Rethinking Distributed Query Execution on High-Speed Networks

Abdallah Salama[†*], Carsten Binnig[†], Tim Kraska [†], Ansgar Scherp[*], Tobias Ziegler[†]
[†]Brown University, Providence, RI, USA
[*]Kiel University, Germany

Abstract

In modern high-speed RDMA-capable networks, the bandwidth to transfer data across machines is getting close to the bandwidth of the local memory bus. Recent work has started to investigate how to redesign individual distributed query operators to best leverage RDMA. However, all these novel RDMA-based query operators are still designed for a classical shared-nothing architecture that relies on a shuffle-based execution model to redistribute the data.

In this paper, we revisit query execution for distributed database systems on fast networks in a more holistic manner by reconsidering all aspects from the overall database architecture, over the partitioning scheme to the execution model. Our experiments show that in the best case our prototype database system called I-Store, which is designed for fast networks from scratch, provides $3\times$ *speed-up over a shuffle-based execution model that was optimized for RDMA.*

1 Introduction

Motivation: Distributed query processing is at the core of many data-intensive applications. In order to support scalability, existing distributed database systems have been designed along the principle to avoid network communication at all cost since the network was seen as the main bottleneck in these systems. However, recently modern high-speed RDMA-capable networks such as InfiniBand FDR/EDR have been become cost competitive. To that end, these networks are no longer used only for high-performance computing clusters, but also become more prevalent in normal enterprise clusters and data centers as well.

With modern high-speed RDMA-capable networks, the bandwidth to transfer data across machines is getting close to the bandwidth of the local memory bus [3]. Recent work has therefore started to investigate how high-speed networks can be efficiently leveraged for query processing in modern distributed main-memory databases [13, 12, 2, 1]. The main focus of this line of work so far was centered around the question of how to redesign individual distributed query operators such as the join to leverage RDMA to make best use of fast networks.

Copyright 2017 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

However, none of the before-mentioned papers has revisited distributed query processing on fast networks in a more holistic manner. Instead, all the novel RDMA-based distributed algorithms are still designed for a classical shared-nothing architecture. In this classical database architecture, data is partitioned across the nodes, whereas each node has direct access only to its local partition. For processing a distributed join operator, data is first shuffled to other nodes based on the join keys before a local join algorithm (e.g., a radix join) is applied to the re-shuffled input. A main benefit of this processing model on slow networks is that it allows the distributed join operators to reduce the amount of data that is transferred over the network; e.g., by applying a semi-join reduction as a part of the data shuffling process. While reducing the bandwidth consumption is a valid reason for slow networks, it should not be the main design consideration when designing distributed query processing algorithms on fast networks. To that end, instead of relying on a shared-nothing architecture and a shuffle-based processing model that was designed to reduce the bandwidth consumption in slow networks, efficient distributed query execution for fast networks should rather focus on other aspects.

First, the main difference of a remote memory accesses in modern RDMA-capable networks compared to a local memory access is not the bandwidth, but the access latency, which is still an order of magnitude higher in the remote case. Thus, relying on an execution model that needs to shuffle the input data for every join operator in a query plan and therefore needs to pay the high latencies of remote data transfers multiple times per query is not optimal. Second, for modern scalable distributed database systems there are other important properties such as elasticity as well as load-balancing that are not naturally supported in a distributed database system that is built on a shared-nothing architecture and uses a shuffle-based execution model.

Contributions: The main contribution of this paper is that it is a first attempt to revisit query execution for distributed database systems on fast networks from the ground up.

First, instead of building on a classical shared-nothing architecture we build our distributed query engine *I-Store* on a more recent architecture for fast networks called the Network-Attached-Memory (NAM) architecture [3]. The main idea of the NAM architecture is that it logically decouples compute and memory nodes and uses RDMA for communication between all nodes as shown in Figure 1. Memory servers provide a shared distributed memory pool that holds all the data, which can be accessed via one-sided RDMA operations from compute servers that execute the queries. In this architecture, compute servers can be scaled independently from memory servers. Furthermore, a compute server can access the data that resides in any of the memory server independent of its location. To that end, elasticity as well as load-balancing can naturally be supported in this architecture.

