Track Join: Distributed Joins with Minimal Network Traffic

Orestis Polychroniou*
Columbia University
orestis@cs.columbia.edu

Rajkumar Sen
Oracle Labs
rajkumar.sen@oracle.com

Kenneth A. Ross†
Columbia University
kar@cs.columbia.edu

ABSTRACT

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 novel 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. Our evaluation based on real and synthetic data shows that track join adapts to diverse cases and degrees of locality. Considering both network traffic and execution time, even with no locality, track join outperforms hash join on the most expensive queries of real workloads.

1. INTRODUCTION

The processing power and storage capacity of a single machine can be large enough to fit small to medium scale databases. Nowadays, servers with memory capacity of more than a terabyte are common. 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. sort, join, and group-by) on RAM-resident data at rates of a few gigabytes per second [2, 3, 29, 34, 36].

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 [32], while accelerators [26] are the implicit manifestation of the same concept.

*Work partly done when author was at the Oracle Labs.
†Work supported by National Science Foundation grant IIS-0915956 and a gift from Oracle Corporation.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610521.

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, 11, 20], hardware advances of today like multi-core parallelism had not yet occurred. Techniques to speed up short-lived distributed transactions [32] target distributed commit protocols, which suffer from network latencies rather than throughput. Queries where communication is inevitable are less popular research topics or are left for data-centric generic distributed systems for batch-processing [7, 24].

The latest network technologies may be slow relative to main-memory-resident processing. A 40 Gbps InfiniBand measured less than 3 GB/s real data rate per node during hash partitioning. If done in RAM, partitioning to a few thousand outputs runs close to the memory copy bandwidth [29, 34]. For instance, a server using 4X 8-core CPUs and 1333 MHz quad-channel DDR3 DRAM achieves a partition rate of 30–35 GB/s, more than an order of magnitude higher than the InfiniBand network. Recent work [3] achieves a hash join rate of 4.85 GB/s of 32-bit key, 32-bit payload tuples on 4X 8-core CPUs. 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 relatively slow compared to local in-memory processing, we expect the execution time to be dominated by network transfers. Thus, any network traffic reduction directly translates to faster execution. In high-end platforms, given that the network still cannot be as fast as the RAM bandwidth, completion times are also reduced if the reduction in network traffic is comparable with the increase in CPU cycles.

In order to show how much time databases can spend on the network, we give an example of a real analytical workload from a large commercial vendor, using a market-leading commercial DBMS. Using 8 machines connected through 40 Gbps InfiniBand (see Section 4 for more details of the configuration), we found that the five most expensive queries spend $\approx$ 65–70% of their time transferring tuples on the network and account for 14.7% of the total time required to execute the entire analytical workload with more than 1500 queries. All five queries have a non-trivial query plan (4–6 joins), but spend 23%, 31%, 30%, 42%, and 43% of their total execution time on a single distributed hash join.

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 [7]. 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 [9, 17] (throughout the paper we will use the term *hash join* to refer to Grace hash join on network [9], rather than disk [17]) is the predominant method for executing distributed joins and uses hash partitioning to split the initial problem into shared-nothing sub-problems that can proceed locally per node. Partitioning both tables works almost independently of the table sizes. However, hash join is far from network-optimal because it transfers almost the full size of both tables over the network. Using pre-determined hash functions guarantees load balancing, but limits the probability that a hashed tuple will not be transferred over the network to $1/N$ on N nodes.

We 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. The decision uses 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 are used in optimally compressed form throughout the join.

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

Algorithm 2-phase track join: process$_R$ − R to S

```
T_R ← {}
for all <key_{R|S} k, payload_R p_R> in table R do
  if k not in T_R then
    send k to process_T (hash(k) mod N)
  end if
  T_R ← T_R + <k, p_R>
end for
barrier
while any process_T n_T sends do
  for all <key_{R|S} k, process_S n_S> from n_T do
    for all <k, payload_R p_R> in T_R do
      send <k, p_R> to n_S
    end for
  end for
end while
```

2. ALGORITHMIC DESCRIPTION

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 others and all links have the same performance. We present three versions of track join gradually evolving in complexity.

The first version 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.

The second version gathers not only an indicator whether a node has tuples of a specific join key or not, but also the number and total size of these tuples. We pick the cheapest side to be selectively broadcast independently for each key.

The third and full version of track join extends the second version by generating an optimal join schedule for each key achieving minimum payload transfers, without hashing or broadcasting. The schedule migrates tuples from one table to potentially fewer nodes than the original placement, before selectively broadcasting tuples from the other table.

The algorithm will be described in this section using a pipelined approach. We assume three processes (R, S operating on tables R and S, and T that generates schedules) running simultaneously on all nodes, but once per node. A non-pipelined approach is also possible (see Section 4).

The algorithmic display convention used throughout this paper 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. $|x|$ denotes set size.

2.1 2-Phase Track Join

The 2-phase (or *single broadcast*) is the simplest version of track join. The first phase is the tracking phase, while the second phase is the selective broadcast phase.

In the first 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. Every node receives unique keys and stores them alongside the id of the source node. In our display of the algorithm, this operation is done by a different process named T, which also generates the transfer schedules. The location of process$_T$ is determined by hashing the join key, thus, distributing the task of scheduling equally across all nodes.

Algorithm 2-phase track join: process$_S$ − R to S

```
T_S ← {}
for all <key_{R|S} k, payload_S p_S> in table S do
  if k not in T_S then
    send k to process_T (hash(k) mod N)
  end if
  T_S ← T_S + <k, p_S>
end for
barrier
while any process_R n_R sends do
  for all <key_{R|S} k, payload_R p_R> from n_R do
    for all <k, payload_S p_S> in T_S do
      commit <k, p_R, p_S>
    end for
  end for
end while
```

Algorithm 2-phase track join: process$_T$ − R to S

$T_{R|S} \leftarrow \{\}$
while any process$_R$ **or any** process$_S$ $n_{R|S}$ **sends do**
  for all <key$_{R|S}$ k> **from** $n_{R|S}$ **do**
    $T_{R|S} \leftarrow T_{R|S}$ + <k, $n_{R|S}$>
  end for
end while
barrier
for all distinct key$_{R|S}$ k **in** $T_{R|S}$ **do**
  for all <k, process$_R$ n_R> **in** $T_{R|S}$ **do**
    for all <k, process$_S$ n_S> **in** $T_{R|S}$ **do**
      send <k, n_S> **to** n_R
    end for
  end for
end for

In the second phase, we only transfer tuples from one table. Assuming we transfer R tuples, node T (process$_T$) sends messages to each location with matching R tuples, including the key and the set of S tuples' locations. Finally, R tuples are selectively broadcast to the tracked S locations, instead of all nodes in the network, and are joined locally.

