DQSim: Does the Task Assignment Slow Down Your Distributed Database System?

Maximilian Rieger
Technical University of Munich
Munich, Germany
max.rieger@tum.de

Thomas Neumann
Technical University of Munich
Munich, Germany
neumann@in.tum.de

Abstract

In distributed OLAP databases, assigning data and tasks to compute nodes is a non-trivial problem with large performance impact. Common strategies include simple round-robin schedulers or minimizing network transfers. However, in some cases these techniques cannot deliver optimal query latency. For example, in scale-out scenarios, existing techniques fail to balance the trade-off between utilization of new nodes and increased network cost from cold caches, leaving substantial room for improvement.

Addressing these problems in existing systems is difficult, because schedulers are often tightly integrated and cannot be easily adapted for experiments. Evaluating a new scheduler in a system is equally difficult, requiring time-consuming and costly experiments for many scenarios.

To tackle this problem, we formalize the distributed task scheduling problem (DTS). Further, we present DQSim, a fine-grained simulator for distributed query execution that estimates the execution time of DTS schedules. This allows us to accurately model the effects of new scheduling algorithms without having to rebuild large parts of a system. DQSim can run thousands of experiments with user-defined system and cluster configurations and queries in seconds to evaluate scheduling techniques quickly.

Finally, we propose two new scheduling algorithms, NetHEFT and cCEFT, that yield good results across a wide range of scenarios. They are on par with round-robin and network-minimizing baselines on simple scenarios where no improvement is expected. In more challenging scenarios, they can reduce query latency by over 2x.

CCS Concepts

• **Information systems** → **Database query processing**.

Keywords

Distributed Database Systems, Scheduling, Query Optimization

ACM Reference Format:
Maximilian Rieger and Thomas Neumann. 2027. DQSim: Does the Task Assignment Slow Down Your Distributed Database System?. In *Proceedings of the 2027 International Conference on Management of Data (SIGMOD'27).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Maximilian Rieger, Technical University of Munich, Munich, Germany, max.rieger@tum.de; Thomas Neumann, Technical University of Munich, Munich, Germany, neumann@in.tum.de.

SIGMOD'27, Huntington Beach, CA, USA

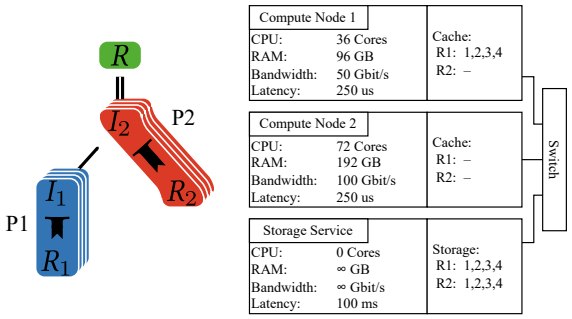

Figure 1: A distributed query and the state of a cluster.

1 Introduction

Motivational Example. Analytical systems are often confronted with drastically different workloads. They should process huge datasets with high performance while being cost-efficient for light workloads. Imagine a system that automatically scales out to new compute nodes when large queries with high expected execution times arrive. Figure 1 shows such a scenario, where one new compute node is added. Now, the distributed execution of the query should be performed as quickly as possible. To do so, the system needs to make a difficult decision: Where to execute the individual parts of the query? The new compute node 2 has more CPU cores and a better network connection. However, node 1 already contains relation 1 in its cache. Further, pipeline 2 depends on pipeline 1 and corresponding partitions should be co-located to avoid unnecessary data transfers. Placing each part of the query on the best compute nodes can have a significant impact on execution performance. This work analyzes the performance implications of such decisions using DQSim, a fine-grained simulator.

Distributed Database Systems. Shared-nothing architectures like SQL Server PDW hash-partitions data over all compute nodes in the cluster [41]. In these systems, the task assignment cannot be changed. In contrast, cloud-native systems like Microsoft Polaris enable a more flexible assignment of data and tasks to compute nodes [2]. More flexible assignments enable improvements in resource utilization by co-locating complementary workloads and dynamically resizing clusters [38]. Further, Polaris implements *state separation* to make execution more robust to node failures. DQSim's execution model is inspired by Polaris and can also represent state separation as discussed in Section 5.

Networks Are Fast Now. In the past, networks were significantly slower than local computation, so it made sense to simply minimize

the data transferred over the network [41]. Modern hardware, however, offers much faster networking. The time it takes to send a tuple of data over the network is now in the same order of magnitude as performing a hash probe [10]. In this work, we present detailed cost models and algorithms to balance the trade-off between local computation and network transfers.

1.1 Content of This Work

DQSim. With this paper, we present and open-source DQSim, a distributed query execution simulator.[1] DQSim is built on the abstraction provided by our formalized distributed task scheduling problem (DTS) as defined below in Section 2. This abstraction enables a system-independent analysis of task assignment to compute nodes in distributed databases. To accurately represent real systems, DQSim considers compute resources and network capabilities of all nodes in a cluster. It simulates the execution and network transfers of each partition individually. We provide a detailed description of DQSim in Section 3 and experimentally validate DQSim's accuracy in Section 6.

Performance Analysis. DQSim enables us to conduct a wide range of experiments varying query, compute node, network, and storage service characteristics. For example, we quantify the cost of state separation by uploading intermediate results to a storage service in Section 5. We also use it to analyze the performance impact of different scheduling strategies in Section 8. By doing so, we identify for which cases simple schedulers already yield good results and for which cases new techniques can improve performance.

Scheduling Algorithms. To show how performance can be improved, we implement and compare six different scheduling algorithms. Our results show that simple strategies work very well in simple cases. However, they fail to find good schedules when applied to heterogeneous clusters or scale-out scenarios. Queries with growing intermediate results magnify the performance impact of bad schedules. We also demonstrate that previous work on heterogeneous scheduling does not work directly for the setting we have in databases, but can be adapted to it [44]. Finally, we present two scheduling algorithms that work well across a wide variety of problems in Section 7.

In summary, we make the following contributions:

- **Simulator:** We present DQSim, a fine-grained simulator that can be used to simulate the distributed execution of queries.
- **Analysis:** We conduct a detailed analysis of scheduling techniques and show the effectiveness of current approaches as well as optimization opportunities.
- **Algorithms:** We present the scheduling algorithms NetHEFT and cCEFT for distributed databases that yield consistently good results compared to all baselines.

2 Distributed Task Scheduling Problem (DTS)

To enable systematic analysis of task assignment, we formalize the distributed task scheduling problem (DTS). First, we define the system architecture for which we optimize the task assignment.

Table 1: Distributed version of the TPC-H schema

Relation	Partition Layout	Partitioning Key	nPartitions
lineitem	Hash-partitioned	`l_orderkey`	64
orders	Hash-partitioned	`o_orderkey`	64
customer	Hash-partitioned	`c_custkey`	64
partsupp	Hash-partitioned	`ps_partkey`	64
part	Hash-partitioned	`p_partkey`	64
supplier	Hash-partitioned	`s_suppkey`	64
nation	Broadcast	—	1
region	Broadcast	—	1

2.1 System Architecture

Schema. We consider a relational OLAP system that stores its base relations on a storage service. Relations can be hash-partitioned by one of their attributes into an arbitrary but fixed number of partitions. The respective partitioning of relations is part of the schema. We will use sensible choices for schemata throughout this work that will favor query execution, such as partitioning by likely join keys, partitioning all tables larger than some threshold, and using the same partition count for all partitioned relations. Partitions can be further divided into blocks which are stored as objects in the storage service. This can be used for pruning using a second partitioning dimension like Polaris [2] or using a sparse index like ClickHouse [40]. Table 1 shows a possible schema for TPC-H that works for all scale factors.

Cluster and Caches. For execution, the system may use an arbitrary number of compute nodes. Compute nodes may differ in CPU cores, memory size, and network speed and latency. Each node can cache any partition from any relation, if it fits in its memory. This allows two nodes to cache the same data and some data to not be cached anywhere. An example of a cluster is given in Figure 1.

Storage Service. In addition to the cache, each partition is stored in the storage service. We use the storage service to represent the decoupled storage as it can be found in many modern query engines [2, 8, 16]. The storage service can be nicely represented by an additional node in the cluster that has 0 CPU cores and a significantly higher bandwidth than the compute nodes. Further, we increase the latency to resemble real conditions in current cloud environments [17].

Execution. When a query is executed, compute nodes work concurrently on their respective tasks. Nodes can start working on a task as soon as all required input data for the task is present. Computation and network transfers are executed concurrently. Data can be transferred between nodes directly using P2P communication or by up- and downloading it to the storage service.

2.2 Task Model

Data Unit. We generalize base relations which are part of the schema, materialized intermediate results, and the final result of a query to data units (DUs). In Figure 1, R_1, R_2, I_1, I_2, and R are all DUs of the query. Data units can be partitioned in the following ways:

[1] github.com/MaxRieger96/dqsim

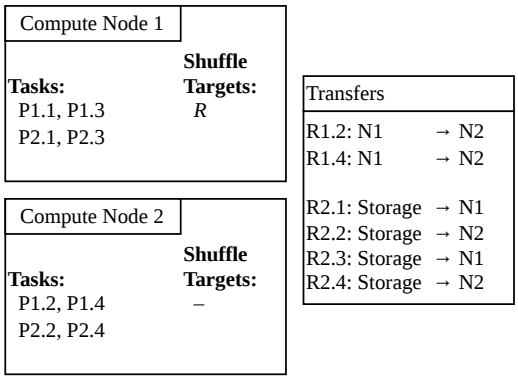

Figure 2: Example of a possible assignment of tasks to nodes

- **Hash-partitioned**: The data unit is hash partitioned by a key.
- **Broadcast**: All data resides in a single partition that is broadcast to all eligible nodes.
- **Single-node**: All data resides on one node, further processing will not be distributed.
- **Scattered**: Data is partitioned without a key.