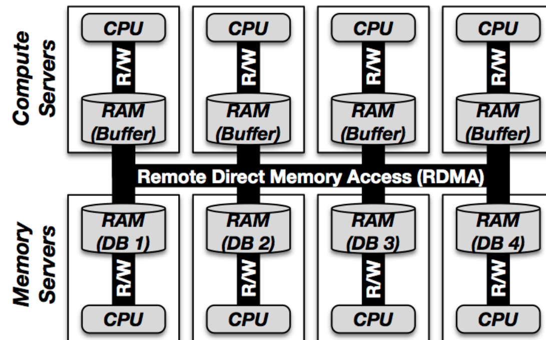


Figure 1: The NAM Architecture

Second, we propose a novel partitioning scheme for the NAM architecture. The main difference to existing partitioning schemes is that tables are split into partitions using multiple partitioning functions (i.e., one per potential join key). Furthermore, even more importantly, partitioning is not used to co-locate data (e.g., for joins) as in a classical shared-nothing architecture. Instead, the partitioning information is used to allow compute servers to efficiently stream over the data in the memory servers at runtime when executing a query.

Third, on top of this partitioning scheme we present an execution model called distributed pipelined query execution for analytical queries on fast networks. The main difference to the existing shuffle-based execution schemes is two-fold: (1) First, in a classical shuffle-based execution scheme intermediate data would need to be re-distributed before each join operator can be executed. In our scheme, we avoid shuffling of intermediates completely. The idea of our model is that we stream data from memory servers to compute servers at runtime while each compute server involved in the query executes a fully pipelined plan without any shuffling operators. In order to achieve this, we partially replicate data when streaming partitions to different compute servers and

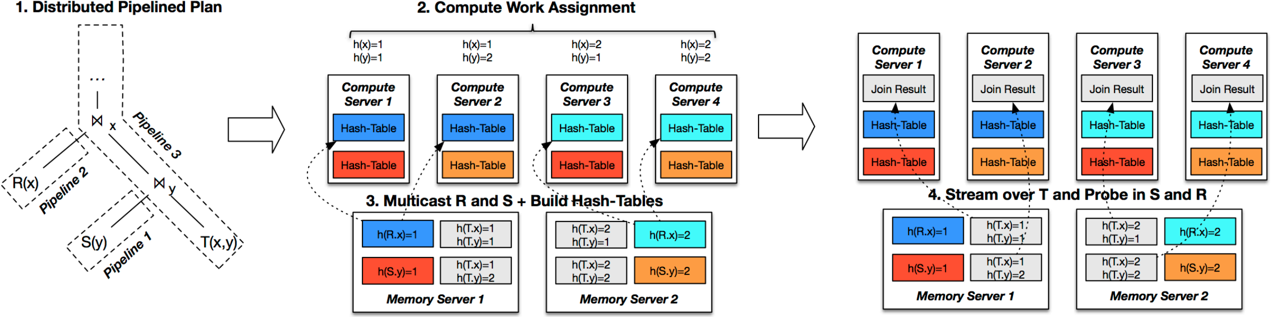


Figure 2: Overview of Distributed Query Execution in *I-Store*

thus trade bandwidth for minimizing the overall network latency of query execution; the main bottleneck in fast networks. (2) Second, our scheme can evenly distribute work across different compute servers and thus efficiently support elasticity and load-balancing.

Fourth, we have implemented all the before-mentioned ideas in a system called *I-Store*. At the moment, in *I-Store* we only support SPJA queries with equi-joins in *I-Store*, an important class of queries often found in OLAP workloads.

Outline: In Section 3, we first give an overview of the main ideas of our query execution scheme as well as how it interacts with the partitioning scheme. Afterwards, in Sections 3 and 4 we discuss the details of partitioning and execution scheme. In our evaluation in Section 5, we then analyze the different design decisions of our execution scheme using microbenchmarks and compare the runtime of our distributed pipelined execution scheme to a shuffle-based execution scheme for queries with a varying number of joins based on the SSB benchmark. Finally, we discuss related work and conclude in Sections 6 and 7.

2 Overview

The distributed query execution model in *I-Store* is based on the notion of a pipelined execution model as it is typically used in single-node parallel databases today, however, adapted for the NAM architecture as outlined before. The main idea is streaming data to multiple compute servers whereas each server executes a query plan in a pipelined manner without any need to shuffle intermediate data between the compute nodes. In order to achieve this, we partially replicate data when streaming data to different compute servers. In consequence, we initially trade some network bandwidth for minimizing the overall network latency; the main bottleneck in fast networks. Moreover, based on the partial replication scheme we can evenly distribute the work to all compute nodes involved in executing a query and thus achieve load balancing as well as elasticity.

