Locality-Sensitive Operators for Parallel Main-Memory Database Clusters

Wolf Rödiger*, Tobias Mühlbauer*, Philipp Unterbrunner†, Angelika Reiser*, Alfons Kemper*, Thomas Neumann*

*Technische Universität München
Munich, Germany
{first.last}@in.tum.de

†Snowflake Computing, Inc.
San Mateo, CA
{first.last}@snowflakecomputing.com

***Abstract*—The growth in compute speed has outpaced the growth in network bandwidth over the last decades. This has led to an increasing performance gap between local and distributed processing. A parallel database cluster thus has to maximize the locality of query processing. A common technique to this end is to co-partition relations to avoid expensive data shuffling across the network. However, this is limited to one attribute per relation and is expensive to maintain in the face of updates. Other attributes often exhibit a *fuzzy* co-location due to correlations with the distribution key but current approaches do not leverage this.**

In this paper, we introduce *locality-sensitive data shuffling*, which can dramatically reduce the amount of network communication for distributed operators such as join and aggregation. We present four novel techniques: (i) *optimal partition assignment* exploits locality to reduce the network phase duration; (ii) *communication scheduling* avoids bandwidth underutilization due to cross traffic; (iii) *adaptive radix partitioning* retains locality during data repartitioning and handles value skew gracefully; and (iv) *selective broadcast* reduces network communication in the presence of extreme value skew or large numbers of duplicates. We present comprehensive experimental results, which show that our techniques can improve performance by up to factor of 5 for fuzzy co-location and a factor of 3 for inputs with value skew.

I. Introduction

Parallel databases are a well-studied field of research, which attracted considerable attention in the 1980s with the Grace [1], Gamma [2], and Bubba [3] systems. At the time, the network bandwidth was more than sufficient for the performance of these disk-based database systems. For example, DeWitt et al. [2] showed that Gamma took only 12% additional execution time when data was redistributed over the network—compared to an entirely local join of co-located data. Copeland et al. [4] stated in the context of Bubba in 1986:

> "[...] on-the-wire interconnect bandwidth will not be a bottleneck [...]."

The game has changed dramatically since the 1980s. On modern hardware, this formerly small overhead of 12% for distributed join processing has turned into a performance penalty of 500% (cf. Section V-B). The gap is caused by the development depicted in Fig. 1: Over the last decades, CPU performance has grown much faster than network bandwidth.

†Work conducted while employed at Oracle Labs, Redwood Shores, CA.

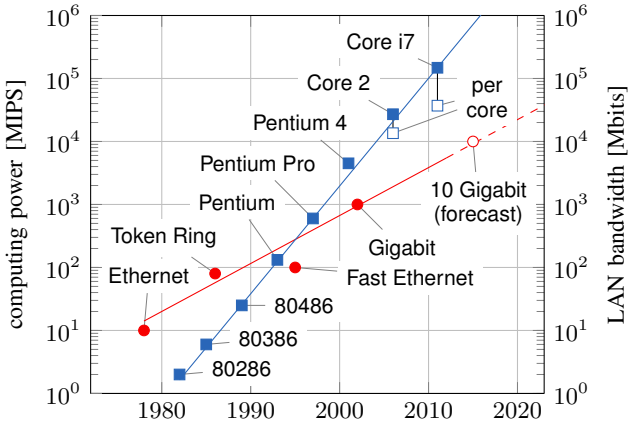


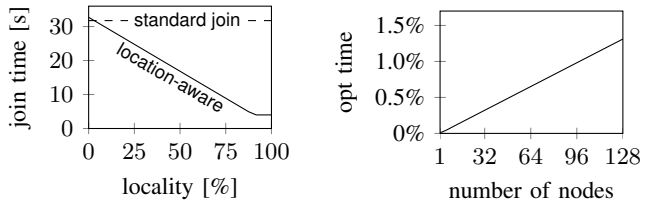
Fig. 1. The need for locality-sensitive distributed operators: CPU speed grows faster than network speed for commodity hardware[1]

This trend is likely to continue in the near future as the widespread adoption of 10 Gigabit Ethernet is currently projected for 2015 [5]. Today, "even inside a single data center, the network is a bottleneck to computation" [6].

Modern main-memory databases such as H-Store [8], HyPer [9], MonetDB [10], or Vectorwise [11] are not slowed down by disk access, buffer managers, or pessimistic concurrency control. The resulting, unprecedented query and transaction performance widens the gap to distributed processing further.

The current situation is highly unsatisfactory for parallel main-memory database systems: Important distributed operators such as join, aggregation, and duplicate elimination are slowed down by overly expensive data shuffling. A common technique to reduce network communication is to explicitly co-partition frequently joined tables by their join key [12]. However, co-partitioning of relations is limited to one attribute per relation (unless the data is also being replicated), requires prior knowledge of the workload, and is expensive to maintain in the face of updates. Other attributes often exhibit a fuzzy co-location due to strong correlations with the distribution key but current approaches do not leverage this.

[1]CPU MIPS from Hennessy and Patterson [7]. Dominant network speed of new servers according to the IEEE NG BASE-T study group [5].



(a) **Benefit:** Join time for 128 nodes, 1 GbE, and 200 M tuples per node (b) **Cost:** Overhead to optimize the partition assignment for data locality

Fig. 2. Comparison of the location-aware join to a standard distributed join

In this paper, we introduce *locality-sensitive data shuffling*, four novel techniques that can significantly reduce the amount of network communication for distributed operators. Its cornerstone is the *optimal partition assignment*, which allows operators to benefit from data locality. Most importantly, it does not degrade when data exhibits no locality. We employ *communication scheduling* to use all the available network bandwidth of the cluster. Uncoordinated communication would otherwise lead to cross traffic and thereby reduce bandwidth utilization dramatically. We propose an *adaptive radix partitioning* for the repartitioning to retain locality in the data and to handle value skewed inputs gracefully. *Selective broadcast* is an extension to the partition assignment that decides dynamically for every partition between shuffle and broadcast. With selective broadcast, our approach can even benefit from extreme value skew and reduce network communication further.

We primarily target clusters of high-end commodity machines, which consist of few but *fat* nodes with large amounts of main-memory. This typically represents the most economic choice for parallel main-memory databases. MapReduce-style systems are, in contrast, designed for large clusters of low-end machines. Recent work [12] has extended the MapReduce processing model with the ability to co-partition data. However, MapReduce-style systems cannot leverage fuzzy co-location yet and have to perform a network-intensive data redistribution between map and reduce tasks. They would therefore also benefit from a locality-sensitive reshuffling.

The immense savings that locality-sensitive data shuffling can achieve (cf. Fig. 2(a)) justify the associated runtime optimization costs (cf. Fig. 2(b)).

In summary, this paper makes the following contributions:

- *Optimal partition assignment*: We devise a method that allows operators to benefit from data locality.
- *Communication scheduling:* Our communication scheduler prevents cross traffic, which would otherwise reduce network bandwidth utilization dramatically.
- *Adaptive Radix Partitioning:* We develop an efficient repartitioning scheme, which retains locality. It further allows us to handle value skewed inputs gracefully.
- *Selective Broadcast:* We extend the partition assignment to selectively broadcast small partitions. This increases the performance for inputs with extreme value skew.

We describe our techniques based on Neo-Join, a network-optimized join operator. They can be similarly applied to other distributed relational operators that reduce to item matching, e.g., aggregation and duplicate elimination.

II. Related Work

As outlined in the introduction, distributed joins have first been considered in the context of database machines. Fushimi et al. introduced the Grace hash-join [13] of which a parallel version was evaluated by DeWitt et al. [2]. These algorithms are optimized for the disk as bottleneck. Disk is still the limiting factor for distributed file systems and similar applications [14], [15]. However, parallel join processing in main-memory database systems avoids the disk and is limited by the network.

