Trees sprout up just about everywhere in computer science.

— Donald E. Knuth

8

Query Representation

A PolyDBMS is inherently a multimodel system that must deal with multiple data models at different stages in the query processing and execution process. According to the definitions introduced in Section 3.1, a PolyDBMS needs to accept queries expressed in query languages based on different data models, internally support multiple data models, and maintain storage and execution engines based on different data models.

A challenge in such a system is the representation of a query in a form that can be processed and enables cross-model queries. The combinatorial complexity of multiple query languages, data models and data stores requires a single form of representation. As outlined in Chapter 4, queries can be represented as operator trees. The operators are defined by the data model. Since the approach to represent a query needs to preserve the semantics of the data model the query language is based on, mapping every query to one data model (for instance the relational data model) is not feasible.

The query representation model introduced in this chapter allows to represent queries based on different data models. It builds upon the schema model introduced in Chapter 7. Similar to this schema model, this query representation model is also introduced for the three data models introduced in Chapter 4.

The Figure 8.1 depicts the various data models that can be found in a PolyDBMS. According to the requirements of a PolyDBMS specified in Section 3.1.2, every PolyDBMS needs to support multiple query languages. Since every query language defines its own set of operators, each language can be considered as an individual data model. In Figure 8.1, these are depicted as *IQL*. All IQLs are translated to one of the algebras defined by the native data models of the PolyDBMS. Native thereby refers to data models supported by the PolyDBMS according to the definition provided in Section 3.1.2 of the PolyDBMS requirements.

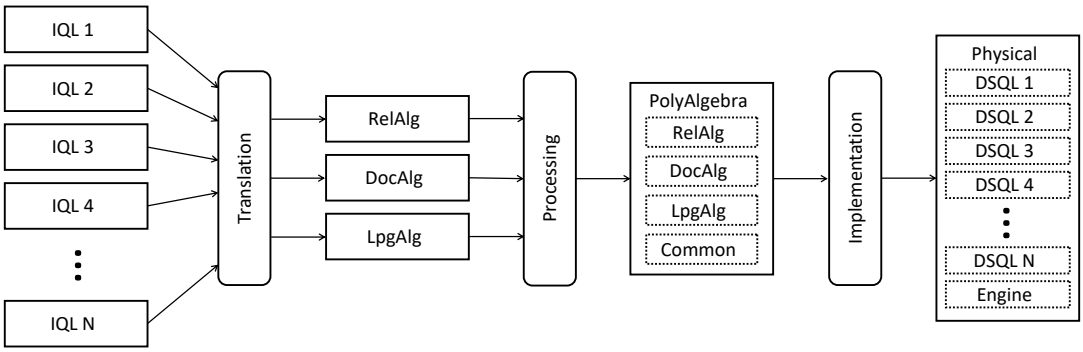

Figure 8.1 The data models used to represent queries through various stages. *IQL* are the data models of the input query languages, the query languages supported by the PolyDBMS (e.g., SQL). *DSQL* are the data models of the query languages of the underlying data stores. *Engine* is the data model of the integrated execution engine (see proposed architecture model).

In the processing stage, the single algebras are represented using an algebra that is constructed as a superset of the data model specific algebras. This algebra is called *PolyAlgebra*. Besides the model specific operators, it also contains a set of special operators for handling DML queries and modeling the enforcement of constraints. Furthermore, the views introduced in Chapter 7 can be expanded in this stage.

In the final step, the PolyAlgebra is converted into a representation based on the data store specific algebras and the operators of the integrated execution engine of the PolyDBMS (see architectural model in Section 3.2.3).

In this chapter, we introduce this PolyAlgebra and outline how the enforcement of constraints and the consistent handling of data modification queries can be expressed using this form of query representation.