Figure 2 gives an overview of all steps required for executing a SQL query that joins three tables using our distributed pipeline model. All steps when executing a distributed pipelined plan are supported by our multi-way partitioning scheme that splits tables by all their join keys using one fixed partitioning function per join key. In the example, we use a hash-based partitioning function (denoted as h): while table R and S are split by one join attribute (either x or y), table T is split by both join keys using two different partitioning functions that are iteratively applied to the table. The result of applying this partitioning scheme to the tables in our example can be seen in Figure 2 (center). Furthermore, partitions are split into smaller fixed sized blocks that enable an efficient load-balancing but can still be efficiently transferred via RDMA.

In the following, we discuss the query execution of the particular SQL query in Figure 2 on top of this partitioning scheme.

In *step 1* (Figure 2, left hand side), the given SQL query is compiled into a so called distributed pipelined plan. The execution of a distributed pipelined plan in parallel using multiple compute nodes is similar to the

parallel execution of a pipelined plan in a single-node using multiple threads. For the query in our example, a single-node pipelined plan would be executed as follows: First, the pipelines over tables R and S would be executed in parallel to build global hash tables that can be seen by all parallel threads. Once the hash tables are built, the last pipeline would be executed by streaming in parallel over table T and probing into the previously built global hash tables for R and S. While the general idea of a distributed pipelined plan is similar, there is an important difference: the pipelines to build and probe do not use a global hash table that can be accessed by any of the compute servers. Instead, in our distributed pipelined model, we partially replicate the data to all compute servers in a streaming manner such that joins can be executed fully locally on each compute server.

After compiling a SQL query into a pipelined plan, in *step 2* (Figure 2, center) we then evenly assign the work to all compute servers that participate in the query execution without yet copying the data over the network. Assigning work to compute servers is only a logical operation and does not involve any data movement yet. In Figure 2, for example, we assign the blocks of different partitions to the four different compute servers required to execute all joins fully locally; e.g., the blocks of the partitions denoted by $h(x) = 1$ and $h(y) = 1$ are assigned to compute server 1 while the blocks of the partition denoted by $h(x) = 1$ and $h(y) = 2$ are assigned to compute server 2 etc. If data is skewed, blocks that belong to the same partition can be assigned to different compute servers. Assigning work evenly to the available compute servers while minimizing replication is a part of our query optimizer to achieve load-balancing.

Based on the assignment, the data of R and S required to build local hash tables is replicated in a streaming manner to the compute servers as shown in *step 3* (Figure 2, center). For example, the partition of R denoted by $h(R.x) = 1$ is replicated to compute server 1 and 2 since both compute servers require this partition to execute the probing pipeline in the subsequent phase when streaming over the assigned T partitions. Furthermore, local hash tables are built while the data is streaming into the compute servers leveraging the fixed-sized blocks as transfer unit to overlap network communication and computation to further reduce network latencies. The result of this phase is shown in Figure 2 (center). It is important to note that in *I-Store* compute servers apply filters when building the hash tables; i.e., we do not push filters into memory servers to avoid that they become a bottleneck. We are also currently working on ideas for distributed indexing where compute servers cache the upper levels of indexes to read only the relevant blocks.

Once the hash tables are built in our example query, in the last *step 4* (Figure 2, right hand side) each compute server streams over the data of the T partitions that it was assigned to in step 2. In our example, compute server 1 fetches the data of the partition of table T denoted by $h(T.x) = 1$ and $h(T.y) = 1$. It is important to note that the compute servers again stream over the data of the partitions of T using the fixed-size blocks and start the probing into the previously created hash tables once the first block is transferred. Moreover, also note that for executing the probing pipeline in a compute server, data can also be replicated in case that T does not contain all join keys.

Finally, if an aggregation is in the query plan and the group-by key is not a join key, then one compute node collects the partial result of all the other compute nodes and applies a post-aggregation over the union of the individual results. This clearly contradicts our goal to not use a shuffle operator in a distributed pipelined plan. However, aggregation results are typically small and do not increase the overall query latency when transferred over the network. Moreover, if scalability of the aggregation is an issue we can also shuffle the data to multiple compute servers based on the group-by keys; i.e., we do not use aggregation trees since they again would increase the network latency.