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.

2-phase track join transfers payloads from one table only. If the input tables have mostly unique keys and the selectivity is high, the cost comprises of tracking plus min($|R|$, $|S|$).

Algorithm 3-phase track join: process$_T$

barrier
$T_{R|S} \leftarrow \{\}$
while any process$_R$ **or any** process$_S$ $n_{R|S}$ **sends do**
  for all <key$_{R|S}$ k, count c> **from** $n_{R|S}$ **do**
    $T_{R|S} \leftarrow T_{R|S}$ + <k, $n_{R|S}$, c>
  end for
end while
barrier
for all distinct key$_{R|S}$ k **in** $T_{R|S}$ **do**
  R, $S \leftarrow \{\}$, $\{\}$
  for all k, process$_R$ n_R, count c> **in** $T_{R|S}$ **do**
    $R \leftarrow R$ + <n_R, $c \cdot width_R$>
  end for
  for all k, process$_S$ n_S, count c> **in** $T_{R|S}$ **do**
    $S \leftarrow S$ + <n_S, $c \cdot width_S$>
  end for
  $RS_{cost} \leftarrow$ broadcast R to S
  $SR_{cost} \leftarrow$ broadcast S to R
  if $RS_{cost} < SR_{cost}$ **then**
    for all <k, process$_R$ n_R> **in** $T_{R|S}$ **do**
      for all <k, process$_S$ n_S> **in** $T_{R|S}$ **do**
        send <k, n_S> **to** n_R
      end for
    end for
  else
    for all <k, process$_S$ n_S> **in** $T_{R|S}$ **do**
      for all <k, process$_R$ n_R> **in** $T_{R|S}$ **do**
        send <k, n_R> **to** n_S
      end for
    end for
  end if
end for

Algorithm 3-phase track join: process$_S$

... symmetric with process$_R$ of 3-phase track join ...

Algorithm 3-phase track join: process$_R$

$T_R \leftarrow \{\}$
for all <key$_{R|S}$ k, payload$_R$ p_R> **in** table R **do**
  $T_R \leftarrow T_R$ + <k, p_R>
end for
barrier
for all distinct key$_{R|S}$ k **in** T_R **do**
  $c \leftarrow |k$ **in** $T_R|$
  send <k, c> **to** process$_T$ (hash(k) mod N)
end for
barrier
while any process$_S$ **or any** process$_T$ **sends do**
  if source is process$_T$ n_T **then**
    for all <key$_{R|S}$ k, process$_S$ n_S> **from** n_T **do**
      for all <k, payload$_R$ p_R> **in** T_R **do**
        send <k, p_R> **to** n_S
      end for
    end for
  else if source is process$_S$ n_S **then**
    for all <key$_{R|S}$ k, payload$_S$ p_S> **from** n_S **do**
      for all <k, payload$_R$ p_R> **in** T_R **do**
        commit <k, p_R, p_S>
      end for
    end for
  end if
end while

2.2 3-Phase Track Join

In the 3-phase (or *double broadcast*) track join, we can decide whether to broadcast R tuples to the locations of S tuples, or vice versa. The decision is taken per distinct join key. In order to decide which selective broadcast direction is cheaper, we need to know how many tuples will be transferred in each case. Thus, instead of only tracking nodes with at least one matching tuple, we also track the number of matches. To generalize for variable lengths, we transfer the sum of matching tuple widths, rather than a count.

Bi-directional selective broadcast can distinguish cases in which moving S tuples would transfer fewer bytes than moving R tuples. The decision whether to selectively broadcast $R \rightarrow S$ or vice versa is taken for each distinct key independently and we do not rely on the optimizer to pick the least expensive direction overall for the entire join.

The cost estimation for one selective broadcast direction is shown below. In practice, we compute both directions and pick the cheapest. The complexity is $O(n)$, where n is the number of nodes with at least one matching tuple for the given join key. The total number of steps required is less than the number of tuples. Thus, the theoretical complexity is linear. The messages that carry location information, logically seen as key and node pairs, have size equal to M.

Algorithm track join: broadcast R to S

$R_{all} \leftarrow \sum_i |R_i|$
$R_{local} \leftarrow \sum_i |R_i|$, **where** $|S_i| > 0$
$R_{nodes} \leftarrow |i$, **where** $|R_i| > 0 \wedge i \neq self|$
$S_{nodes} \leftarrow |i$, **where** $|S_i| > 0|$
$RS_{cost} \leftarrow R_{all} \cdot S_{nodes} - R_{local} + R_{nodes} \cdot S_{nodes} \cdot M$
return RS_{cost}

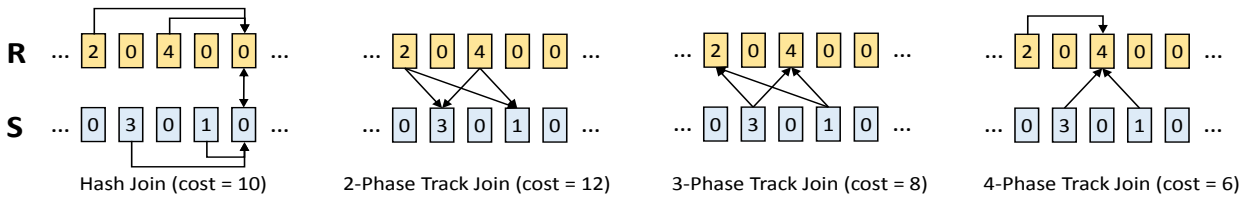

Figure 1: Example of network transfers in hash join and track join per distinct join key

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 unique 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 we cannot optimize, 4-phase track join transfers the minimum amount of payload data possible for an early-materialized distributed join.

4-phase track join extends 3-phase track join by adding a more complicated scheduling algorithm and a migration phase. It 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. We will prove that optimal network traffic scheduling still takes linear time.

In Figure 1 we show an example of how the different algorithms behave for a single key join. In hash join, we assume the hash destination is the fifth node. In 2-phase track join we selectively broadcast R tuples to matching S tuple locations and in 3-phase track join we choose the opposite direction, since it is cheaper. The 4-phase track join first migrates R tuples, before selectively broadcasting S tuples.

The schedules by 4-phase track join sometimes exhibit behavior similar to that of hash join, where all tuples are sent into a single node to perform the join. Even in such a case, the node will be the one with the most pre-existing matching tuples, as seen in Figure 1, maximizing locality.