8.1 The PolyAlgebra

With the PolyAlgebra, we introduce an approach to represent queries originating from different query languages and based on different data models using one algebra. This algebra consists of the algebras defined by the data models supported by the PolyDBMS and thus contains multiple operators for similar concepts originating from different data models. This is different from the typical approach of defining an algebra based on a coherent set of operators; which we will refer to as *unified approach*. In a PolyDBMS, this requires similar operators from different data models to be merged into one operator that encompasses the semantic of the data model specific operators. However, we see

three strong arguments against the unified approach and in favor of our PolyAlgebra approach:

Firstly, the *preservation of the data model specific semantics of the operators*: A unified approach comes with the risk of implying a specific data model, since the dependencies and the meaning of a certain combination of operators can be different between data models. Ensuring the semantic is preserved through all stages of query processing and optimization is challenging.

Secondly, the *mapping from and to the algebra*: If a query language is based on a data model similar to one of the data models supported by the PolyDBMS, the operators of the query language can be mapped to the operators defined by the data model of the PolyDBMS without a sophisticated logic. With a unified approach, the mapping is much more complex since the operators themselves are much more complex in order to encompass the different semantics. Furthermore, this complex mapping needs to be done individually for every supported query language and data store, duplicating and spreading logic across the PolyDBMS which is also suboptimal from an implementation perspective.

Thirdly, the *expandability*: This is much easier in the PolyAlgebra approach than in the unified approach. If the PolyDBMS is extended to support an additional data model, an additional set of operators can be added to the PolyAlgebra without any modifications to the operators of other data models. In the unified approach, such an extension might require complex changes to the operators to encompass the additional semantic and behavior. These changes then in turn require changes to all adapters and query language translators.

However, including the full set of operators from all supported data models in the PolyAlgebra has one main disadvantage: it results in a larger set of operators that need to be handled. However, due to the reduced complexity of the operators, this can also be seen as an advantage.

For the algebras composing the PolyAlgebra, we refer to the operators introduced in Chapter 4. However, we have not introduced an approach for specifying on which data the operators should be applied on. Similar to Section 7.3, we define a data entity E representing the object of a data model that represents the data:

$$E := \mathcal{T} \vee \mathcal{C} \vee \mathcal{N}^{LPG} \vee \mathcal{V}^{REL} \vee \mathcal{V}^{DOC} \vee \mathcal{V}^{LPG} \tag{8.1}$$

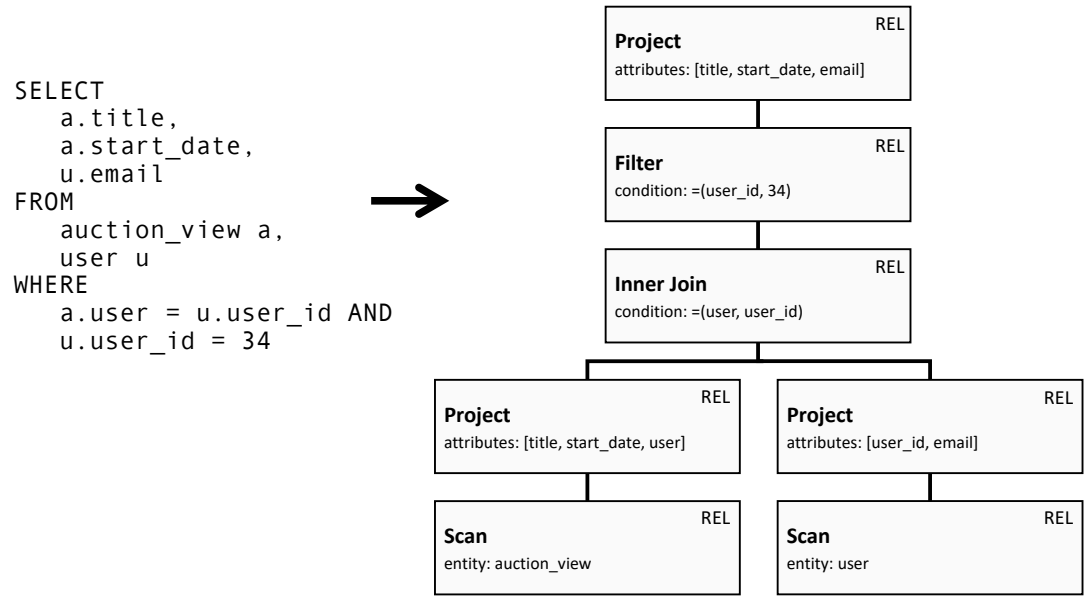

Figure 8.2 An example of a query plan consisting of relational operators. It represents the SQL query depicted on the left side.

