At the time, Nixon was normalizing relations with China. I figured that if he could normalize relations, then so could I.

— Edgar F. Codd

7

Schema Model

The *schema* of a database system is the blueprint on how the data is structured in the database and defines how it can be accessed. The level of detail and the available building blocks for defining a schema depends on the data model (see Chapter 4). Since the schema is usually defined by a database administrator depending on the specific use-case and might also be subject of change over time, a database system provides the means for defining and altering schemas at runtime. In this chapter, a conceptual model for expressing and managing schemas in a PolyDBMS is introduced.

In Chapter 4, three data models have been introduced. These data models provide different levels of detail when it comes to defining schemas. The relational data model is very specific and enforces a strict schema. The document model is the opposite, enforcing no schema at all. In the LPG model, a schema is optional. Since a PolyDBMS needs to support multiple data models (see Section 3.1.1), a conceptual model for building schemas based on different data models and with various levels of detail is required (see Section 3.1.4). The schema model introduced in this chapter is based on the three data models already introduced in Chapter 4. However, the relational, document and LPG model are not only very popular and widely used database data models, they are also conceptually very distinct. Other popular data models like the wide-column, key-value, RDF or object relational model can be seen as subset of the combination of these three data models.

The discussion of this schema model is structured in three sections: The first section deals with the provided building blocks for modeling the *logical schema* and its properties. Furthermore, we define the mapping of schema between data models. In our schema model, the logical schema is the central schema of the PolyDBMS and the one against which queries are expressed. Since a PolyDBMS combines multiple underlying data stores based on different data models, a strategy is required for mapping the log-

ical schema to a *physical schema*. This strategy is introduced in the second section. In the third section we introduce a sophisticated model for allocating logical entities to underlying database systems.

7.1 Logical Schema

The logical schema is the central schema of the PolyDBMS. It is exposed to the user and represents the schema against which queries are expressed. Since a PolyDBMS supports multiple data models, and the requirements outlined in Section 3.1.4 require that there is only one logical schema, an approach is required for accommodating schema definitions based on different data models in one logical schema. We do this using a hierarchical approach.

The logical schema $\mathcal{S}$ of a PolyDBMS consists of multiple namespaces $\mathcal{N}$.

$$\mathcal{S} := \{\mathcal{N}_0, ..., \mathcal{N}_m\} \tag{7.1}$$

Every namespace $\mathcal{N}$ has a unique name and is of a specific data model $\mathcal{M}$ with $\mathcal{M} := \{REL, DOC, LPG\}$. The symbol $\mathcal{N}^{REL}$, for example, denotes a namespace of type relational. All names and labels introduced in this schema model must not contain dots ('.') and may not be empty.

An important element of this schema model is, that it defines two approaches for mapping data between data models: Firstly, there is a defined mapping for queries accessing schema objects of a namespace with a different data model. This is introduced in Section 7.1.4. Secondly, there is the notion of *views* for every data model. A view is a schema object that represents the result of an arbitrary, read-only query submitted in any of the query languages supported by the PolyDBMS. The difference between those two approaches is, that the first approach maps the data to the data model of the query language and then applies the query, while the view-based approach does the mapping on the result represented by the view. Views also allow fulfilling the multifaceted schema requirement outlined in Section 3.1.5.

7.1.1 Relational

A namespace of model relational $\mathcal{N}^{REL}$ consists of a set of tables and views. Tables and views have names which are unique within the namespace. There can also be no view

and table with the same name.

$$\mathcal{N}^{REL} := \left(name, \{\mathcal{T}_1, ..., \mathcal{T}_n\}, \{\mathcal{V}_1^{REL}, ..., \mathcal{V}_k^{REL}\}\right) \tag{7.2}$$

A table $\mathcal{T}$ corresponds to a relation in the relation model introduced in Section 4.1. However, in contrast to the set-semantic of relations which allows no duplicate tuples, we define tables as a bag of tuples. We furthermore refer to the tuples as records. A table consists of an ordered, non-empty list of columns $(COL_1, ..., COL_i)$ and a set of constraints $\{CSTR_1, ..., CSTR_j\}$. A table $\mathcal{T}$ can be described as a triple:

$$\mathcal{T} := \left(name,\ (COL_1, ..., COL_i),\ \{CSTR_1, ..., CSTR_j\}\right) \tag{7.3}$$