Partitions are identified by their DU and their individual partition index. Additionally, each DU has a fixed size in bytes. If the DU is partitioned, all partitions are set to the same size by default. Partition sizes can also be configured to represent skew.

Pipeline. We decompose queries into pipelines which represent the fused computation of operators that is not interrupted by data materialization or shuffles [31]. A pipeline always scans one data unit (DU) and returns another DU as a result. Additionally, a pipeline can depend on further DUs. For example, P2 in Figure 1 that performs a hash-join will scan the probe side DU (R_2) but will depend on the build side DU (I_1).

Task. For distributed execution, we split each pipeline into tasks. A task performs the computation of a pipeline for one input partition of the scanned DU. Consequently, it also returns only a partition of the output DU and only depends on one partition of the required DUs. For example, both pipelines in Figure 1 are split into four tasks each.

Shuffle. DUs have to be partitioned on the correct attribute in some cases to enable correct distributed execution. For example, to perform a hash distributed join, both inputs of the join have to be partitioned on the respective join key. To change the partition key of a DU, we introduce shuffles. A shuffle takes one DU as input and returns a new DU containing the same data but with a different partitioning. For example, the hash-partitioned I_2 in Figure 1 is shuffled to a single node so it can be returned as a result.

Shuffle Targets. When the output of a shuffle is partitioned, nodes need to know where to send each tuple after re-partitioning. A shuffle target describes the location of each shuffle output partition.

Transfer. Sometimes a consumer-task is executed on a different compute node than the producer-task, which creates its input. In this case, the data needs to be transferred from one node to another. We call this a transfer, which consists of the respective partition to be transferred, a source node that should send the partition, and a target node that receives it. Data can also be transferred between the storage service and a compute node.

Query. In summary, a query $Q = (D, P, S)$ consists of a set of data units D, a set of pipelines P, and a set of shuffles S.

2.3 Task and Data Assignment

Given the system architecture above, we define the distributed task scheduling problem (DTS) as follows: Given a query with its task set T and a cluster C consisting of the node set N, find an assignment $\mathcal{A} : T \to N$ that minimizes the makespan (the execution time) of the schedule. Further, we require the solution to specify a set of transfers $\mathcal{T}_r : PR \to \mathcal{P}(N \times N)$ that specifies which nodes send and receive partitions (PR) from each other. Finally, a solution to the DTS problem includes shuffle targets $\mathcal{T}_s : PR^{sh} \to N$ that specify which nodes shall receive each partition of a data unit that is created by a shuffle. Tasks depend on their input data, so the schedule needs to ensure that all input data of a task is present on the assigned node. This effectively forms a task DAG, where a task only starts after all of its input data becomes available. Figure 2 shows an example of a possible solution to the DTS problem using round robin assignment. In the following section, we describe how DQSim computes the execution time of an assignment. This is an extended version of the scheduling problem that Ullman proved to be NP-hard over 50 years ago [45]. Ullman's scheduling problem optimizes the execution time of a task DAG where tasks can be processed on any node without any communication. Ullman's scheduling problem can be reduced to the DTS problem by setting all network speeds to infinity, latencies to 0 and using only nodes of the same type. Consequently, the DTS problem is NP-hard as well.

3 DQSim

The formal problem description above defines the possible scheduling decisions a system can make. In this section, we show how a schedule can be evaluated.

Experiments on Distributed Systems Are Difficult. The effects of scheduling decisions can be complicated and hard to analyze. One needs to apply a scheduling algorithm to a variety of scenarios, including different queries, cluster sizes, and compute and networking configurations. Conducting such experiments on real hardware can be very expensive and time-consuming. Further, the effort would not be transferable to other database systems, making it difficult to evaluate schedulers with different system performance characteristics.

Simulation Is Fast. We approach this problem by simulating the distributed execution of queries. DQSim deliberately abstracts away many low-level execution effects and simplifies aspects like congestion to provide a tractable model for scheduling effects. Using our flexible and generic simulation has several advantages. First, experiments that would take hours can be conducted within seconds. This allows us to evaluate new ideas much faster. Second, thanks to its flexibility, we can easily adapt to new scenarios. We can resize clusters to thousands of nodes, change network throughput and latency and add new components like a storage service or caching layer. Third, we can use different performance characteristics for individual queries. One can obtain query plans and task/pipeline execution times from traces of real systems to tailor scenarios for

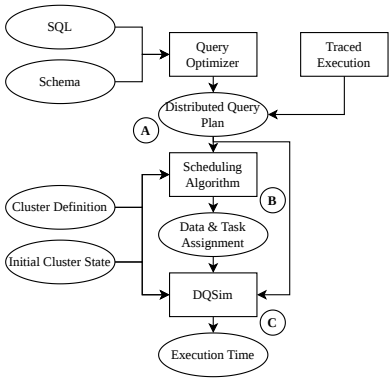

Figure 3: Overview of the simulation workflow.

```
def run_simulation(plan):
  Jr, Jp = {}, plan.all_jobs()
  t = 0
  while !Jr.empty() or !Jp.empty():
    ready_jobs = find_ready_jobs()
    Jp -= ready_jobs
    Jr += ready_jobs
    dt = min(remaining_job_times(Jr))
    t += dt
    finished_jobs = progress_jobs(Jr, dt)
    store_outputs(finished_jobs)
    Jr -= finished_jobs
  return t
```

Listing 1: Simulation loop

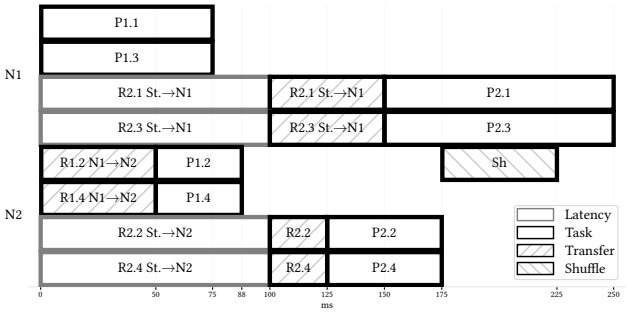

Figure 4: Simulation timeline for running example.

specific systems and use-cases. For faster experiments, one can also leverage recent work on cost models to estimate the execution times of individual pipelines [24, 34, 43, 52]. Finally, conducting experiments in the simulator does not cost money beyond running a small machine, making planning optimizations a viable research topic for anyone.

3.1 Simulator Overview.

Figure 3 depicts how DQSim can be used. First, we need to obtain query plans Ⓐ, either from execution traces or from the query optimizer of a system. Second, a scheduling algorithm Ⓑ can assign tasks and data to nodes for each query. These assignments are schedules. Scheduling algorithms also need access to the cluster definition and current state of the cluster cache. Next, DQSim Ⓒ can simulate the schedule with the same cluster information and the original distributed query. Finally, DQSim returns the simulated execution time.

Simulation State. The simulation closely follows the task model as described in Section 2.2. For the simulation it maintains the state of **tasks**, **transfers**, and **shuffle operations**. We will call tasks, transfers and shuffle operations *jobs*. There are two sets for each kind of job: pending and running. Before the simulation starts, all jobs are added to their respective pending set. Then, in each step, we check which jobs can be started, because their input data is available at the respective node. A job that can be started is removed from the pending set and added to the running set. Further, DQSim tracks the execution time of each step and the used memory for each node. Step by step, it transitions jobs from pending to running and removes finished jobs in a loop, advancing the time in each step.

Simulation Loop. As outlined in Listing 1, the simulation loop begins by finding ready jobs. Here we check all pending jobs and add them to the running tasks if the input data they require is present on the node that should execute the job. Then, we compute the remaining time each job will take until it finishes. This works slightly differently for each type of job. We will cover the details in the following paragraphs. Now we compute the time until the first job will finish, i.e. the minimum of the remaining times. We use this time as our step time t_s. As all running computations and network transfers / shuffles will progress with constant speed until at least one job finishes, we can now advance our simulation by t_s. We store the progress of each task T_i at step s as $p(T_i)^{(s)} \in [0, 1]$. For all tasks that do not finish, we add the progress $\Delta p(T_i)^{(s)}$ they made in step s.

$$p(T_i)^{(s+1)} = p(T_i)^{(s)} + \Delta p(T_i)^{(s)} \tag{1}$$

We can compute this number by checking which percentage of the remaining job time $t_r(T_i)$ is covered by the step time t_s:

$$\Delta p(T_i)^{(s)} = (1 - p(T_i)^{(s)}) \cdot \frac{t_s}{t_r(T_i)} \tag{2}$$

For all jobs that finish, we remove them from the running set and make their output data available on the respective node. With this newly available data, the simulator will be able to find new jobs that can transition from pending to running in the next step. The following paragraphs explain in greater detail how job latencies can be computed.