In namespaces of the type relational or document, the data is held by tables ($\mathcal{T}$) or collections (C). In namespaces of type LPG, the data is held by the namespace itself $\mathcal{N}^{LPG}$. Furthermore, data is also represented by views $\mathcal{V}$.

Since databases usually consist of multiple such data entities, a special operator is required to select the data entity holding the data the operators should be applied on. This operator is called *scan operator*. There is one such operator for every supported data model, taking a schema object E as parameter and returning the data as specified by the corresponding data model.

The LPG algebra has two result representations: *graph* and *graph relation*. This is required since the *match* operator needs to operate on the whole graph in order to match paths and build a graph relation. The LPG node operator introduced in Section 4.3 is thereby subsumed by the match operator. It therefore takes a graph as input and returns a graph relation as output. The union operator is defined for both graph and graph relation. All other LPG operators solely operate on graph relations.

Besides the scan operator there is also the *values* operator. This operator can be seen as a special form of the scan operator which instead of a data entity directly specifies the data. This for instance allows to model *insert* operations; see Section 8.2.

With the scan and the values operator, queries can be represented as operator trees. Such a tree is depicted in Figure 8.2. Every valid tree must either have a values or a scan operator at each of its leaves. These operator trees are called *query plans*.

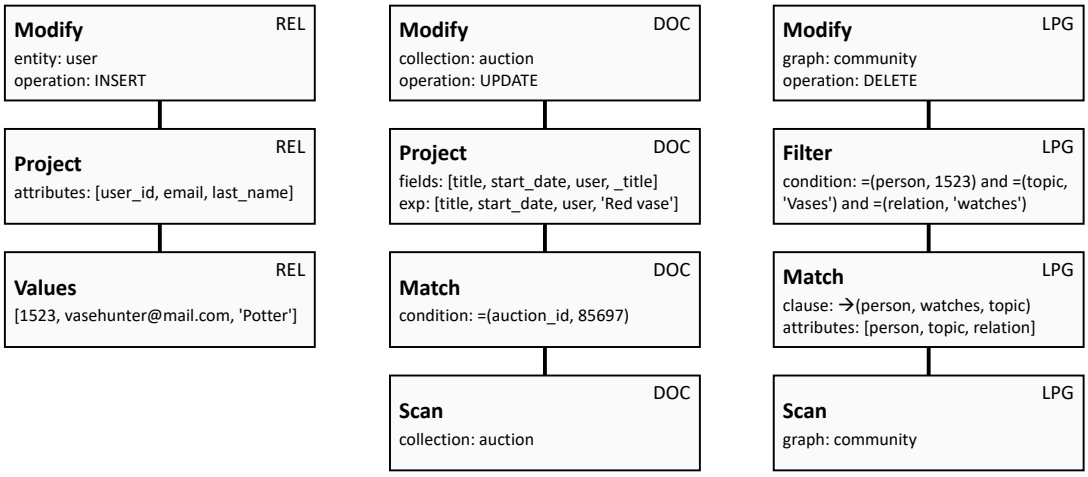

Figure 8.3 Structure of different data manipulation operations represented as query plans using the PolyAlgebra. The left plan shows a relational insert query, the plan in the middle an update of a collection in a document model, and the one on the right depicts a plan for deleting an edge in an LPG graph.

8.2 Representation of DML Queries

The algebras introduced in Chapter 4 only specify approaches for querying the data, but not for manipulating it. We therefore introduce the *modify* operator. This operator allows to modify the data hold by a data entity, according to the specified operation. The modify operator is a unary operator, taking the records to be dealt with as input and returning an integer indicating the number of inserted, updated or deleted data items as output.