Algorithm 4-phase track join: process$_S$

... symmetric with process$_R$ of 4-phase track join ...

Algorithm 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
        $T_R \leftarrow T_R - <k, p_R>$
      end for
    end for
  else if source is process$_R$ n_R **then**
    for all <key$_{R|S}$ k, payload$_R$ p_R> **from** n_R **do**
      $T_R \leftarrow T_R + <k, p_R>$
    end for
  end if
end while
barrier
... identical to the 3rd phase of 3-phase track join ...

We formalize the problem of network traffic minimization for a single key (cartesian product) join, assuming for simplicity that the plan is known to every node and no location messages are sent. Assume that x_{ij} is the binary decision of sending R tuples from node i to node j, while y_{ij} is the binary decision of sending S tuples from j to i. $|R_i|$ represents the total size of R tuples residing in node i, while $|S_j|$ 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. One way is by sending R tuples from i to j, setting $x_{ij} = 1$. The second 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. In short, we need some node k, where $x_{ik} = 1$ and $y_{kj} = 1$. Local sends do not affect network traffic, thus, all x_{ii} and y_{jj} are ignored.

$$\text{min: } \sum_i \sum_{j \neq i} x_{ij} \cdot |R_i| + y_{ij} \cdot |S_j| \quad \text{s.t. } \forall i, j \sum_k x_{ik} \cdot y_{kj} \geq 1$$

Algorithm 4-phase track join: process$_T$

barrier
... identical to the 2nd phase of 3-phase track join ...
barrier
for all distinct key$_{R|S}$ k **in** $T_{R|S}$ **do**
  $R, S \leftarrow \{\}, \{\}$
  for all <k, process$_R$ n_R, count c> **in** $T_{R|S}$ **do**
    $R \leftarrow R + <n_R, c \cdot width_R>$
  end for
  for all <k, process$_S$ n_S, count c> **in** $T_{R|S}$ **do**
    $S \leftarrow S + <n_S, c \cdot width_S>$
  end for
  $RS_{cost}, S_{migr} \leftarrow$ migrate S & broadcast R
  $SR_{cost}, R_{migr} \leftarrow$ migrate R & broadcast S
  if $RS_{cost} < SR_{cost}$ **then**
    for all <k, process$_S$ n_S> **in** S_{migr} **do**
      $d_S \leftarrow$ **any** process$_S$ **not in** S_{migr}
      send <k, d_S> **to** n_S
      $T_{R|S} \leftarrow T_{R|S} - <k, n_S,$ count $c_{migr}>$
      $T_{R|S} \leftarrow T_{R|S} - <k, d_S,$ count $c_{dest}>$
      $T_{R|S} \leftarrow T_{R|S} + <k, d_S, c_{migr} + c_{dest}>$
    end for
  else
    for all <k, process$_R$ n_R> **in** R_{migr} **do**
      $d_R \leftarrow$ **any** process$_R$ **not in** R_{migr}
      send <k, d_R> **to** n_R
      $T_{R|S} \leftarrow T_{R|S} - <k, n_R,$ count $c_{migr}>$
      $T_{R|S} \leftarrow T_{R|S} - <k, d_R,$ count $c_{dest}>$
      $T_{R|S} \leftarrow T_{R|S} + <k, d_R, c_{migr} + c_{dest}>$
    end for
  end if
end for
barrier
... identical to the 3rd phase of 3-phase track join ...

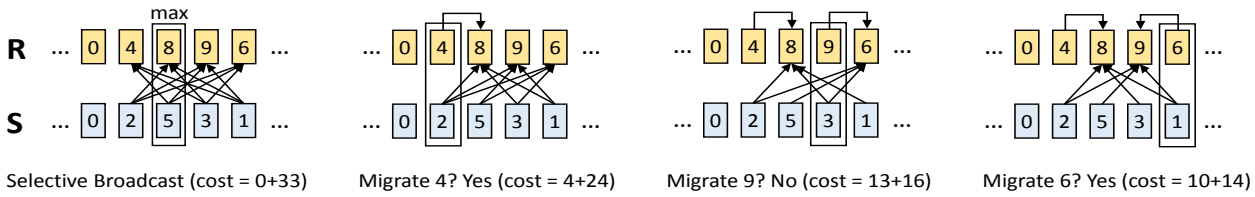

Figure 2: Example of 4-phase track join optimal schedule generation per distinct join key

We solve the traffic minimization problem by optimizing selective broadcasts using migrations. Before the $R \rightarrow S$ selective broadcast, we migrate S tuples across nodes, depending on whether the cost of the later broadcast of R tuples is reduced. The mechanism allows only S tuples to be transferred, thus, we can easily decide whether the S tuples of some node should stay in place, or be transferred somewhere with more S tuples. We reduce the cost of the later selective broadcast also considering the migration cost. The algorithm shown below includes location messages of size M.

Algorithm track join: migrate S & broadcast R

$R_{all} \leftarrow \sum_i |R_i|$
$R_{local} \leftarrow \sum_i |R_i|$, **where** $|S_i| > 0$
$R_{nodes} \leftarrow |i$, **where** $|R_i| > 0 \wedge i \neq self|$
$S_{nodes} \leftarrow |i$, **where** $|S_i| > 0|$
$RS_{cost} \leftarrow R_{all} \cdot S_{nodes} - R_{local} + R_{nodes} \cdot S_{nodes} \cdot M$
$max_S \leftarrow i$, **where** $|S_i| > 0 \wedge \forall j : |R_i| + |S_i| \geq |R_j| + |S_j|$
$S_{migr} \leftarrow \{\}$
for $i \leftarrow 1$ **to** $|\{S_1, S_2, ...\}|$ **do**
  if $|S_i| > 0$ **and** $i \neq max_S$ **then**
    $\Delta \leftarrow |R_i| + |S_i| - R_{all} - R_{nodes} \cdot M$
    if $i \neq self$ **then**
      $\Delta \leftarrow \Delta + M$
    end if
    if $\Delta < 0$ **then**
      $RS_{cost} \leftarrow RS_{cost} + \Delta$
      $S_{migr} \leftarrow S_{migr} + \{i\}$
    end if
  end if
end for
return RS_{cost}, S_{migr}

THEOREM 1. *We can generate optimal selective broadcast schedules ($R \rightarrow S$) by allowing the migration of (S) tuples.*

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

$$(\sum |R_i|) \cdot (\sum z_i) - (\sum |R_i| \cdot z_i) + \sum |S_i| \cdot (1 - z_i)$$