A column corresponds to an attribute in the relational model. Every column COL has a unique name within the table, a data type, a set of pairs with data type properties and a nullability information.

$$COL := \left(name,\ type,\ \{length : ...,\},\ nullable\right) \tag{7.4}$$

A constraint $CSTR$ has a name unique within the namespace and is of a specific type. Types are *primary*, *foreign* and *unique*. A primary key constraint $CSTR^{PK}$ specifies a set of columns of the table whose values uniquely identify each record in the table. The set of columns is represented by a key.

$$\begin{aligned} \mathcal{KEY} &:= \left\{COL_1, ..., COL_k\right\} \\ CSTR^{PK} &:= \left(name,\ \mathcal{KEY}\right) \end{aligned} \tag{7.5}$$

The uniqueness criterion imposed by this constraint is enforced by the PolyDBMS. The columns building the primary key must not be nullable (i.e., are not allowed to contain null values). Furthermore, there can only be one $CSTR^{PK}$ per table. To enforce uniqueness on a second set of columns, uniqueness constraints can be defined. There can be multiple uniqueness constraints $CSTR^{UNIQUE}$ for a table. In contrast to primary constraints, the key on which the unique constraint is defined may consist of columns allowing null values.

$$CSTR^{UNIQUE} := \left(name,\ \mathcal{KEY}\right) \tag{7.6}$$

Foreign key constraints expresses a relationship between two tables of a namespace. A foreign key $CSTR^{FK}$ is defined by a *local key* and a *remote key*. The local key specifies a set of columns in the table for which the constraint is defined. The remote key specifies

a key in another table for which a primary or unique constraint is defined. The columns forming the local key does not need to be unique.

$$CSTR^{FK} := \left(name,\ \mathcal{K}EY_{LOCAL},\ \mathcal{K}EY_{REMOTE}\right) \tag{7.7}$$

In the relational model, a view $\mathcal{V}^{REL}$ is modeled as a special kind of table, representing the result of a query. Similar to a table, it has a list of columns. However, there are no constraints.

$$\mathcal{V}^{REL} := \left(name,\ query,\ (COL_1, ..., COL_i)\right) \tag{7.8}$$

7.1.2 Document

A namespace of the type document $\mathcal{N}^{DOC}$ contains a set of collections and views. Both, the collection C and the view $\mathcal{V}^{DOC}$ have a unique name within the namespace.

$$\mathcal{N}^{DOC} := \left(name,\ \{C_1, ..., C_i\}, \{\mathcal{V}_1^{DOC}, ..., \mathcal{V}_k^{DOC}\}\right) \tag{7.9}$$

According to the definition introduced in Section 4.2, the document model does not impose any schema, except that every document needs to have an *_id* field. However, for this model, we extend this and introduce the concept of *defined fields*. A field thereby refers to an edge on the document. This allows to define additional fields besides the *_id* that need to be present on every document. Furthermore, the fields also serve a second purpose: they allow for a vertical partitioning of collections (see Section 7.3). With *label* being a name according to the introduced definition, *required* being a boolean specifying whether the field is required on every document, *type* being a data type, and $\{length : ..., \}$ a set of pairs with properties of the data type, a collection is defined as:

$$\begin{aligned} F &:= (label, required) \\ F^{TYPED} &:= \left(label,\ type,\ \{length : ..., \},\ nullable,\ required\right) \\ C &:= \left(name,\ \{F_1, ..., F_n\},\ \{F_1^{TYPED}, ..., F_m^{TYPED}\}\right) \end{aligned} \tag{7.10}$$

The set of typed fields at least contains an entry for the *_id* field. A view $\mathcal{V}^{DOC}$ represents the result of an arbitrary query. It is similar to a collection; however, there are no constraints.

$$\mathcal{V}^{DOC} := (name,\ query) \tag{7.11}$$

7.1.3 Labeled Property Graph

In contrast to our definition of a relational namespace or a document namespace, the LPG namespace does not consist of entity structures like tables or collections. Instead, the namespace itself represents one graph. Ever node and every edge of this graph have a unique identifier. This identifier is randomly generated by the PolyDBMS system.