For insert operations, the input defines the data to be added to the schema object while for delete operations, it defines the records to be deleted. For update operations, the input to the modify operator specifies both, the current values of the records to be updated and the new values. Figure 8.3 depicts the structure of different data modification operations. The query plan beneath the the modify operator can be arbitrarily complex. This, for example, allows to copy data from other data entities or model complex conditions.

8.3 Physical Algebra

The selected architecture model for a PolyDBMS makes use of multiple heterogeneous execution engines to process queries. Primarily, it can be distinguished between the internal execution engine of the PolyDBMS and the underlying data stores. Determining the optimal strategy for executing queries (i.e., decomposing the query and selecting

where the subqueries should be executed), is called *query routing* and will be discussed in Chapter 9. In this section, we discuss the resulting implications and requirements for representing query plans across multiple execution engines.

As mentioned before, every underlying data store provides its own set of operators. Together with the operators of the internal execution engine, these operators form the *physical algebra*; see Figure 8.1. While the PolyAlgebra provides a logical representation of a query, the physical algebra describes where and how a query is executed. It is an intermediate representation which is used by the PolyDBMS to implement the query on the underlying data stores using their native query methods (i.e., query language). Furthermore, it contains the instructions for the internal execution engine.

Database systems typically do not expose their internal query representation, but expect queries to be expressed in a query language (e.g., SQL). The PolyDBMS must therefore interact with its underlying data stores using this query language. The operators that comprise the physical algebra are therefore defined based on the structure and operators of this query language. A convenient side effect is that data stores supporting the same query language can use the same or a similar set of operators.

Every data store needs to provide support for a scan operator that reads and streams the whole data of a specified data entity. For modifying data, each data store furthermore needs to provide support for a data modification operator, a values operator, a basic filter operator, and a basic project operator. Basic in this case means, that only a small subset of the functionality is required. The filter operator needs to support an equals comparison off a field or attribute with a provided value. The project operator needs at least support removing attributes or path and adding attributes or path with a specified value.

To meet the independence of storage configuration requirement demanded by in Section 3.1.3, the integrated engine must provide full support for all operators supported by the PolyDBMS except for the *scan* and the *modify* operator. These two operators interact with the data. The internal engine required by the architecture model of a PolyDBMS however does not store any data. It is only used to process queries spanning multiple data stores and to compensate for missing features. Nevertheless, it would be possible to also store data in this internal engine. In this case, it would also need support for a scan and modify operator.

Since reading a data entity from an underlying data store and streaming the whole data into the internal execution engine of the PolyDBMS is expected to be less efficient than executing the query on the underlying data store, the PolyDBMS makes use of additional