Wolf et al. proposed heuristics for distributed sort-merge [16] and hash-join [17] algorithms to achieve load balancing in the presence of skew. However, their approach targets CPU bound systems and does not apply when network is the bottleneck. Wilschut et al. [18] devised a distributed hash-join algorithm with fewer synchronization requirements. Again, CPU costs were identified as the limiting factor. Stamos and Young [19] improved the fragment-replicate (FR) join [20] by reducing its communication cost. However, partition-based joins still outperform FR in the case of equi-joins. Afrati and Ullman [21] optimized FR for MapReduce, while Blanas et al. [22] compared joins for MapReduce. MapReduce focuses especially on scalability and fault-tolerance, whereas we target per-node efficiency. Frey et al. [23] designed a join algorithm for non-commodity high-speed Infiniband networks with a ring topology. They state that the network was not the bottleneck.

Systems with non-uniform memory access (NUMA) distinguish expensive remote from cheaper local reads similar to distributed systems. Teubner et al. [24] designed a stream-based join for NUMA systems. However, it does not fully utilize the bandwidth of all memory interconnect links. Albutiu et al. [25] presented MPSM, a NUMA-aware sort-merge based join algorithm. Li et al. [26] applied data shuffling to NUMA systems and in particular to MPSM. They showed that a coordinated round-robin access pattern increases the bandwidth utilization by up to 3× but improves the join by only 8% as sorting dominates the runtime. We show that data shuffling applied to distributed systems achieves much higher benefits. Li et al. did also not consider skew in the data placement.

Bloom filters [27] are commonly used to reduce the network traffic. They are orthogonal to our approach and can be used as a preliminary step. However, they should not be applied in all cases due to their incurred computation and communication costs. Dynamic bloom filters [28] lower the computation costs as they can be maintained continuously for common join attributes. Still, the exchange of the filters itself causes network traffic that increases quadratically in the number of nodes. They should only be used for joins with a small selectivity.

CloudRAMSort [29] introduced the idea to split tuples into key and payload. This is compatible with our approach and should be applied to reduce communication costs. A second data shuffling phase is merges result tuples with their payloads. While this effectively represents a second join, it is computed locally on the nodes where the payloads reside.

III. Neo-Join

Neo-Join is a distributed join algorithm based on locality-sensitive data shuffling. It computes the equi-join of two relations R and S, which are horizontally fragmented across the nodes of a distributed system. Neo-Join exploits locality and handles value skew gracefully using optimal partition assignment. Communication scheduling allows it to avoid cross-traffic. The algorithm proceeds in four phases: (1) *repartition* the data, (2) *assign* the resulting partitions to nodes for a minimal network phase duration, (3) *schedule* the communication to avoid cross traffic, and (4) *shuffle* the partitions according to the schedule while *joining* incoming partition chunks in parallel. In the following we describe these phases in detail.

A. Data Repartitioning (Phase 1)

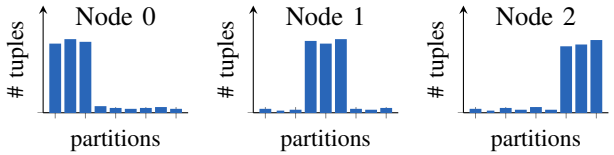
The idea to split join relations into disjoint partitions was introduced by the Grace [1] and Gamma [2] database machines. Partitioning ensures that all tuples with the same join key end up in the same partition. Consequently, partitions can be joined independently on different nodes. We first define locality and skew before covering different choices for the repartitioning.

Locality. We use the term locality to describe the degree of local clustering of a partitioning. Fig. 3(a) shows an example with high locality. We specify locality in percent, where $x\%$ denotes that on average for each partition the node with its largest part has $x\%$ of its tuples with an additional $1/n$-th of the remaining tuples (for n nodes). 0% thus corresponds to a uniform distribution where all nodes own equal parts of all partitions and 100% to the other extreme where nodes own partitions exclusively. Locality in the data distribution can have many reasons, e.g., a (fuzzy) co-partitioning of the two relations, a distribution key used as join key, a correlation between distribution and join key, or time-of-creation clustering. Locality can be created on purpose during load time to benefit from the significant savings possible with locality-sensitive operators. One may also let tuples wander with queries to create fuzzy co-location for frequently joined attributes.

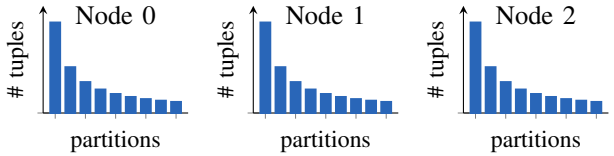
Skew. The term skew is commonly used to denote the deviation from a uniform distribution. Skewed inputs can significantly affect the join performance and therefore should be considered during the design of parallel and distributed join algorithms (e.g., [16], [25], [30]). We use the term *value skew* to denote inputs with skewed value distributions. Skewed inputs can lead to skewed partitions as shown in Fig. 3(b).

In the following, we assume the general case that distribution and join key differ. Otherwise, one or both relations would already be distributed by the join attribute and repartitioning becomes straightforward: The existing partitioning can be used to repartition the other relation. Our optimal partition assignment can automatically exploit the resulting locality.

There are many options to repartition the inputs. Optimal partition assignment (Section III-B) and communication scheduling (Section III-C) apply to any of them. However, repartitioning schemes such as the proposed radix partitioning



(a) **Locality:** the partition fragments exhibit local clustering on the nodes



(b) **Skew:** the partition sizes are skewed according to a Zipfian distribution

Fig. 3. Example for locality and skew with three nodes and nine partitions

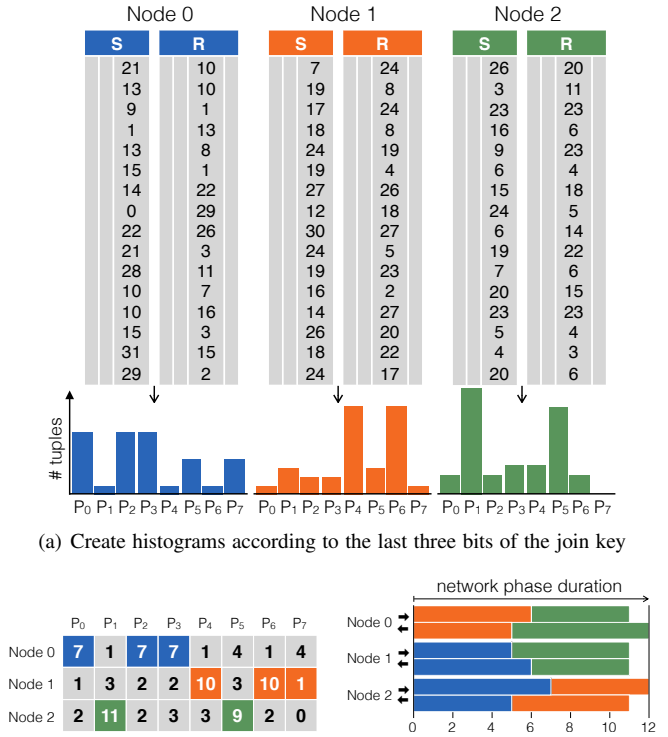
that retain locality in the data placement can improve the join performance dramatically, as we describe in the following.

1) Hash Partitioning: Hashing is commonly used for partitioning as it achieves balanced partitions even when the input exhibits value skew. However, hash partitioning can be sub-optimal. An example is an input that is almost range-partitioned across nodes (e.g., caused by time-of-creation clustering as is the case with *order* and *orderline* in TPC-H). With range-partitioning, the input could be assigned to nodes so that only the few tuples that violate the range partitioning are transferred. Hash partitioning destroys the range partitioning. The resulting network phase is much longer than necessary.

2) Radix Partitioning: We propose *radix partitioning* [31] of the join key based on the most significant bits[2] (MSB) to retain locality in the data placement. MSB radix partitioning is a special, "weak" case of hash partitioning, which uses the b most significant bits as the hash value. Of course, using the key directly does not produce such balanced partitions as proper hash partitioning. But more importantly, MSB radix partitioning is order-preserving and thereby a restricted case of range partitioning, which allows the algorithm to assign the partitions to nodes in a way that reduces the communication costs significantly. By partitioning the input into many more partitions than there are nodes, one can still handle value skew, e.g., when there is a bias towards small keys. Section IV covers techniques that handle moderate and extreme cases of value skew while keeping the number of partitions low.