Example. Figure 4 shows how the query from Figure 1 is simulated with the assignment from Figure 2. The whole simulation is performed in 9 steps. At time 0 several jobs start concurrently: (1) Node 1 runs the tasks P1.1 and P1.3, which will take 75 ms. (2) Node 1 sends the partitions $R_1.2$ and $R_1.4$ to node 2 which will take 50 ms. For simplicity, latency between compute nodes is omitted in this example. (3) Both nodes load partitions from R_2 from the storage node. First, they need to wait for the storage latency of 100ms. The shortest jobs are the transfers of $R_1.2$ and $R_1.4$, so the first step ends after 50 ms. After this step, the remaining latency for

the storage transfers is updated, and the progress of tasks P1.1 and P1.3 is advanced to 66.7%.

3.2 Job Times

Task Times. Tasks are annotated with an expected execution time on a reference machine. The total execution time $t_t(T_i, N_j)$ on the respective node N_j can be computed by simply comparing the core count of the selected node with the reference machine.

$$t_r(T_i) = (1 - p(T_i)^{(s)}) \cdot t_t(T_i, N_j) \cdot n_{\text{tasks}}(N_j) \quad (3)$$

Then, we count the number of tasks $n_{\text{tasks}}(N_j)$ the node is currently executing. We multiply the total execution time of each task with the number of currently running tasks on the respective machine. Then, we compute the currently remaining execution time $t_r(T_i)$ by multiplying this task time by the remaining fraction of the task $(1 - p(T_i)^{(s)})$. For example, in Figure 4 a task from pipeline 1 takes 37.5 ms on node 1. As node 1 runs P1.1 and P1.3 concurrently, it needs 75 ms to complete them. Node 2 has twice as many cores, so it completes P1.2 and P1.4 concurrently in 37.5 ms. While this model simplifies the effects of congestion, it maintains the overall required execution time of all tasks. This model also assumes that performance of all tasks can scale linearly with the number of cores. Since many algorithms show diminishing returns at large core counts, our implementation also supports an optional scaling model in which the marginal speedup from additional cores decays beyond a configurable threshold. Note that the maximum number of tasks executed in parallel on a single machine can be limited to any number ≥ 1.

Transfer Times. A transfer sends a partition from one source node to one target node. To compute the time it takes to transfer data from source to target, we consider two factors: network latency and network throughput. Network latencies can contribute significantly to execution times, especially when accessing slow storage services [17]. However, they do not have a large effect on the network throughput of concurrent transfers. We approximate latencies by simply using the provided network latencies of the two nodes involved in the transfer. Then, we delay the data transfer by the maximum of the two values. Only after the latency time has passed, we simulate any data flowing between the nodes. To fit this in the step model of our simulator, we split each transfer into two jobs. Once the latency job has finished, we run the actual data transfer job. For example, at the beginning of Figure 4, the transfers from the storage node wait 100 ms before the actual data transfer starts. To compute the time for the transfer job, we need an individual throughput in bytes per second for each network transfer. A detailed description of how this throughput can be computed is given below. Using the throughput and the size of the partition transferred, we can compute the total transfer time. Updating the progress works analogously to tasks.

Shuffle Operations. Shuffles are more complex than transfers. First, they describe the transfer of one data unit to another, where each data unit can consist of multiple partitions. Second, they can involve all compute nodes in the cluster, as each of them might send source partitions and receive target partitions. To make this more manageable, we decompose a single shuffle into many shuffle operations. A shuffle operation reads a whole partition of the input DU of the shuffle but only outputs a partial partition. For example, in Figure 4, all partitions of I_2 are shuffled to a single partition of R. After finishing P2.2 and P2.4, node 2 has two partitions of I_2. When their data is shuffled to the shuffle target (node 1), the partition $R.1$ is only 50% complete. Only after node 1 finishes P2.1 and P2.3 and obtains $I_2.1$ and $I_2.3$, the shuffle output partition $R.1$ is complete and becomes available on node 1. We track the progress of collecting each shuffle output partition received at the shuffle targets again by percentage. So when a shuffle operation finishes, it updates the progress of the collected shuffle partition. If this increases the progress of the collected shuffle to 100%, the partition is made available at the target node. This breakdown allows us to treat a shuffle operation the same way as a network transfer to compute its transfer time and progress.

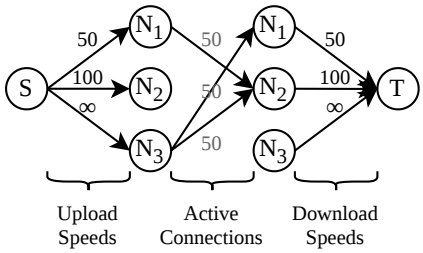

Figure 5: Modeling the network throughput in Gbit/s of data streams as a max-flow problem.

3.3 Network Simulation

Network Throughput As Max-Flow. While we can use execution traces to define the local execution time of pipelines, we have to fully simulate the network. Currently, DQSim only supports a fully switched network topology. Here, each node has its own upload and download speed, as well as a latency. As there can be many concurrent data streams between compute nodes, they incur congestion on the network bandwidth. There is at most a single data stream between a pair of nodes. If multiple jobs transfer data between the same pair of nodes, they have to share the bandwidth. Still, computing the available bandwidth for all streams is non-trivial. Figure 5 shows how this problem can be modeled as a max-flow problem [19]. There are two groups of nodes representing each compute node in the graph. The flow first needs to pass the respective upload speeds in Gbit/s of all source nodes represented by the edges from the source nodes to the first group of compute nodes. Next, it can only take the connections that have open streams, represented by edges that connect the respective nodes in both groups. Finally, the download speeds of nodes are represented by edges connecting the second group of nodes to the target nodes. Now we can use any max-flow algorithm to compute the bandwidths. The edge weights of the active connections between the two sets of nodes represent the available bandwidth between the respective nodes. The data stream throughput is divided equally for all transfers that are transferring data between the respective nodes. This fair-share model abstracts away the complexity of network congestion. We used the Boykov-Kolmogorov algorithm to compute a solution for this problem [11]. As this still requires significant computation time,

we also implemented a simple greedy approximation for this family of max-flow problems.

Greedy Approximation of Network Throughput. Our approximation works as follows: First, it collects the number of outgoing and incoming streams at each node. Second, it computes the available throughput, each node can provide for each stream, if the bandwidth is shared equally among all streams. Next, for each stream we select the minimum of the available bandwidths of the source and target nodes and initialize the transfer speeds with it. Then, we iterate over all streams and check for each whether both nodes still have bandwidth available. If so, we add the minimum of both available bandwidths to the stream. After iterating over all streams, we reach a Pareto optimum which we return as a solution.

4 Creating Distributed Workloads

To evaluate scheduling algorithms with DQSim, workloads of different queries are essential. So we fully automated the process of converting SQL queries to DTS problems, given a distributed schema. As depicted in Figure 3, a scheduling algorithm takes three inputs: Cluster definition, initial cluster state, and a distributed query plan.

Creating Cluster Definition and State. Cluster definitions are rather easy to create. We choose common compute instances from cloud providers and use their core count, memory size, and network bandwidth. For example, many of the experiments in this paper are based on the c5n18xlarge instance type from AWS [6]. An initial cluster state can be created by assigning partitions to nodes. We first manually create a distributed version of the database schema we are using. Here we define the layout of each relation, which is either hash-partitioned on a key or broadcast. See Table 1 for an example of a distributed version of the TPC-H schema. A simple way to do so is to assign the partitions of each relation in a round-robin fashion to all nodes. To create scale-out scenarios, we leave out the newly added nodes that are not assigned any partitions in the initial state.

Creating Distributed Query Plans. There are two ways to obtain a distributed plan from a SQL query. First, we can simply run the query on a distributed system, collect the plan with execution traces, and use these traces to create a DTS problem. We did this for the simulator validation in Section 6. Second, we use the parser and optimizer of the single-node database system Umbra to obtain a query plan and annotate it with cardinalities and sizes [32]. We then split the plan into pipelines that can be executed distributedly. This includes placing new shuffle stages where distributed joins require a repartitioning. To do so, we make use of the distribution properties required by each operator as described in [2]. For small intermediate results, we prefer to broadcast instead of hash-partitioning. Finally, we set the result and small intermediate results close to the result to single node distribution. To obtain execution times for tasks, we estimate the expected execution time of each pipeline on a reference machine using the pipeline-based learned cost model T3 [34]. With this SQL frontend, one can create very large workloads quickly from SQL queries, once the distributed schema is defined.

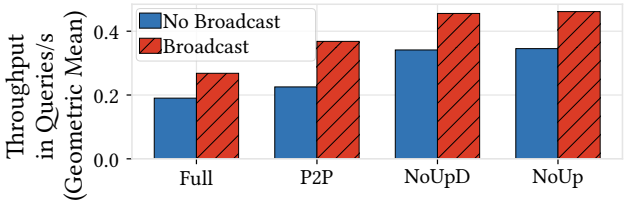

Figure 6: Query throughput for variations of state separation across TPC-H, TPC-DS, and JOB.

