload X contains more than 1,500 queries and the same most expensive hash join appears in the five slowest queries, taking 23%, 31%, 30%, 42%, and 43% of their execution time. The slowest five queries require 14.7% of the total time required for all queries in the workload (>1500) cumulatively and spend ≈65–70% of their time on the network. Time was measured by running the entire X workload on the commercial DBMS we use. The hash join in all five queries operates on the same intermediate result for the key columns, but each query uses different payloads. Queries Q2–Q5 are similar to Q1 and do 4–6 joins followed by a final aggregation step.

In Figure 11, we compare hash and track join on the slowest five queries, using dictionary coding with the minimum number of bits, which is also the optimal compression scheme that we can apply here. The total bits per tuple for R:S are 79:145, 67:120, 60:126, 67:131, and 69:145, respectively. The network traffic reduction is 53%, 45%, 46%, 48%, and 52%, respectively. Here, both inputs have almost entirely unique keys. Thus, assuming we pick the table with shorter tuples (R here) to selectively broadcast during 2-phase track join, all track join versions have similar performance. Note that the joins are not identical, they have minor variations in the projected columns.

In our last experiment, we use a second real workload (Y) that has ≈90 analytical queries. We use the slowest query in Y that executes 9 joins and isolate the most expensive hash join, based on profiling results from the commercial DBMS. The most expensive query takes 20% of the time required to run all queries of Y cumulatively and the hash join takes about 36% of the total



(a) Original data placement (b) Shuffled data placement

Fig. 12. Slowest join in slowest query of workload Y using variable byte encoding.

execution time of the most expensive query. In Figure 12(a), we show the network traffic of the join using the original data placement. In Figure 12(b), we shuffle the inputs to remove all pre-existing locality.

The R table has 57,119,489 rows and the S table has 141,312,688 rows. The join output consists of 1,068,159,117 rows. The data are uncompressed and use variable byte encoding. The tuples are 37 and 47 bytes wide, the largest part of which is due to a character 23-byte character column. The large output of the query makes it a good example of high join selectivity that has different characteristics from foreign and candidate key joins. Late materialization could cause this query to be excessively expensive, if payloads are fetched at the output tuple cardinality.

When the input is shuffled, the 4-phase version is better than hash join, while the other versions almost broadcast R due to key repetitions. The 2-phase track join can be prohibitive if there is no locality, and we choose to broadcast S tuples to R tuple locations. This is due to sending S tuples to too many nodes to match with R and is shown in Figure 12(b). The opposite broadcast direction is not as bad but is still three times more expensive than hash join. 4-phase track join adapts to the shuffled case and transfers 28% less data than hash join. This experiment is a good example of the adaptiveness of track join. We need many repeated keys, since the output cardinality is 5.4 times the input cardinality per distinct join key. The naïve selective broadcast of 2-phase and 3-phase track join are similar to broadcasting R to all nodes.

5.2 Track Join Implementation

For the next set of experiments, we implement hash join and track join in C++ using POSIX system calls but no extra libraries. The network communication uses TCP and all local processing is done on RAM. We use four identical machines connected through an 1 Gbit Ethernet switch, each with 2X Intel Xeon X5550 (4 physical cores with two hardware threads per physical core) CPUs at 2.67 GHz and 24GB ECC RAM. We measured the RAM bandwidth at 28.4GB/s for reading data from RAM, at 16GB/s for writing data to RAM, and at 12.4GB/s for RAM to RAM copying. We compile the C++ source code using GCC 4.8 with `-O3` optimization. The operating system is the Ubuntu 14.04 GNU/Linux distribution and the Linux kernel version is 3.2. In all of our experiments, we use all 16 hardware threads of each machine.

Our platform is severely network bound, since each edge can transfer 0.093GB/s, if used exclusively. Although track join was presented using a pipelined approach in Section 2, we separate CPU and network utilization by de-pipelining all operations of track join. Thus, we can estimate performance on perfectly balanced and much faster networks by dividing the time of network transfers accordingly. We assume the data are initially on RAM and the join results are also materialized on RAM. If the result is too large to fit on RAM, then we overwrite the earlier parts of output on RAM. In Table 4, we show the CPU and network use for hash join and all track join versions.

Table 4. CPU & Network Time of the Slowest Join of the Slowest Query of X and Y

Dataset	Placement	Time	HJ	2TJ	3TJ	4TJ
X	Original	CPU time (s)	4.308	5.396	6.842	7.500
		Network time (s)	87.754	38.857	44.432	44.389
	Shuffled	CPU time (s)	4.598	6.457	7.601	8.290
		Network time (s)	87.828	61.961	67.117	67.518
Y	Original	CPU time (s)	2.301	2.279	3.355	2.400
		Network time (s)	30.097	10.800	11.145	10.476
	Shuffled	CPU time (s)	2.331	2.635	3.536	2.541
		Network time (s)	30.191	28.674	29.520	18.230

We present more details on the data encoding used in Table 4. Our implementation uses fixed byte widths. For X, we use 4-byte keys, 7-byte Rpayloads, and 18-byte S payloads. For Y, we use 4-byte keys, 33-byte R payloads, and 43-byte S payloads. The node locations use 1 byte and the counts use 1 byte for X and 2 bytes for Y.

Using workload X in the original order, 2-phase track join (using the $R \rightarrow S$ selective broadcast direction) increases the CPU time by 25%, but reduces the network time by 56% compared to hash join. 3-phase and 4-phase track join increases the CPU time by 59% and 74% without further reducing the network traffic. If X is shuffled, then 2-phase track join increases the CPU time by 40% and reduces the network time by 29%.

Using Y in the original order, 2-phase track join suffices to reduce the network time by 64% compared to hash join without affecting the CPU time significantly. If Y is shuffled, then 2-phase and 3-phase track join increase the CPU time without saving any network traffic, due to excessive key repetitions. In fact, we would transfer more data than hash join, as shown Figure 12(b), if we used more nodes, because the number of nodes is less than the key repeats. However, 4-phase track join increases the CPU time by 9% compared to hash join but reduces the network traffic by 40%.

