But wherever there is man, there must be some sort of route.

— Robert Edison Fulton Jr.

9

Query Routing

In a PolyDBMS, data can be replicated and partitioned across multiple data stores. To execute a query, the PolyDBMS needs to plan the execution by decomposing the query and selecting which data store has the best characteristics for executing the query or parts of it. This process of planning the execution of a query in terms of the optimal utilization of the available data stores is referred to as *query routing*.

What makes query routing complicated is that the execution time of a query on a data store depends on several factors, such as the amount and type of data stored or the performance of the system on which the data store is deployed. In addition, these factors may also change over time. This requires the query routing to be adaptive and to adjust to the conditions of the environment and the use case.

Simply speaking, query routing is about transforming the scan operators of a query plan. This includes resolving the partitioning and modeling the access to the individual logical partitions, as well as deciding from which data store the data should be read. In this chapter, we introduce our *four phase query routing* approach. As the name implies, it consists of four phases:

The first phase is the *resolving phase*. In this phase, the data entities are broken down into their individual logical partitions. The PolyDBMS thereby analyzes the query and determines which of the partitions must be considered for the execution of the query.

In the *parameterization phase*, the query is transformed into a form that allows it to be reused for similar queries. This allows subsequent routing and query processing steps to be cached. Furthermore, the identification of similar queries is also important for the selection strategy.

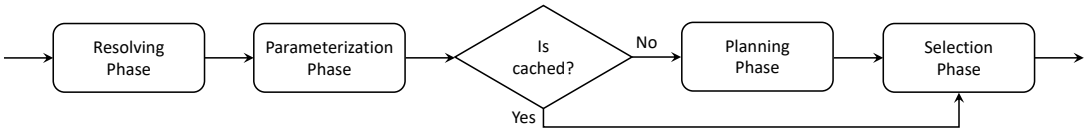

Figure 9.1 The four phases of the introduced query routing model. As depicted, the time-consuming *panning phase* is cached for similar queries.

In the *planning phase*, candidate plans are generated by selecting for each of the scan operators, on which data store it should be executed. This phase results in a set of query plans.

The final *selection phase* it is selected which of the previously generated candidate plans should be used to execute the query.

Figure 9.1 depicts the four phases. The caching allows to skip the time-consuming planning phase if a similar query has already been planned. In the following sections, these phases are elaborated in more detail.

9.1 Resolving Phase

In the resolving phase, data entities are resolved to the required set of corresponding logical partitions. As outlined in Section 7.3.1 all data entities have a partition function assigned. Thus, all data entities are inherently partitioned. However, due to the NONE partition function, a data entity can consist of only one partition. Since the horizontal partitioning is defined on the logical data entity, the resolution is independent from which data store(s) the partitions are read.

Figure 9.2 depicts the transformation of a query plan done in the resolving phase. The query in this example selects two users (the one with the ID 34 and the one with the ID 2584). In this example, one user record is stored in the partition 0 and one in the partition 2. Both partitions are therefore unioned together.

The scan operators for the individual partitions contain a list of all available placements (i.e., allocations to a data store) of this partition. Depending on the freshness requirements of the query (see Section 7.3.2), this list also contains outdated placements which—at this point in time—meet the the requirements. Furthermore, it also contains a list of all *vertical partition entities* (see Equation (7.19)) available at this placement.

The obtained query plan depicted in Figure 9.2 is not optimized. For instance, it would probably be beneficial to move the filter bellow the union operator in order to reduce