As outlined in Section 4.3, the LPG model can be described as schema-optional. This means, there can be a schema, but it is not mandatory. A schema can be enforced using a special graph (see Section 4.3.3). With $\mathcal{G}$ being such a graph defining a graph schema, a namespace $\mathcal{N}^{LPG}$ is defined as:

$$\mathcal{N}^{LPG} := (name,\ \mathcal{G},\ \{\mathcal{V}_1^{LPG}, ..., \mathcal{V}_k^{LPG}\}) \tag{7.12}$$

With ℓ being labels and $p := (key, type)$ being a pair of a property key and a property type, the graph schema $\mathcal{G}$ is defined as:

$$\begin{aligned} P &:= \{p_1, ..., p_i\} \\ g &:= (\ell_{from},\ \ell_{to},\ \ell_{edge},\ P_{from},\ P_{to},\ P_{edge}) \\ \mathcal{G} &:= \{g_1, ..., g_m\} \end{aligned} \tag{7.13}$$

It specifies that every edge connecting a node with the label ℓ_{from} to a node with the label ℓ_{to}, must have the label ℓ_{edge}. Further it specifies that in this case the node with the label ℓ_{from} must have the set of properties specified by the set P_{from}, the node with the label ℓ_{to} must have the set of properties specified by the set P_{to} and the edge must have the set of properties specified by the set P_{edge}. There can be multiple entries for the same pair of labels ℓ_{from} and ℓ_{to}.

Similar to the relational and document namespace, a view in the namespace $\mathcal{N}^{LPG}$ represents the result of an arbitrary query. A view $\mathcal{V}^{LPG}$ is defined as:

$$\mathcal{V}^{LPG} := (name,\ query) \tag{7.14}$$

7.1.4 Mapping

An important part of the PolyDBMS concept is the ability to access data represented in different data models using the same query language. This requires a mapping between different schemas. In this section, we outline the mapping between the relational, document and LPG model. The introduced mappings are designed so that they can be

done on-the-fly and potentially be implemented using operators of the data model to be mapped.

Since the data model is defined by the namespace, an approach for accessing other namespaces is required. As mentioned before, names must not contain a dot ('.'). This allows to use this character for expressing access paths (as it is already common is several query languages like SQL or the MongoDB query language).

The arrows in the headings of the following sections indicate the direction in which the schema is being mapped. The section "Relational → Document" for example, outlines the mapping of a relational schema to a document schema. This is for instance required when accessing a relational schema with a query language based on the document data model.

7.1.4.1 Relational → Document

The mapping of a relational schema to a document schema is trivial: a table is represented as a collection. Every record of the table is represented as a document containing a field for each column of the table.

7.1.4.2 Relational → LPG

From the perspective of the LPG model, the tables of the relational model are viewed as labels. For every record of the table, a node is created having the name of the table as label. The value for the individual columns are represented as properties. The edges between the nodes are constructed based on the existing foreign key constraints. The name of the foreign key constraint is used as label for the edge. The edge points from the table referenced by $\mathcal{KEY}_{LOCAL}$ to the table referenced by $\mathcal{KEY}_{REMOTE}$.

This approach does not result in a "perfect" graph since, for example, join tables required for modeling m-to-n relationships are not resolved and foreign key columns are not filtered from the list of properties. However, this approach can be done on-the-fly and does not require any artificial names to be generated.

7.1.4.3 Document → Relational

For this mapping, we use the previously introduced notion of *defined fields*. A collection is mapped to a table with the defined fields as columns. Furthermore, there is an addi-

tional column named *_data* containing all other fields and nested structures represented as JSON. Since it can be specified whether a defined field is required on every document, the table may contain *null* values for those columns where the required flag is not set. For typed fields, the specified data type is being used. For non-typed fields, the special data type *JSON* is being used. This type represents nested data as JSON.

In case the mapping is done on the result of a relational view, the list of columns defined for every relational view is used as projection list, reducing the result to the specified list of columns. Since the relational model is strictly typed, the values are converted into the type specified by the projection list. If a value cannot be converted to the specified data type, *null* is being used.