5 Case Study: State Separation

Before diving into solutions to the distributed task scheduling problem, we first perform a case study to show how DQSim can be used to evaluate system designs. In this case study, we look at the cost of state separation achieved by uploading all shuffled intermediate results. The state separation design in this experiment is inspired by Polaris [2]. However, the details of their implementation are not public, so the numbers in this experiment might not reflect its performance accurately. We consider the following variants in our experiment:

- **Full:** All shuffle targets are located on the storage service. When a shuffle input partition becomes available, the producer node shuffles its data to the storage service. If some input data of a task is not available on the node that executes it, it is loaded from the storage service.
- **P2P:** Shuffle targets are on the storage service, but partitions can also be shared with P2P connections. The durability is the same as in Full.
- **NoUpD:** Shuffle targets are located on compute nodes and P2P connections are used for all data transfers. Shuffled partitions are written to disk for durability.
- **NoUp:** No state separation. Same as NoUpD, but nothing is written to disk.

For each of those variants, we have two versions. Broadcast versions execute broadcast tasks on all nodes. The versions with no broadcast can only run each broadcast task once and have to transfer its results to all nodes that use them. Tasks are assigned to the nodes that have the largest fraction of their input data. The partitions of the base relations are distributed and cached equally to all nodes.

The Cost of State Separation. Figure 6 shows the query throughput of all TPC-H, TPC-DS, and JOB queries for all variants. There is a significant difference between the performance of state separation compared to the NoUp variants that do not upload intermediate results. This happens because network speeds of compute nodes are limited and the storage service has high latency. We observed much larger differences for smaller queries due to the latency of the storage service. In contrast, the local-disk durability variant (NoUpD) is only marginally slower than NoUp baseline, despite assuming relatively slow SATA write bandwidth of 560 MB/s.

The Cost of Design Decisions. Another interesting result of this experiment is the difference between the variations that actually do implement state separation. The variant that allows both P2P connections and repeated execution of broadcast tasks is over 1.5x faster than the variant that disallows both. When designing a cloud-based execution engine, this performance impact might not be obvious

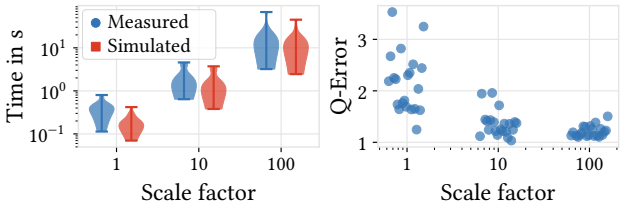

Figure 7: Simulated execution of all TPC-H queries with different scale factors compared to Trino execution.

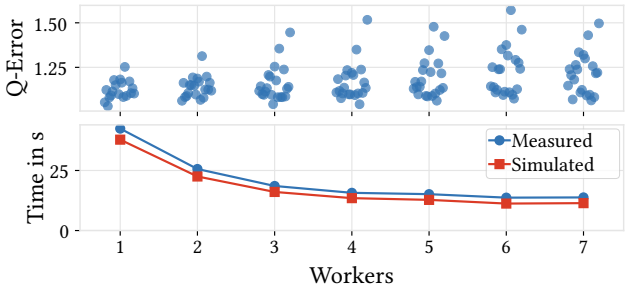

Figure 8: Simulated and Trino execution of all TPC-H queries across varying numbers of workers.

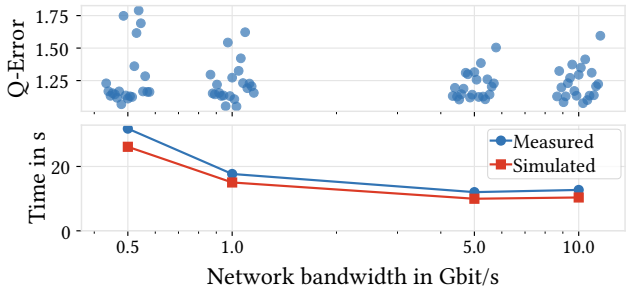

Figure 9: Simulated and Trino execution of all TPC-H queries across varying network speeds.

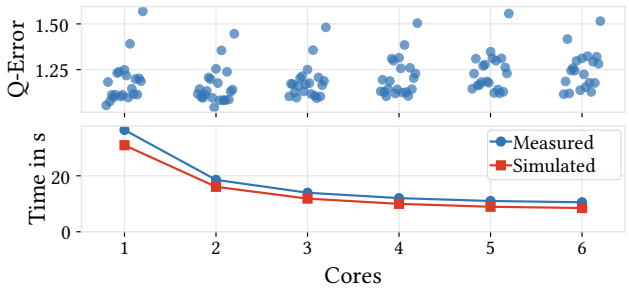

Figure 10: Simulated and Trino execution of all TPC-H queries across varying worker sizes.

without conducting experiments. It might look like a good idea to avoid repeatedly computing the same result on different nodes by simply loading the broadcast results from other nodes. Also, setting up P2P communication between compute nodes introduces system complexity that can be avoided by transferring all data over the storage service. Without knowing the performance impact, it is hard to make a good decision whether this optimization is worth the development time. Using our simulator, these discoveries can be made quickly without a full implementation of the approaches. The results can also be computed in seconds compared to benchmarks that need to be set up and run for some time. Implementing all of these variants and performing the experiment in a full distributed system would require substantial engineering effort.

Improving Performance For State Separation. Seeing the performance overhead of state separation, we think that there could be some improvements over locating all shuffle targets on the storage service. For some queries, it might be cheaper to restart each query on a failure than using state separation. A simple idea would be to classify each query by its expected execution time. For short-running queries, one could simply toggle off state separation, as a restart would not be very expensive. Another idea would be to set only specific durability points within a query plan instead of storing all intermediate results.

Summary. In summary, the analysis above shows that DQSim can be used to reason about design decisions. It is easy to model multiple different designs and DQSim can evaluate them quickly.

6 Simulation Validation

The case study above illustrates how DQSim can be used to analyze the performance impact of system design decisions. However, such experiments are only meaningful if DQSim's predictions translate to real effects in full systems. In this section, we show that although DQSim underestimates absolute latency, especially for smaller scale factors, it supports relative schedule comparison. To validate DQSim's execution model given traced plans, we reconstruct distributed query plans from Trino's execution metrics and simulate these plans in DQSim [1]. Figure 7 compares DQSim's estimated execution times to Trino benchmarks. We observe systematic under-estimation, as DQSim models an idealized execution. In contrast, there are several sources of overhead in Trino, like thread scheduling, coordination, and IO. For scale factor one, these effects make a noticeable difference and DQSim underestimates by 2.2x per query on average. For larger scale factors, the simulations become more accurate (1.4x for sf 10 and 1.2x for sf 100). Figure 8, Figure 9, and Figure 10 show that DQSim can also reflect the performance effects of varying worker nodes, network speeds, and worker size. Experiments with Spark SQL [7] and Firebolt [18] yield similar results. In conclusion, DQSim's abstractions of the execution model preserve performance trends well enough to reason about scheduling quality.

7 Scheduling Algorithms

Having defined the DTS problem and DQSim's execution, we now cover algorithms that address the problem. We start with simple baselines such as round-robin, minimizing network transfers, and random sampling. Then, we show more sophisticated algorithms based on the well-established HEFT algorithm for DAG scheduling with communication delays [44]. As the DTS problem is NP-hard, all presented algorithms are heuristics that do not solve the problem optimally.

```python
def round_robin(plan, nNodes):
  result  = {}
  for i, t in enumerate(plan.flatTasks):
    result[t] = i % nNodes
  return liftFlatAssignment(result)

def aligned_rr(plan, nNodes):
  result  = {}
  for t in plan.flatTasks:
    result[t] = t.out.partitionIndex % nNodes
  return liftFlatAssignment(result)
```

Listing 2: Round-robin assignment algorithm.

7.1 Round Robin

A trivial way to assign tasks is using a simple round-robin fashion as outlined in Listing 2. However, there are some interesting details here.

Broadcast Tasks. First, we make a distinction between broadcast tasks and flat tasks. A *broadcast task* is a task that represents the execution of a broadcast pipeline. In contrast, a *flat task* is a task that should only be executed on a single machine. For example, a TPC-H query joining both `region` and `nation` with other tables can execute that join on all nodes to have this tiny intermediate result available on all nodes. We assign the flat tasks in a round-robin fashion, but each broadcast task can be assigned to many nodes. To avoid unnecessary work, we only perform broadcast tasks on nodes that actually use their output data. We compute this usage mapping from partitions to using nodes by iterating over all tasks that have been already assigned to nodes. Here, we have to make sure that broadcast tasks whose output is only used by other broadcast tasks are assigned as well. This can be done by iterating over the broadcast pipelines in inverse execution order when assigning broadcast tasks and adding the required input partitions of each newly assigned broadcast task.

Data Assignment. As described in Section 2, the result of a scheduling algorithm consists of task assignment, transfers, and shuffle targets. After task placement, we ensure that each task's required data is present on its node. Again, we use a mapping from partitions to all its users. First, we assign the shuffle targets. For each partitioned or single node shuffle target, we select the first node that actually uses the shuffle target partition. For broadcast shuffles, there is only a single output partition and we select all nodes that use the shuffle target partition.

Aligning Partitions. A simple optimization we can use to improve data locality is to assign partitions aligned to compute nodes. The idea is simple: Assign task i of each pipeline to the same node. If there is a join that does not need a shuffle, the data will already be co-located. We also use the same idea to assign partitions of base relations to nodes in the initial state. We observed that aligned assignment improves execution time by about 10%.