3 Multiway-Partitioning and Storage Layout

As mentioned before, our partitioning scheme splits the tables of a given database based on all the possible join keys. As a first step in our partitioning scheme, we therefore define one partitioning function for each join key. This function is then used to split a table which contains that join key into n parts. For example, in Figure 2 we

use the function $h(x) = (x\%2) + 1$ to partition all tables that contain the join key x into two parts (which are tables R and T in the example). If a table contains more than one join key, we iteratively apply all the according partitioning functions to this table (e.g., table T contains x and y).

At the moment, we only support hash-based partitioning functions. However, in general we could also use other partitioning functions such as a range-based function as long as the output is deterministic (i.e., round robin is not supported). Moreover, the number of partitions a function creates should best reflect the maximum number of compute servers that are going to be used for query processing. That way, we can efficiently support elasticity by assigning different partitions to distinct compute servers.

Another important aspect, as mentioned before, is that partitions are split into smaller blocks of a fixed size that enable load-balancing but can still be efficiently transferred via RDMA. In our current system, we currently use a fixed size of 2KB for splitting partitions into blocks since this size allows us to leverage the full network-bandwidth in our own cluster with InfiniBand FDR $4\times$ as shown in a previous paper already [3]. The blocks that result after partitioning a table are distributed to the different memory servers. For distributing the blocks to memory servers, co-location does not play any role. Instead, it is only important to equally distribute all blocks of a table to different nodes to avoid network skew when transferring the table data to the compute servers.

Finally, for storing all blocks in the memory servers we use a distributed linked-list structure where each block logically stores a remote pointer to the next block as shown in Figure 3. A remote pointer consists of the node identifier and an offset in the remote memory region of the memory node the next block resides in. The main purpose for using a linked list is that for metadata handling we only need to know the head of the list instead of storing the addresses of all blocks of a table. However, we still want to enable that blocks can be efficiently retrieved by to compute servers using RDMA. In *I-Store*, the preferred access method of a compute server to retrieve blocks from memory servers is to use one-sided RDMA reads in order to not push computation to the memory servers.

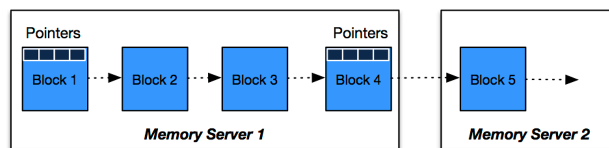


Figure 3: Storage Layout in Memory Servers

In order to avoid remote pointer chasing, which would increase the latency since we need to read one block before we can read the next block, we physically store pointers together in groups. For example, in Figure 3 every fourth block stores pointers for the next four blocks. This storage layout allows us, for example, to efficiently implement remote pre-fetching when streaming over the data of a table in a compute server; i.e., after fetching the first block into a compute server we can fetch the next 4 blocks in the background while streaming over the first block to execute a query pipeline.

4 Distributed Pipelined Query Execution

Based on our partitioning scheme, we can execute queries using our distributed pipelined scheme as outlined already in Section 3 . The two main tasks of this scheme are (1) compute a work assignment for each compute server and (2) based on the assignment execute the pipelines on each compute server by streaming data from memory severs to compute servers. In the following, we discuss the details for both tasks.

4.1 Computing the Work Assignment

The goal of computing the work assignment is that every compute server has to execute approximately the same amount of work to achieve a load-balanced execution. Otherwise, individual compute nodes might run longer and thus have a negative impact on the overall query runtime. In order to enable a balanced work distribution, we implement the following assignment procedure for a given partitioning scheme as shown in Figure 4 (left hand side). The example partitioning is based again on three tables R, S, and T with join attribute x and y as

before. Different from the partitioning used in the first example, the tables R and S are partitioned by a hash partitioning function h which splits the tables into 4 parts on either x or y. Moreover, table S is thus partitioned into 16 parts using the same partitioning functions, however, iteratively applied first partitioning function to x and then splitting the output using the partitioning function on y.

Based on the partitioned data and a given pipelined plan, we compute the assignment as follows:

Assume that we want to execute the same query as shown already in Figure 2 (left hand side) over the partitioning given by Figure 4 (left hand side). First, for every possible combination of partitions that need to be shipped to a compute node together to enable a fully local pipelined execution, we estimate the execution cost. For example, the partition of R with $(h(x) = 1$ (blue partition 1), the partition , $h(y) = 1)$ of S (red partition 1) as well as the partition $(h(x) = 1, h(y) = 1)$ of T (grey partition 1.1) must be shipped to the same compute server the join query can be executed fully locally on that compute node. At the moment, we leverage a very simple cost function to compute the execution cost for a set of partitions, which only reflects the total size of the partitions that need to be transferred together to a compute node. In future, we plan to include not only the data transfer cost but also the cost of executing the pipelined query plan over the transferred partitions.