Fig. 4 (on the next page) depicts a simple example with 5 bit join keys ($0 \leq \text{key} < 32$). First, the nodes compute histograms for their local input by radix-clustering the tuples into eight partitions $P_0, \ldots, P_7$ according to their 3 most significant bits $\mathbf{b_4 b_3 b_2} b_1 b_0$ as shown in Fig. 4(a). Next, the algorithm assigns these eight partitions to the three nodes so that the duration of the network phase is minimal. As we show in Section III-C, the network phase duration is determined by the maximum straggler, i.e., the node that needs the most time to receive or send its data. An optimal assignment, which minimizes the communication time of the maximum straggler, is shown in Fig. 4(b). With this assignment both node 0 and node 2 are

[2]More precisely, the most significant *used* bits to avoid leading zeros.



(a) Create histograms according to the last three bits of the join key

(b) Compute an optimal partition assignment based on the histograms

(c) Resulting send/receive costs determine the network phase duration

Fig. 4. Example for the optimal partition assignment which aims at a minimal network phase duration with three nodes and eight (2^3) radix partitions

maximum stragglers with a cost of 12 as depicted in Fig. 4(c). For a perfect hash partitioning one would expect that every node has to send $1/n$-th of its tuples to every other node (≈ 21) and also receive $1/n$-th of the tuples from every other node (also ≈ 21). In this simplified example, radix partitioning reduced the duration of the network phase by almost a factor of two compared to hash partitioning.

B. Optimal Partition Assignment (Phase 2)

The previous section described how to repartition the input relations so that tuples with the same join key fall into the same partition. In general, the new partitions are *fragmented* across the nodes. Therefore, all fragments of one specific partition have to be transferred to the same node for joining. This section describes how to determine an assignment of partitions to nodes that minimizes the network phase duration.

We define the *receive cost* of a node as the number of tuples it receives from other nodes for the partitions that were assigned to it. Similarly, its *send cost* is defined as the number of tuples it has to send to other nodes. Section III-C4 shows that the minimum network phase duration is determined by the node with the maximum send/receive cost. The assignment is therefore optimized to minimize this maximum cost.

A naïve approach would assign a partition to the node that owns its largest fragment. However, this is not optimal in general. Consider the assignment for the running example in Fig. 4(b). Partition 7 is assigned to node 1 even though node 0 owns its largest fragment. While the assignment of partition 7 to node 0 reduces the send cost of node 0 by 4 tuples, it also increases its receive cost to a total of 13 tuples. As a result, the network phase duration increases from 12 to 13 (cf. Fig. 4(c)).

1) Mixed Integer Linear Programming: We phrase the partition assignment problem as a mixed integer linear program (MILP). As a result, one can use an integer programming solver to solve it. The linear program computes a configuration of the decision variables $x_{ij} \in \{0, 1\}$. These decision variables define the assignment of the p partitions to the n nodes: $x_{ij} = 1$ determines that partition j is assigned to node i, while $x_{ij} = 0$ specifies that partition j is not assigned to node i.

Each partition has to be assigned to exactly one node:

$$\sum_{i=0}^{n-1} x_{ij} = 1 \qquad \text{for } 0 \le j < p \tag{1}$$

The linear program should minimize the duration of the network phase, which is equal to the maximum send or receive cost over all nodes. We denote the send cost of node i as s_i and its receive cost as r_i. The objective function is therefore:

$$\min \max_{0 \le i < n} \{s_i, r_i\} \tag{2}$$

Using the decision variables x_{ij} and the size of partition j at node i—denoted with h_{ij}—we can express the amount of data each node has to send (s_i) and receive (r_i):

$$s_i = \sum_{j=0}^{p-1} h_{ij} \cdot (1 - x_{ij}) \qquad \text{for } 0 \le i < n \tag{3}$$

$$r_i = \sum_{j=0}^{p-1} \left(x_{ij} \sum_{k=0, i \ne k}^{n-1} h_{kj} \right) \qquad \text{for } 0 \le i < n \tag{4}$$

Equation 3 computes the send cost of node i as the size of all local fragments of partitions that are not assigned to it. Likewise, equation 4 adds the size of remote fragments of partitions that were assigned to node i to the receive cost.

MILPs require a linear objective, which minimizing a maximum is not. Fortunately, we can rephrase the objective and instead minimize a new variable w. Additional constraints take care that w assumes the maximum over the send/receive costs:

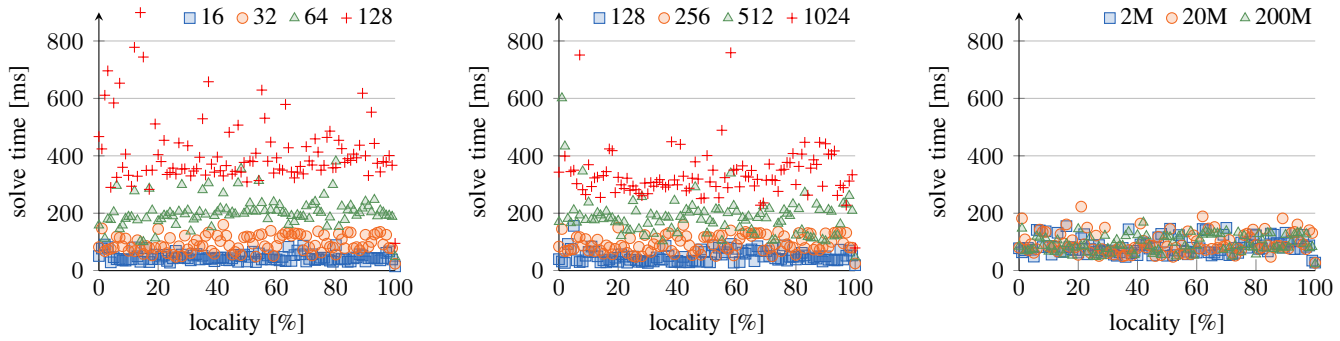
minimize w, subject to

$$\text{(OPT-ASSIGN)} \quad \begin{aligned} w &\ge \sum_{j=0}^{p-1} h_{ij}(1 - x_{ij}) && 0 \le i < n \\ w &\ge \sum_{j=0}^{p-1} \left(x_{ij} \sum_{k=0, i \ne k}^{n-1} h_{kj} \right) && 0 \le i < n \\ 1 &= \sum_{i=0}^{n-1} x_{ij} && 0 \le j < p \end{aligned}$$

One can obtain an optimal solution for a specific partition assignment problem (OPT-ASSIGN) by passing the mixed integer linear program to an optimizer such as Microsoft Gurobi[3] or IBM CPLEX[4]. These solvers can be linked as a library to create and solve linear programs via API calls.

[3] http://www.gurobi.com

[4] http://ibm.com/software/integration/optimization/cplex



(a) Nodes (256 partitions, 200M tuples per node) (b) Partitions (32 nodes, 200M tuples per node) (c) Tuples per node (32 nodes, 256 partitions)

Fig. 5. Time to compute the optimal partition assignment while increasing the locality and varying the number of nodes, partitions, and tuples per node

2) Argument Size Balancing: In general, one would like to avoid that large fractions of the argument relations are assigned to a single node. This could lead to exhaustion of the resources on this node (e.g., main memory) during join computation or when a (potentially very large) result set is generated. The following constraint restricts the input size for all nodes i to a multiple of the ideal input size:

$$\sum_{j=0}^{p-1} x_{ij}(h_j^R + h_j^S) \leq (1+o) \cdot \frac{|R|+|S|}{n} \quad \text{for } 0 \leq i < n \qquad (5)$$

where $o \in [0, n-1]$ is the overload factor, which is allowed in addition to the ideal input size, $|R|$ and $|S|$ denote the size of the argument relations, and h_j^R and h_j^S are the total size of partition P_j for relation R and S, respectively. Argument size balancing is not used in the following experiments.

3) NP-hardness: We provide a proof sketch to show that OPT-ASSIGN is NP-hard. We show that its *decision variant* (ASSIGN) is NP-complete by reducing the known NP-complete *partition problem* (PARTITION) to it. We recall from [32] that as a consequence OPT-ASSIGN is NP-hard. ASSIGN decides whether the objective function of OPT-ASSIGN is smaller or equal to a given constant k. PARTITION determines whether a given bag B of positive integers can be partitioned into bags S_1 and S_2 that have an equal sum.

