Tables, columns, views, and columns

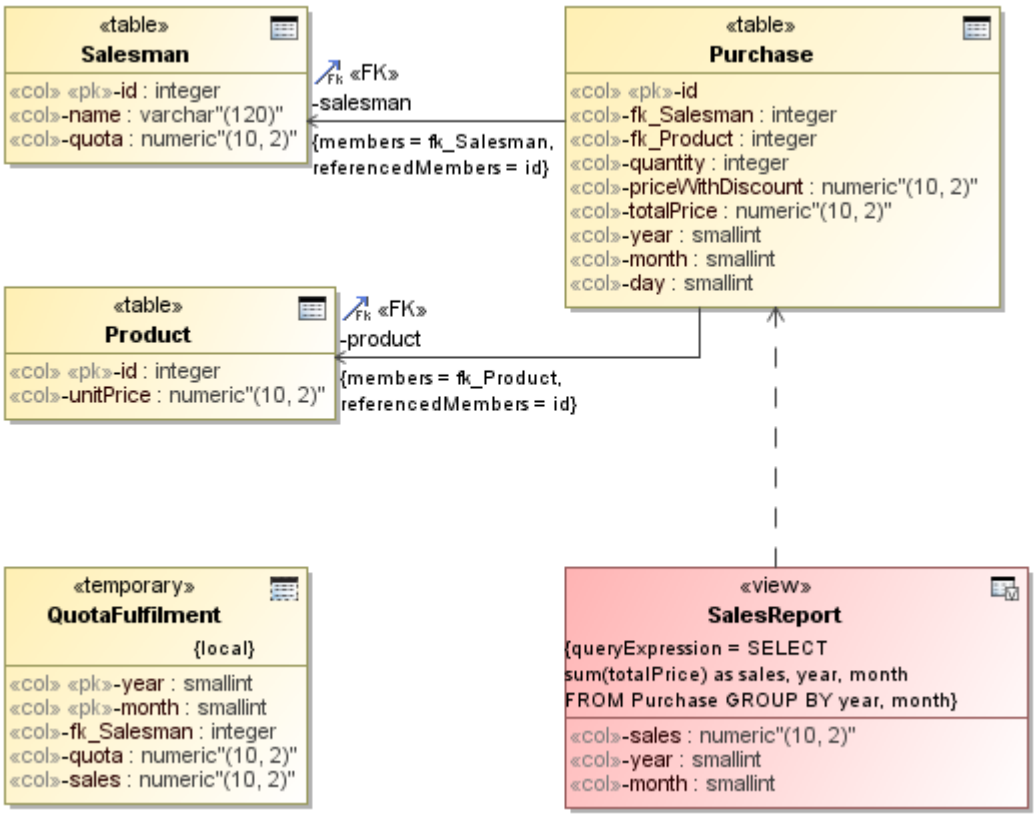
On this page

Tables and their constituent columns are the main elements for describing database data structures. Table stores multiple rows of data each consisting of several columns. Each cell holds one data value (or is empty). All values of one column are of the same type. Correspondingly each table description consists of the table name and a set of column descriptions. Additionally there are various kinds of constraints (including the all- important primary key and foreign key constraints), that can be applied on tables and triggers, specifying additional actions to be performed during data manipulation.

See Constraints and Triggers for triggers. There can be various kinds of tables as follows.

- Persistent table
- Temporary table
- View (derived table)
- Column

The following figure illustrates various kinds of tables that can be modeled on the diagram.



Various kinds of tables: persistent tables, temporary tables, and views.

Tables can have generalization relationships between them. These relationships correspond to the following SQL syntax in the create table statement.

```
CREATE TABLE <name> OF <UDT name> [UNDER <supertable>]
```

There can be at most 1 outgoing generalization. Generalizations are not widely supported in database management systems. As of v17.0.1 Cameo Data Modeler supports modeling of these structures. Generation of corresponding script code is not supported yet.

Persistent table

Note
SQL Persistent Table is modeled as UML Class with the «Persistent- Table» stereotype applied. For the sake of compactness, these tables are displayed with the «table» keyword (instead of the long form - «PersistentTable») on the diagram.

Persistent table is the most often used kind of table. Besides the standard SQL element properties, persistent table has the following properties available in the Specification window (these properties are only available in Expert mode).