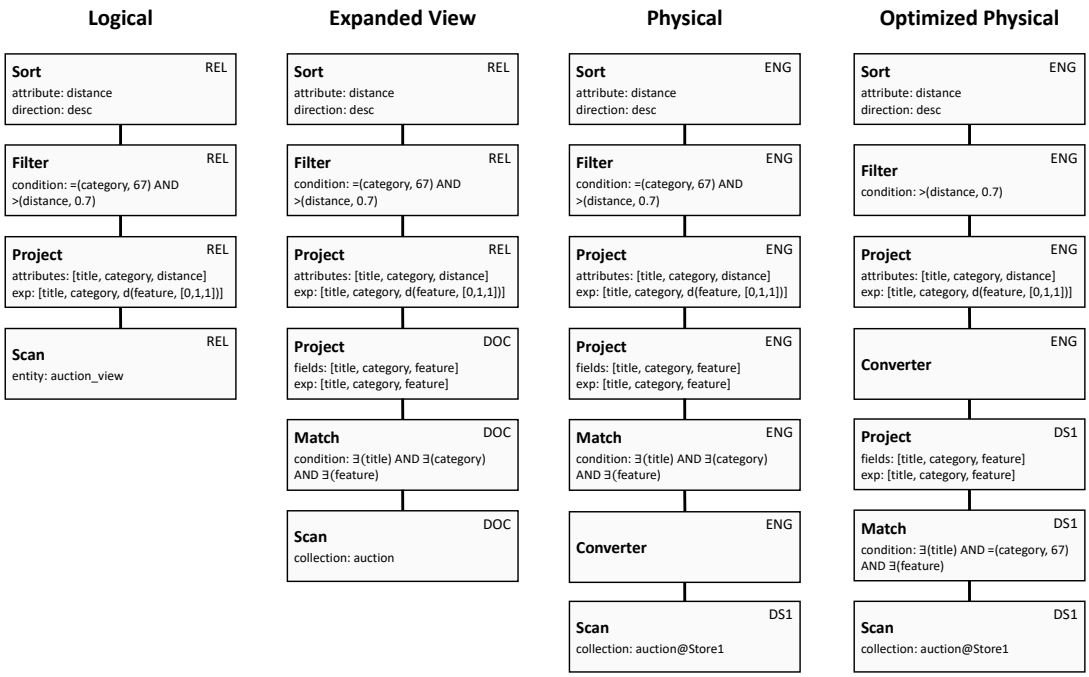

Figure 8.4 Different representations of the same cross-model query.

operators provided by the data stores. This avoids IO heavy data streaming and makes use of the optimizations of the data store (see Section 3.2.3). This execution of an operator on an underlying data store is referred to as *push down*.

Figure 8.4 depicts the various representations of a query that retrieves all auctions belonging to the category with the ID 67 and having at least a certain similarity to a specified vector. This similarity is calculated using the distance function d. This function returns the Euclidean distance between two vectors. The depicted logical plan on the left side is the result of a query based on the relational model (e.g., SQL). Since *auction* is a collection of documents, there are two approaches for querying it from a relational context (see Section 7.1.4.3): directly or by means of a view. In this example, the query operates on a view that has been defined using a query language based on the document data model. Thus, the query plan obtained when expanding the view contains operators from the relation algebra and from the document algebra.

To execute this query plan, it needs to be implemented using operators of the underlying data stores and the integrated execution engine. In this example, we assume that the auction collection is neither replicated nor partitioned. Hence, there is only one data store holding the data. In Figure 8.4 the operators of this data store are labeled as *DS1* while the operators of the integrated execution engine are labeled as *ENG*.

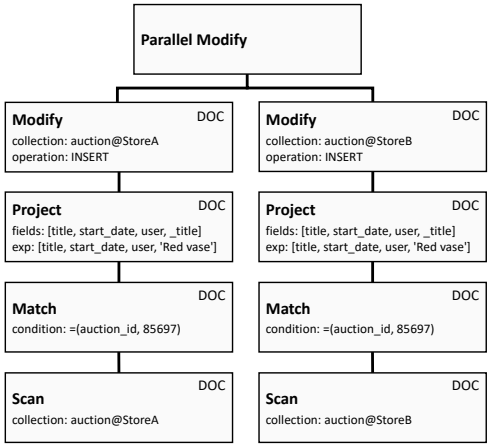

Figure 8.5 An exemplary query plan representing the execution of an insert on two data stores.

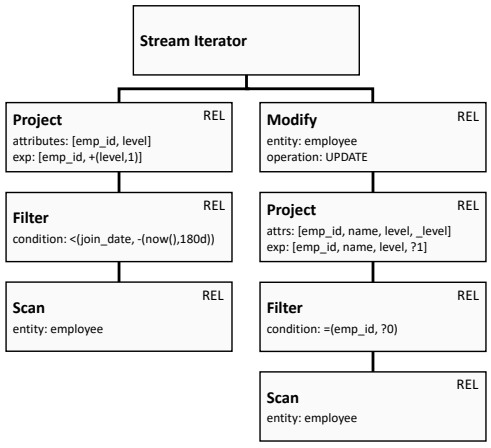

Figure 8.6 A complex update statement on a data store that does not supported the required operations.

The third query plan depicted in the figure is not optimized at all. The whole data of the collection is read from the data store and converted to the internal representation of the PolyDBMS. All operators are implemented using the internal execution engine. To improve the performance, in the fourth plan, some of the operators are implemented using operators of the underlying data store. Since the underlying data store in this example does not support the distance function d, some parts of the query still need to be executed using the integrated engine. However, the amount of data that needs to be streamed to and processed by the internal engine of the PolyDBMS got significantly reduced due to the push down of operators to the underlying data store.