The $\sum |R_i|$ term is the total size of R tuples, the $\sum z_i$ is the number of locations with S tuples, the $\sum |R_i| \cdot z_i$ are the R tuples that were local during broadcast and $\sum |S_i| \cdot (1 - z_i)$ is the cost of migrating S tuples. Since $\sum |R_i| (\equiv R)$ and $\sum |S_i| (\equiv S)$ are independent of all z_i, the formula can be minimized by checking each z_i independently. Finally, since S_{migr} cannot contain all nodes, we force the node with the largest $|R_i| + |S_i|$ and $|S_i| > 0$, out of S_{migr} ($z_i = 1$). □

THEOREM 2. *The minimum optimized selective broadcast direction ($R \rightarrow S$, or $S \rightarrow R$) is also the minimum network traffic schedule for single key (cartesian product) joins.*

PROOF. Given a computed S_{migr}, migrating any R_x to any R_y can only increase the cost. If $y \in S - S_{migr}$, the cost remains the same, as we moved R_x once, but now it is local to S_y and skip y during the selective broadcast. If $y \in S_{migr}$, we increase the cost of the migrating phase by $|R_x|$, but the cost of the selective broadcast remains the same. □

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 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, and count (or size). If a node does not contain a key, the triplet is not considered at all. Thus, scheduling is in the worst case linear in the total number of input tuples.

Using multiple threads per process is allowed and is, of course, essential to achieve good performance on CPU intensive tasks. 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.

2.4 Traffic Compression

Track join, as described so far, makes absolutely no effort to reduce the data footprint by compression. Modern analytical database systems employ distinct column value dictionaries [26, 35]. The compression process can occur offline, and data are 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.

The simplest case is delta encoding [18, 31]. With sufficient CPU power, we can sort all columns sent over the network. Track join imposes no specific message order for all types of messages, besides the barrier synchronization across phases. Thus, the potential compression rate increases.

A second technique is to perform partitioning at the source to create common prefixes. For instance, we can radix partition the first p bits and pack $(w-p)$-bit suffix with a common prefix. We can even tune the compression rate, by employing more partition passes to create wider prefixes. Each pass has been shown to be very efficient on memory-resident data, close to the RAM copy bandwidth [29, 34]. If the inputs retain dictionary encoding through the join, the number of distinct values using the same prefix is maximized.

A similar optimization we can use for track join is to avoid sending node locations alongside keys by partitioning in corresponding groups. We avoid sending the node part in messages containing key and node pairs by sending many keys with a single node label after partitioning by node.

3. QUERY OPTIMIZATION

The formal model of track join is used by the query optimizer to decide whether to use track join in favor of 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. Finally, we discuss how track join interacts with semi-join filtering on selective joins.

3.1 Network Cost Model

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

The network cost of the ubiquitous hash join, omitting the $1/N$ in place transfer probability on N nodes, is:

$$t_R \cdot (w_k + w_R) + t_S \cdot (w_k + w_S)$$

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 later used in 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 refer to these quantities as $n_R \equiv \min(N, t_R/d_R)$ and $n_S \equiv \min(N, t_S/d_S)$.

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. We refer to these quantities as $m_R \equiv \min(N, (t_R \cdot s_R)/d_R)$ and $m_S \equiv \min(N, (t_S \cdot s_S)/d_S)$.

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

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

For 3-phase track join, we transfer counters during tracking. 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 $c_S \equiv \log(t_S/(d_S \cdot n_S))$. If some keys' repetitions exceed the maximum count, we can aggregate at the destination. The cost of 3-phase track join is:

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

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. We briefly explain how these classes are estimated, shortly.

The query optimizer should pick 2-phase track join rather than 4-phase when both tables have almost entirely unique keys, which is enabled by the distinct values estimation. Simple broadcast join can be better if one table is very small.

If actual numbers are required, instead of relative comparison of available algorithms, we can estimate the network cost of track join using correlation classes. 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 $R \rightarrow$ (for R_1), or the $S \rightarrow R$ (for R_2) selective broadcast direction. To populate the classes, we can use *correlated sampling* [37], a recently proposed technique that preserves the join relationships of tuples, is independent 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 keys to correlation classes based on traffic levels, and combine their cardinality and average traffic. A simplified version of 4-phase cost is shown below, using the classes that combine 3-phase track join (R_1, S_1, R_2, S_2) with hash join (R_3, S_3):

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

These three correlation classes merge hash join with 3-phase track join and are the most common in real queries. More classes can be defined by combining them in a step function.

A limitation of track join is that it invests significant time working with keys compared to hash join. When the payloads are small, we expect track join to perform worse than hash join, unless there is locality. For instance, if the joined tables have equal cardinality ($t_R = t_S$) and entirely unique keys, in order to not transfer more than hash join, we need:

$$\begin{aligned} 4 \cdot w_k + \min(w_R, w_S) &\leq 2 \cdot w_k + w_R + w_S \\ \Leftrightarrow \qquad 2 \cdot w_k &\leq \max(w_R, w_S) \end{aligned}$$

i.e., the larger payload width must be two times bigger than the join key width, assuming there is no pre-existing locality.

3.2 Tracking-Aware Hash Join

Late materialization is a popular approach for designing main memory databases. The technique is typically used in column stores [26, 31], but can also be viewed as an optimizer decision, whether deferring column access and carrying record identifiers (rids) costs less than early materialization. An analysis of tuple reduction by interleaving late materializated operations is out of the scope of this paper.

In the simple case, keys are hashed, rids are implicitly generated, and payloads are fetched afterwards. The cost is:

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

where t_{RS} is the number of output joined tuples.

When distributed, rids contain a local id and a node id. To use the implicit tracking information carried in a 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 of the resulting method is:

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

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 subsumes the rid-based tracking-aware hash join.

The extra cost of transferring rids is non trivial. 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. The rids of both tables together may be wider than the smaller payloads, making late materialization less effective for distributed compared to local in-memory processing.

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, 22] implemented using Bloom filters [6], which are optimized towards network traffic. In our analysis, we assume a two-way semi-join.

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 filtered early materialized hash join is:

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

where e is the relative error of the Bloom filters.

The cost of filtered late materialized hash join, is:

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

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

The cost of 2-phase track join, using Bloom filtering, is:

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

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 can exceed the cost of sending a few columns for reasonable cluster size N. Track join does perfect semi-join filtering during tracking. Thus, it is more likely to subsume the Bloom filtering step compared to early materialized hash join, which sends all columns, and late materialized hash join, which sends keys and rids.