Second, based on the computed costs we then assign the partitions to a given number of compute nodes such that every compute node gets approximately the same amount of work. In case all partitions have the same size (i.e., if data is distributed in a uniform way), the assignment becomes trivial as shown in Figure 4 (right hand side). However, computing the assignment under data skew where individual partitions have different sizes becomes more complex. Actually, computing the work assignment under data skew into a given number of compute servers essentially is a k-partitioning problem where k is the number of compute servers and the costs per partition combination represent the numbers in the set that needs to be divided equally. Since this problem is NP-hard, we plan to leverage approximate solutions based on heuristics (e.g., the largest differencing method) to solve the assignment problem. Furthermore, since partitions are further split into smaller fixed size blocks, we also plan to use these blocks to readjust the assignment to achieve a fairer distribution under heavy skew after applying the heuristics.

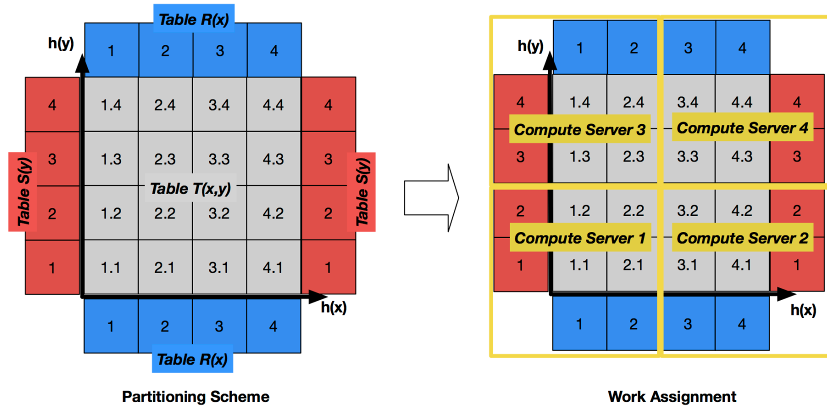


Figure 4: Computing the Work Assignment

4.2 Streaming Data to Compute Servers

Once the work assignment is computed, each compute server executes the pipelines in a query plan over the partitions it was assigned to. An example of a pipelined plan was already shown in Figure 2 (left hand side). For executing the pipelines in the plans, we transfer the partitions to the compute servers in a streaming manner such that pipelines can be executed in compute servers while data is streaming in from memory servers.

The methods we use for streaming partitions to compute servers can be differentiated into two cases: (1) Partitions are streamed to multiple compute servers, (2) Partitions are streamed to one compute server. For example, to execute the pipelines 1 and 2 in Figure 2 (center), each partition of R and S is streamed to two of the four compute servers. However, for the partitions of T, no replication is required to execute pipeline 3. In order to implement these two cases in a streaming manner, we using different techniques as explained next.

Streaming to one Compute Server: Streaming data from memory servers to one compute servers (i.e.,

without replication) is implemented in *I-Store* using one-sided RDMA reads that are issued by the compute server. For example in Figure 2 (right hand side) the compute server 1 issues RDMA reads to retrieve the data for partition 1.1. The main reason is that the CPU of the memory server is not involved in the data transfer and thus does not become a bottleneck when we scale-out the compute servers to support elasticity. The advantage of using one-sided operations to scale-out computation in the NAM architecture has been shown in a previous paper already [3]. Streaming is implemented on a block basis; i.e., each block is retrieved by a compute server using a separate RDMA read.

Furthermore, we additionally implement pre-fetching to further reduce the latency of retrieving each block as shown in Figure 5. We therefore reserve a read buffer for n blocks and leverage the queue-based programming model of RDMA where a read request can be separated from the notification of its completion. The main idea of pre-fetching is that a compute server requests the NIC to read the next n blocks without waiting for the completion event (CE) for the RDMA reads. Instead, the compute server only polls for completion event (CE) in the completion queue of the NIC when it needs to process the next block for executing a query pipeline (e.g., to apply a filter predicate on all tuples in the block and add them to a hash table). While the compute server is executing the query pipeline for the current block, the NIC can process further outstanding read requests in the background. Moreover, once a block is processed by the query pipeline executed by a compute server, the compute server can request the NIC to read the next block into the free slot in the read buffer again without waiting for its completion. For more details about the queue-based programming model supported by RDMA, we refer to [3].

