Considering Distributed Processing in the Query Optimizer

Maximilian Rieger

supervised by Prof. Dr. Thomas Neumann, Technische Universität München

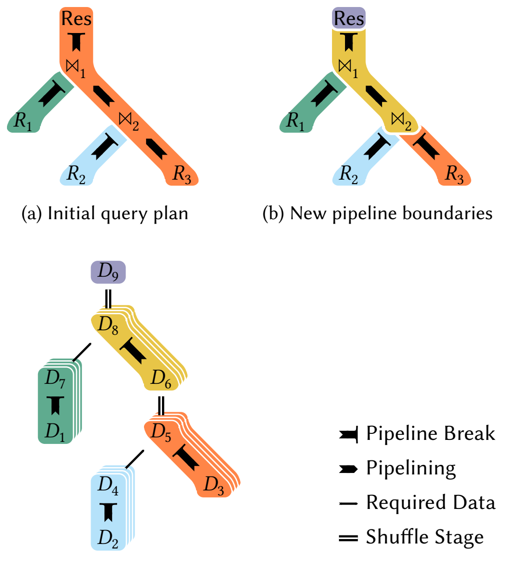
Abstract

Distributed database systems gain relevance both in industry and academia. However, existing research on query optimization for relational database systems focuses largely on systems running on a single machine. Work on distributed systems neglects available workload information in database systems. In this work, we present optimization strategies to fully leverage the potential of distributed systems running on modern cloud architectures with fast networks. We focus on the optimal assignment of tasks to compute nodes and the joint optimization of join ordering and distribution layout of data. Furthermore, we introduce distributed plans and simulation-based evaluations using a new cost model for computation time.

1. Introduction

Considering the very high bandwidth available in modern cloud systems, distributed processing becomes more and more attractive to not only handle large data sizes but also to improve processing performance. Still, good physical plans are key for efficient execution. We argue that distributed execution engines require changes to existing query optimizers for optimal performance. Existing join ordering algorithms yield suboptimal results because they fail to model the cost of data transfers. Furthermore, they need to spread computational load while avoiding waiting on data when assigning tasks to machines. We plan to contribute the following components to investigate new optimization opportunities:

1. A strategy to transform query plans for efficient distributed execution. Methods like hash distributed joins require repartitioning of their input data. Our strategy chooses favorable distributions and introduces necessary data shuffling.
2. A new operator-based computation time estimation method that allows us to compare the cost of transferring data over the network against local processing time.
3. An optimization method for the task assignment problem which determines on which node each part of a distributed query should be executed to minimize query response time.
4. A simulator that models the execution of distributed query plans on a cluster considering each nodes computation and network capabilities. This simulator uses our previous computation time estimations to track execution times accurately. We can verify the quality of assignments across various cluster setups using simulation-based evaluation.
5. A new join ordering algorithm that can take effects of distributed data partitioning and execution into account to jointly optimize distribution layout.



(a) Initial query plan (b) New pipeline boundaries

(c) Distributed query plan

Figure 1: Stages of distributing a query plan with partitioned base relations.

(3) will directly improve query response time. It makes use of (2) to estimate computational load and can be evaluated using (4). (5) also has a direct impact on query performance and can be compared against existing algorithms using (1).

VLDB 2023 PhD Workshop, co-located with the 49th International Conference on Very Large Data Bases (VLDB 2023), August 28, 2023, Vancouver, Canada

max.rieger@tum.de (M. Rieger)

2. Related Work

There are several distributed database systems, but the number of publications on distributed optimization is rather limited. Microsoft extended the search space of SQL Server's query optimizer with data distribution information for cost-based search [1]. Its cloud-native successor Polaris avoids the task assignment problem by writing and reading all intermediate results from a decoupled storage service [2]. This removes many effects of data locality. Redshift automatically chooses partition keys and distribution for observed query workloads, but there is little information published about its query optimizer [3]. Snowflake [4] uses a classical Cascades-like query optimizer [5] but fixes the data distribution at query runtime. Vertica segments tables by their columns instead of hash partitioning [6]. It chooses join ordering with a worklist-based approach that considers distribution information and terminates when the memory budget is exhausted. MemSQL performs cost-based query rewrites in a heuristically pruned search space, weighing data transfers with a constant factor [7]. SparkSQL uses cost and rule-based optimizations to broadcast small tables and perform preaggregations [8]. Rödiger et al. propose network optimal partition assignment using MILP for single join operations in [9]. They approach data skew with selective broadcast and Flow-Join that dynamically broadcast partitions and tuples respectively [10].

There is also a lot of related work in the area of big data that covers similar optimization aspects, such as task scheduling [11]. In contrast to big data systems like Hadoop and Spark, relational database systems have much more information ahead of time. We build upon that research utilizing this additional information and database-specific optimizations such as join ordering.