The polynomial-time reduction is achieved as follows: Every integer c_i of the bag B corresponds to a partition P_i of size $2 \cdot c_i$ where two nodes n_1 and n_2 both own a fragment of size c_i. The send and receive cost for partition P_i is by construction equal to c_i for both nodes. ASSIGN can be used to decide whether an assignment exists in which both nodes have the same send and receive cost ($r_1 = r_2 = s_1 = s_2 = \text{sum}(B)/2$) by choosing $k = \text{sum}(B)/2$. If this is possible, the partitions assigned to node n_1 represent the subset S_1 and those for node n_2 the subset S_2. Therefore, a solution to the assignment problem is also a solution to the original partition problem. An example is shown in Fig. 6.

ASSIGN is in NP, with the partition assignment as a certificate. PARTITION is NP-complete [32] and we have constructed a reduction to ASSIGN. Therefore, ASSIGN is NP-complete and OPT-ASSIGN NP-hard. □

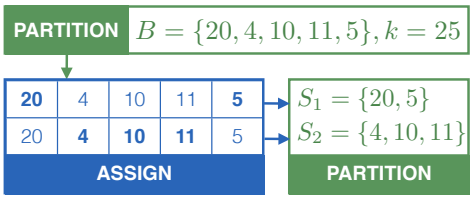


Fig. 6. An example for the reduction of PARTITION to ASSIGN

4) Optimization Time: Despite the NP-hardness of the partition assignment problem it is possible to solve real-world instances in reasonable time, i.e., much faster than the potential savings in network communication time. Fig. 5 depicts the solve time using the linear programming solver CPLEX for variations of the problem while increasing the locality.

In general, the solve time for a linear program increases with the number of variables. For the partition assignment, every combination between nodes and partitions is represented by a variable. Consequently, there is a direct correlation between the number of nodes/partitions and the solve time as visible in Fig. 5(a) and 5(b). The number of tuples per node has no impact on the solve time as expected and shown in Fig. 5(c).

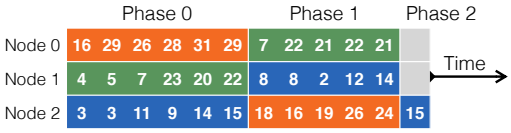
Fig. 5 indicates that the optimal assignment becomes expensive for clusters with hundreds or even thousands of nodes. As outlined in the introduction, our target are mainly clusters with fewer but fatter nodes as this is typically the most economic choice for parallel main-memory database clusters. Consequently, we only show the benefits and costs associated with the optimal solution. Efficient approximations should be possible to support larger clusters. We further expect an even slower network performance for very large clusters due to a shared network infrastructure, which would also increase the savings possible with locality-sensitive operators.

There are several options to minimize the optimization time: One can reduce the number of partitions by adaptively combining small partitions (cf. Section IV-A). The assignment can be precomputed eagerly or cached for recurring queries to avoid the runtime optimization overhead completely. Lastly, the locality can be estimated to invest the optimization time only when the expected savings are big enough.

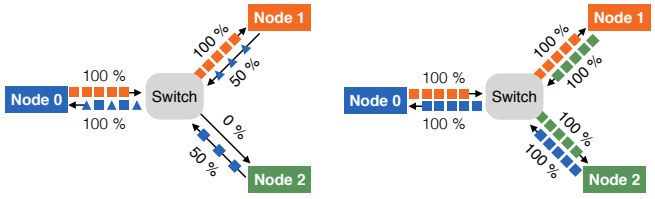
TABLE I
TERMINOLOGY MAPPING

Open Shop	Auto Shop	Network Transfer
job	car	sender
task	check engine	data transfer
processor	engine test bench	receiver
execution time	time for the check	message size
preemption	suspend check	split message

(a) A naïve schedule which uses only 67 % of the available bandwidth



(b) An optimal schedule which uses 94 % of the available bandwidth



(c) The naïve schedule leads to network congestion

(d) The optimal schedule avoids harmful cross traffic

Fig. 7. Naïve and optimal schedule for our running example with three nodes

C. Communication Scheduling (Phase 3)

The previous section described how to compute an optimal assignment of partitions to nodes. The next step is to redistribute the partitions according to this assignment. However, when the nodes use the network without coordination, the available bandwidth is utilized poorly. The scheduling of communication tasks can improve the bandwidth utilization significantly. We assume a star topology with uniform bandwidth that is common for small clusters. Our approach can be extended to non-uniform bandwidths by adjusting the partition assignment problem without increasing its complexity by adding variables or constraints (omitted due to space restrictions).

1) Network Congestion: A naïve distribution scheme would let all the nodes send their tuples to the first node, then to the second node, and so on. Fig. 7(a) depicts a naïve schedule for the running example. This simple scheme leads to significant network congestion since the nodes compete for the bandwidth of a single link while other links are not fully utilized. Fig. 7(c) visualizes the reason for the network congestion: Both, node 1 and node 2 send data to node 0 at the same time and therefore share the bandwidth of the link that connects node 0 to the switch. Node 0 can send with only 1 Gbit/s to either node 1 or node 2 although both could each receive simultaneously. Ultimately, 1 Gbit/s of bandwidth remains unused.

Network congestion can be avoided entirely by dividing the communication into distinct phases. In each phase a node has a single target to which it sends, and likewise a single source from which it receives. However, it is not obvious how to determine the phases so that a schedule with minimum finish time is realized. In practice, the nodes send different amounts of data to different nodes, which renders a simple round-robin scheme impractical. The problem to devise a communication schedule with minimum finish time corresponds to the open-shop scheduling problem [33]. It can be solved in polynomial time when preemption is allowed, which in this case corresponds to splitting network transfers into smaller chunks.

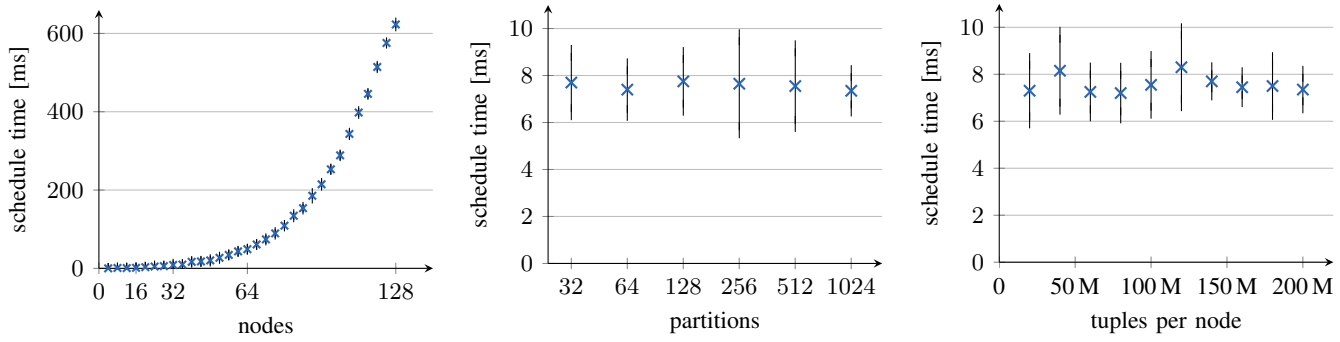
2) The Open Shop Scheduling Problem: The open shop problem is defined for abstract jobs, tasks, and processors. We explain it with the example of an auto shop. Afterwards, we translate the problem of computing optimal network phases into an open shop problem. As a result, one can use the polynomial-time algorithm that solves open shop problems to compute communication schedules.

An auto shop consists of m processors each dedicated to perform a specific repair task, e.g., the engine test bench, the wheel alignment system, and the exhaust test facility. There are multiple jobs, i.e., cars that need maintenance, consisting of m tasks, which need to be performed, e.g., check the engines, align wheels, and test the exhaust system. Each task of a job is performed by the corresponding processor. Every task has an associated processing time. The tasks may be performed in any order as it is irrelevant if the engine or the wheel alignment is checked first. However, two repair tasks cannot be performed for the same car simultaneously, since the processors are all located in different buildings. Similarly, a processor can check only one car at a time. Suspension of tasks is allowed. The goal is to find a schedule with minimal total processing time.