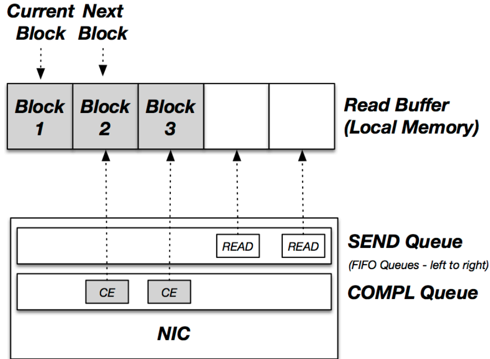


Figure 5: Prefetching using RDMA Reads

Streaming to Multiple Compute Servers: Streaming data to multiple compute server can be implemented using the same technique as discussed before by multiple compute servers. For example, in Figure 2 (center) the compute server 1 and 3 need to read the the same partitions of table R that is denoted by $h(R.x) = 1$ (red). To that end, compute server 1 and 3 can use the same technique as discussed before to stream the partition independently to both compute servers using RDMA reads. However, when both compute servers read the same partition from memory server 1, the available bandwidth on the memory server drops by half for each compute server as shown in Figure 6 (left hand side). The reason is that the same blocks of the partition $h(R.x) = 1$ (red) needs to be transferred from memory server 1 to both compute servers. This gets even worse, if we further scale-out the number of compute servers and thus need to replicate the same partition to more compute servers.

To that end, we use RDMA multicast operations in *I-Store* to stream partitions if they are required by multiple compute servers. The advantage when using multicast operation is that the memory server only needs to transfer the same blocks once to the switch which then replicates the blocks to all compute servers in the multicast-group. The only disadvantage is that multicast operations in RDMA are two-sided (i.e., the memory server's CPU is actively involved in the data transfer). However, when using multicast operations, each block of a partition only needs to be send once independent of

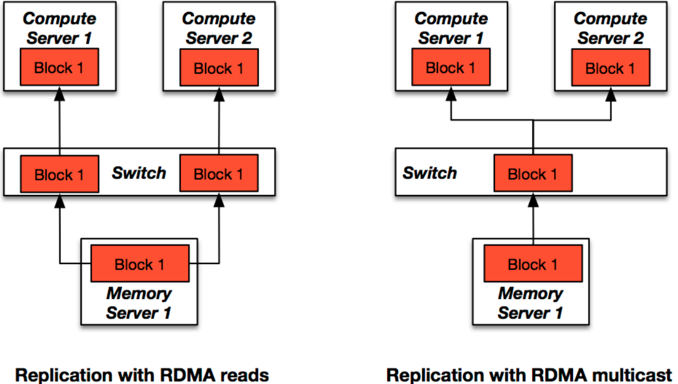


Figure 6: RDMA Read vs. Multicast

the number of compute servers which receive the block as shown in Figure 6 (right hand side). Thus, using two-sided multicast operations does not limit the scalability of our processing model. Finally, it is important to note that for multicast operations, we can implement a similar prefetching technique as described before for RDMA reads. The only difference is that both sides (compute and memory servers) must be involved in the prefetching implementation; i.e., a memory server needs to register multiple RDMA send operations at its NIC while the compute server needs to register multiple RDMA receive operations at its NIC.

5 Experimental Evaluation

In this section, we present the results of our initial evaluation of our distributed query execution engine implemented in *I-Store*. For the evaluation we executed different experiments: (1) In a first experiment, we show the end-to-end performance when running analytical queries. The main goal of this experiment is to compare our distributed pipelined model to a classical shuffle-based model. (2) In a second experiment, we discuss the results of different microbenchmarks that we executed to validate our design decisions (e.g., using RDMA multicast operations vs. reads).

For executing all the experiments, we used a cluster with 8 machines connected to a single InfiniBand FDR 4X switch using a Mellanox Connect-IB card. Each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256GB RAM. The machines run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as their operating system and use the Mellanox OFED 2.3.1 driver for the network.