We can project track join performance on 10× faster network (10Gbit Ethernet) by scaling the network time. For X, track join is ≈ 29% faster than hash join. For Y, track join is ≈ 37% faster than hash join. Note that the hardware bundle of the commercial DBMS uses both faster CPUs (2X 8-core CPUs at 2.9GHz) and network (40Gbit InfiniBand). If the network is too fast compared to the CPU, then track join is less useful. Still, if the network is slow, track join is likely to reduce the total execution time, even if there is no locality in the dataset. If the dataset exhibits locality, then track join will further increase the performance gap with hash join.

Table 5 shows the execution times (in seconds) per step for hash join. To execute hash join, we first hash partition both tables based on the join key. Then, we transfer each partition of both tables to the respective network node. Once the tables have been transferred over the network, each node has keys with the same hash destination. We then sort the tables locally using MSB radix-sort and perform a final merge-join on RAM. Thus, local joins are executed using a sort-merge-join approach. The transfer steps that use the network are significantly slower, since the experiment is measured on 1Gbit Ethernet. Nevertheless, since the steps are completely depipelined, we can scale the network time for faster networks independently of the other steps.

Table 6 shows the execution times (in seconds) per step for 4-phase track join. We start by sorting both tables locally. Then, we aggregate the tuples by key and count the number of tuples per group, to gather the information needed in the tracking phase. We then hash partition the keys and the counts by key and distribute them over the network. We can now merge the keys and counts to create a mapping of matching tuples for each distinct join key. This mapping is

Table 5. Distributed Hash Join Steps (in Seconds)

Step description	Dataset X		Dataset Y	
	Original	Shuffled	Original	Shuffled
Hash partition R tuples by key	0.347	0.350	0.054	0.054
Hash partition S tuples by key	0.478	0.477	0.167	0.167
Transfer R tuples	29.464	29.925	7.197	7.392
Transfer S tuples	57.199	57.142	22.550	22.945
Local copy tuples	0.115	0.115	0.039	0.039
Sort received R tuples by key	1.145	1.288	0.176	0.179
Sort received S tuples by key	1.627	1.777	0.535	0.572
Final merge-join	0.601	0.602	1.322	1.321

Table 6. 4-phase Track Join Steps (in Seconds)

Step description	Dataset X		Dataset Y	
	Original	Shuffled	Original	Shuffled
Sort local R tuples by key	0.979	1.300	0.182	0.182
Sort local S tuples by key	1.401	1.792	0.534	0.565
Aggregate (local) keys → counts	0.229	0.227	0.022	0.025
Hash partitition keys & counts by key	0.373	0.372	0.011	0.018
Transfer keys & counts	26.800	27.339	0.977	1.378
Local copy keys & counts	0.034	0.034	0.093	0.001
Merge received keys & counts	0.506	0.507	0.015	0.022
Schedule & partition by node id	1.627	1.650	0.035	0.047
Transfer $R \rightarrow S$ keys & node ids	7.277	10.913	0.346	0.532
Transfer $S \rightarrow R$ keys & node ids	6.046	1.562	0.135	0.247
Local copy keys & node ids	0.016	0.016	0.000	0.000
Merge received keys & node ids	0.237	0.235	0.007	0.012
Merge-join $R \rightarrow S$ keys & node ids ⇒ payloads & partition by node id	0.315	0.456	0.068	0.098
Merge-join $S \rightarrow R$ keys & node ids ⇒ payloads & partition by node id	0.355	0.204	0.067	0.082
Transfer $R \rightarrow S$ tuples	2.664	27.532	6.086	9.600
Transfer $S \rightarrow R$ tuples	0.001	0.001	3.235	6.462
Local copy $R \rightarrow S$ tuples	0.067	0.017	0.007	0.009
Local copy $S \rightarrow R$ tuples	0.138	0.037	0.021	0.008
Merge received $R \rightarrow S$ tuples	0.161	0.531	0.045	0.067
Merge received $S \rightarrow R$ tuples	0.141	0.066	0.043	0.045
Final merge-join $R \rightarrow S$	0.419	0.555	0.822	0.793
Final merge-join $S \rightarrow R$	0.342	0.161	0.518	0.556

used to generate the schedule of transfers that minimizes the network traffic for each cartesian product. We now transfer keys and count pairs that participate in the $R \rightarrow S$ selective broadcast and pairs that participate in the $S \rightarrow R$ selective broadcast separately. In fact, we actually have four cases, since each direction further splits the transferred pairs to those about migrating tuples and those about the later selective broadcast. After the pairs are merged on each node, the keys

are (locally) matched to their payloads and are transferred over the network. This phase includes both migrating and selectively broadcast tuples. At the final phase, we merge the received tuples by key and perform a final merge-join.

The approach used in Table 6 to execute 4-phase track join keeps the tables sorted by key to ensure we remain cache-conscious and avoid relying on large hash tables that would incur large numbers of cache misses. Furthermore, since track join needs the data grouped by key in multiple steps, it is preferable to re-merge parts sorted by key, rather than to scatter in a new large hash table each time. The pipelined approach that was described in Section 2 would benefit more from using large hash tables. Still, track join cannot overlap the separate *phases*, pipelining occurs inside each phase.

The scheduling cost and the local join cost vary depending on the dataset. For X, where we have mostly unique keys, scheduling is faster than the local sort-merge-join. However, Y features smaller inputs and a much larger output. In such a case, creating a cartesian product transfer schedule is more expensive than local sorting, but still insignificant compared to the total execution time. Similarly to the scheduling step, the final merge-join is much more expensive on Y, since it creates a far larger number of output tuples compared to the inputs.

5.3 Data Placement Optimization

In this section, we present the experimental evaluation of our data placement optimization algorithm (Section 4). The experiments in this section are simulated, since we are interested in the network traffic of subsequent joins, rather than specific execution time. Since we do not have access to the complete proprietary workloads of X and Y except for the slowest join, we will use the openly available dataset of the TPC-H benchmark [9], which is very frequently used to represent data warehousing applications.