7.2 Enumeration

Number of Possible Assignments. Using the same distinction into flat and broadcast tasks as above, we can enumerate all possible assignments of flat tasks. First, we count the number of flat tasks n_f and the number of compute nodes n_c. Naturally, each flat task has n_c possible locations. So, the number of possible assignments is $n_c^{n_f}$. Here, we can already see that this number grows extremely large. For example, consider a query with 4 distributed pipelines that is executed on 4 nodes and the data is partitioned into 8 partitions per relation. This moderate query results in $4 \cdot 8 = 32$ tasks and $4^{32} \approx 10^{19}$ possible assignments.

Ranking Assignments. However, conceptually, nothing prevents us from enumerating or ranking all of these assignments. For very small problem instances, this can be used to exhaustively enumerate all possible assignments. For larger instances, we can randomly sample a rank r with $0 \leq r < n_c^{n_f}$ and construct the corresponding assignment. In both cases, we can simply evaluate all created assignments using the simulator and choose the best observed assignment. We can obtain an assignment from a rank using base-n unranking. The vector of nodes assigned to each task $(a_0, ..., a_{n_f-1}) \in 0, ..., n_c - 1$ can be computed using:

$$a_i = \left\lfloor \frac{r}{n_c^i} \right\rfloor \mod n_c \tag{4}$$

We analyze this approach with a sample size of 1000 under the name *Rand1k* in Section 8. It is usually slow and returns bad schedules. It is slow, because we need to run the simulator very often. Even though the simulator is implemented rather efficiently and we can use multithreading to evaluate plans concurrently, sampling and evaluating 1000 assignments takes about 140 ms per query, longer than a scheduler typically should take for simple queries. Still, because there is such a huge number of possible assignments, a random sample of 1000 is very small in comparison and the likelihood of finding a good assignment is rather low.

7.3 GMN: Greedily Minimize Network Transfers

A straightforward idea to create good task schedules is to place tasks where most of their input data is present. This reduces the time spent on network transfers and network latency, if all data is already locally available. We compute the best node by collecting a list of all nodes that have at least some of the input data of the current task. Then, we choose the node with the maximum fraction of present data measured in bytes as a result for the current task. When no node has any of the input data stored, we fall back to round-robin assignment. This approach works well in many cases. However, its greedy nature makes it unsuitable for queries where intermediate results can grow larger than their inputs. In such cases it is oblivious to the cost of subsequent transfers and places tasks whose results should be co-located on different nodes. Also, it comes to its limits when applied to heterogeneous compute clusters, because it does not consider properties of the compute nodes. We also created variants of this algorithm with different fallbacks. Using aligned round-robin instead of regular round-robin makes no significant difference. We also implemented a simplified version of earliest-start-time and earliest-finish-time that only considers computation time and ignores network transfers. The earliest-finish-time variant appears to work slightly better for heterogeneous clusters.

7.4 Earliest-Finish-Time Algorithms for Heterogeneous Clusters

In modern cloud systems, cost is becoming an increasingly important factor [27]. One opportunity to optimize cost is to make use of so-called spot instances [5, 42, 51]. Cloud instances that are currently in low demand can be rented temporarily at a lower price. As the instance types of spot instances are volatile, systems can make best use of them by supporting clusters with a mix of different instance types. In this section we propose a number of approaches to schedule tasks on such heterogeneous clusters.

The Heterogeneous Task Scheduling Model (HTS). A very similar problem to task scheduling on distributed database systems has been studied for decades. In 2002, Topcuoglu et al. published their work on the task scheduling problem on *heterogeneous processors* with contention-free communication delays [44]. We will call this problem setting the *HTS model*. In this well-defined problem setting, processors correspond to compute nodes in database systems. Tasks are defined as a DAG and can depend on each other, analogously to the DTS model in this paper. The execution time of each task can be different on each node and is defined by a large matrix. This is different to the DTS model, where the execution time of tasks is simply scaled by the compute capabilities of nodes. Finally, the network is modeled very differently. The HTS model assumes that all nodes have a direct connection to each other (fully connected topology) with individual bandwidths. Further, the HTS model ignores contention on these communication channels. We think that this model does not fit the setting of distributed databases well. For example, if all nodes would transfer their data to a single node n_0, the network connection of n_0 will become a bottleneck. In the HTS model, however, these transfers would not interfere with each other. Even multiple transfers between the same two nodes would not congest, resulting in a 2x faster transfer time than in the DTS model. Also, it does not support broadcast tasks that may be executed on more than a single node, shuffles that can consist of multiple inputs and outputs, or node caches that are already present. Nevertheless, we created an adapter that enables us to use algorithms for the HTS model for DTS problems.

The HEFT Algorithm. The heterogeneous earliest-finish-time (HEFT) algorithm is a rather simple but very effective algorithm for the HTS problem [44]. There are two main ideas. First, we sort tasks by a *rank* that is determined by the amount of work that depends on the result of this task. The more work depends on a task, the earlier it will be picked up by HEFT and assigned to a node. Second, when a task is selected, HEFT computes the finish time of the task for each node. It places the task on the node that has the earliest finish time. When a task is placed on a node, it occupies the node for a contiguous time interval to represent the computation time. This forms a CPU schedule for each node. Because HEFT considers many dependent tasks and many placement possibilities for each task, the optimization time of HEFT is significantly longer than the greedy algorithms above.

The NetHEFT Algorithm. While HEFT is very useful for the HTS problem, the model differences render it quite ineffective for our use-case (see Section 8). To overcome this issue, NetHEFT explicitly models the DTS problem. NetHEFT's key extension over HEFT is the maintenance of *resource calendars* (a concept also used in civil engineering to schedule construction work with constrained resources [28]). When NetHEFT schedules a job, it blocks the respective resources CPU, network upload, and network download in a resource calendar. A HEFT schedule resembles a single resource calendar for the CPU time. In contrast, NetHEFT can represent congestion on CPU and network. Resource calendars enable NetHEFT to make accurate ahead-of-time finish time estimations. We also create special handling for cached inputs, shuffles, and storage services. To enable cached inputs, we create a map from each partition to the time it became available on each node that has it. Cached inputs are simply available at time 0. When creating the assignment, NetHEFT iterates over all pipelines in a topological ordering and assigns the tasks of each pipeline. If the output of a pipeline is shuffled, we directly block an upload slot of each node that computes a partition of the shuffle input. The download slots of shuffled partitions are only computed when assigning the tasks that require them. When a task is placed on a node, corresponding download slots are blocked for shuffled inputs. Note that we relax the requirements of resource calendars here so upload and download can happen at distinct times. We use this approximation to make this algorithm practical. Otherwise, dependent tasks would need to be assigned immediately, preventing step-by-step assignment. With DQSim, we can see that NetHEFT is very effective despite this approximation. For regular transfers from nodes n_1 to n_2, however, we block the same time slot in the upload calendar of n_1 and the download calendar of n_2. As we assume a storage service with very high bandwidth, we omit the calendar for the storage service node and thereby ignore all congestion on the storage service. Network latencies are incorporated by increasing the lower bound of network time slots by the maximum of both involved network latencies from the time at which the partition becomes available. NetHEFT yields much better results than HEFT and is faster in most instances (see Section 8.7).

7.5 cCEFT: Schedule Connected Components

Finally, we present cCEFT, a variant of NetHEFT. Instead of assigning each task individually, cCEFT schedules larger task groups. The idea behind this grouping is to always co-locate shards of pipelines that do not need to be shuffled. First, cCEFT divides query plans into connected components that are not separated by shuffles. Then, it groups tasks that belonged to different pipelines within the same component. These task groups are mostly equivalent to Polaris tasks, which are maximal units of work by default [2]. Then, the task groups are scheduled like individual tasks in NetHEFT. To do so, we iterate over components in topological order. Then, we pick each task group within the component. For each task group, we compute the earliest finish time on each node using the same resource schedules as NetHEFT. This algorithm has the advantage that tasks, which can be executed without transfers, will always be co-located despite greedy decisions. On the other hand, cCEFT can never assign two tasks within a component to different nodes, even if it would be advantageous, e.g. for load balancing.

Comparison to Polaris. cCEFT's components are constructed similarly to Polaris' tasks, which groups tasks into maximal units of work [2]. However, the two algorithms differ fundamentally in their approach to assigning tasks to nodes. While the Polaris paper

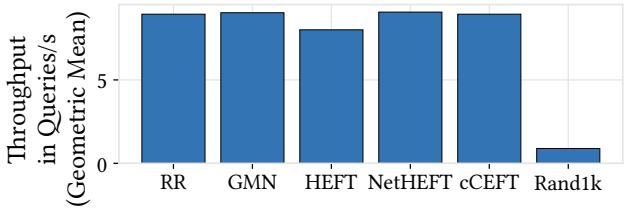

Figure 11: Execution throughput in common scenario

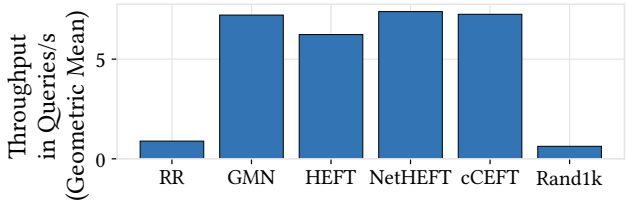

Figure 12: Execution throughput after 2x scale-out.