The network scheduling problem can be translated to an *open shop scheduling problem* with preemption as summarized in Table I: A *task* is the data transfer from one node to another and it has an *execution time* corresponding to the size of the data transfer. The *job* to which the task belongs is the sending node and the *processor* is the receiving node. A node should not send to several nodes simultaneously, similarly to the tasks of a job, which cannot be processed at different processors at the same time. A node should receive from at most one other node, just as a processor can execute only one task. The data transfer between two nodes can be split into multiple transfers, just as tasks can be *preempted*.

3) Solving Open Shops: Gonzales and Sahni [33] describe a polynomial time algorithm, which computes a minimum finish time schedule for open shops. The algorithm is based on finding perfect matchings in bipartite graphs. We explain it on the basis of the running example with three nodes.

The algorithm starts by generating the two vertex sets of the bipartite graph. The first set of vertices consists of a vertex for each sender and an equal number of additional vertices for virtual senders. Similarly, the second set has vertices for normal and virtual receivers. In the example, there are $N = 3$



(a) Nodes (256 partitions, 200M tuples per node) (b) Partitions (32 nodes, 200M tuples per node) (c) Tuples per node (32 nodes, 256 partitions)

Fig. 8. Time to schedule the network communication between nodes while varying the number of nodes, partitions, and tuples per node

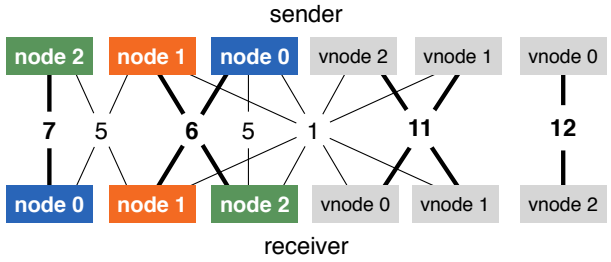


Fig. 9. The bipartite graph for our example with initial matching

nodes and therefore a total of 12 vertices in the bipartite graph as shown in Fig. 9. Each network transfer is represented as an edge connecting a sender with a receiver. Every edge is weighted with the transfer size, e.g., node 2 has to send 7 tuples to node 0. For optimality, it is important that all nodes have a weight equal to α, the maximum send or receive cost across nodes (12 for the running example as shown in Fig. 4(c)). Gonzales and Sahni describe how to insert edges between nodes and their virtual partners to achieve this.

The second step of the algorithm repeatedly finds perfect matchings. Every matching corresponds to a network phase and the edges of the matching define which nodes communicate in this phase. The minimal edge weight in the matching determines its duration. All matching edges are decreased by this amount and edges with weight zero are removed. Senders that are matched to virtual receivers do not send in this phase and receivers matched to virtual senders do not receive. This process is repeated until no edges remain.

The matching highlighted in Fig. 9 corresponds to the first phase of the schedule. Every transfer in this phase sends 6 tuples as this is the minimum edge weight. The edges of the matching specify that node 0 sends to node 1, node 1 to node 2, and node 2 to node 0. The resulting optimal schedule is shown in Fig. 7(b). It consists of three phases and achieves a network bandwidth utilization of 94%. In contrast, the naïve schedule utilizes only 67% of the available bandwidth.

4) Optimality: A schedule with a duration of less than α is not possible since α is the maximum send or receive cost across all nodes. At least one node has this send or receive cost and cannot finish earlier. Surprisingly, the algorithm always finds an optimal schedule with duration equal to α. The proof by Gonzales and Sahni can be found in [33].

5) Time Complexity: The runtime of the algorithm is in $\mathcal{O}(r^2)$ where r is the number of non-zero tasks [33]. Every transfer of the communication scheduling problem translates to a non-zero task. A system with n nodes has no more than $n(n-1)$ transfers because each of the n nodes sends to at most all other nodes. The runtime is therefore in $\mathcal{O}(n^4)$.

Fig. 8(a) compares the time needed to schedule the communication for a varying number of nodes. It is apparent that the problem size increases with the number of nodes. Fig. 8(b) and 8(c) show that the number of partitions or tuples do not affect the schedule time. The error bars show the standard deviation.

While the network scheduling is quite fast for up to 64 nodes where it takes about 50 ms, this increases to 623 ms for 128 nodes and even further for more nodes. Still, this is not a problem as the communication schedule can be computed incrementally and in parallel to the actual data shuffling. The nodes can start communicating as soon as the first phase is computed—which takes less than a millisecond. The remaining phases are then computed during the data shuffling.

6) Simultaneous Communication: We have until now assumed that no other communication happens over the network during the data shuffling. In the general case where simultaneous communication takes place, all network traffic needs to be scheduled to achieve the total available bandwidth of the database cluster. In this case the communication scheduling should be extended to guarantee fairness, so that no operator can "starve". However, this is out of scope for this paper.

D. Partition Shuffling and Local Join (Phase 4)

At this point, we have a partition assignment and a schedule, which describe how to redistribute the partitions. The only task that remains is to actually transfer the partitions over the network and join incoming partition chunks in parallel.

1) Partition Shuffling: In theory, there is no need to synchronize between the phases of the communication schedule. All nodes that participate in a phase send the exact same amount of data and should therefore also finish together. In reality, some nodes stop sending a little bit earlier than others

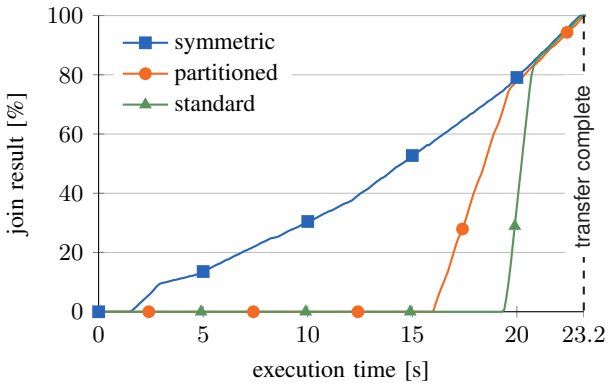


Fig. 10. Result availability over time for three different hash-join variants on 4 nodes with 200 M tuples each

due to variations in the TCP throughput, e.g., caused by packet loss. These nodes then send to their next target, which still receives from another node. As a result, both nodes share the bandwidth of the same link and are slowed down. The problem intensifies when other nodes start to use the links still occupied by the slower nodes. The situation is similar to a traffic jam.

To mitigate this problem, all nodes synchronize before they begin the next phase. While this avoids cross traffic, it introduces a synchronization barrier at which nodes could be forced to wait for a node with temporarily less bandwidth. Waiting for synchronization did not noticeably impact the performance in our experiments. Nevertheless, we propose a solution for it: The nodes stop sending when they exceed the time limit for the current phase and report their remaining tuples. The communication scheduler updates the bipartite graph, which represents the network transfers, accordingly. It then computes a perfect matching to determine the next phase.

2) Local Join: CPU performance and network bandwidth have grown at different speeds over the last decades. In today's systems the runtime of a distributed join is dominated by the network, whereas the local join computation on the nodes is less critical. Arriving tuples can be joined in parallel to the network transfer. Fig. 10 shows the result generation for three hash-join variants over time. It shows that the join finishes instantaneously when the last tuple arrives independent of the choice for the local join. Consequently, we have so far focused on the optimization of the data shuffling. Still, there are implementation choices to be made for the local join.

The *standard hash-join* [34] first builds a hash-table for the smaller input, which is then probed with the larger input. This two-phase approach effectively blocks for the probe input. In a distributed join, the probe input needs to be cached until after the entire build input has been received and processed. Only then can probing of the hash-table start to produce result tuples. This is visible as a steep incline in the result size about 20 seconds into the join computation.