In Section 4.1, we noted that the optimization technique focuses on a single fact table and multiple dimension tables in a star schema. While TPC-H is not a star schema, we can use the fact table and the largest dimension tables that are still organized as a star (sub-)schema. We use the LINEITEM table as our fact table. The first dimension table we use is the PARTSUPP table, joined with the fact table using the attributes L_PARTKEY and L_SUPPKEY (from the fact table LINEITEM), which is a composite foreign key to PS_PARTKEY and PS_SUPPKEY (from PARTSUPP). The second dimension table we use is the ORDERS table, joined with the fact table using the attribute L_ORDERKEY (from LINEITEM), which is a foreign key to O_ORDERKEY (from ORDERS). In the sequel, we shall denote the join key of the first dimension table (attributes *_PARTKEY and *_SUPPKEY) by A, and denote the join key of the second dimension table (attributes *_ORDERKEY) by B. Note that the width of the composite key A is 8 bytes, and key width of B is 4 bytes.

We first conduct experiments on joining the above three tables and then expand to other tables. We use a TPC-H scale factor of 10 for all experiments, to be able to simulate a large number of network nodes. The scale factor of the dataset is large enough to show the improvements of our data placement optimization. We use both the standard TPC-H dataset as well as another version of TPC-H that introduces skew in the dataset [7]. The skewed version of TPC-H repeats some fact table values more often than others, to represent real world data warehouses that exhibit skew. Specifically, we set the input parameter of the skewed data generator to 1.0, and as a result, the cardinality of key A of the fact table exhibits a zipfian distribution, while the distribution of key B remains unchanged. The number of distinct key A values in the skewed fact table is 7,028,277, while the standard LINEITEM table has 7,995,953 distinct key A values. The number of distinct key B values is 15 million in both standard and skewed tables. Because of this skewness, the number of distinct (A, B) key combinations in the skewed fact table is 43,361,365, much smaller than that in the standard fact table where almost all 60 million key combinations are unique.

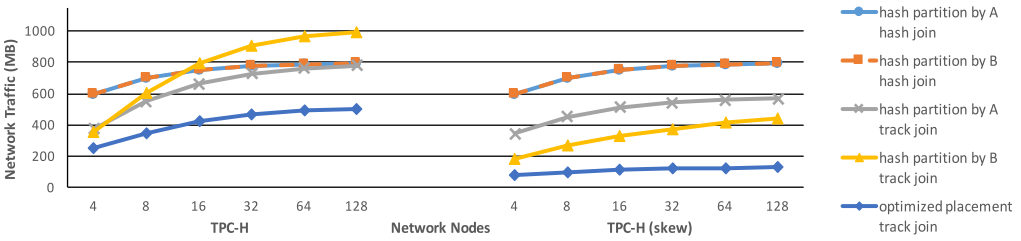


Fig. 13. Network cost of joins per data placement of TPC-H with varying number of nodes.

We compare the optimized placement against hash co-partitioning the fact table by one of the join keys, which is the optimal placement for the join on the partitioning key. Specifically, we show the network cost if we use hash join to execute the two joins, when the fact table is hash partitioned by A, or when it is partitioned by B. The dimension tables are hash partitioned by their keys, respectively. We also show the network cost for using track join instead of hash join, using the same two placements. Finally, we show the network cost for executing the two track joins after we optimize the data placement of the three participating tables with the algorithm described in Section 4. We use the 4-phase variant for track join results in this section. By default, we set the number of nodes to 16 and the payloads for every table to 16 bytes. Assume also in the default workload the joins are executed at the same frequency and thus their weight is 0.5. In the following experiments, we shall vary each one of these parameters and report the total weighted costs.

Figure 13 shows the network cost of executing track join across a varying number of nodes for TPC-H and the skewed TPC-H. Hash join performance is the same regardless if we partition by A or B. When hash partitioning is used, the total cost shown here is essentially the cost of the join on the non-partitioning key, since the other join can be executed locally and leads to zero network cost. Since the dimension tables are hash partitioned by their join keys, respectively, the hash join on the non-partitioning key always sends the fact table tuples to the hashed locations, no matter they are partitioned by A or B. Thus, their total costs are the same. As the number of nodes increases, the cost increases, since there is increasingly lower chance for the two hash functions to map the A and B values of a fact table tuple to the same location.

The network costs of using track join also increase as the number of nodes increases, due to the diminishing effect of data locality. For the standard TPC-H, using track join when the fact table partitioned by A performs better than partitioned by B, because the key width of A (8 bytes) is larger than that of B (4 bytes). When partitioned by A, both the tracking phase and the subsequent joining phase will send less data over the network. At the tracking phase, we transfer the counts of the 4-byte B keys instead of the 8-byte A keys. After tracking, 4-phase track join will send dimension table tuples to the node locations of the matching fact table tuples due to the smaller tuple width of dimension table tuples. When `LINEITEM` is partitioned by A, the smaller `ORDERS` tuples will be sent over the network, resulting in overall smaller costs. In fact, when the fact table is partitioned by B and track join transfers the wider `PARTSUPP` tuples, the cost becomes higher than using hash join when the number of nodes becomes large. Although in this case track join chooses the better direction to send `PARTSUPP` tuples, rather than the `LINEITEM` tuples as in hash join, the overhead of the tracking phase outweighs the benefit of a better transfer direction, leading to even higher cost than hash joins.

For TPC-H, after optimizing the placement, we reduce the network traffic by 30–35% compared to the second best approach, typically track join partitioned by B. Compared to hash joins, the savings can be up to 60%. When TPC-H has no skew, fact table key combinations are mostly unique.