As outlined in Chapter 7, views can be defined using a query language based on a different data model. Since querying a view results in replacing the scan operator with the definition of the view, this can result in trees containing operators from different data models. In order to model the necessary mapping between data models, we introduce *converter* operators. These operators take a result represented according to one data model and convert it into another data model. The conversion is done according to the definitions provided in Section 7.1.4.

Since the schema model introduced in Section 7.3.1 allows data to be replicated across multiple underlying data stores, data modifications need to be performed on all data stores in order to keep the data consistent. To model this in one query plan, we introduce the *parallel modify* operator. This operator executes data modification queries simultaneously on multiple data stores. This is depicted in Figure 8.5.

As mentioned before, the algebra of a data store does only need to be able to represent the following five operators of the PolyAlgebra: scan, values, modify, basic filter and basic project. While the algebra of a data store might encompass additional operators, this is not mandatory. This causes issues with data modification operations.

Consider an update query that increases the "level" of every employee working for the company since at least 180 days. This query requires the data store to subtract and compare timestamps and increment an integer. These are operations not mandatory to be supported by every data store.

Theoretically, executing such queries on a data store without support for these operations is no issue since the internal engine of the PolyDBMS provides support for all operations and therefor compensates for missing functionality of the data stores. However, in practice, we run into issues with the length of the resulting query string (e.g., the resulting SQL string) used to interact with the data store. Executing the operations of the example query not supported by the underlying data store in the internal engine of the PolyDBMS would cause the query string to contain the individual value of the level attribute for every employee. Furthermore, a query changing multiple tuples to different values is rather complex and therefore might also exceed the capabilities of the data store.

We therefore introduce the *Stream Iterator* operator depicted in Figure 8.6. This operator is implemented by the integrated engine of the PolyDBMS. It iterates on the result received from the left input branch and invokes the right branch with these values as parameters. This allows the PolyDBMS to extract the unsupported operations and reduce a data modification query to a set of basic operations supported on every data store. Since the left branch containing the complex operation can be routed independently (see Chapter 9), it can be executed on a data store optimized for this type of query. The stream iterator can also be combined with the parallel modify operator. If supported by the underlying data stores, the actual execution can also be done in batches.

Besides unsupported operations, the stream iterator also presents a solution for handling data manipulation operations referencing data that is not allocated to a data store (e.g., due to partitioning of the schema entity). Furthermore, it enables the modification of data stored according to the schema mapping outlined in Section 7.2. This allows to meet the 'independence of storage configuration' requirement for arbitrarily complex data modification queries.

8.4 Enforcement of Constraints

As outlined in Chapter 7, the data model also defines the ability for specifying constraints to be imposed on the data governed according to this data model. It needs to be distinguished between two types of constraints: those applying to an individual data item (e.g., nullability constraints or the data type) and those applying to the data entity or the whole schema object. While the former type can easily be enforced, the latter is more challenging.

One approach to enforce constraints in a PolyDBMS is to delegate the enforcement to one of the underlying data stores supporting the enforcement of this type of constraint. However, this requires all affected data to be placed on this data store. Since not all data stores support the enforcement of constraints and this approach cannot be applied for partitioned data entities, other solutions are required.

In database systems uniqueness constraints are typically enforced using indexes. This is also a solution in a PolyDBMS. However, maintaining indexes on the PolyDBMS level introduces a significant overhead, especially with complex update queries. Correctly implementing such index updates is possible using the previously introduced stream iterator construct.

However, indexes are only feasible solution for enforcing existence-based constraints (uniqueness, foreign key) but they are not feasible for other kinds of constraints like check-constraints or the enforcement of a graph schema. These can only be enforced using a query that checks whether a query violates any constraints. The efficiency of this approach depends on the underlying data stores. The uniqueness of a certain value can for instance be checked very efficiently if there is a data store with a matching index. Since indexes are cheaper to maintain on an underlying data store than on the PolyDBMS level, this approach can also be efficient for enforcing existence-based constraints. However, this heavily depends on the data, the workload and the involved data stores.