7.1.4.4 Document → LPG

The mapping of a document schema to a LPG schema is rather trivial: every document is considered as node with the *_id* field used as label. All other keys of the document are modeled as properties of the node with the data type *JSON*. From the graph perspective, a document namespace is accessed using the name of a collection as label.

7.1.4.5 LPG → Relational

In the relational context, a LPG namespace is accessed using a specific label of the graph as table name (e.g., `SELECT * FROM mygraph.employee`, with `mygraph` being the name of the namespace and `employee` the label). All nodes with this label are represented as records of this table. The table has three columns: An *id* column containing the unique id of the node, a *properties* column containing a map with the property key and value pairs and a *labels* column containing a list of all other labels of this node.

Edges are also represented using tables, mimicking a join table. However, they are not accessed directly by their label but mapped based on the nodes they connect, by using two node labels separated by an "->" as table name (e.g., `SELECT * FROM mygraph.employee->department`). This table contains all edges from nodes with the label "employee" to nodes with the label "department". The table contains a record for every edge between those two nodes in this direction. The table has four columns: The first column has the name of the outgoing node label (in our example this is "employee") and contains the unique id of this node. The second column has the name of the target node label ("department" in our example) and contains the unique id of the target

node. The third column has the name “label” and contains the label of the edge, and the fourth column contains a map with key-value pairs of the properties of the edge.

7.1.4.6 LPG → Document

From the document perspective, a LPG namespace is accessed using a specific label. Each node with this label is represented as a document. All parts of the graph reachable from this node are added to the document as arbitrarily deep nested structures. The label and properties of an edge are merged with the linked node. Self-references are represented by a special key in the respective (sub)document.

7.2 Physical Schema

The physical schema is the schema maintained on the underlying data stores. Due to the *Independence of Storage Configuration* requirement outlined in Section 3.1.3, a model is required for mapping logical schemas of different data models to a physical schema for storing and processing the data. The physical schemas are hidden from the user (consequence of Section 3.1.4). The mapping to a physical data store is different than the on-the-fly mapping between the logical schemas model outlined in Section 7.1.4.

In the following sections, we define strategies for mapping logical schemas based on a data model different than the native data model of the data store. The arrows in the headings of the following sections indicate the direction of the mapping. In “Relational → Document” for instance, it is outlined how a relational schema is mapped to a document schema.

7.2.1 Relational → Document

Storing relational tables in a document database is straightforward. Every table is mapped to a collection on the document database. Every record of the table is represented as a document, storing the values for the columns of the table as fields of the document. The primary key of the table is used as value for the *_id* field.

7.2.2 Relational → LPG

TODO

7.2.3 Document → Relational

A document schema is mapped to a relational data store by mapping a collection to a table. The table has at least two columns: An *_id* column storing the value of the primary key and a *_data* field containing the values for all fields and eventual nested structures as JSON. If there are *defined fields* for the logical collection to be mapped, these fields are mapped to individual columns. Every document of the collection is stored as a record of the table. Since it can be specified whether a defined field is required on every document, the table may contain *null* values for those columns where the required flag is not set.

7.2.4 Document → LPG

TODO

7.2.5 LPG → Relational

TODO

7.2.6 LPG → Document

TODO

7.3 Allocation of Data

After we have introduced the logical schema model (Section 7.1) and how logical schemas are mapped to physical schemas (Section 7.2), this section introduces a model for putting both together. Instead of simply allocating a namespace to one underlying data store, our model allows to replicate and partition data across multiple underlying data stores. Furthermore, it provides the capabilities to adjust the allocation depending on the access frequency and allows to delay the update on some of the replicas. This enormous flexibility clearly sets this model apart from existing polystore and multimodel database systems.

7.3.1 Replication and Partitioning

As introduced in Section 6.1, it can be distinguished between *data replication* and *data partitioning*. For the later it can furthermore be distinguished between *vertical partitioning* and *horizontal partitioning*. While support for replication or partitioning of data is not required by the definition of a PolyDBMS introduced in Chapter 3, we consider it crucial for providing high query performance.

In a PolyDBMS with multiple, heterogenous underlying data stores, replicating data on multiple data stores allows optimizing for different kinds of queries. Vertical partitioning allows to reduce the storage overhead by only storing certain parts of a schema object (e.g., certain columns of a table) on specific data stores. With horizontal partitioning it is possible to implement load-balancing between multiple data stores or—in combination with the concept introduced in Section 7.3.3—to optimize the performance for current or recently used data. However, the full potential of these techniques is revealed when they are getting combined. This will be discussed in more detail in Chapter 9.