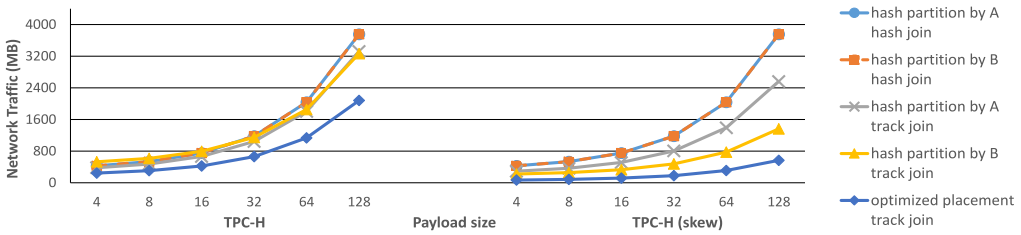


Fig. 14. Network cost of joins per data placement of TPC-H with varying payload size.

The network cost reduction mainly stems from intentionally collocating the fact table tuple with matching dimension table tuples, where matching tuples from multiple dimensions are collocated with the fact table tuple. In contrast, hash partition by a specific attribute would collocate the fact table with tuples from a single dimension table based on the attribute hashed. Joins on the other dimensions incur high costs, since there is little chance for the matching tuples to be collocated.

For the skewed TPC-H, in addition to intentionally collocating tuples, the placement optimization prioritizes key combinations of higher cardinality and further reduces network traffic. Also, when a key combination has multiple placements, we refine the placement to pick the best option. The refinement step takes into account low granularity information that corresponds to single-key joins (cartesian products) across multiple dimensions. As a result, track join with optimized data placement reduces the network traffic by 84–87%, compared with hash joins with hash partitioning. Compared with track joins with the best hash partitioning, the saving in total network cost is 58–71%.

We also observe on the right of Figure 13 that track joins with hash partitioning achieves lower costs after introducing the skewness, which is expected, since track join can leverage the data distribution to send less data. Interestingly for track join, in the skewed TPC-H, partitioning by B is the better option, whereas in the standard TPC-H partitioning by A is better. This result is because of different cardinality distribution of key A in the fact table. When the fact table is partitioned by B, track join will send the PARTSUPP tuples to matching node locations. For each foreign key, the number of matching locations is bounded by the number of total nodes and the number of matching tuples. In the standard TPC-H, the cardinality of key A in the fact table follows a normal distribution. On average, each foreign key is mapped to 7.5 fact tuples. In the skewed TPC-H, the cardinality of key A in the fact table follows a zipfian distribution, so a lot of keys have very few tuples. Although a very small number of keys have a lot of tuples, the number of matching node locations in track join is bounded by the number of total nodes, so the average number of node locations is much smaller than 7.5. For example, with 16 nodes, the average is only 3.1 when the matching node locations are capped by 16. Therefore, even though the key width of A is larger, track join still sends much less data over the network and the total cost becomes lower than partitioning by A in the skewed TPC-H.

Figure 14 repeats the same experiment as Figure 13, but sets the number of nodes to 16 and varies the size of the payloads. In this figure, we set the same payload size for each of the three tables, so track join always chooses to selectively broadcast the narrower dimension tuples. Note that track join performs better with larger payload size, since it avoids sending unnecessary tuples by tracking only the keys. For example, for hash partitioning by B, the cost of track join becomes lower than the cost of hash join if the payload size is larger than 32 bytes, while track join actually performs worse at the default 16-byte payload, as we discussed earlier. For the standard TPC-H, track join sends about 12% less data over the network compared to hash join, under the best hash partitioning. If we optimize the data placement, then track join further reduces the network traffic

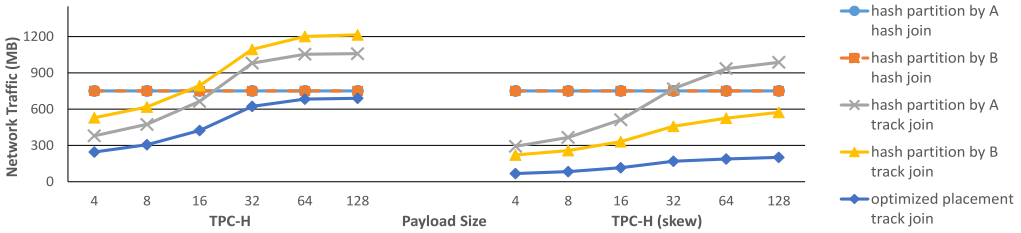


Fig. 15. Network cost of joins with varying payload size of dimension tables only.

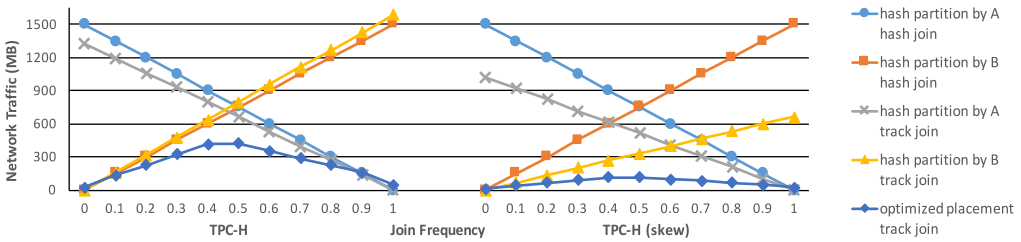
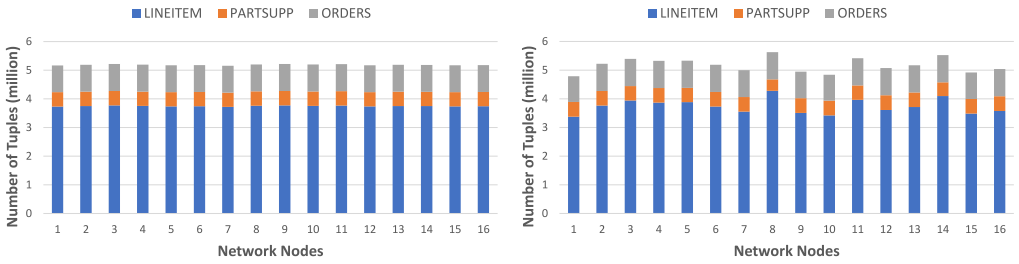


Fig. 16. Network cost of joins per data placement of TPC-H with varying join weight.

by ≈35%. For the skewed dataset, even without optimizing the placement, track join sends 49–64% less data over the network compared to hash join. If we also optimize the data placement, then track join further reduces the network traffic by 60–70%. In contrast with hash join, track join with optimized data placement reduces the network traffic by 85%, sending almost 7 times less data over the network. This result shows that the data placement optimization is even more useful when the payload size is large.