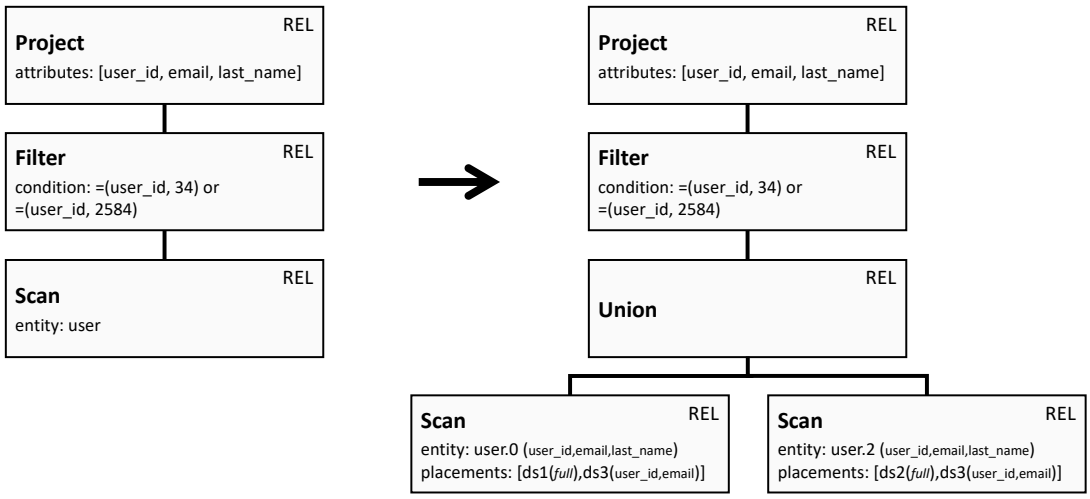

Figure 9.2 This example shows a query retrieving to user accounts by their IDs. In the resolving phase, it is identified that the query access data spread across two partitions. The corresponding partition are therefore unioned together. Furthermore, the scan operations on the individual partitions contain a list of all available placements of this partition.

the amount of data that needs to be unioned. However, since it has not yet been decided from which data stores the data will be read, the potential for optimization is rather limited at this stage.

The required set of partitions is determined based on the conditions in the query plan. Since these conditions can contain function calls or conditions that cannot be evaluated without partially executing the query, the fallback is to union all partitions and to rely on subsequent optimization steps of the PolyDBMS and the underlying data stores.

9.2 Parameterization Phase

The routing of queries and the subsequent steps in the query processing can be quite complex. This complexity can introduce a significant overhead that is especially problematic for short running transactional queries. To reduce the amount of work that needs to be done for every query, parts of the query routing and processing should be cached and only done once for *similar* queries.

We define the similarity of queries based on the structure of the query plan. This is based on the observation, that especially transactional workloads produced by applications are generated based on templates. The application typically derives queries from these templates by replacing defined placeholders with literals. By removing literals from the

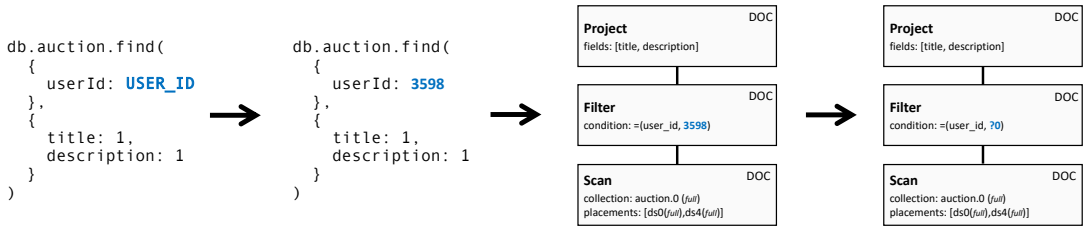

Figure 9.3 The template of a MongoDB query is used to retrieve the title and description of the auction with the ID 3598. As depicted, this results in a query plan with this ID in the filter condition. By replacing the literal 3598 with a parameter, we obtain a query plan that can be reused for all queries retrieving an auction that is stored on this partition.

query plan, we get a query plan which can be reused for queries with a different set of literals.

The process of removing literals from the query plan is called *parameterization*. In this process, all literals in the query plan are replaced by placeholders. Figure 9.3 depicts the parameterization for a simple query plan originating from a MongoDB query. The parameterization only replaces literals. Operators, functions or identifier names are not substituted.

Since the parameterization is done on the query plan obtained from the resolving phase, it results in a generic query plan that can be reused for all queries where the same set of partitions is accessed; thus, where the literals (e.g., the user ID in the example) belong to the same set of partitions. Together with the list of placements of the respective partitions available at this point in time, all information needed for the next routing step is encoded in this generalized query plan.

This approach results in a robust and easy to maintain solution for caching the time-consuming planning phase. Since the list of available placements of a partition added to each scan operator in the previous phase are part of the query plan, adding or removing placements will result in a different query plan and thus a new planning. This also covers data freshness: if a placement no longer meets the requirements of a query, it will not be added to the list of placements in the resolving phase.

9.3 Planning Phase

The planning phase is the most complex phase of the four phase query routing. In this phase, the query plan expressed in the logical PolyAlgebra introduced in Section 8.1 is implemented using the physical algebra (see Section 8.3). If there are multiple place-

ments of a logical partition on different data stores, the planning needs to decide from which data store, the data should be read (i.e., where to execute the scan operator).

The decision on which data store a scan operator is executed determines whether other operators can also be pushed down. As outlined in Section 8.3, *push down* refers to the execution of operators on the underlying data store instead of the internal engine of the PolyDBMS. Since pushing down operators avoids streaming the whole data entity to the internal engine of the PolyDBMS, it usually results in significantly better performance.

The decision on which data store to execute the scan operator therefore influences the potential for pushing down additional operators. For an operator to be pushed down, this operator and all other operators between this operator and the scan operator need to be supported by this data store. Since not all data stores provide the same set of operators, the decision on which data store to execute a scan operator can therefore have a crucial impact on the performance of the query.