The *symmetric hash-join* [18], a variant thereof known as XJoin [35], consists of only one phase: An incoming tuple is first probed into the hash-table of the other input and then added to the hash-table of its own input. The symmetric hash-join processes each tuple immediately, thereby avoids to postpone work, and still computes the correct result. It finishes at the same time than the standard hash-join even though it processes every tuple twice, once for each hash-table. The network transfer dominates the runtime to such a large extent that this additional work has no impact. The continuous processing of incoming tuples is reflected in the steady incline in the number of result tuples as shown in Fig. 10.

The *partitioned hash-join* is a third hash-join variant that performs no more work than the standard hash-join and can produce results earlier. It maintains a hash-table for every partition and can probe those hash-tables that are fully built, while build tuples for other partitions are still missing. This is evident in Fig. 10 as result tuples are generated already 16 seconds into the join computation.

Neo-Join supports all three hash-joins. It uses the partitioned hash-join by default since it is able to perform its work earlier than the standard hash-join. Further, the partitioned hash-join processes every tuple only once in contrast to the symmetric hash-join, which maintains two hash-tables. This becomes important for inputs with high locality since Neo-Join reduces the network transfer time. Partitioned and standard hash-join are in this case up to twice as fast as the symmetric hash-join. All subsequent experiments use the partitioned hash-join.

E. Shuffling the Join Result

So far, we have not considered the cost for reshuffling a large join result in preparation for the next join or aggregation operator in the query plan. There is no need to redistribute the join result when the subsequent operator references the same attribute as the current join. In fact, locality-sensitive data shuffling identifies and exploits the resulting co-location automatically. In all other cases, reshuffling is necessary and can have a significant impact on the total query execution time. All techniques described in this paper should be applied for the optimization of this additional data shuffling phase.

A combined optimization of successive operators could result in further improvements. The assignment of partitions to nodes influences the result sizes on the nodes, which in turn determine the cost for shuffling the result. We leave the adjustments to the partition assignment model that are needed to consider the duration of a succeeding result transfer phase as future work as this goes beyond the scope of this paper.

Instead of a separate phase, the result could already be redistributed during the join computation. However, it is hard to estimate at which point in time result tuples are available at specific nodes. Moreover, the bandwidth of the nodes should already be fully utilized during the data shuffling for the join. The redistribution of the result could instead start after the data shuffling for the join has finished, yet possibly before the end of the join computation. This further increases the overlapping of communication and computation. However, in a network-bound system this situation will only arise when the data shuffling phase of the join has a short duration due to data co-location.

IV. Handling Skew

We extend the locality-sensitive data shuffling to handle value skew and high numbers of duplicates without increasing the problem size for the partition assignment. Extreme value skew can even be leveraged to reduce communication further.

A. Adaptive Radix Partitioning

The optimal partition assignment as presented so far handles skewed inputs by balancing larger partitions with many smaller ones. This handles those cases quite well where for example 80% of the data lies in the first 20% of the value range. However, for extreme cases of value skew a considerable number of partitions is needed for a balanced partition assignment. This is highly undesirable as the runtime of the partition assignment increases with the number of partitions.

We extend radix partitioning to handle inputs with extreme value skew by adaptively combining small partitions, hence called adaptive radix partitioning (ARP). With ARP the nodes create wide histograms with many buckets. The partition assigner aggregates these into a global histogram and combines buckets that are smaller than a certain threshold. The resulting partitions are better balanced than with standard radix partitioning. Most importantly, the number of partitions used in the optimal optimal partition assignment is kept small.

We evaluate ARP with the Zipf distribution, which is commonly used to model extreme cases of value skew and high numbers of duplicates. The Zipf factor $s \geq 0$ controls the extent of skew, where $s = 0$ corresponds to a uniform distribution. Zipf is known to model real world data accurately, including the size of cities and word frequencies [36].

Our micro-benchmark consists of two relations, *city* and *person*, where person has a foreign key *hometown* referencing the city relation. Both relations contain 400 M tuples. The *hometown* attribute of the *person* table is skewed to model the fact that most persons live in few cities. We varied the Zipf factor s from 0 to 1. $s = 0$ corresponds to a uniform distribution as mentioned before, while $s = 1$ implies that 77% of the values are in the first 1% of the value range. Note that for $s = 1$ the number of duplicates is also quite high: the value 0 occurs in 21 M tuples (5%), the value 1 in 10 M (2.4%), the value 2 in 7 M (1.6%), etc. The Zipf factor n denotes the number of elements, in this case $n = 400$ M.

Table II shows the join duration using 4 nodes for an increasingly skewed *person* table. With simple radix partitioning, even 512 partitions do not suffice to maintain the join duration for $s = 1$. On the other hand, adaptive radix partitioning needs only 16 partitions to sustain the duration of the uniform case.

Since ARP produces better results for the same number of partitions, it should be used as a replacement for the standard radix partitioning we described in Section III-A2.

TABLE II
Join duration in seconds for increasing value skew

	Zipf factor s				
	0.00	0.25	0.5	0.75	1.00
16 partitions	27 s	24 s	23 s	29 s	44 s
512 partitions	23 s	23 s	23 s	23 s	33 s
16 partitions (ARP)	23 s	24 s	24 s	24 s	24 s
16 partitions (SB)	24 s	24 s	23 s	20 s	10 s
16 partitions (SB + ARP)	23 s	23 s	24 s	20 s	10 s

B. Selective Broadcast

Selective broadcast (SB) extends the optimal partition assignment so that it dynamically decides for every partition whether to assign it to a node or broadcast one of its relation fragments instead. This achieves two things: First, it covers the case when one relation is significantly smaller than the other so that broadcasting it is more efficient than partition shuffling. Second, the ability to decide between broadcast and shuffle for every single partition is highly beneficial for skewed inputs.

1) Shuffle or Broadcast: There are two fundamental options for distributed joins: (i) shuffle both relations so that tuples with the same key end up on the same node, or (ii) let one relation remain fragmented across the system and broadcast the other—also known as the fragment-replicate join [20]. Shuffling relations R and S incurs communication costs of $\frac{n-1}{n} \cdot (|R| + |S|)/n$ as each of the n nodes sends $1/n$-th of its fragments of R and S to the other $n-1$ nodes. Broadcasting R costs $(n-1) \cdot |R|/n$ as every node sends its fragment of R to every other node. Broadcast thus performs better than shuffling when the ratio between relations is higher than $1 : (n-1)$.

The locality-sensitive data shuffling as presented so far only considers shuffling. Selective broadcast extends our approach so that it decides for every partition whether to shuffle or broadcast it. In particular, this covers the case where one relation is much smaller than the other and should be broadcast as a whole. As a consequence, selective broadcast performs always at least as good as shuffling both relations or broadcasting the smaller as illustrated in Fig. 11 for 4 nodes. Selective broadcast can even outperform broadcast and shuffle for inputs with high value skew as we explain in the next section.

2) Skew: Selective broadcast can lead to significant speed-ups in the case of extremely skewed inputs. It broadcasts those partition fragments of one relation that are significantly smaller than their counterpart of the other relation. The remaining partitions are either broadcast by the other relation or assigned to nodes as before. Fig. 12 illustrates this for two relations R and S where S is skewed towards small values. The first five partitions of R are broadcast as the corresponding partitions of S are much larger. The remaining partitions are assigned to nodes and shuffled as before. This has the additional benefit that partitions are kept local that are large due to a high number of duplicates, which is common for Zipf distributions.

To show the potential savings with selective broadcast, we come back to the example of the *city* and *person* relations. For Zipf factor $s = 1$ and 16 partitions, the join performance increases by a factor of 2.8 when compared to the non-skewed case $s = 0$. In particular, the first three partitions are broadcast by *city*, the next four partitions are shuffled, while the remaining nine partitions are broadcast by *person*.