Different from Figure 14, Figure 15 varies only the payload size of the dimension tables, while the size of fact table payload is the default 16 bytes. Since the size of fact table tuples is fixed ($8 + 4 + 16 = 28$ bytes), the cost of hash joins remains unchanged. When the payload size is small (4–16 bytes), track join chooses to send dimension table tuples to matching locations; when the payload size is large (32–128 bytes), track join chooses to send fact table tuples to the node locations of matching dimension tuples. For the standard TPC-H, when the payload size of dimension tables is large, track join performs worse than hash join if the fact table is hash partitioned, since track join does the same work as hash join for sending the tuples, and incurs some extra overhead for the tracking phase. By intentionally collocating tuples for both dimensions, the optimized placement still reduces the total network traffic by 9–35%, compared with hash partitioning by using the best of track join and hash join. The reduction is up to 70% on the skewed dataset compared with the second best approach, i.e., using the track join when the fact table is partitioned by B.

Next, we vary the weight between the two joins while fixing both the number of nodes and the size of payloads to their default values. Figure 16 shows the total weighted network traffic for the two joins, where the horizontal axis value θ represents the weight of joining the first dimension (i.e., LINEITEM $\bowtie$ PARTSUPP). The weight of the other join LINEITEM $\bowtie$ ORDERS is then $(1 - \theta)$. Note that the weight θ encapsulates both the frequency of the join compared to the other join, and the average selectivity of the dimension table before participating in the join. When θ is small, the total weighted cost is mainly the cost of the join with ORDERS. For hash partitioning by B, this join can be executed locally, so the costs of both track join and hash join are low; for hash partitioning by A, this join incurs a high cost, since matching tuples are not collocated. As θ increases, the cost of the join with PARTSUPP becomes more important, so the total cost of hash partitioning



(a) TPC-H (b) TPC-H (skew)

Fig. 17. Load balancing after optimizing data placement.

by A gradually decreases, while the cost of partitioning by B increases. For hash join, the better dimension for partitioning is by A if $\theta > 0.5$ and by B if $\theta < 0.5$. For track join, the choice of the better dimension for partitioning must consider the differences in key width and cardinality distribution, as we discussed earlier for Figure 13. In contrast, the optimized data placement avoids choosing a fixed dimension for partitioning, and collocates tuples that have more impact on the overall costs across different weights. As shown in Figure 16, the data placement optimization reduces the network traffic by up to 36% in the standard TPC-H and by 65% in the skewed TPC-H. Even if one join dominates the workload, the network cost of track join after optimizing the data placement remains low, offering a robust option for reducing the network traffic across multiple joins that use different attributes of the fact table.

Under the default parameter settings, Figure 17 shows the number of tuples for every joining table after the data placement optimization. For the standard TPC-H, as shown in Figure 17(a), optimizing the data placement leads to almost perfect load balancing across the 16 nodes, thanks to the randomness in choosing node locations for a particular key. For the skewed version of TPC-H, Figure 17(b) shows that the tuples of the three tables are also placed relatively balanced across the nodes, despite of the zipfian distribution of key A in the fact table. Specifically, Node 8 has the most number of `LINEITEM` tuples, 14.2% more than the average number of fact table tuples over all nodes, while Node 1 has the least number of `LINEITEM` tuples, 9.9% less than the average. We achieve this load balancing result, because we pick node locations at random from mapped location candidates in Algorithm 12. In the absence of an explicitly adversarial input order of key combinations to this algorithm, our placement optimization avoids mapping all tuples to one or two nodes, and results in reasonably balanced partitions even in the presence of skew.

Since the optimized data placement does not collocate tuples by one particular dimension and instead clusters tuples from multiple dimensions together with fact table tuples, join algorithms that are unaware of this data locality can not benefit from such a placement. Figure 18 shows the network traffic of join algorithms on different data placement plans, under the default parameter settings. For each join algorithm, we show the total cost on the best hash partition (between by A and by B), the placement by only clustering tuples (Section 4.3) without further refinement using the cost model of track join (Section 4.4), and the complete optimized placement using both clustering and refinement steps.

For hash join, the cost of using best hash partition is essentially one of the joins that cannot be executed locally. However, if we cluster fact table tuples with matching tuples from both dimensions, then neither of the two joins can be executed locally using hash joins with standard hash functions. Both joins have to redistribute most tuples over the network, even though some tuples may have been placed at the same node by the cluster step. As shown in the figure, the total cost is almost twice of the cost under the best hash partition. Further placement refinement has no effect

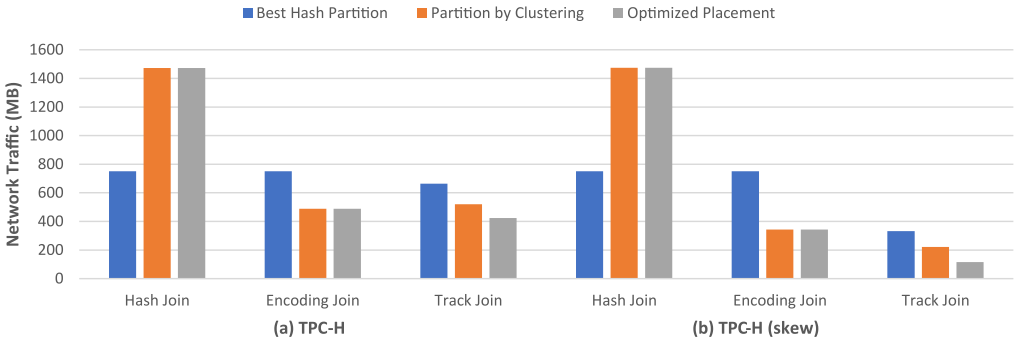


Fig. 18. Effect of different placement strategies on join algorithms.

on hash joins, since the process uses the cost model specific to track joins. The degree of skew also has almost no effect, because the tables have the same cardinalities in the two datasets.