4. EXPERIMENTAL EVALUATION

Our evaluation is split into two parts. First, we simulate distributed joins to measure network traffic across encoding schemes (Section 4.1). Second, we implement and measure track join against hash join execution times (Section 4.2).

4.1 Simulations

In the simulations we assume 16 nodes and show 7 cases: broadcast $R \to S$ (BJ-R) and $S \to R$ (BJ-S), hash join (HJ), 2-phase track join $R \to S$ (2TJ-R) and $S \to R$ (2TJ-S), 3-phase track join (3TJ), and 4-phase track join (4TJ).

In the three experiments shown in Figure 3, the tuple width ratio of R and S are 1/3, 2/3, and 3/3. The key width is included in the tuple width and is 4 bytes. The tables have equal cardinality (10^9) and almost entirely 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 the 2-phase version suffices.

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 was the result of hashing the 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, the difference is maximized since $k \approx 2$ is the lower bound for equi-joins.

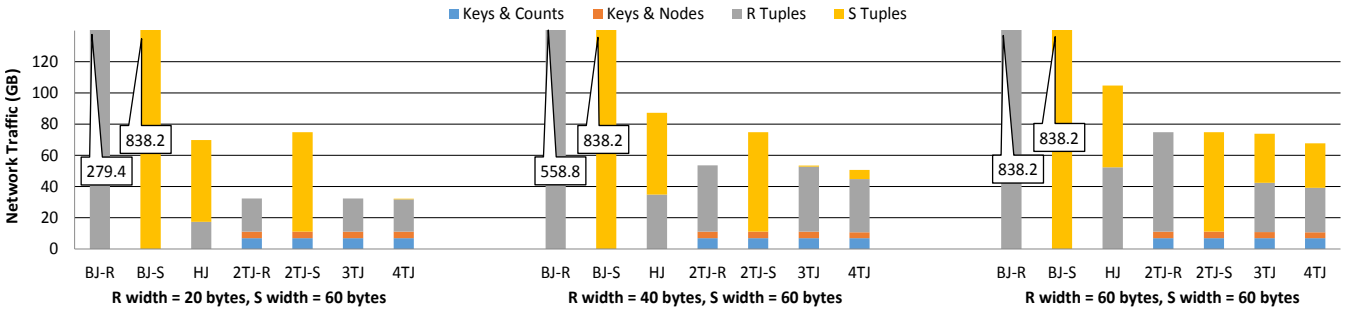

Figure 3: Synthetic dataset of 10^9 vs. 10^9 tuples with $\approx 10^9$ unique join keys

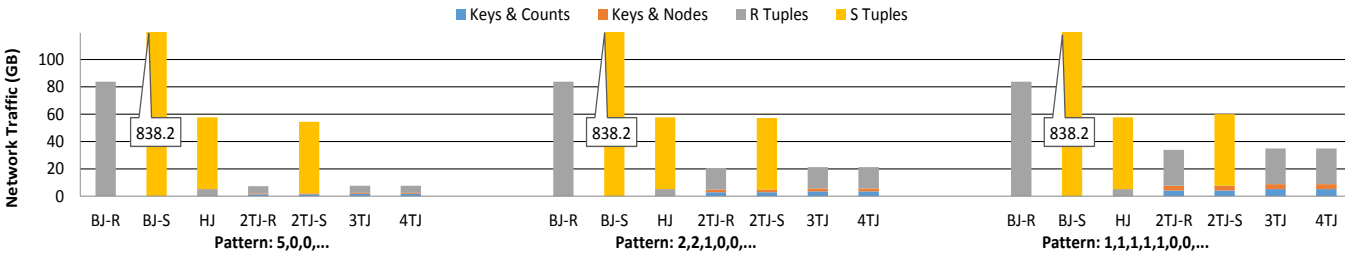

Figure 4: Synthetic dataset of $2 \cdot 10^8$ unique vs. 10^9 tuples (single side intra-table collocated)

In Figure 4, we show how locality can affect track join performance. We assume S to have 1 billion 60-byte tuples and R to have 200 million 30 byte tuples with unique keys. S repeats each key 5 times but key locations follow patterns specified below each chart. The pattern 5,0,0,... means all 5 key repeats are on a single node. The pattern 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,... means all repeats are on different nodes. This placement is 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 pattern, still better than uniform placement, the traffic is still reduced compared to hash join. Note that random placement collocates repeats to some degree.

We do the same experiment using small tables for both inputs. 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 unique key. These numbers were picked so that the outputs on all figures of the current page are of the same size. In Figure 4 we produce 1 billion tuples as output. In Figures 5 and 6 we use 200 million tuples with 40 million distinct join keys for both tables, thus each key has 25 output tuples. The tuple widths are still 30 bytes for R and 60 bytes for S.

The difference between Figure 5 and Figure 6 is whether the tuples across tables are collocated. In the first, the 5,0,0,... configuration means all repeats from R are on a single node and all repeats from S are on a (different) single node. We term this case, shown on Figure 5, *intra-table* collocation. When same key tuples from across tables are collocated, locality is further increased. We term this case, shown on Figure 6, *inter-table* collocation. When all 10 repeats are collocated, track join eliminates all transfers of payloads. Messages used during the tracking phase can only be affected by the same case of locality as hash join.

Figures 4, 5, and 5 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, 4-phase track join is similar to hash join. However, track join sends far less data when matching tuples are collocated, inside or across tables.

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

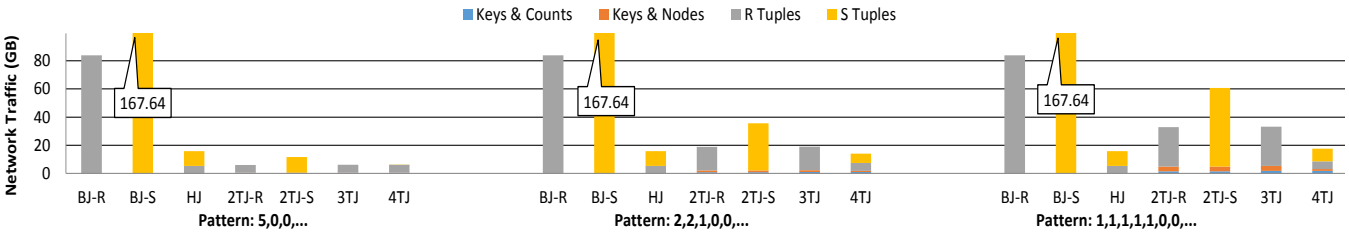

Figure 5: Synthetic dataset with $2 \cdot 10^8$ tuples per table with $4 \cdot 10^7$ unique keys (intra collocated)