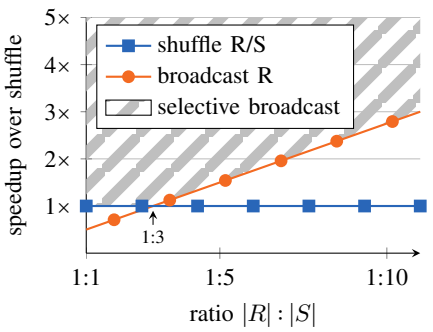


Fig. 11. The selective broadcast of R outperforms a complete broadcast or shuffling R/S by design

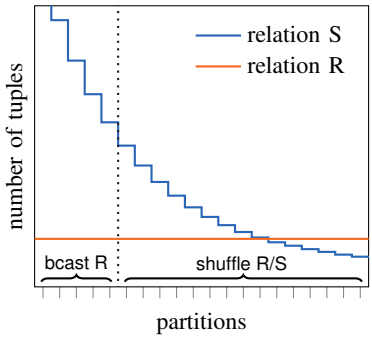


Fig. 12. Selective broadcasting decides dynamically whether to broadcast or shuffle partitions

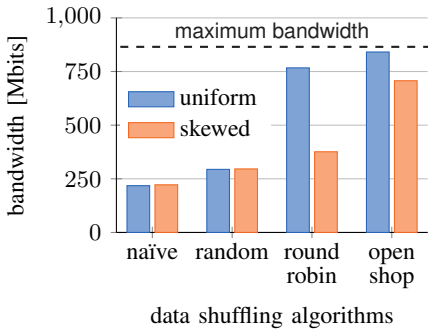


Fig. 13. Comparison of *data shuffling algorithms* for uniform and skewed inputs on 4 nodes

3) Model Extension: In the following, we describe how to extend the mixed integer linear program of the partition assignment to support selective broadcasting of partitions.

We have to model the choice between assigning a partition to a node or broadcasting it by either relation S or relation R. The existing binary variables $x_{ij} \in \{0,1\}$ specify whether partition j is assigned to node i. We add two new variables $y_j, z_j \in \{0,1\}$ per partition j that denote if its fragment for relation R respectively S is broadcast instead.

The constraints have to be updated using the new variables. Previously, the model included the restriction that every partition has to be assigned to exactly one node (cf. Equation 1). In the new model, each partition j is either assigned to a node or broadcast for one of the two relations. All variables that refer to the same partition therefore have to be mutually exclusive:

$$\left(\sum_{i=0}^{n-1} x_{ij}\right) + y_j + z_j = 1, \qquad \text{for } 0 \le j < p-1 \qquad (6)$$

The constraints for the send and receive costs have to be updated as well. The send cost r_i of node i was previously defined as the sum of all partitions it has to send because they were assigned to other nodes (cf. Equation 3). For selective broadcast, one has to account for the additional cost for partitions that are broadcast instead: If a partition is broadcast, all nodes that own a relation fragment of this partition have to send it to all $n-1$ other nodes. The additional send cost of node i for broadcasts s_i^B is therefore:

$$s_i^B = \sum_{j=0}^{p-1} \left(y_j(n-1)h_{ij}^R + z_j(n-1)h_{ij}^S\right) \quad \text{for } 0 \le i < n \qquad (7)$$

where h_{ij}^R and h_{ij}^S denote the size of the relation fragments of partition j at node i for R and S, respectively. In addition to the receive cost r_i for partitions that were assigned to node i (cf. Equation 4), it also receives all relation fragments from the other nodes for partitions that are broadcast. The additional receive costs of node i for broadcasts r_i^B are:

$$r_i^B = \sum_{j=0}^{p-1} \left(y_j \sum_{k=0, i\neq k}^{n-1} h_{kj}^R + z_j \sum_{k=0, i\neq k}^{n-1} h_{kj}^S \right) \quad \text{for } 0 \le i < n \qquad (8)$$

The objective (cf. Equation 2) remains unchanged, the program still minimize the maximum send or receive cost across all nodes. One can now update the linear program for the partition assignment with equations 6-8 to support selective broadcasts, modulo a smaller adjustment to the send cost:

minimize w, subject to

$$\text{(SEL-BCAST)} \quad \begin{aligned} w &\ge s_i + s_i^B - \sum_{j=0}^{p-1} (y_j h_{ij} + z_j h_{ij}) && 0 \le i < n \\ w &\ge r_i + r_i^B && 0 \le i < n \\ 1 &= \left(\sum_{i=0}^{n-1} x_{ij}\right) + y_j + z_j && 0 \le j < p \end{aligned}$$

The runtime of the partition assignment increases by an average of 39% with selective broadcast enabled (comparing the geometric mean of 720 experiments: 8 to 64 nodes, 4 to 16 partitions per node, 20 levels of locality, 3 repetitions).

V. Evaluation

All experiments of this paper were conducted on a shared-nothing [37] cluster of four identical machines except the scale up experiment. The cluster is connected via Gigabit Ethernet. Each of the four nodes has 32 GB of RAM, an Intel Core i7-3770 processor with four cores at 3.4 GHz each. The machines run Linux 3.8 as operating system. The implementation links to the IBM CPLEX library for MILP solving.

A. Data Shuffling Alternatives

Fig. 13 compares the bandwidth utilization of four different data shuffling schemes for uniform and skewed inputs. For the naïve scheme all nodes first send to node 0, followed by node 1, and so on. In the random data shuffling scheme targets are selected at random. Both the naïve and the random scheme are not synchronized as this decreases their performance. The round-robin scheme orders the nodes in a cycle. Each node sends first to its clock-wise neighbor, then to the one after that, etc. The phases are synchronized to avoid cross traffic. Open shop is our data shuffling scheme, which is based on an open shop schedule and is also synchronized.

The naïve and random distribution scheme perform similar for both inputs. The bandwidth utilization of round-robin deteriorates for the skewed input. Open Shop handles the skewed input better due to its optimal communication scheduling.

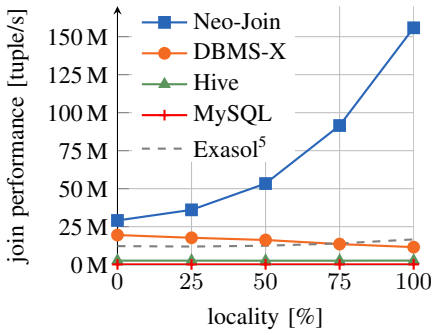


Fig. 14. Join performance for increasing locality with one coordinator and three data nodes

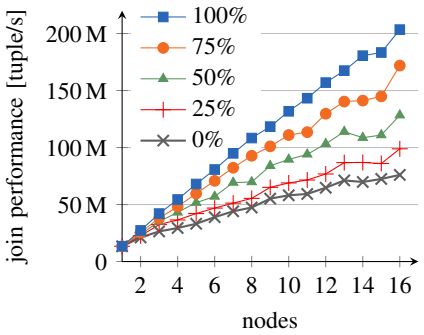


Fig. 15. *Scale up* across 16 nodes with 100 M tuples each, while varying the degree of locality

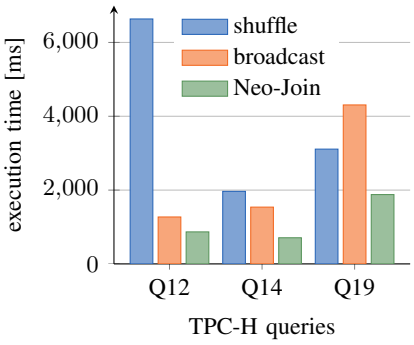


Fig. 16. Execution time of selected TPC-H queries for hash-based shuffling, broadcast, and Neo-Join

B. Locality

Fig. 14 compares Neo-Join to MySQL Cluster 7.2.10, Hive 0.10 on a Hadoop 1.1.1 cluster, and the commercial system DBMS-X for five different levels of locality. MySQL Cluster is a distributed variant of the MySQL database, which uses main memory as storage. Hive is a data warehouse system based on Hadoop, the open source implementation of MapReduce. For better comparability, we use the in-memory file system `ramfs` instead of a disk and tuned the number of map/reduce tasks. DBMS-X is a disk-based column store, which is configured likewise to use main memory. However, whether disk or main memory were used did not result in noticeable differences in performance, which indicates that indeed the network bandwidth is the limiting factor. All systems were configured to use three data nodes and one coordinator.

The experiments use a total of 600M tuples (100M tuples per relation per node). Each tuple consists of a key and a payload, both defined as `DECIMAL(18, 0)`, which allows for a physical representation as two 64bit integers. Neo-Join is in general independent of the data layout, but we implemented it for a row-store. The join key is constrained to $[0, 2^{32})$ so that the join result is not nearly empty—which is the case for 64 bit uniform join keys. We varied the locality, where $x\%$ denotes that for each partition the node with its largest part owns $x\%$ of its tuples with an additional $1/n$-th of the remaining tuples (for n nodes). 0% thus corresponds to a uniform distribution where all nodes own equal parts of all partitions and 100% to the other extreme where nodes own partitions exclusively.