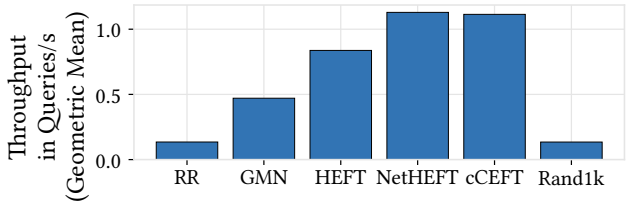

Figure 13: Execution throughput after 2x scale-out from small to large nodes.

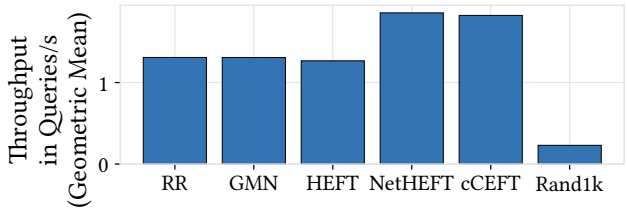

Figure 14: Execution throughput on a heterogeneous cluster.

does not provide full details of the resource governor, it states that a task's target location is fixed by data affinity. This resembles the strategy of our GMN baseline, which greedily places tasks where most of their input data is already available. In contrast, NetHEFT and cCEFT keep CPU and network resource calendars for all nodes to choose the node with the earliest finish time.

Further, Polaris' scheduler is described as only scheduling tasks after they become ready. In contrast, the DTS problem requires scheduling to be finished ahead of execution. This difference matters for shuffles and transfers in low-latency query processing. Only if the consuming nodes are already fixed can data be sent immediately to the target node once it becomes available.

8 Scheduling Experiments

In this section, we evaluate the scheduling algorithms introduced in Section 7. In all experiments, we report throughput as the inverse of the geometric mean of individual query latencies: $(\prod_{i=1}^{n} l_i)^{\frac{-1}{n}}$. We scale task execution speeds linearly with core count in all experiments and do not use the diminishing-returns scaling configuration described in Section 3.2.

8.1 Default Scenario

First, we consider a common scenario: There are 8 identical compute nodes of type c5n18xlarge, relations are split into 16 partitions. In the initial state, each node has two partitions of each base relation. We used all TPC-H, TPC-DS, and JOB queries with lifted single-node plans from Umbra [32] and T3 execution-time estimates [34], and scaled cardinalities by a factor of 100. Figure 11 shows that most algorithms yield similar performance. Notably, Rand1k which is introduced in Section 7.2 yields drastically worse results. This supports our assumption that random search does not work well for this problem, because the search space is too large. HEFT also performs worse, showing that the DTS problem is very different from the HTS problem. Surprisingly, simple aligned round-robin scheduling (RR) performs just as well as the other algorithms. The reason for this is that in this simple case where base relations are also distributed in an aligned round-robin fashion, round-robin schedules will always have good data locality.

8.2 Scale-Out Scenarios

Doubling the Cluster Size. Figure 12 depicts performance in a simple scale-out scenario where there are 4 nodes that have all data cached and 4 newly added nodes that have no data cached. Here, we can see that the round-robin algorithm performs significantly worse. It will ignore data locality completely and place half of the tasks on the newly added nodes, even when they need to load a lot of data. We can also see that GMN, NetHEFT, and cCEFT all yield about the same performance. The reason for this is that in this scenario, scaling out never pays off. Placing all tasks on the machines with warm caches is optimal because network transfers would be more expensive than doing all computation on the original 4 nodes. While NetHEFT and cCEFT can make this decision deliberately, GMN always chooses machines with warm caches to optimize data locality.

Adding Larger Machines. Figure 13 shows the performance when starting with a 4-node cluster of small machines (c5nxlarge) and adding 4 large (c5n18xlarge) machines. In this scenario, scaling out pays off, because the small machines have so little computational power that the execution actually becomes the bottleneck. It is faster to transfer some data and utilize the compute capabilities of the large machines. We can see that in this scenario, simply maximizing data locality does not work. Execution with GMN schedules is about 2x slower than the best. Also, HEFT can find rather good schedules, even though it has an inaccurate model of the network, because network transfers are not the bottleneck here. NetHEFT and cCEFT yield similarly good results.

8.3 Heterogeneous Clusters

Figure 14 shows that GMN does yield worse results on heterogeneous clusters, even if all caches are warm. In this case, it behaves the same as round-robin does. This result indicates that there is an

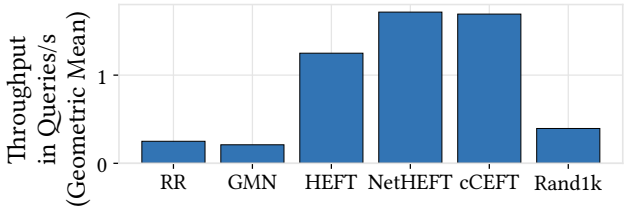

Figure 15: Execution throughput of queries with growing intermediate results after 2x scale-out from small to large nodes.

opportunity to improve performance with NetHEFT or cCEFT on heterogeneous clusters.

8.4 Growing Cardinalities

Growing Queries. For the queries above, pipeline outputs were generally smaller than pipeline inputs. In many analytical queries, base relations are filtered and subsequent joins reduce the cardinalities even further. There are, however, also workloads, e.g. graph workloads, where intermediate result cardinalities can grow much larger. We obtained a set of such queries, by filtering the SQLStorm queries to queries that have at least one join result that is larger than both of its inputs [39]. We used 996 such queries for this benchmark.

Scheduling Growing Queries. For workloads with growing intermediate results, greedy decisions can have stronger negative effects. If two tasks that depend on each other are not co-located, the intermediate result needs to be transferred. For small intermediate results, these transfers are usually cheap. But for larger intermediate results, the transfer can be much more expensive than co-locating the tasks in the first place. This effect does not occur, however, if the base relations are distributed equally as in Section 8.1. In this scenario, simple round-robin assignment works very well. Figure 15 shows that GMN cannot find good schedules when scaling out queries with growing intermediate results. In this case, it is even worse than round-robin. Both NetHEFT and cCEFT show good performance.

8.5 Schedule Quality

While the experiments above compared schedule quality of the different algorithms, we want to check the overall schedule quality of all algorithms. Ideally, we want to compare the schedules from the algorithms against the optimal schedule for each query. To do so, we use schedule enumeration and brute-force all possible assignments from non-broadcast tasks to nodes. As the number of possibilities grows rapidly, we can only evaluate this on a very restrictive set of queries. For this experiment, we choose a cluster of three different node types and a partition count of four. To make the problems more challenging, we assign each base relation partition to a random node. Although this is not very realistic for real systems, this adversarial experiment shows schedule quality in very challenging cases. We use a subset of the TPC-H, TPC-DS, and JOB queries. To keep the brute force times manageable, we do not include queries with more than 16 tasks. Figure 16 shows that NetHEFT and cCEFT return optimal schedules in many cases. Their worst schedules cause a

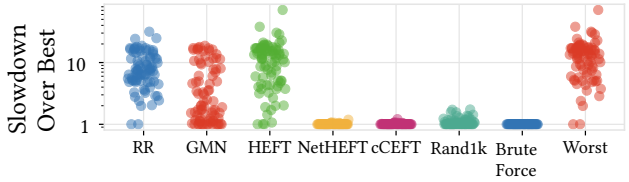

Figure 16: Execution slowdown compared to best schedule. Each marker represents the slowdown on one query.

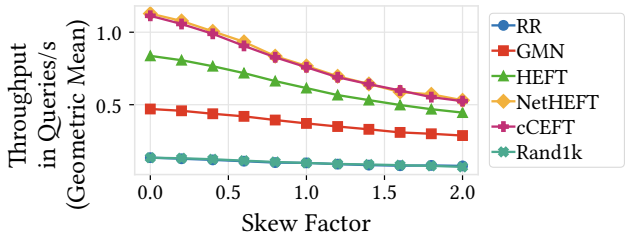

Figure 17: Execution speed with increasing partition skew.

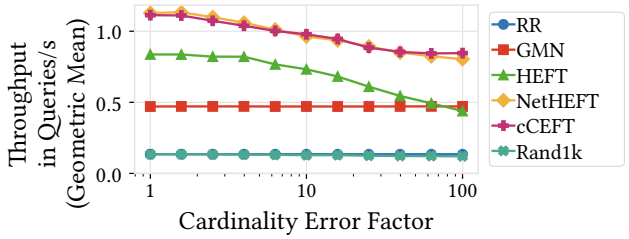

Figure 18: Execution speed with increasing cardinality estimation errors.

slowdown below 1.4x. Figure 16 also shows the worst schedule found by all algorithms. Interestingly, the brute force algorithm is not always guaranteed to find the optimal schedule. The reason for this is the lifting heuristic we use to go from a flat assignment to a full schedule. In this step we assign broadcast tasks to all nodes that need the output data. In cases where transferring the broadcast output to a node would be faster than executing the broadcast pipeline the brute force approach cannot find the optimal schedule. Nevertheless, this experiment gives us confidence that NetHEFT and cCEFT do find very high quality schedules, even in complicated scenarios.

8.6 Scheduling Robustness

In the experiments above, the schedulers were given perfect information about the execution. However, real systems do not have perfect cardinality estimates, are subject to skew, and can have stragglers. In the following, we modify the scale out scenario from Figure 13 to analyze the robustness of the scheduling algorithms.