Our schema model supports arbitrary combinations of horizontal and vertical data partitioning and data replication. The only constraint is, that there is the same set of schema object parts (e.g., columns of a table) for all partitions assigned to a particular data store.

A *data entity* is the structure holding data. In namespaces of the type relational or document, the data is hold by tables ($\mathcal{T}$) or collections ($\mathcal{C}$). Furthermore, data is also represented by views. If the result represented by a view is physically stored, the view is called a *materialized view*. In namespaces of type LPG, the data is hold by the namespace itself. A data entity E is therefore defined as:

What is with $\mathcal{V}^{LPG}$

$$E := \mathcal{T} \vee \mathcal{C} \vee \mathcal{N}^{LPG} \vee \mathcal{V}^{REL} \vee \mathcal{V}^{DOC} \tag{7.15}$$

Every data entity E has a horizontal partition function ϱ. A partition function maps a value v to a partition:

$$\varrho : v \longrightarrow P \tag{7.16}$$

In our model, all data entities have a partition function assigned, making all entities partitioned. However, this partition function can be the *NONE* partition function, assigning all data to one partition P. Besides the *NONE* partition function, available for every data model, the set of partition functions depends on the data model of the namespace. For relational namespaces, we define a *HASH*, *LIST* and *RANGE* partition function. All three partition functions are applied on a partition column $COL^P \in \mathcal{T}$. The *RANGE* partition function needs to be applied on column with a numerical data type while the

LIST function requires a string or integer type. The *HASH* partition function can be applied to an arbitrary data type.

$$\begin{aligned} \varrho^{HASH} &: COL^P, n \rightarrow P \quad \text{with } n \in \mathbb{N} \text{ being the number of partitions} \\ \varrho^{LIST} &: COL^P, \big(\{v_1, ..., v_i\}, ..., \{v_1, ..., v_j\}\big) \rightarrow P \\ \varrho^{RANGE} &: COL^P, \big((v_{min}, v_{max}), ..., (v_{min}, v_{max})\big) \rightarrow P \end{aligned} \tag{7.17}$$

Since the list of values and the list of min and max pairs specified for the list and range partition function do not need to be closed (i.e., there might be values which do not appear in the lists or intervals), both partition functions define an *unbound* partition for all those records. However, the specified mapping must be distinct.

Is distinct the right word?

For the document model, a *HASH* partition function is defined similar to the relational model. However, this function always operates on the *_id* of the documents. For LPGs, horizontal partitioning is more complex. Graph partitioning—which is the theory of reducing a graph into a mutually exclusive sets of nodes—is typically considered a NP-hard problem. We therefore decided against an automatic partitioning; instead, the partitioning can be defined based on the labels. With ℓ being a node or edge label, the partition function is defined as:

$$\varrho^{LABELS} : \mathcal{N}^{LPG}, \big(\{\ell_1, ..., \ell_i\}, ..., \{\ell_1, ..., \ell_j\}\big) \rightarrow P \tag{7.18}$$

The set of node labels must not overlap. If a graph is partitioned, the PolyDBMS does not allow edges between nodes residing in different partitions. Further, it does not allow assigning labels belonging to different partitions to the same node. An edge connecting two nodes must have a label belonging to the set of labels of the corresponding partition.

A vertical partition entity B is the substructure of a data entity that is subject to vertical partitioning. In the relational model these are the columns (COL) of a table and in the document model the typed (F^{TYPED}) and untyped (F) fields of a document. In the LPG model, vertical partitioning is done on the properties of a node or edge (p).