Different from hash join, the encoding join discussed in Section 4.3.1 directly uses the mapping obtained at the clustering step to decide the locations for sending tuples, so it avoids sending a majority of the tuples that have been collocated, reducing over 66% of the network traffic compared with hash join on the clustered placement. Since encoding join is aware of the matching node locations, it is able to exploit some of the benefits of track join, but has no further improvement for refinement specific to track join.

For track join, both the clustering step and the refinement step reduce the network traffic. Compared with the best hash partition, for the standard TPC-H the clustering step reduces 22% of the network traffic and the refinement step further reduces 15%. For the skewed TPC-H, the refinement step has more impact on reducing the network traffic. Compared with hash partition, the clustering and refinement steps reduces 31% and 33% of the network traffic, respectively.

For data warehouse with more complex schemas, the data placement optimization can be applied to the central star sub-schema including the fact table and first-level dimension tables. In our final experiment, we expand the schema used in previous experiments by including a second layer of dimension tables PART and CUSTOMER in TPC-H. In this small snowflake schema, PART is linked with PARTSUPP via the foreign key PARTKEY, and CUSTOMER is linked with ORDERS via the foreign key CUSTKEY. The key width of both PARTKEY and CUSTKEY is 4 bytes. To study the effect of different data placement, we execute a denormalization query that joins all five tables and compare the total network costs. For this experiment, the number of nodes is set to 64, and the payload size of every table is set to the default 16 bytes.

For such a schema, simply co-partitioning the fact table LINEITEM with one first-layer dimension table is not sufficient, and a better way is to use reference partitioning [16], where a table can be co-partitioned by another table referenced by the foreign key, and chains of tables linked by foreign keys can be co-partitioned. Similar to the case in a star schema, there are two options depending on the foreign key used to partition the fact table LINEITEM: (1) we can co-partition CUSTOMER and ORDERS by CUSTKEY, and reference partition LINEITEM by the already partitioned ORDERKEY in table ORDERS; (2) we can also co-partition PART and PARTSUPP by PARTKEY, and reference partition LINEITEM by table PARTSUPP. As a result of partitioning by foreign key constraints, the joins on the partitioning key can be executed locally. For the first option, LINEITEM $\bowtie$ (ORDERS $\bowtie$ CUSTOMER) is executed locally, and PARTSUPP $\bowtie$ PART is executed locally. The final results can be obtained by joining these two intermediate results. The execution is similar for the second option. Figure 19 shows the better result of these two options. The total network traffic is 3.52GB if hash join is used. For track join, the total cost is 2.55GB, about 27% less than using the hash join.

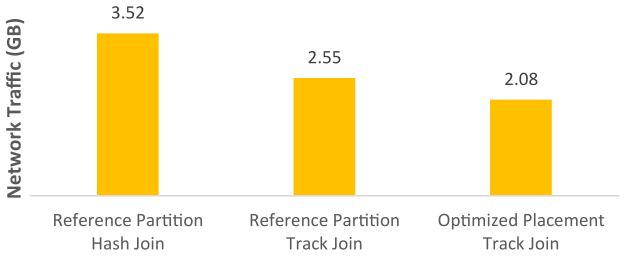


Fig. 19. Network cost of joins denormalizing a snowflake (sub)schema from TPC-H.

To reduce the cost of the final join, we can optimize the data placement to collocate LINEITEM tuples with matching PARTSUPP and ORDERS tuples. The other two dimension tables PART and CUSTOMER are still partitioned by their primary keys. Without changing the query plan, the total network traffic using track join is 2.08GB. Compared with reference partitioning, the reduction of network traffic is 18% after optimizing the placement. Although the joins ORDERS ⋈ CUSTOMER and PARTSUPP ⋈ PART are no longer local, their network costs are only 0.28 and 0.15GB using track joins. These intermediate results further join with LINEITEM. Different from reference partition, neither of these joins is local, but because of the collocation around fact table tuples, the total cost is much lower. This result demonstrates that in a typical analytic database like TPC-H, by optimizing the central star sub-schema, the saving in network cost is so significant that even if the cost of joining peripheral dimension tables increases, we can still improve the overall network costs.

6 RELATED WORK

The foundations of distributed query processing and database implementation [5, 13, 14, 19, 31, 52] have been laid some decades back. Hash join was introduced as part of a database machine [26] and was later parallelized [12]. Distributed algorithms that reduce network traffic by filtering tuples have been extensively studied, namely semi-joins [4, 50, 53], Bloom-joins [35], and other approaches [30].

Contemporary database systems are currently optimized for specific needs [56], the most important cases being analytical and transactional systems. Distributed transactional databases are targeting higher transaction throughput, by eliminating unnecessary network communication [56] due to latency induced delays of distributed commit protocols. Storage managers have also been improved accordingly [22, 23]. Analytical database systems have evolved into column stores [46, 55] and query execution works primarily on RAM [32]. There has also been significant work in the literature toward optimizing main-memory operators for analytical workloads, namely scans [29, 43, 61], joins [2, 3, 24], group-by aggregations [34, 63], sorting [42, 51, 60], and across multiple operators [40, 41].