Skew. Figure 17 shows how execution performance degrades under partition skew. We distorted the partition sizes of each data unit following a Zipf distribution. The total cardinality of each data unit remains equal, but now the size of partition p_i is proportional to $\frac{1}{i^s}$, where s is the skew factor and i is the rank of the partition according to a random order. So, while all partitions have the same size at

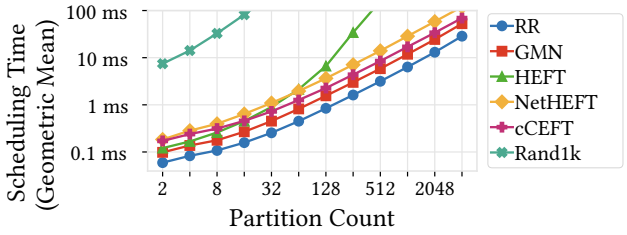

Figure 19: Scheduling time for different partition counts.

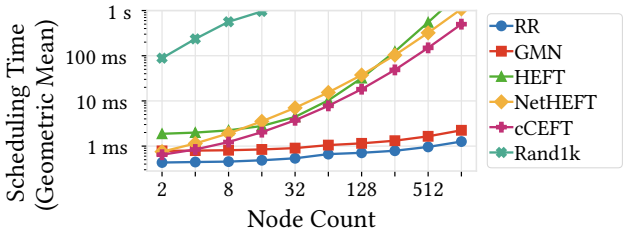

Figure 20: Scheduling time for different node counts.

$s = 0$, for $s = 2$ the largest partition alone covers 63% of the data and the smallest only 0.2%. We can see that performance degrades under skew.

Cardinality Estimation Errors. Figure 18 shows that cardinality estimation errors also degrade execution performance. For this experiment, we randomly multiply or divide all cardinalities of non-base-relations by the error factor. We use these distorted cardinalities to run the planners and evaluate their assignments on the original plans. We observe that round robin and GMN schedules, which do not consider cardinalities at all, are robust in this case. In contrast, NetHEFT and cCEFT schedules can degrade. For simple scenarios, this can lead them to find slightly worse assignments than GMN. However, when scaling out to larger nodes, NetHEFT and cCEFT can still find better schedules for errors of 100x.

Outlook. Clearly, these differences between expected and real execution times can have a significant impact on performance. Robustness against such unexpected events could be improved at runtime. For example, systems could adaptively reschedule during execution. Future work could extend the scheduling algorithms to adaptively reschedule during execution (see Section 10).

8.7 Schedule Optimization Times

While our new algorithms find better schedules, a scheduler also must compute them quickly to improve performance. The scheduling time itself directly adds to the query latency and should, hence, be as short as possible. In this section we show the optimization times of scheduling algorithms for different scenarios.

Partition Count Scaling. In Figure 19, we see how the different algorithms scale to larger partition counts. The partition count is the number of partitions we split each data unit into. Depending on the system, a large range of partition counts could be realistic. Shared nothing systems might create one partition per node which would make partition counts below 16 realistic. Large scale cloud systems can use a fixed partitioning in the storage service and could profit from over-partitioning. For example, Polaris supports up to 60 partitions [29]. We consider 64 a reasonable partition count. In this experiment we evaluated the schedulers on all TPC-H queries for 8 nodes. Figure 19 shows that most algorithms can build a schedule in less than one millisecond. Here we can see that random sampling is over 100x slower than the other algorithms in most cases. Also, the not-adapted HEFT algorithm becomes significantly slower for partition counts above 100. Of course, round-robin is the fastest algorithm and the simple heuristic of GMN is also rather quick to evaluate. NetHEFT and cCEFT, however, are usually less than 5x slower than GMN. cCEFT is faster than NetHEFT and remains below 10 ms for up to 500 partitions.

Node Count Scaling. The second important factor for scheduling times next to partition count is node count. Figure 20 shows that Round Robin and GMN scale very well to huge cluster sizes of up to 1000 nodes. HEFT, NetHEFT, and cCEFT all show strongly increasing optimization times for larger node counts. NetHEFT and cCEFT finish in about 5 ms for up to 16 nodes and handle about 100 nodes in 20 ms. This optimization time could be significant, as many workloads contain a large fraction of short-running queries below 100 ms [52]. One might want to use an adaptive scheduler that uses GMN for simple cases or short-running queries and only employ cCEFT when the query is expected to run for a longer time. In this experiment we evaluated the schedulers on all TPC-H queries using 64 partitions.

Performance Summary. In conclusion, our experiments suggest that cCEFT is typically better than NetHEFT with slightly faster optimization speed and almost equal schedule quality. For cases where all nodes have warm caches and intermediate results do not grow the lower optimization time of GMN might make it more attractive than always running cCEFT.

9 Related Work

Modern Analytical Database Systems. Current analytical systems usually employ a simpler approach to task placement. For example, Polaris assigns tasks based on data collocation, similarly to our GMN algorithm [2]. Snowflake uses consistent hashing to assign data to compute nodes [16]. While they employ work stealing during scans, the initial assignment of partitions resembles round-robin distribution like traditional shared-nothing systems. Similarly, Redshift re-assigns partitions statically after resizing the cluster and hydrates caches in the background [8]. Vertica introduces dynamic rescaling using shared storage in *Eon mode*, and uses a similar partition assignment as Redshift which they call subscription to shards [46].

Formal Scheduling Problems. There has been ample work on formalized scheduling problems that are very similar to the distributed task scheduling problem (DTS) covered by this paper. In scheduling problems, one minimizes the *makespan*, the finish time of the last task. The class of scheduling most closely related to DTS is DAG scheduling, which has been studied since the 1960s [23]. Veltman et al introduced communication delays between processors in 1990 [47]. Topcuoglu et al. published the HEFT algorithm for DAG scheduling on heterogeneous processors with communication

delays in 2002 [44]. This problem setting is quite similar, but assumes that there is a fully connected network between processors and there is no congestion on network bandwidth. NetHEFT is our adaptation of the HEFT algorithm for the DTS problem that models the network more closely. More recent work introduced CEFT as an improvement over HEFT leveraging critical components [26]. The problem was also approached using a mixed integer program formulation [12]. While all of these scheduling problems are NP-hard, we can leverage additional knowledge about typical database queries to get practically good results with moderate computational complexity.

Scheduling on Databases. There have been approaches to scheduling problems very similar to the DTS problem in database research. For example, Gouda and Dayal studied schedules for semijoin queries where relations are stored at different sites [22]. Garofalakis and Ioannidis optimize query scheduling with multi-dimensional resource usages [20]. Giceva et al. optimized co-location of dependent tasks on multi-core CPUs [21]. Sabek et al. apply reinforcement learning to task scheduling on multi-core CPUs [37].

Network-Aware Scheduling. There is also work on network-aware scheduling in related settings. ShuffleWatcher improves shuffle locality with better placement of MapReduce tasks [3]. Unlike later work and DTS, it does not model dependencies between jobs. Corral assigns whole jobs to rack-sets for better locality in multi-tenant settings and considers DAG-structured jobs [25]. Dawn also considers job dependencies using DAG structures to improve locality of dependent tasks [49]. Corral and Dawn are closest to DTS, but they primarily address settings where limited cross-rack bandwidth makes network locality the central scheduling objective. Rödiger et al. proposed to compute optimal partition assignment for shuffles using a MILP formulation of the problem [36]. That work also targets systems where network bandwidth is a major bottleneck. Rödiger et al. stated that their partition assignment optimization is not worthwhile on faster networks [35]. In contrast, DTS also targets relatively short-running database workloads on clusters with much faster modern networks [10].

Flow-Based Scheduling. Flow-based network scheduling improves distributed execution by controlling the ordering, prioritization, or bandwidth allocation of transfers. Orchestra achieves significant speedups by improving network schedules of broadcast and shuffle operations and prioritizing transfers [14]. Varys extends the scope by using the coflow abstraction for network transfers [15]. It can schedule multiple coflows concurrently to improve overall latency and meet individual deadlines. Rödiger et al. also control network schedules to avoid congestion [35, 36]. In contrast to these approaches, the DTS problem optimizes data and task placement. Thus, flow-based scheduling is complementary to DTS. It can optimize transfer and shuffle execution after a DTS algorithm has fixed the transfer endpoints.

Related Work In Similar Systems. Other work focuses on similar problems. For example, Cardellini et al. optimize operator replication in streaming systems [13]. Further, Wang et al. optimize the assignment of streaming operators to compute nodes [50]. Xu et al. jointly optimize task scheduling and cache management in Spark [53]. Google's Borg manages job scheduling to large scale clusters [48]. Agile Ant dynamically optimizes caches and autonomously scales clusters if needed [4].

Database Simulation. There is also work on simulators for distributed data systems. SparkTune is a detailed cost model that considers compute as well as network and disk IO [9]. As it is fine-tuned to Spark it achieves good accuracy, but in its current form lacks the generality to represent distributed relational systems accurately. PerfOrator aims to find the best hardware for queries by estimating performance and resource usage [33]. While it schedules queries to suitable hardware, it does not consider intra-query scheduling decisions. CloudGlide is a tool to explore data processing on modern cloud infrastructure [30]. It models queries as IO, CPU, and shuffle stages. This information is gathered from execution traces without detailed information about queries. While CloudGlide accurately predicts execution times with this coarse view, it is not suitable to evaluate fine-grained execution decisions like task placement.