As already mentioned for data modification queries in Section 8.3, it would theoretically be possible to implement individual operators on a data store by streaming the data from and to the data stores. However, due to length limitations of query languages, this option is not feasible. Hence, for an operator to be pushed down on a data store, all previous operators starting from the scan operator need to be pushed down as well.

In order to come up with an optimal plan for executing a query, the capabilities and the expected performance of the data stores need to be considered. Especially for heavily replicated and partitioned data entities, finding such an optimal plan can be complex. However, due to the parameterization of the query plan, this phase can be cached. This is especially important for short running transactional queries where the the total execution time should be a few milliseconds.

9.3.1 Query Decomposition

There are data stores with different sets of features and capabilities. While a data store might excels for some types of data and workloads, it performs bad for other types. Since database queries can be arbitrarily complex, there might be no single data store suitable for it. The idea of query decomposition in a PolyDBMS is to split the query into multiple parts that are then executed on different data stores. This allows to utilize and combine the strength of different execution engines.

Using the query representation introduced in this thesis, decomposing queries is fairly simple. Since every read-only query plan has scan operators on all of its leaves, a query can be decomposed by implementing the scan operator using operators from different data stores. In the subsequent query optimization, supported operators are pushed down, resulting in a decomposed query.

9.3.2 Vertical Partitioning

As outlined in Section 7.3.1, data can be partitioned horizontally and vertically. The horizontal partitioning has already been resolved in the first phase. The vertical partitioning, however, has not yet been dealt with by inserting the required union operations. How it is dealt with vertical partitioning depends on the data model. In the three previously introduced data model, vertical partitioning results in some sort of join operation that "glues" the distributed parts of a record, document, node or edge together.

As part of the resolution phase, every scan operation has been extended with a list of data placements including their vertical partition entities and a list of required vertical partition entities. In the planing phase, these lists are used to determine an optimal combination of multiple partitions. Since vertical partitioning always results in multiple data stores being involved, these join operators cannot be pushed down. The routing therefore tries to first deal with the horizontal partitioning and implement as many operators as possibles before addressing the vertical partitioning. The rules for merging and splitting operators and changing their order is the subject of query optimization. Since there are no fundamental differences to "traditional" database management systems, we do not go into more detail. How the query optimization is done in our implementation Polypheny-DB is described in Chapter 10.

9.3.3 Static Planning Rules

One approach for selecting where (i.e., on which data store) to execute a scan operator and to find a valid execution plan for data spread across different data stores is by means of a static set of rules. This set of rules can be extended based on the use-cases and the requirements of the scenario. Every rule can return one or multiple candidate plans.

An obvious approach is to select based on the data model of the scan operator and prefer native data stores. This increases the chance of pushing down additional operators. Furthermore, native data stores are expected to perform better than data stores based on a different data model.

Another approach is to identify the type of workload and select the data store based on the identified workload. This is especially interesting for simple queries where everything can be pushed down to one data store.

The third approach is to decide based on the usage of certain features (e.g., nearest neighbor search or geospatial functions). If a query contains such a special function, it should be preferred to utilize a data store with native support for this feature.

The fourth rule deals with the latency of the communication between the data store and the PolyDBMS. It prefers data stores with a low latency (e.g., those deployed on the same machine as the PolyDBMS). This is especially beneficial for simple, short running queries where the communication overhead quickly exceeds the execution time.

Every candidate plan proposed by the planning rules is optimized in a subsequent step. This includes identifying operators that can be pushed down to the underlying data stores, merging and splitting operators and adjusting their order.

The advantage of the rule-based approach is, that it is very fast. The rules can be applied in parallel and result in a set of query plans. However, it is not very adaptive. While it is possible to adapt the rules at runtime, especially with complex queries, this approach reaches its limit.

9.3.4 Cost-based Planning

A major disadvantage of the static planing rules is the split between the decision where to execute a query and the optimization of the query. Merging these two steps would allow to select the data store according to the potential optimizations its choice enables.

By using a cost-based query optimizer, this can be modeled. However, the difficulty is to design the cost-model. This cost-model must assign a costs to every operator of the physical algebra. The ability to dynamically adjust this cost model results in a highly adaptive approach for query planning. However, building and adjusting this cost model is quite difficult.

Optimally, the cost model is designed such that the cost for a query plan implemented using operators from store A and a query plan implemented using operators from store B reflect the the relative difference between the actual execution time of the two query plans. Since the execution time of a query plan depends not only on the data store but also on the environment and the data, a static cost model has its limitations. We are therefore working on an approach for learning the cost model based on the previously

executed queries. While the first results are promising, this is subject to future work. But even with a static cost model, we achieved good results.