Since all approaches have their strength and weakness, a PolyDBMS should consider all three approaches. For constraints that are enforced on query time, this requires a universal approach to express the enforcement as part of the query plan. We therefore introduce the *conditional execute* operator. This operator only executes the right branch iff the result of the left branch fulfills a specified condition. The left branch can be used to model all three approaches. Figure 8.7 depicts a plan enforcing the uniqueness of the foreign keys auction and user of the bid table in the context of an insert query. Instead

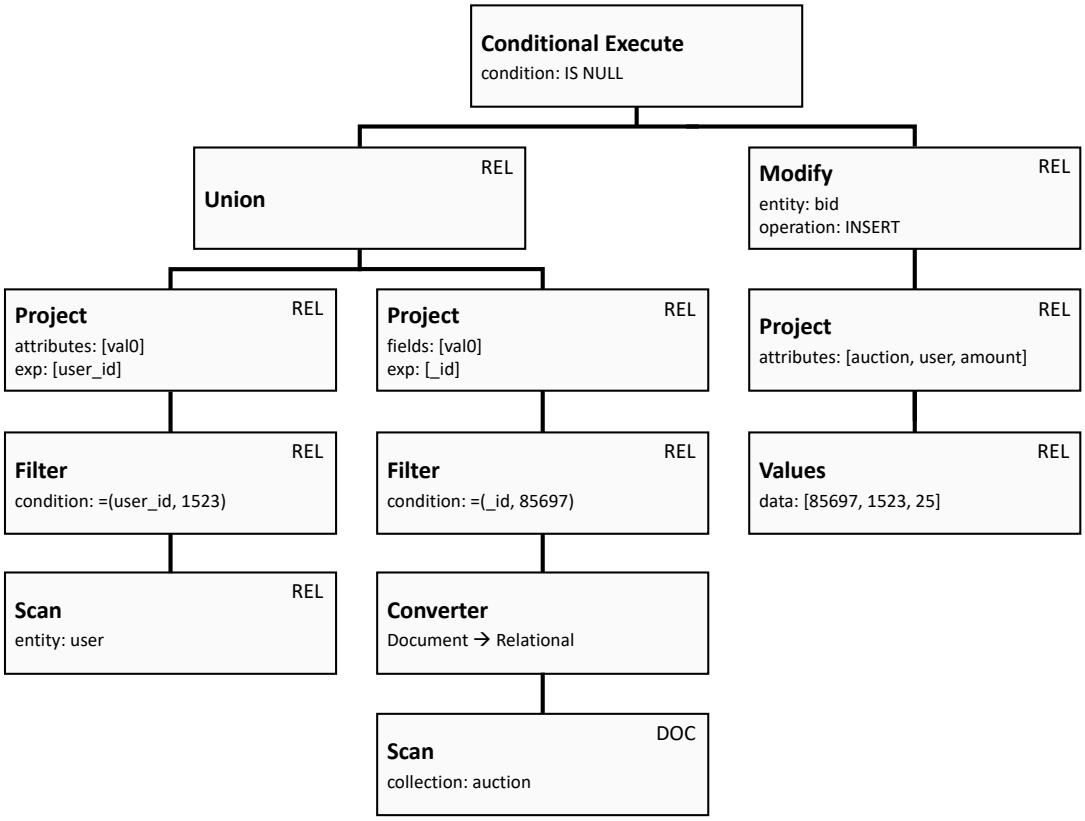

Figure 8.7 The conditional execute operator is used to enforce two foreign key constraints. Since auctions are stored using a document data model, the enforcement combines operators from different data models. The necessary converts would be added in the implementation step.

of the union construct, it would also be possible to stack multiple conditional execute operators.

The advantage of this construct is that the left part can be arbitrarily complex. This allows to represent even complex check constraints. Since everything is represented in one plan, synergies and redundancies can be eliminated. Furthermore, it is possible to combine different enforcement techniques.