In distributed sorting [25], CPU and network time can be overlapped by transferring keys and payloads separately. Hash join has also been adapted to reduce the response time [59]. Other work proposed a distributed join algorithm for faster networks where the network is not the bottleneck [20]. Reducing the execution time for distributed joins has also been discussed in recent work [49]. The optimization is designed as an NP-complete scheduling problem and therefore cannot be applied per key. In contrast, track join minimizes network volume using linear-time scheduling applied per key, achieving a finer granularity optimum. Nevertheless, track join can still utilize the same temporal scheduling to reduce end-to-end execution time. Recent work proposed a variant of hash join that handles skew by detecting heavy hitters [48]. Skew awareness has also been studied in the context of array databases [15]. Elseidy et al. studied an online dataflow join operator

with arbitrary predicates that minimizes the size of state maintained and number of messages communicated [17]. Since our original publication of the track-join paper [44], Tian et al. [58] have proposed methods to join data between a distributed database and a hadoop distributed file system. The proposed zigzag join uses Bloom filters both ways to prune unmatched tuples, and then performs a partitioned hash join on the HDFS side. In a pure distributed database, the network costs of such joins are analyzed and compared with track joins in Section 3.3. The tracking phase of track join also prunes unmatches tuples, and further chooses an optimal schedule for joining the tuples, instead of using an uninformative hash-based schedule.

Generic batch processing distributed systems [11] have been used for database operations such as joins [1, 8, 37], minimizing the network cost by placing "map" and "reduce" operators close to relevant data. In such approaches, tuples are grouped using higher granularity criteria, but optimal network schedules of payloads are only achieved by using the lowest granularity of each distinct join key.

Track join (Sections 2, 5.1, and 5.2), also presented in our earlier work [44], is a distributed join algorithm that minimizes the network traffic (Figures 4 and 5 do not appear in Reference [44]). Track join finds the locations of matching tuples in the cluster per distinct join key, and because it is locality-aware, it provides more flexibility for data placement. Relying on hash partitioning to place the data across the nodes eliminates network transfers only for a single join. Analytical workloads typically utilize multiple attributes (dimensions) and the remaining joins cannot be optimized. Here, we additionally propose a data placement optimization (Section 4) that reduces the network traffic of distributed joins for the entire workload, by placing matching tuples across multiple dimensions to the same node, and by relying on track join to exploit the generated locality. We show that the data placement optimization works using representative OLAP benchmarks, including skew (Section 5.3).

Multiple data placement optimizations have been proposed in literature. Data declustering [33] allows databases to achieve high parallelism and load balancing. When the network is the bottleneck, approaches like using a smaller number of machines per query by converting the problem to hypergraph partitioning [27]. However, this approach does not directly minimize network costs. Graph partitioning has been used to minimize the network cost directly [21], but requires each query is executed at a single site, an unrealistic assumption for distributed databases. Data analytics systems collocate related files [18], but require manual tuning from a database administrator.

In transactional databases, automatic partitioning is a popular technique [10, 38, 45, 57] to reduce the number of distributed transactions. Here, we focus on analytical workloads with mostly read-only queries that span across all partitions. For star schema workloads, Reference [54] studies data allocation for star-join query processing with bitmap indices, to reduce I/O overhead. Automated partitioning advisors based on query optimizers were also developed in References [36, 47, 62]. However, we showed that hash partitioning by the best attribute is not sufficient. For multiple tables with parent-child relationships in a complex schema, reference partitioning is developed to partition a table with respect to another table based on a referential constraint [16]. By recursively applying reference partitioning, chains of tables linked by foreign keys can be co-partitioned. Partition-wise joins can then be used to execute foreign key joins between co-partitioned tables, thereby making queries efficient. Improving upon Reference [16], predicate-based reference partitioning has been recently proposed to increase data locality [65] but relies on replicating matching tuples.

7 CONCLUSIONS

We present track join, a new algorithm for distributed joins that minimizes network traffic. To minimize the number of tuples transferred across the network, track join generates an optimal

transfer schedule for each distinct join key after tracking the initial locations of tuples. Track join makes no assumptions about the DBMS storage and does not rely on schema properties or pre-existing favorable data placement.

For three versions of track join, we studied cost estimation for query optimization, compared the interaction with semi-join filtering, and showed track join to be better than tracking-aware hash join. Our experimental evaluation shows the efficiency of track join at reducing network traffic and shows its adaptiveness on various cases and degrees of locality. Our evaluation shows that we can reduce both network traffic and total execution time. The workloads were extracted from a corpus of commercial analytical workloads and the queries were profiled as the most expensive out of all the queries in each workload using a market-leading commercial DBMS.

Building upon track join, we further proposed a data placement algorithm that reduces the total network cost of joins for analytical workloads. By collocating tuples from the fact table alongside joining tuples from multiple dimension tables, and by using the per-key cost analysis of track join to pick the best placement for each fact table key combination, our placement optimization significantly reduces the total network cost of multiple track joins across multiple dimensions, which otherwise have conflicting favorable placements if hash partitioning is used. Our approach also automatically responds to correlation and skew to further reduce the network cost.

REFERENCES

[1] Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In *Proceedings of the EDBT*. 99–110.

[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB* 5, 10 (June 2012), 1064–1075.

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Ozsu. 2013. MultiCore, main-memory joins: Sort vs. hash revisited. *PVLDB* 7, 1 (Sept. 2013), 85–96.

[4] Philip A. Bernstein and D. W. Chiu. 1981. Using semi-joins to solve relational queries. *J. ACM* 28, 1 (Jan. 1981).

[5] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie. 1981. Query processing in a system for distributed databases. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 602–625.

[6] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM* 13, 7 (July 1970).

[7] Surajit Chaudhuri and Vivek Narasayya. 2016. TPC-H Data Generation with Skew. Retrieved from https://www.microsoft.com/en-us/download/details.aspx?id=52430.

[8] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the SIGMOD*. 63–78.

[9] Transaction Processing Performance Council. 2014. The TPC-H Benchmark, 2.17.1. Retrieved from http://www.tpc.org/tpch.

[10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A workload-driven approach to database replication and partitioning. *PVLDB* 3, 1–2 (Sept. 2010), 48–57.

[11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the OSDI*. 137–150.

[12] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The gamma database machine project. *IEEE Trans. Knowl. Data Engin.* 2, 1 (Mar. 1990), 44–62.

[13] David J. DeWitt and Jim Gray. 1992. Parallel database systems: The future of high performance database systems. *Comm. ACM* 35 (1992), 85–98.

[14] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the SIGMOD*. 1–8.

[15] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. 2015. Skew-aware join optimization for array databases. In *Proceedings of the SIGMOD*. 123–135.