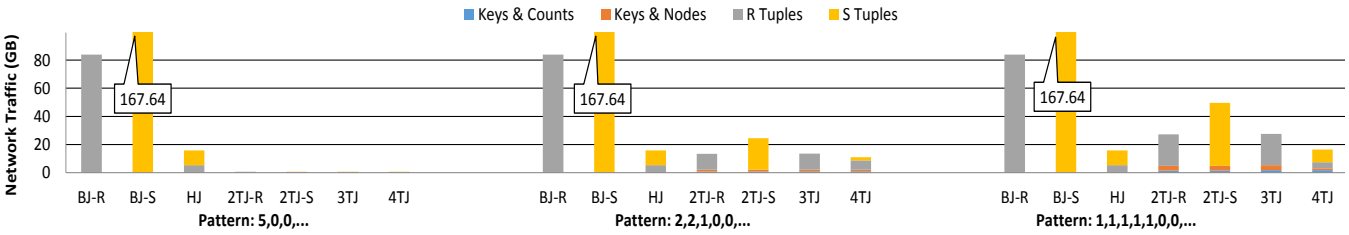

Figure 6: Synthetic dataset with $2 \cdot 10^8$ tuples per table with $4 \cdot 10^7$ unique keys (inter & intra collocated)

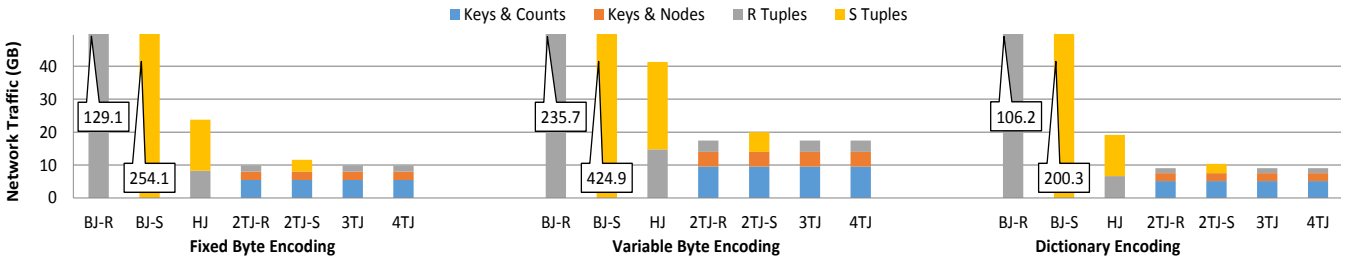

Figure 7: Slowest join sub-query of slowest query (Q1) of real workload X (original tuple ordering)

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. Q1 includes a hash join that we profiled to take 23% of the total execution time. The two join inputs, shown in table 1, are both intermediate relations and consist of 769,845,120 and 790,963,741 tuples. The join output has 730,073,001 tuples. Q1 joins 7 relations, after applying selections on 4, and performs one final aggregation.

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

Table 1: R (≈ 770M tuples) ⋈ S (≈ 791M tuples)

In Figure 7, we use 3 different encoding schemes, fixed byte (1, 2 or 4 byte) codes, fixed bit dictionary codes, and variable byte[1] encoding. The input displayed locality, since broadcasting R transferred ≈ 1/3 of R tuples compared to hash join. We would expect both to transfer the whole R table. We shuffle and repeat the experiments in Figure 8.

Workload X contains more than 1500 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 aggregation. We study the same expensive join as in Q1.

All columns of the join are of number type and would use variable byte encoding if left uncompressed, which is expensive for network use. As shown in Figures 7 and 8, where we presented results for the most expensive query (Q1), using uncompressed keys increases the cost of track join. Since we cannot optimize the tracking phase, it is preferable to compress the key columns rather than the payload columns.

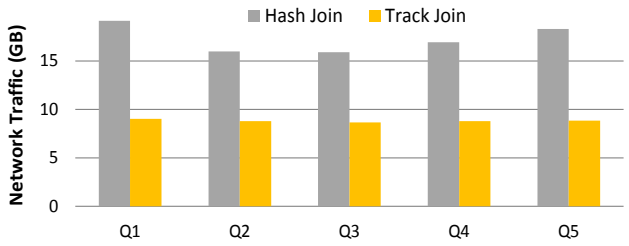

Figure 9: Common slowest join sub-query in slowest queries 1–5 of X (optimal dictionary compression)

In Figure 9, 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) as the one we selective broadcast during 2-phase track join, all track join versions have similar performance, sending each tuple from table with shorter tuples to the single location of the match from the table with wider tuples.

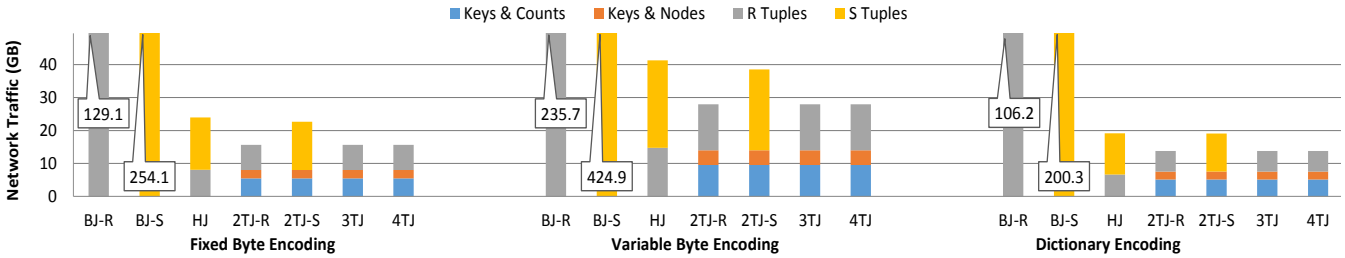

Figure 8: Slowest join sub-query of slowest query (Q1) of real workload X (shuffled tuple ordering)

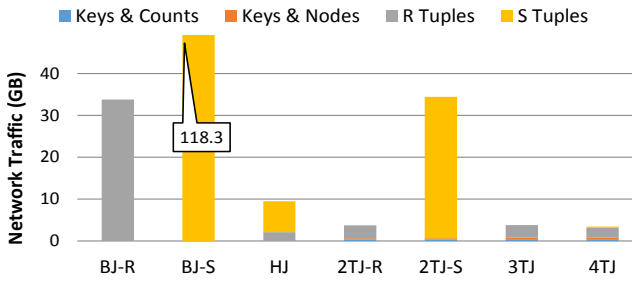

Figure 10: Slowest join in slowest query of uncompressed variable byte[1] Y (original tuple ordering)