10 Discussion

Use of Formalized Problem and Simulation. DQSim abstracts away from several aspects of real-world execution. This abstraction enables it to run experiments with much lower cost and latency than any full system could while offering large flexibility in system design and workloads. For example, it does not model noisy neighbors, network latency variations, network congestion in more complex network topologies, or CPU clock variations caused by spikes and idle phases of local computation. Our experiment in Section 6 suggests that DQSim's level of abstraction includes enough details to meaningfully analyze the relative performance impact of scheduling decisions. However, DQSim's abstractions make it more useful for relative performance evaluation than for making precise execution time predictions in realistic environments.

Holistic Workload Scheduling. This paper considers scheduling one query at a time, which already is a challenging task. This simplification makes it easier to interpret results. The scheduling of whole workloads consisting of multiple concurrent queries is an interesting topic for future work. Individual query scheduling allows better interpretability and we consider it a logical first step. The distributed query plan structure, however, already fits multiple queries rather well. By matching the common base relations between queries to single data units and allowing results from multiple queries, a distributed plan should be able to represent a whole workload. This suggests a straightforward extension of DQSim and scheduling algorithms for workloads consisting of multiple queries. Future research could focus on trade-offs between throughput, latency, fairness, and other workload-level challenges.

Adaptive Re-Scheduling. Further, DQSim can be extended to interactively change states of the simulated systems by adding new queries and removing finished ones. Simulation and scheduling can then be run in an interleaved pattern. This would enable simulating a continuous workload execution instead of simulating each scenario individually. Finally, unexpected events like cardinality mispredictions, skew, or node failures could be handled as well in such an interactive setting. Note that assigning tasks for entire queries ahead of time is necessary to enable P2P shuffles and transfers to start immediately once their input data becomes available.

Scheduling Algorithm Choice. The newly proposed scheduling algorithms NetHEFT and cCEFT provide consistently good query

latencies for all scenarios we tested. For many cases, simply optimizing data locality like GMN is suboptimal and the new algorithms deliver strong speedups. In our experiments, cCEFT generally outperforms NetHEFT. As the optimization time of cCEFT can be significantly higher than GMN's, it might be worth it to detect whether the current scenario could benefit from cCEFT and default to GMN if not. For example, GMN is sufficient for homogeneous clusters with warm caches, which are easy to identify.

Spilling. Many OLAP systems spill intermediate state to disk when it exceeds local memory. DQSim tracks memory usage of compute nodes, but currently does not model memory-pressure effects. Consequently, our evaluation does not cover spilling effects.

Extending DQSim for spilling, however, would fit its abstractions naturally. Spillable state like group-by or hash-join build tables is already represented by data units at the ends of pipelines. A spilling extension in DQSim could detect memory pressure and switch affected tasks to a spilling variant. They would write spilled state to disk. Consuming tasks would then read this state from disk. DQSim could model disk reads and writes as a new resource, similar to network to incorporate bandwidth limitations. Further, spilling variants of tasks could have a higher CPU cost.

11 Conclusion

In this work, we formalize the distributed task scheduling problem (DTS) as an abstraction of scheduling decisions in distributed databases. Further, we present DQSim, a fine-grained simulator for distributed analytical query execution that is designed to evaluate task schedules. We demonstrate how it can be used to evaluate the performance impacts of high-level system architecture decisions like state separation. Using DQSim, we show that for many scenarios, simple solutions like round-robin or minimizing network transfers achieve competitive performance. We also show that in challenging scale-out scenarios, more refined algorithms offer significantly better performance. Finally, we propose two new scheduling algorithms that deliver consistently strong query latency in all of our experiments.

References

[1] 2026. Trino: Distributed SQL Query Engine for Big Data. https://trino.io/. Accessed: 2026-05-06.

[2] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.

[3] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. 2014. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *USENIX ATC*. USENIX Association, 1–12.

[4] Hani Al-Sayeh, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2024. Agile-Ant: Self-managing Distributed Cache Management for Cost Optimization of Big Data Applications. *Proc. VLDB Endow.* 17, 11 (2024), 3151–3164.

[5] Amazon Web Services. 2025. Amazon EC2 User Guide Spot Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html. Accessed: 2026-01-11.

[6] Amazon Web Services. 2026. Amazon EC2 C5n Instance Types. https://aws.amazon.com/ec2/instance-types/c5/. Accessed: 2026-01-11.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394.

[8] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD Conference*. ACM, 2205–2217.

[9] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Trans. Knowl. Data Eng.* 31, 5 (2019), 819–832.

[10] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (2016), 528–539.

[11] Yuri Boykov and Vladimir Kolmogorov. 2004. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 9 (2004), 1124–1137.

[12] Abdessamad Ait El Cadi, Rabie Ben Atitallah, Saïd Hanafi, Nenad Mladenovic, and Abdelhakim Artiba. 2017. New MIP model for multiprocessor scheduling problem with communication delays. *Optim. Lett.* 11, 6 (2017), 1091–1107.

[13] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal Operator Replication and Placement for Distributed Stream Processing Systems. *SIGMETRICS Perform. Evaluation Rev.* 44, 4 (2017), 11–22.

[14] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. 2011. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*. ACM, 98–109.

[15] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with Varys. In *SIGCOMM*. ACM, 443–454.

[16] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[17] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782.

[18] Firebolt. 2026. Firebolt Cloud Data Warehouse Whitepaper. https://www.firebolt.io/resources/firebolt-cloud-data-warehouse-whitepaper. Accessed: 2026-05-19.

[19] Lester R Ford Jr and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian journal of Mathematics* 8 (1956), 399–404.

[20] Minos N. Garofalakis and Yannis E. Ioannidis. 1997. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *VLDB*. Morgan Kaufmann, 296–305.

[21] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of Query Plans on Multicores. *Proc. VLDB Endow.* 8, 3 (2014), 233–244.

[22] Mohamed G. Gouda and Umeshwar Dayal. 1981. Optimal Semijoin Schedules For Query Processing in Local Distributed Database Systems. In *SIGMOD Conference*. ACM Press, 164–175.

[23] Ronald L Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell system technical journal* 45, 9 (1966), 1563–1581.

[24] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374.

[25] Virajith Jalaparti, Peter Bodík, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 2015. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*. ACM, 407–420.

[26] Minhaj Ahmad Khan. 2012. Scheduling for heterogeneous Systems using constrained critical paths. *Parallel Comput.* 38, 4-5 (2012), 175–193.

[27] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612.

[28] Ming Lu and Hoi-Ching Lam. 2008. Critical Path Scheduling under Resource Calendar Constraints. *Journal of Construction Engineering and Management* 134, 1 (2008), 25–31. doi:10.1061/(ASCE)0733-9364(2008)134:1(25)

[29] Microsoft Corporation. 2024. *Massively Parallel Processing (MPP) Architecture in Azure Synapse Analytics.* https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/massively-parallel-processing-mpp-architecture

[30] Michail Georgoulakis Misegiannis, Daniel Ritter, Viktor Leis, and Jana Giceva. 2025. CloudGlide: Deconstructing the Landscape of Cloud-Based Analytics. *Proc. VLDB Endow.* 18, 13 (2025), 5638–5651.

[31] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[32] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[33] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: eloquent performance models for Resource Optimization. In *SoCC*. ACM, 415–427.

[34] Maximilian Rieger and Thomas Neumann. 2025. T3: Accurate and Fast Performance Prediction for Relational Database Systems With Compiled Decision Trees. *Proc. ACM Manag. Data* 3, 3, Article 227 (June 2025), 27 pages. doi:10.1145/3725364

[35] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proc. VLDB Endow.* 9, 4 (2015), 228–239.

[36] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2014. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*. IEEE Computer Society, 592–603.

[37] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *SIGMOD Conference*. ACM, 1228–1242.
[38] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *SIGMOD Conference Companion*. ACM, 225–237.
[39] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.
[40] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744.
[41] Srinath Shankar, Rimma V. Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David J. DeWitt, and César A. Galindo-Legaria. 2012. Query Optimization in Microsoft SQL Server PDW. In *SIGMOD Conference*. ACM, 767–776.
[42] Supreeth Shastri and David E. Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *SoCC*. ACM, 493–505.
[43] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.
[44] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distributed Syst.* 13, 3 (2002), 260–274. doi:10.1109/71.993206
[45] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384–393. doi:10.1016/S0022-0000(75)80008-0
[46] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *SIGMOD Conference*. ACM, 797–809.
[47] Bart Veltman, B. J. Lageweg, and Jan Karel Lenstra. 1990. Multiprocessor scheduling with communication delays. *Parallel Comput.* 16, 2-3 (1990), 173–182.
[48] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*. ACM, 18:1–18:17.
[49] Shaoqi Wang, Wei Chen, Xiaobo Zhou, Liqiang Zhang, and Yin Wang. 2019. Dependency-Aware Network Adaptive Scheduling of Data-Intensive Parallel Jobs. *IEEE Trans. Parallel Distributed Syst.* 30, 3 (2019), 515–529.
[50] Yuanli Wang, Lei Huang, Zikun Wang, Vasiliki Kalavri, and Ibrahim Matta. 2025. CAPSys: Contention-aware task placement for data stream processing. In *EuroSys*. ACM, 654–670.
[51] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* 15, 11 (2022), 2966–2979.
[52] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *SIGMOD Conference Companion*. ACM, 280–294.
[53] Yinggen Xu, Liu Liu, and Zhijun Ding. 2020. DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters. In *IPDPS*. IEEE, 378–387.

Received January 2026; revised May 2026; accepted June 2026