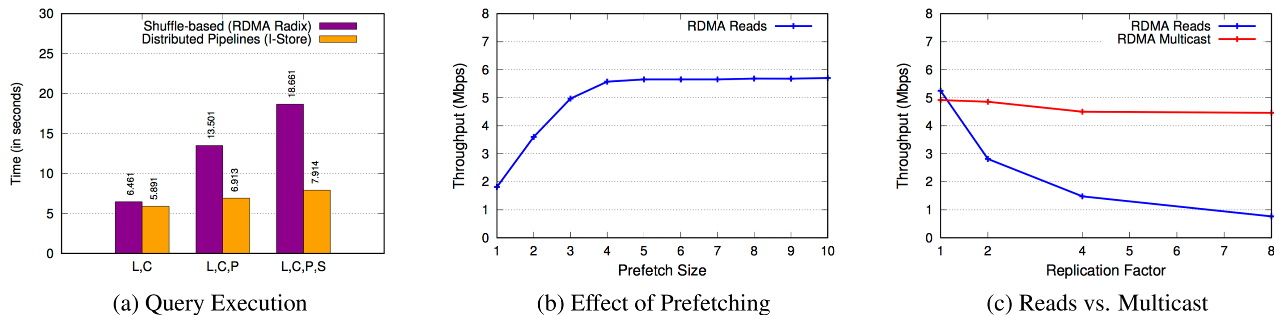
5.1 Exp. 1: System Evaluation

As data set, we used the schema and data generator of the Star Schema Benchmark (SSB) [10] and created a database of $SF = 100$. Afterwards, we partitioned the database using our partitioning scheme as follows: The individual dimension tables (`Customer`, `Parts`, `Supplier`) were hash-partitioned into 8 parts using their primary keys (which is the join key). We did not include the `Date` table into our queries since it is small and thus could easily be replicated at runtime entirely to all compute nodes. The fact table was partitioned into into $8^3 = 256$ partitions using the join keys of the three before-mentioned dimension tables.

As a workload, we executed queries with a different number of joins. We started with a query that joins only two tables; the fact table and the `Customer` (C) table. Afterward, we subsequently added one more of the dimension tables (i.e., `Parts` (P), `Supplier` (S)) to the query. For executing the queries in *I-Store*, we used 8 memory and 8 compute servers (i.e., one memory and compute server were co-located on one physical node) and used 10 threads per compute server for query execution.

In order to compare our approach to a shuffle-based execution model that needs to re-distribute the inputs for every join operator, we used the code that is available for the RDMA radix join as presented in [2]. The RDMA radix join shuffles the data in the first partitioning phase over the network. Since the code is designed to join only two tables, we simulated queries with more than two tables by running the algorithm multiple times on different input tables without including to the cost of materializing the join output. For executing our workload using the RDMA radix join, we partitioned all tables into 8 partitions using a hash-based scheme and distributed the partitions to all 8 nodes in our cluster. Furthermore, we configured the join algorithm to use 10 threads per node as for our join. For loading the data, we modified the code of the RDMA radix join to use the key distribution of the before-mentioned partitioned SSB tables and the tuple-width as specified by the schema of the SSB benchmark [10].

The result can be seen in Figure 7a. We see that both approaches have a similar runtime for a binary join query. The reason is that both approaches need to transfer approximately the same amount of data over the network. However, since we stream data to compute nodes and overlap the pipeline execution with the data transfer, we can achieve a slightly better performance. Furthermore, for queries which join 3 or 4 tables our



(a) Query Execution (b) Effect of Prefetching (c) Reads vs. Multicast

Figure 7: Experimental Evaluation of *I-Store*

approach shows a clear benefit: While the RDMA radix join needs to shuffle data for every join, we can fully stream the pipelines without any further shuffling which results in a performance gain of $3\times$ in the best case.

In this experiment, only one of the joined tables in a query was large while all other tables where relatively small. We believe that this is a common in most data warehousing scenarios with fact and dimension tables. In future, we plan to also look into large-to-large table joins.

5.2 Exp. 2: Microbenchmarks

In this experiment we validated the efficiency of our decisions for streaming data from memory servers to compute servers in *I-Store* to enable a pipelined query execution in the compute servers.