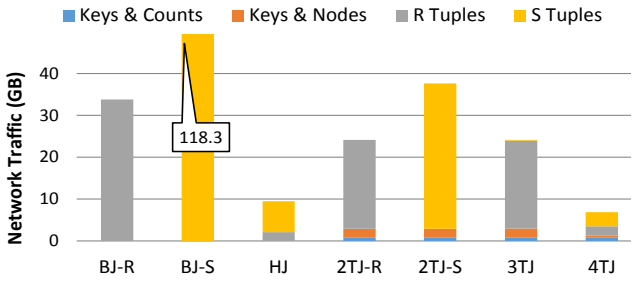

Figure 11: Slowest join in slowest query of uncompressed variable byte[1] Y (shuffled tuple ordering)

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 cases, we can effectively reduce the data width using simple compression schemes [18] that allow fast decompression and do not require accessing large dictionaries that are unlikely to remain CPU-cache-resident. On the other hand, 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 [18, 35]. However, in many operations, including non-selective equi-joins, dictionary accesses are redundant. In our results for compressed workload X, we omit dictionary dereference traffic as the join can proceed solely on compressed data.

In all experiments we omit dictionary dereference traffic as the join can proceed solely on compressed data. Dictionaries are also compressed before the join using the minimum number of bits required to encode the distinct values of the intermediate relation. This is the optimal encoding scheme, when most keys are unique and have no specific value ordering. If the dictionaries are not compressed at runtime, we need to rely on the original dictionary that encodes all distinct values of original tables. Encoding to the optimal compression is not always the best choice as the total gain may be outweighed by the total time needed to re-compress. The decision to compact the dictionary and re-compress an intermediate relation, is taken by the query optimizer.

In our last simulation we use a second real workload (Y). We isolate the most expensive hash join out of 9 joins from the slowest query out of the $\approx$ 90 analytical queries in Y. According to our profiling, the top query takes 20% of the time required to run all queries cumulatively and the hash join takes about 36% of the total query execution time.

In Figure 10 we show the running time using the original data ordering. 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.

The last 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 which also applies per distinct join key. The naïve selective broadcast of 2-phase and 3-phase track join almost broadcasts R.

In Figure 11, we re-run the experiment on the same workload as Figure 10, but first we shuffle the input to remove all locality. 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 11. 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.

4.2 Implementation

For the next set of experiments, we implement hash join and track join in C++ using system calls but no libraries. The network communication uses TCP and all local processing is done in RAM. We use four identical machines connected through 1 Gbit Ethernet, each with 2X Intel Xeon X5550 (4-core with 2-way SMT) CPUs at 2.67 GHz and 24 GB ECC RAM. The RAM bandwidth is 28.4 GB/s for reading data from RAM, 16 GB/s for writing data to RAM, and 12.4 GB/s for RAM to RAM copying. We compile with GCC 4.8 using `-O3` optimization, the OS is Linux 3.2, and we always use all 16 hardware threads on each machine.

In Table 2 we show the CPU and network use for hash join and all track join versions. Our platform is severely network bound since each edge can transfer 0.093 GB/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.[2]

Input		Time	HJ	2TJ[3]	3TJ	4TJ
X	Orig.	CPU	4.308	5.396	6.842	7.500
		Network	87.754	38.857	44.432	44.389
	Shuf.	CPU	4.598	6.457	7.601	8.290
		Network	87.828	61.961	67.117	67.518
Y	Orig.	CPU	2.301	2.279	3.355	2.400
		Network	30.097	10.800	11.145	10.476
	Shuf.	CPU	2.331	2.635	3.536	2.541
		Network	30.191	28.674	29.520	18.230

Table 2: CPU & network time (in seconds) on the slowest join of the most expensive query of X and Y

[1] The variable byte scheme of X, Y uses base 100 encoding.
[2] We materialize Y in chunks as it does not fit in RAM.
[3] 2TJ uses the $R \rightarrow S$ direction for selective broadcasts.

Step Description	Workload X		Workload Y	
	Orig.	Shuf.	Orig.	Shuf.
Hash partition R tuples	0.347	0.350	0.054	0.054
Hash partition S tuples	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	1.145	1.288	0.176	0.179
Sort received S tuples	1.627	1.777	0.535	0.572
Final merge-join	0.601	0.602	1.322	1.321

Table 3: Distributed hash join steps (in seconds)

Our implementation supports fixed byte widths. For X, we use 4-byte keys, 7-byte payloads for R, and 18-byte payloads for S. For Y, we use 4-byte keys, 33-byte payloads for R, and 43-byte payloads for S. The node locations use 1 byte and the tracking counts use 1 byte for X and 2 for Y.

Using workload X in the original order, 2-phase track join 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%, but do not reduce network more than 2-phase track join. If X is shuffled, 2-phase track join, the best version here, increase the CPU time by 40% 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. If Y is shuffled, 2-phase and 3-phase track join increase the CPU time without saving network, due to key repetitions. In fact, we would transfer more data than hash join, as shown Figure 11, if we used more nodes. 4-phase track join, the only useful version here, increases the CPU time by 9% and reduces network by 40%.

We project track join performance on 10X faster network (10 Gbit Ethernet) by scaling the network time. For X, we project track join to be $\approx$ 29% faster than hash join. For Y, track join would be $\approx$ 37% faster. Note that the hardware bundle of the commercial DBMS uses both faster CPUs (2X 8-core CPUs at 2.9GHz) and network (40 Gbit InfiniBand).

Tables 3 and 4 show the times per step for hash join and 4-phase track join. We use sort-merge-join (MSB radix-sort) for local joins. Data movement is separated between remote and local transfers. For track join, we sort both tables and aggregate the keys. We shuffle keys and counts across the network and generate schedules by splitting pairs of nodes into four distinct parts for migrating and broadcasting R and S. The transferred pairs are merge-joined to translate nodes into payloads. The payloads are shuffled and the final join occurs. For X, scheduling takes half the time of local hash join, but is redundant since 2-phase track join suffices. For Y, scheduling is crucial and takes almost negligible time.

5. FUTURE WORK

The original ordering of tuples in X exhibits locality skew among nodes, reducing the potential for improvement over hash join. If some nodes exhibit more locality than others, we need to take into account the balancing of transfers among nodes and not only aim for minimal network traffic.