3. Distributed Processing Model

First, we define the characteristics of distributed systems for which we want to optimize. We focus our work on OLAP systems, but the concepts are also applicable to transactional workloads. The system of concern has disaggregated compute from storage to allow flexible scaling of compute nodes similar to Snowflake [4]. Nodes can vary in computational and network capabilities to use available cloud instances cost-effectively. The full dataset has to be stored on a storage service. Tables are stored in a columnar fashion and hash partitioned on user-defined distribution keys. Similarly to Polaris [2], nodes may cache arbitrary partitions locally. In contrast, we explicitly do not disaggregate query state from compute nodes to avoid the latency overhead of writing back and reading all intermediate results from the storage service.

We generalize relations, intermediate results, and final query results to *data units*. Hash-distributed execution can be used to effectively perform aggregations and joins on large amounts of data. However, the processing speed of joins can be improved by broadcasting data units in cases of skewed data or vast differences in cardinalities [9]. Furthermore, it is possible that the overhead of data transfers in further stages outweighs the advantages of distributed processing. Thus, the optimizer should also be able to decide that a data unit should only reside on a single node. In summary, data units can have the following four partition layouts:

- **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**: Tuples are partitioned without a key.

There are many metrics, such as throughput, query latency, cloud cost, and energy consumption. We choose to optimize for latency, as we expect that optimizing for lower execution and transfer times will improve results in all metrics.

4. Components For Distributed Query Optimization

The main components of our research project are distributed plan generation, computation time estimation, task assignment, a simulator for distributed execution, and a new join ordering optimizer.

4.1. Distributed Plan Generation

We compose distributed query plans from three main components:

Data Units can be base relations from a database, intermediate results, or the final result of a query. They contain the available attributes and the estimated number of tuples in the data unit. Each data unit is annotated with a *partition layout* determining the type of partitioning and the partition key if any.

Pipelines represent the fused computation of operators that is not interrupted by data materialization or transfers. A pipeline always takes one data unit as an input and creates one data unit as an output. Additionally, a pipeline may require the presence of further data units, e.g., a pipeline performing a hash-join would require the data unit of the build side while taking the data unit of the probe side as input.

Shuffle Stages repartition data. A shuffle stage takes one data unit as input and returns one data unit as output.

The only difference between input and output data unit is their respective partition layout.

We initially create distributed query plans from physical plans created for single machines, as depicted in Figure 1a. Our method takes a query plan and partition layout information for all base relations and distributes the plan in several passes.

First, the operators in the plan are combined to pipelines. Next, we determine the best partition layout at each operator and split pipelines where necessary, as in Figure 1b. Finally, we explicitly name the data units at the ends of each pipeline. The output of a pipeline may have a different partition layout than the required input layout of its scanning pipeline. In this case, we create two data units and link them with a shuffle stage, as shown for D_5 and D_6 in Figure 1c.

Distributing single-node plans like this will not yield optimal results as the original plan does not incorporate any information about the distributed system. Ultimately, the optimizer should consider distribution in all phases. Most stages of this method can be reused for such an end-to-end optimizer, and we can create distributed plans to conduct experiments early. Our work on this rule-based plan generation is mostly done.

4.2. Computation Time Estimation

Traditional single-node query optimizers rely on relatively simple cost models because cardinality estimation errors outweigh the effect of more detailed models [12]. For distributed processing, however, we need to compare the relative cost of data transfers over the network to local computation to find good execution strategies. In the presence of fast modern networks, it is no longer sufficient to simply rely on the sizes of intermediate results and ignore the computation time. For an exact comparison, we want to accurately predict the computation time of pipelines. We build a fine-grained operator-based cost model to predict the average computation time required for each tuple at each operator. The profiling method proposed by Beischl et al. [13] provides very detailed data for modern compiled data processing engines. We use this data and available information about each operator, such as tuple size, input cardinality, and expression complexity, to create a detailed performance model. Optimization stages can consider the performance estimations of this model if they are included in the query plan. We have yet to implement this method.

4.3. Task Assignment Optimizer

Single pipelines can be distributed using data-parallelism. If the scanned data unit is partitioned into n partitions, we create n tasks for this pipeline, where each task scans one partition and outputs one partition of the pipelines result.

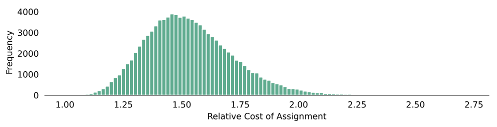


Figure 2: Cost distribution of 100 thousand sampled task assignments for TPC-H Q21 on 16 machines.

The task assignment optimizer focuses on the choice of which node should execute which tasks. Each node can execute any task, if the necessary data is transferred accordingly. However, this will have significant impact on performance. We want to evenly spread the computational load among nodes and minimize time spent waiting on data transfers. Each assignment also has an effect on subsequent execution, as it determines on which node the resulting partition will reside.

As depicted in Figure 2, good task assignments can improve performance over 2x. The number of possible assignments $n_{\text{assign}} = n_{\text{nodes}}^{n_{\text{tasks}}}$ grows exponentially in the number of tasks, which renders exhaustive enumeration of assignments infeasible. Using our computation time estimates and estimated time spent for data transfers, we plan to build a heuristic optimizer for the task assignment problem that is able to generate good assignments in short time. We will consider sampling-based and greedy methods to find good initial plans in short time and refining methods like iterative improvement and simulated annealing to further improve these plans. We have implemented first approaches to this problem.