[16] George Eadon and others. 2008. Supporting table partitioning by reference in oracle. In *Proceedings of the SIGMOD*. 1111–1122.

[17] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *PVLDB* 7, 6 (Feb. 2014), 441–452.

[18] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. CoHadoop: Flexible data placement and its exploitation in Hadoop. *PVLDB* 4, 9 (June 2011), 575–585.

[19] Robert Epstein, Michael Stonebraker, and Eugene Wong. 1978. Distributed query processing in a relational data base system. In *Proceedings of the SIGMOD*. 169–180.
[20] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. 2009. Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the DaMoN*. 27–33.
[21] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. 2014. Distributed data placement to minimize communication costs via graph partitioning. In *Proceedings of the SSDBM*. Article 20.
[22] Martin Grund and others. 2010. HYRISE: A main memory hybrid storage engine. *PVLDB* 4, 2 (Nov. 2010), 105–116.
[23] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: A scalable storage manager for the multicore era. In *Proceedings of the EDBT*. 24–35.
[24] Changkyu Kim and others. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB* 2, 2 (Aug. 2009), 1378–1389.
[25] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. 2012. CloudRAMsort: Fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *Proceedings of the SIGMOD*. 841–850.
[26] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of hash to data base machine and its architecture. *New Gen. Comput.* 1 (1983), 63–74.
[27] K. Ashwin Kumar, Amol Deshpande, and Samir Khuller. 2013. Data placement and replica selection for improving co-location in distributed environments. *CoRR* abs/1302.4168 (2013). Retrieved from http://arxiv.org/abs/1302.4168.
[28] Viktor Leis et al. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the SIGMOD*. 743–754.
[29] Daniel Lemire et al. 2015. Decoding billions of integers per second through vectorization. *Softw.: Pract. Exper.* 45, 1 (Jan. 2015), 1–29.
[30] Zhe Li and Kenneth A. Ross. 1995. PERF join: An alternative to two-way semijoin and Bloomjoin. In *Proceedings of the CIKM*. 137–144.
[31] Lothar F. Mackert and Guy M. Lohman. 1986. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the VLDB*. 149–159.
[32] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. Optimizing database architecture for the new bottleneck: Memory access. *VLDB J.* 9, 3 (2000), 231–246.
[33] Manish Mehta and David J. DeWitt. 1997. Data placement in shared-nothing parallel database systems. *VLDB J.* 6, 1 (Feb. 1997).
[34] Ingo Müller et al. 2015. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the SIGMOD*. 1123–1136.
[35] J. K. Mullin. 1990. Optimal semijoins for distributed database systems. *IEEE Trans. Softw. Eng.* 16, 5 (May 1990).
[36] Rimma Nehme and Nicolas Bruno. 2011. Automated partitioning design in parallel database systems. In *Proceedings of the SIGMOD*.
[37] Alper Okcan and Mirek Riedewald. 2011. Processing theta-joins using mapreduce. In *Proceedings of the SIGMOD*. 949–960.
[38] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the SIGMOD*. 61–72.
[39] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the SIGMOD*. 165–178.
[40] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo—A vector algebra for portable database performance on modern hardware. *PVLDB* 9, 14 (Oct. 2016), 1707–1718.
[41] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the SIGMOD*. 1493–1508.
[42] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the SIGMOD*. 755–766.
[43] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient lightweight compression alongside fast scans. In *Proceedings of the DaMoN*.
[44] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. 2014. Track join: Distributed joins with minimal network traffic. In *Proceedings of the SIGMOD*. ACM, 1483–1494.
[45] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the EDBT*. 430–441.
[46] Vijayshankar Raman et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *PVLDB* 6, 11 (Aug. 2013), 1080–1091.
[47] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *Proceedings of the SIGMOD*. 558–569.

[48] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *Proceedings of the ICDE*. 1194–1205.
[49] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2014. Locality-sensitive operators for parallel main-memory database clusters. In *Proceedings of the ICDE*. 17–30.
[50] N. Roussopoulos and H. Kang. 1991. A pipeline N-way join algorithm based on the 2-way semijoin program. *IEEE Trans. Knowl. Data Engin.* 3, 4 (Dec. 1991), 486–495.
[51] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the SIGMOD*. 351–362.
[52] Donovan A. Schneider and David J. DeWitt. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the SIGMOD*. 110–121.
[53] James W. Stamos and Honesty C. Young. 1993. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distrib. Syst.* 4, 12 (Dec. 1993), 1345–1354.
[54] Thomas Stöhr et al. 2000. Multi-dimensional database allocation for parallel data warehouses. In *Proceedings of the VLDB*. 273–284.
[55] Mike Stonebraker et al. 2005. C-store: A column-oriented DBMS. In *Proceedings of the VLDB*. 553–564.
[56] Michael Stonebraker et al. 2007. The end of an architectural era: (It's time for a complete rewrite). In *Proceedings of the VLDB*. 1150–1160.
[57] Rebecca Taft et al. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB* 8, 3 (Nov. 2014), 245–256.
[58] Yuanyuan Tian, Fatma Özcan, Tao Zou, Romulo Goncalves, and Hamid Pirahesh. 2016. Building a hybrid warehouse: Efficient joins between data stored in HDFS and enterprise warehouse. *ACM TODS* 41, 4 (2016), 21.
[59] Tolga Urhan and Michael J. Franklin. 2000. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engin. Bulletin* 23 (June 2000), 27–33.
[60] Jan Wassenberg and Peter Sanders. 2011. Engineering a multi core radix sort. In *Proceedings of the EuroPar*. 160–169.
[61] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB* 2, 1 (Aug. 2009), 385–394.
[62] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the SIGMOD*. 13–24.
[63] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *Proceedings of the DaMoN*.
[64] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: A new database synopsis for query estimation. In *Proceedings of the SIGMOD*. 469–480.
[65] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-aware partitioning in parallel database systems. In *Proceedings of the SIGMOD*. 17–30.

Received March 2017; revised June 2018; accepted July 2018