A pipelined implementation can reduce end-to-end time by overlapping CPU and network [16]. Track join is more complex than hash join, offering more choices for overlap. Thus, investigating alternative implementations that overlap CPU and network to minimize time is a crucial problem.

Step Description	Workload X		Workload Y	
	Orig.	Shuf.	Orig.	Shuf.
Sort local R tuples	0.979	1.300	0.182	0.182
Sort local S tuples	1.401	1.792	0.534	0.565
Aggregate keys	0.229	0.227	0.022	0.025
Hash part. keys, counts	0.373	0.372	0.011	0.018
Transfer key, count	26.800	27.339	0.977	1.378
Local copy key, count	0.034	0.034	0.093	0.001
Merge recv. key, count	0.506	0.507	0.015	0.022
Generate schedules and partition by node	1.627	1.650	0.035	0.047
Tran. $R \rightarrow S$ keys, nodes	7.277	10.913	0.346	0.532
Tran. $S \rightarrow R$ keys, nodes	6.046	1.562	0.135	0.247
Local copy keys, nodes	0.016	0.016	0.000	0.000
Merge rec. keys, nodes	0.237	0.235	0.007	0.012
Merge-join $R \rightarrow S$ keys, nodes $\Rightarrow$ payloads and partition by node	0.315	0.456	0.068	0.098
Merge-join $S \rightarrow R$ keys, nodes $\Rightarrow$ payloads and partition by node	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 rec. $R \rightarrow S$ tuples	0.161	0.531	0.045	0.067
Merge rec. $S \rightarrow R$ 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

Table 4: Track join (4-phase) steps (in seconds)

6. RELATED WORK

The foundations of distributed query processing and database implementation [5, 8, 10, 11, 20] have been laid some decades back. The baseline hash join algorithm we also use in our work is the version presented in the Grace database machine [17], later parallelized by the Gamma project [9]. Distributed algorithms that filter tuples to reduce network traffic have been extensively studied, namely semi-joins [4, 28, 30], Bloom-joins [22], and other approaches [19].

Contemporary databases, as mentioned, shift from the one-size-fits-all paradigm [32] and are designed as optimized programs for specific needs, the most important cases being analytical and transactional systems. Transactional distributed databases are targeting higher transaction throughput, by eliminating unnecessary network communication [32] due to latency induced delays of distributed commit protocols. Storage managers have also been improved since [13, 14] to keep up with the increasing speed. Analytical systems have evolved into column stores [26, 31] and query execution works primarily in RAM [21]. Using the latest hardware platforms, the network bandwidth remains an order of magnitude lower than the RAM bandwidth. There has also been significant work towards optimizing main-memory resident database operators, namely linear scans [18, 35], joins [2, 3, 15], group-by aggregations [36], and sorting [25, 29, 34].

In distributed sorting [16], CPU and network time can be overlapped by transferring keys and payloads separately. If the response time is important, a modified hash join has been proposed [33]. On faster networks, distributed joins have been evaluated [12], but the network was not the bottle-

neck for the algorithm used. Recent work discusses network time optimization [27] covering joins. The optimization is NP-complete and therefore cannot be applied per-key. In contrast, track join minimizes network volume using linear scheduling applied per key, achieving a finer granularity optimum. While completion time is not necessarily a better metric than network volume when partial results can be consumed as soon as they are generated, track join can utilize the temporal scheduling of [27] to reduce end-to-end time.

Popular generic distributed systems [7, 24] have been used for database operations such as joins [1, 23], minimizing network transfers by carefully mapping "map" and "reduce" operators to the most relevant data. However, "tracking" is still required to achieve high granularity collocation. Based on our non-pipelined implementation, track join can be reimplemented for MapReduce. Thus, network optimizations can occur both at a coarse-grain granularity by the framework and at a fine-grain granularity by using track join.

7. CONCLUSION

We presented track join, a novel algorithm for distributed joins that minimizes communication and therefore, 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 initial locations of tuples. Track join makes no assumptions about data organization in the DBMS and does not rely on schema properties or pre-existing favorable data placement to show its merits.

We described three versions of track join, 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 DBMS.

8. ACKNOWLEDGMENTS

We thank John Kowtko, Nipun Agarwal, and Eric Sedlar of the Oracle Labs for their invaluable help and support.

9. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.
[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, June 2012.
[3] C. Balkesen et al. Multicore, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, Sept. 2013.
[4] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, Jan. 1981.
[5] P. A. Bernstein et al. Query processing in a system for distributed databases. *ACM Trans. Database Systems*, 6:602–625, 1981.
[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426, July 1970.
[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[8] D. J. DeWitt et al. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.
[9] D. J. DeWitt et al. The Gamma database machine project. *IEEE Trans. Knowl. Data Engin.*, 2(1):44–62, Mar. 1990.
[10] D. J. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Comm. ACM*, 35:85–98, 1992.
[11] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, pages 169–180, 1978.
[12] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: high-speed networks for distributed join processing. In *DaMoN*, pages 27–33, 2009.
[13] M. Grund et al. HYRISE: A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, Nov. 2010.
[14] R. Johnson et al. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.
[15] C. Kim et al. Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, Aug. 2009.
[16] C. Kim et al. CloudRAMsort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *SIGMOD*, pages 841–850, 2012.
[17] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1:63–74, 1983.
[18] D. Lemire et al. Decoding billions of integers per second through vectorization. *Software: Pract. Exper.*, May 2013.
[19] Z. Li and K. A. Ross. PERF join: an alternative to two-way semijoin and Bloomjoin. In *CIKM*, pages 137–144, 1995.
[20] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, pages 149–159, 1986.
[21] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231–246, 2000.
[22] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Engin.*, 16(5):558–560, May 1990.
[23] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.
[24] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.
[25] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, 2014.
[26] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, Aug. 2013.
[27] W. Roediger et al. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, 2014.
[28] N. Roussopoulos and H. Kang. A pipeline n-way join algorithm based on the 2-way semijoin program. *IEEE Trans. Knowl. Data Engin.*, 3(4):486–495, Dec. 1991.
[29] N. Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.
[30] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distributed Systems*, 4(12):1345–1354, Dec. 1993.
[31] M. Stonebraker et al. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.
[32] M. Stonebraker et al. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
[33] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engin. Bulletin*, 23:27–33, June 2000.
[34] J. Wassenberg and P. Sanders. Engineering a multi core radix sort. In *EuroPar*, pages 160–169, 2011.
[35] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, Aug. 2009.
[36] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9, 2011.
[37] F. Yu et al. CS2: a new database synopsis for query estimation. In *SIGMOD*, pages 469–480, 2013.