For the case of uniformly distributed data, MySQL needs about an hour for the join while Hive finishes in 4 minutes, DBMS-X in 30 seconds, and Neo Join in 20 seconds. The join performance of MySQL and Hive stays mostly unchanged with increasing locality while DBMS-X's performance deteriorates by 41%. Neo-Join is able to improve its performance with increasing locality from 30 million tuples/s to more than 150 million tuples/s. The join performance for perfectly co-located data (100% locality) is 5× higher compared to a uniform distribution (0% locality). This emphasizes the importance of locality-sensitive distributed database operators. None of the contenders were able to take advantage from locality.

[5]Exasol, record holder in the TPC-H cluster benchmark, was measured with the same input on a different cluster with lower CPU performance but the same network bandwidth—which is the bottleneck for distributed joins.

C. Scale Up

The scale up experiment was conducted on a cluster of 16 nodes. Each node has an Intel Core 2 Quad Q6700 CPU with four cores at 2.66 GHz and 8 GB of main memory. They are connected over Gigabit Ethernet—still the dominant connection speed for new servers [5]. The nodes are somewhat aged as visible in the rather inferior single node join performance. Note that a higher processing power would also further improve the join performance for joins with high locality.

A linear scale up is defined as a linear increase in join performance when the number of nodes and the size of the input is increased proportionally. In this case, the input is increased by 100M tuples for every additional node. Fig. 15 demonstrates that Neo-Join scales linearly with the number of nodes. Moreover, it shows that locality-aware data-shuffling does not deteriorate when there is no locality in the data.

D. TPC-H

We chose the TPC-H benchmark to test our approach with a more realistic data set. We generated the relations for a scale factor of 100 and split them into four parts, one per node. The resulting data set has about 100 GB. We compare our approach to a hash-based shuffle of both relations and a broadcast of the smaller relation. These are the two state-of-the-art choices for the data shuffling phase of a distributed query. The results for selected single-join queries are shown in Fig. 16.

The selection of Q12 is so restrictive that one of the join inputs is 45× larger than the other. Consequently, a broadcast of the smaller relation is much faster than shuffling both relations. Neo-Join improves over this by exploiting the near-perfect co-location of the *orders* and *lineitem* tables caused by time-of-creation clustering. Note that Neo-Join does not assign the few partitions that violate the co-partitioning but instead selectively broadcasts them by the smaller relation, which is even faster. It achieves a speedup of 7.6× over shuffling and 1.5× over broadcasting. For Q14, Neo-Join is able to exploit the time-of-creation clustering of the *part* relation and

repartitions the *lineitem* table, which exhibits no locality on the join attribute. This improves the execution time by a factor of 3.4 over shuffling respectively 1.2 for broadcast. The size of the input relations for Q19 differ only by 55%, thus shuffling becomes faster than broadcast. Neo-Join again exploits the locality in the data placement of the *part* relation and is 1.7× faster than a shuffle and 2.3× faster than a broadcast.

VI. Conclusion

Over the last decades, compute speed has grown much faster than network speed. In parallel main-memory database clusters, it is thus of utmost importance to maximize the locality in query processing. A common technique is to co-partition relations to reduce the expensive data shuffling. However, co-partitioning is restricted to one attribute per relation (unless it is also being replicated) and expensive to maintain under updates. Other attributes often exhibit a *fuzzy* co-location but current approaches do not leverage this.

In this paper, we have introduced *locality-sensitive data shuffling*, a set of four techniques that can dramatically reduce the amount of network communication of distributed operators. We have presented four novel techniques: (i) *optimal partition assignment* computes an assignment with minimum network phase duration given any repartitioning of the input while considering locality, skew and the case that some nodes own larger parts of a relation than others; (ii) *communication scheduling* leverages all the available network bandwidth in a cluster; (iii) *adaptive radix partitioning* retains locality in the data and handles value skew gracefully; and (iv) *selective broadcast* allows to reduce network communication for cases with extreme value skew by dynamically deciding whether to shuffle or broadcast a partition. We have presented comprehensive experimental results, which show that our approach can improve performance by up to a factor of 5 for fuzzy co-location and a factor of 3 for inputs with value skew.

Acknowledgments

Wolf Rödiger is a recipient of the Oracle External Research Fellowship. Tobias Mühlbauer is a recipient of the Google Europe Fellowship in Structured Data Analysis. This work has further been partially sponsored by the German Federal Ministry of Education and Research (BMBF) grant HDBC 01IS12026. We would like to thank Maximilian Schlund for his advice and helpful suggestions.

References

[1] S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of a parallel relational database machine GRACE," in *PVLDB*, 1986.

[2] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The Gamma database machine project," *TKDE*, vol. 2, no. 1, 1990.

[3] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, a highly parallel database system," *TKDE*, vol. 2, no. 1, 1990.

[4] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," *SIGMOD Record*, vol. 17, no. 3, 1988.

[5] IEEE study group on Next Generation 802.3 BASE-T, "Next generation BASE-T call for interest," http://www.ieee802.org/3/NGBASET, 2012.

[6] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," *CACM*, vol. 54, no. 3, 2011.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture*, 5th ed. Morgan Kaufmann, 2011.

[8] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a high-performance, distributed main memory transaction processing system," *PVLDB*, vol. 1, no. 2, 2008.

[9] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011.

[10] S. Manegold, M. L. Kersten, and P. Boncz, "Database architecture evolution: mammals flourished long before dinosaurs became extinct," *PVLDB*, vol. 2, no. 2, 2009.

[11] M. Zukowski, M. van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical DBMS," in *ICDE*, 2012.

[12] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *SIGMOD*, 2013.

[13] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *NGC*, vol. 1, no. 1, 1983.

[14] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *SoCC*, 2012.

[15] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage," in *OSDI*, 2012.

[16] J. L. Wolf, D. M. Dias, and P. S. Yu, "A parallel sort merge join algorithm for managing data skew," *TPDS*, vol. 4, no. 1, 1993.

[17] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias, "A parallel hash join algorithm for managing data skew," *TPDS*, vol. 4, no. 12, 1993.

[18] A. N. Wilschut and P. M. G. Apers, "Dataflow query execution in a parallel main-memory environment," in *PDIS*, 1991.

[19] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joins," *TPDS*, vol. 4, no. 12, 1993.

[20] R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in a relational data base system," in *SIGMOD*, 1978.

[21] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *EDBT*, 2010.

[22] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MapReduce," in *SIGMOD*, 2010.

[23] P. Frey, R. Goncalves, M. Kersten, and J. Teubner, "Spinning relations: high-speed networks for distributed join processing," in *DaMoN*, 2009.

[24] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *SIGMOD*, 2011.

[25] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *PVLDB*, vol. 5, no. 10, 2012.

[26] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman, "NUMA-aware algorithms: The case of data shuffling," in *CIDR*, 2013.

[27] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *CACM*, vol. 13, no. 7, 1970.

[28] M. A. Bender, "Don't thrash: how to cache your hash on flash," *PVLDB*, vol. 5, no. 11, 2012.

[29] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, "CloudRAMSort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster," in *SIGMOD*, 2012.

[30] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs," *PVLDB*, 2009.

[31] S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *TKDE*, vol. 14, no. 4, 2002.

[32] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[33] T. Gonzalez and S. Sahni, "Open shop scheduling to minimize finish time," *JACM*, vol. 23, no. 4, 1976.

[34] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *SIGMOD*, 1984.

[35] T. Urhan and M. J. Franklin, "XJoin: A reactively-scheduled pipelined join operator," *IEEE Data Engineering Bulletin*, vol. 23, no. 2, 2000.

[36] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *SIGMOD Record*, vol. 23, no. 2, 1994.

[37] M. Stonebraker, "The case for shared nothing," *IEEE Database Enginieering*, vol. 9, no. 1, 1986.