The most important factor is, that the query optimizer takes the amount of data that needs to be streamed to the internal engine of the PolyDBMS into account. This results in query plans that maximize the push down of operators to the underlying data stores. However, the disadvantage of this approach is the required time to find an optimal query plan, especially with a large number of placements. Furthermore, it is difficult to design a cost model for simple transactional query plans since for such queries aspects like the latency for communication between the PolyDBMS and the data store exceeds the cost for actually executing the query.

9.3.5 Combining Both Approaches

Both approaches have their advantages and weaknesses. The integration of the data store selection and planning into the query optimization results in good query plans. With a learned cost model, it is also very adaptive. However, it is time-consuming. Furthermore, simple query plans are challenging since network latency and other aspects are more important than the actual execution time on an underlying data store.

While we are currently working on integrating these aspects by learning and adjusting the operator costs at runtime, this introduces another issue: In order to learn the operator costs, enough data points need to be available. After significant changes to the data allocation and the set of data stores, the PolyDBMS suffers from an outdated cost model. Furthermore, the cost-based optimization is time-consuming if a large number of placements needs to be considered (e.g., with partitioned data entities).

Our approach is to combine the cost-based planning with the static routing rules introduced before. These rules are especially strong for simple, short running queries and can be adjusted to the scenario. They are furthermore very fast.

The idea is to identify the type of query: simple and potentially short running or complex and potentially long-running. For simple queries, only the static routing rules are used and the resulting query plans are optimized. For long-running queries, both the routing rules and the cost-based planning approach are being used.

In both cases, we obtain a set of potential query execution plans. Duplicate plans are removed. These plans are then cached for subsequent queries which are similar to this one.

9.4 Selection Phase

The selection phase is the final phase of the four phase query routing. Its purpose is to select an execution plan from the set of candidate plans generated by the planning phase. While the planing phase is cached for subsequent similar queries, the selection phase is applied to every query. The reason for this split of planning and selection phase is the mentioned caching of the planning phase: the dedicated selection phase allows to make a final decision on the query plan based on the current state of the system and by taking previous executions or updates to the cost model into account. This avoids constantly re-planning query plans.

The selection is done using *score functions*. These score functions allow to select which of the previously proposed execution plans is optimal given the current state of the system and collected knowledge about previous executions of similar queries.

In the following sections, we outline multiple score functions. Every score function $s(\cdot)$ returns a value between zero and one. Furthermore, there is an adjustable weight w for every score function. The score S of an execution plan q is calculated as

$$S := \sum_{i=1}^{n} w_i \cdot s_i(q) \tag{9.1}$$

with n being the number of score functions. Instead of always executing the plan with the highest score, it has shown advantageous to normalize the scores and use them as probabilities. This makes things even more adaptive, especially with the *previous execution time scoring function*. The plan with the highest score is selected the most, while plans with lower scores are less likely to be selected.

9.4.1 Previous Execution Time

A very powerful technique is to score the query plans based on the actual execution time of this execution plan. This is an approach we have first described in [VSS17]. It makes use of the introduced parameterization technique for identifying similar queries and the observation that queries are typically derived from templates.

This score function requires a subsystem in the PolyDBMS which constantly monitors the execution times and stores them together with the identifier of the chosen execution plan. This data is then used by this score function.

The average previous execution time of an execution plan is calculated by normalizing the average execution times. If there are no times for an execution plan, the maximum score of 1 is assigned. Depending on the other scoring functions and their weights, this fosters the execution of those plans.

With a function $at(\cdot)$ that returns the average execution time of an execution plan or zero if there are no previous executions, the scoring function s_{pev} for an execution plan q of the set of candidate plans Q is given as

$$s_{pev}(q, Q) := 1 - \frac{at(q)}{max(\{at(q_i) \mid q_i \in Q\})} \tag{9.2}$$

This approach is very adaptive, especially if only recent execution times are considered for calculating the average execution times (i.e., using a sliding window approach). An issue of this scoring function are parallel workloads. The obtained values are influenced by parallel executions on the data store. However, we realized that is also an advantage in scenarios with relatively constant workloads. In such scenarios, it perfectly depicts the actual execution times that could be achieved on this data store and thus also results in some sort of load-balancing between the data stores.

9.4.2 Operator Cost Model

This scoring function uses the operator cost model introduced in the planning phase to calculate the cost of a query plan. With a function $c(\cdot)$ that calculates the cost of an execution plan based on its operators, the scoring function s_{cost} for an execution plan q of the set of candidate plans Q is given as

$$s_{cost}(q, Q) := 1 - \frac{c(q)}{max(\{c(q_i) \mid q_i \in Q\})} \tag{9.3}$$

Using the cost-model in the selection phase makes the query routing more adaptive since changes to the cost-model influences the routing even for cached execution plans.