4.4. Distributed Execution Simulator

The best way to evaluate optimizations is to conduct benchmarks on a real system. However, it is intricate to conduct thorough large scale benchmarks on distributed systems. Experiments on large compute clusters are expensive and take substantial effort to realize with a work-in-progress system. Hence, we decided to simulate the distributed system without the need for a real implementation. This simulator is much more flexible, as we can make fundamental changes to the execution model with little effort. Also, the structure of the simulated cluster can be changed in compute nodes count, hardware, and network speeds effortlessly. It can also be used as a direct cost function for cost-based optimization methods.

The simulator takes a distributed query plan, a cluster definition, each nodes cached partitions of base relations, and a task assignment as input. It maintains pending and currently active tasks and data transfers over the network and their current progress in percent. First, it computes the estimated remaining time to finish for each active task and transfer. The shortest time t_{min} determines when the set of currently running tasks and transfers changes.

The simulator advances the progress of all operations by $t_{\min}$. At least one of them will finish and therefore make a new partition available at some node. Finally, it finds all pending operations that can start since now new partitions are available. By accumulating all values of $t_{\min}$, we can compute the overall runtime of the query.

Our implementation of this simulator is ready for use. We use it to evaluate randomly sampled task assignments and give their execution time distribution in Figure 2. As our implementation is fast, it can easily simulate tens of thousands of executions per second and is hence suitable for direct integration in the optimization loop.

4.5. Join Ordering Optimizer

Not only the new problem of task assignment has optimization potential. Join ordering algorithms for single-node execution yield deficient plans for distributed execution [7]. As large base relations are likely to be hash partitioned on join keys, it will be advantageous to execute the respective join first and avoid reshuffling the data, even if that might not be optimal for single-node execution. Furthermore, the join ordering algorithm can directly choose the distribution (hash distributed, broadcast, or simply using only a single node) of intermediate results. We will investigate the feasibility of applying exhaustive dynamic programming algorithms that extend solutions by physical properties similar to SQL server PDW [1]. This will enlarge the search space significantly, and exhaustive search will be infeasible in many cases. Hence, we will work on further possibilities to restrict the search space and gracefully fall back to fast approximations. Additionally, we will work on a new cost model for enumeration algorithms which incorporates both computation and network time. We have not yet started working on this problem.

5. Conclusion

Distributed query processing opens potential for optimizations at many different stages of physical plan generation. This work proposes approaches to use that potential in several ways. We describe a way to lift current physical query plans for distributed execution. Then, we create a simulation-based evaluation method for these plans. We highlight the importance of the task assignment problem and sketch several methods to find good assignments. Finally, we present our vision for new enumeration-based join ordering algorithms that jointly optimize the distribution of data with the join ordering.

References

[1] S. Shankar, R. V. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. J. DeWitt, C. A. Galindo-Legaria, Query optimization in microsoft SQL server PDW, in: SIGMOD Conference, ACM, 2012, pp. 767–776.

[2] J. Aguilar-Saborit, R. Ramakrishnan, POLARIS: the distributed SQL engine in azure synapse, Proc. VLDB Endow. 13 (2020) 3204–3216.

[3] N. Armenatzoglou, S. Basu, N. Bhanoori, et al., Amazon redshift re-invented, in: SIGMOD Conference, ACM, 2022, pp. 2205–2217.

[4] B. Dageville, T. Cruanes, et al., The snowflake elastic data warehouse, in: SIGMOD Conference, ACM, 2016, pp. 215–226.

[5] G. Graefe, The cascades framework for query optimization, IEEE Data Eng. Bull. 18 (1995) 19–29.

[6] N. Tran, A. Lamb, L. Shrinivas, S. Bodagala, J. Dave, The vertica query optimizer: The case for specialized query optimizers, in: ICDE, IEEE Computer Society, 2014, pp. 1108–1119.

[7] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, M. Andrews, The memsql query optimizer: A modern optimizer for real-time analytics in a distributed database, Proc. VLDB Endow. 9 (2016) 1401–1412.

[8] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, M. Zaharia, Spark SQL: relational data processing in spark, in: SIGMOD Conference, ACM, 2015, pp. 1383–1394.

[9] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, T. Neumann, Locality-sensitive operators for parallel main-memory database clusters, in: ICDE, IEEE Computer Society, 2014, pp. 592–603.

[10] W. Rödiger, S. Idicula, A. Kemper, T. Neumann, Flow-join: Adaptive skew handling for distributed joins over high-speed networks, in: ICDE, IEEE Computer Society, 2016, pp. 1194–1205.

[11] M. Soualhia, F. Khomh, S. Tahar, Task scheduling in big data platforms: A systematic literature review, J. Syst. Softw. 134 (2017) 170–189.

[12] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proc. VLDB Endow. 9 (2015) 204–215.

[13] A. Beischl, T. Kersten, M. Bandle, J. Giceva, T. Neumann, Profiling dataflow systems on multiple abstraction levels, in: EuroSys, ACM, 2021, pp. 474–489.