$$B := COL \vee F \vee F^{TYPED} \vee p \tag{7.19}$$

With ℓ being a label and P_E being a partition of the data entity E, a partition group G of an entity E is defined as:

$$G_E := \big(\ell, \{P_E^1, ..., P_E^j\}\big) \tag{7.20}$$

A data placement L describes the mapping of data to underlying data stores. The mapping of an entity E to a data store A is described by a pair

$$L_E^A := \big(\{(G_E^1, \Upsilon_1), ..., (G_E^n, \Upsilon_n)\}, \{B_E^1, ..., B_E^m\}\big) \tag{7.21}$$

with $\Upsilon \in \{\text{EAGER}, \text{LAZY}\}$ describing the update strategy of the data placement (see Section 7.3.2). With a function $E(L)$ that returns the data entity of a data placement, a underlying store can be described by a set of data placements:

$$A := \{L^1, ..., L^n\} \text{ with } \forall\, L^1, L^2 \in A : E(L^1) \neq E(L^2) \tag{7.22}$$

Since the set of partition groups and the set of vertical partition entities are independently defined for the entire data placement L_E^A, and since there can only be one data placement for an entity assigned to a specific adapter, the constraint that all partitions placed on a data store must have the same set of columns is fulfilled.

7.3.2 Data Freshness

In Section 6.4 we have introduced the concept of data freshness in the context of distributed database systems. Depending on the strategy with which updates are propagated in the distributed system and depending on the distribution protocol, individual sites can have outdated versions of certain elements. It depends on the use-case and often also on the type of query, whether a certain degree of outdatedness is an issue or not [ASS08].

In the context of a PolyDBMS, this is interesting since it allows to increase the performance of OLTP workload by applying data manipulation operations only to a subset of the stores immediately. on the other data stores, the operations are defered. The update strategy $\Upsilon \in \{\text{EAGER}, \text{LAZY}\}$ is defined individually for every partition group placed on a data store. *EAGER* means that DML operations are applied immediately while *LAZY* allows data manipulation operations to be executed later, resulting in outdated data.

A read-only query can specify a required level of data freshness (see Chapter 9). This model defines data freshness based on the time elapsed since the partition has been refreshed the last time. If there are no deltas to be applied, the partition is always considered up-to-date. With t_s being to timestamp of the last dml operation applied to the data store, t_{now} being the current timestamp, d the accepted outdatedness, and n the number of pending DML operations, the freshness function is defined as:

$$freshness(t_{store}, t_{now}, d, n) := \begin{cases} true & \text{if } n = 0 \\ (t_{now} - t_{store}) \leq d & \text{else} \end{cases} \tag{7.23}$$

The function returns *true* iff the freshness requirements of the query can be fulfilled. The function is applied for every partition accessed by the query.

7.3.3 Temperature-aware Data Management

As outlined in Section 6.2, temperature-aware data management refers to the concept of storing frequently used data in faster (but usually also more expensive) data stores while old and only rarely accessed data is stored on slower (but also cheaper) storage. In the context of a PolyDBMS with its heterogeneous data stores, this is even more interesting. Besides replicating frequently used data to stores running on faster storage (e.g., in-memory), data can also be replicated to a larger set of heterogeneous data stores. This allows to optimize frequently used data for a large spectrum of workloads without huge storage costs.

The partition group G introduced above allows grouping multiple partitions and allows assigning them a label. Since the mapping of partitions to partition groups can be changed, temperature-aware data management can easily be implemented in the above model using two partition groups, one with the label "hot" and one with the label "cold". To implement the temperature-aware partitioning, a special partition function assigns the partitions to the corresponding partition groups based on the access frequency. This assignment is updated at a certain interval. To avoid partitions oscillating between hot and cold, a hysteresis is specified.

Instead of assigning the hot partition group only to one or multiple high performance data stores, it can additionally also be assigned to the data stores holding the cold partition group. This approach is especially beneficial if there are analytical queries, since it eliminates the need to union the data from the stores holding the hot and the cold data together. However, this impacts the performance of data modification queries on the "hot" data since the data also needs to be updated on the slower data store holding all data.

However, our model also offers a solution for this: Since the update strategy Υ can be set individually for every partition group assigned to a data placement, the temperature-aware data management can be combined with data freshness, resulting in a very potent combination. This allows to allocate the hot partition group with an eager update strategy to the fast store and with a lazy update strategy to the cold store. This results in high performance for transactional workload including data manipulation queries on the current data while still being able to execute analytical queries—for which slightly outdated data is acceptable—on the cold data store. If the latest data is required, the query can always be executed on a different data store or the queued delta operations can be applied prior to the query.