Exp. 2a - Streaming to one Compute Server: First, we analyzed the approach when streaming data to one compute server (i.e., blocks do not need to be replicated) and compared a streaming approach using a varying number of prefetched blocks starting with only 1 block (no pre-fetching) up to 10 blocks. As block size we used $2KB$ and the tuple width was set to $8B$ (i.e., one large integer). Moreover, on top of the incoming data stream we executed a simple scan pipeline that reads each incoming block and computes a sum operator over the $8B$ integer values for all tuples in the block. The results can be seen in Figure 7b where the y-axis shows the number of scanned blocks per second (million blocks per second, Mbps) for different pre-fetch sizes. We can see that compared to no pre-fetching (size=1) we can gain a speed-up of up to factor $3\times$ using pre-fetching (size=10). Moreover, pre-fetching more than 10 blocks also does not seem to further increase the throughput in that case. In fact, even when pre-fetching only 5 blocks we already reach almost the maximum throughput in this experiment.

Exp. 2b - Streaming to multiple Compute Servers: Second, we also analyzed the effect of using RDMA reads vs. RDMA multicasts to stream blocks to multiple servers that require replication. We varied the replication factor from 1 to 8 to show the effect of streaming blocks from 1 memory server to a maximum number of 8 compute servers. For simulating query pipelines, we used the same setup as before: As block size we used $2KB$ and the tuple width was $8B$ (i.e., one large integer). Moreover, on top of the incoming data stream we executed the same simple scan pipeline as before. The results can be seen in Figure 7c: the x-axis shows the number of receiving compute servers and the y-axis shows the average number of scanned blocks (million blocks per second, Mbps) per compute server. We can see that when using RDMA reads the throughput in a compute server drops linearly from $5m$ to $0.8m$ when increasing the number of compute servers involved. However, when using RDMA multicasts the throughput remains constant at around $4m$ which is a little lower in case no replication is required but significantly higher when replication is required (from 2 to 8 compute nodes in this experiment).

6 Related Work

In this paper, we made the case that networks with RDMA capabilities should directly influence the architecture and algorithms of distributed DBMSs. Many projects in both academia and industry have attempted to add RDMA as an afterthought to an existing DBMS [13, 11, 9, 5, 6]. Some recent work has investigated building RDMA-aware DBMSs [14, 15] on top of RDMA-enabled key/value stores [7], but again query processing in these systems comes as an afterthought instead of first-class design considerations.

Furthermore, the proposed ideas for RDMA build upon the huge amount of work on distributed processing (see [8] for an overview). Similar to our work, other work [12, 2, 1] also discussed distributed query processing for RDMA but focused only on the redesign of individual algorithms without rethinking the overall query processing architecture. For example, [12, 2] implements RDMA-based versions of two existing join algorithms (an RDMA radix join as well as an RDMA sort merge join). Furthermore, [12] adapts the shuffle operator of a classical shared-nothing DBMS but includes some interesting ideas of how to handle data skew and achieve load-balancing using this operator. Finally, SpinningJoins [4] also make use of RDMA. However, this work assumes severely limited network bandwidth (only 1.25GB/s) and therefore streams one relation across all the nodes (similar to a block-nested loop join).

7 Conclusions

In this paper we revisited query execution for distributed database systems on fast networks. While recent work has already started to investigate how to redesign individual distributed query operators to best leverage RDMA, in this paper we reconsidered all aspects from the overall database architecture, over the partitioning scheme, to the query execution model. Our initial experiments showed that in the best case our prototype system provides $3\times$ speed-up over a shuffle-based execution model that was optimized for RDMA. In future, we plan to investigate aspects such as elasticity as well as efficient load-balancing in more depth since these properties are natural to our execution model.

References

[1] C. Barthels, G. Alonso, T. Hoefler, T. Schneider, and I. Müller. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[2] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1463–1475, 2015.

[3] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[4] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 283–292, 2010.

[5] N. S. Islam, M. Wasi-ur-Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of HDFS over infiniband. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 35, 2012.

[6] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance RDMA capable interconnects. In *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*, pages 743–752, 2011.

[7] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306, 2014.

[8] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[9] X. Lu, N. S. Islam, M. Wasi-ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop RPC with RDMA over infiniband. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 641–650, 2013.

[10] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, pages 237–252, 2009.

[11] A. Pruscino. Oracle RAC: architecture and performance. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 635, 2003.

[12] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205, 2016.

[13] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[14] C. Tinnefeld, D. Kossmann, J. Böse, and H. Plattner. Parallel join executions in ramcloud. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 182–190, 2014.

[15] C. Tinnefeld, D. Kossmann, M. Grund, J. Boese, F. Renkes, V. Sikka, and H. Plattner. Elastic online analytical processing on ramcloud. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 454–464, 2013.