Property name	Description
User-defined type	Points to structured user defined type, which serves as a base for the row type of the table.
Supertable	Points to the parent (base) table. Can only be used together with userdefined type.
Self Ref Column Generation	Describes the self-referencing column generation options. Can only be used together with user-defined type. Corresponds to the following subclause of SQL create table statement `REF IS <column name> [SYSTEM GENERATED|USER GENERATED|DERIVED]`
Referencing Foreign Keys	This is back reference from foreign keys, referencing this table. This field is for information purposes only. If you want to change it, change **Referenced Table** field of the foreign key instead.
Insertable	These are two derived (non editable) fields, describing table data editing capabilities. At the moment calculation of these properties is not implemented - they are always set to false.
Updatable	

Temporary table

Note
SQL Temporary Table is modeled as UML Class with the «Temporar- yTable» stereotype applied. For the sake of compactness, these tables are displayed with the «temporary» keyword (instead of the long form - «TemporaryTable») on the diagram.

Temporary table is a kind of table, where data is held only temporary. There are two kinds of temporary tables. Local temporary table persists for the duration of user session and is visible only for the creator user. Global temporary table is long lived and visible for all users. Note that **data** in the global temporary table is different for different users and does not persist throughout user sessions (only global table definition persists).

Temporary tables are created using SQL create table statement (using TEMPORARY option).

```
CREATE (GLOBAL | LOCAL) TEMPORARY TABLE <table name> ...
[ON COMMIT (PRESERVE | DELETE) ROWS]
```

Besides the standard SQL element properties and persistent table properties (see Persistent table), temporary table has the following properties available in the Specification window.

Property name	Description
Local	Marks the table as local or global temporary table.
Delete On Commit	Regulates whether data is deleted or retained on commit.

View (derived table)

Note
SQL View is modeled as UML Class with the «ViewTable» stereotype applied. For the sake of compactness, views are displayed with the «view» keyword (instead of the long form - «ViewTable») on the dia- gram.

View is a table, whose data is derived from data of other tables (by applying some SQL query). Views are created using SQL create view statement.

```
CREATE VIEW <name> [<view column list>]
AS <query expression>
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Note that since column definition list is optional in SQL syntax, specifying column definitions in the view is also optional (columns can be inferred from query expression of the view). However, it is often a good idea to include column definitions, since this allows to see view data structure on the diagram / in the model at a glance, without parsing the query expression text.

Besides the standard SQL element properties and persistent table properties (see Persistent table), view has the following properties available in the Specification window.

Property name	Description
Query Expression	A query expression, defining how data is calculated / retrieved for this view. This is an SQL SELECT statement.
Check Type	Describes how check is performed on the data update through the view. Only meaningful for updateable views (which is rare).

Query expression of the view modeling deserves a special attention. Query expression, defining the view, is not just a simple string, but a (stereotyped) UML model element. By default query expression model object is stored within the view definition itself. There is a special constraint, automatically created inside the view, to hold this expression. When the view is created, **Query Expression** field (which is a tag of stereotype, applied on the view) is automatically pointed to this expression.

So by default you just need to fill in the **Body** text of the expression. To do that you need to double-click on the **Query Expression** field. This opens Specification window for the expression itself, where **Body** can be filled in. This is the default, no-hassle way to specify view. It is easy. But it has one deficiency. Views created this way do not have any model references to the underlying table model elements. This may be undesirable from the dependency tracking standpoint (in the dependency analysis). To remedy this, you can draw an additional **Dependency** relationships between the view and base tables.

There is also another way to model the query expression, defining the view. If you click on the **...** button of the **Query Expression** field, this action opens the element selection dialog, allowing to retarget the **Query Expression** pointer choose another expression object, located somewhere else in the model. For example view definition expression can be located inside the **Abstraction** relationship, drawn from the view to the base table (**Mapping** field of the **Abstraction**).

To model view queries using abstractions

---

1. Draw an abstraction relationship between a View and a Table.
2. In the abstraction's Specification window, fill in the **Mapping** cell. This will be an inner UML OpaqueExpression model element with language and body cells. Set language to "SQL" and fill in the body with the necessary "SELECT ..." expression text.
3. Further open the Specification window of the mapping expression, and apply the «QueryExpressionDefault» stereotype.
4. Open the Specification window of the view. Click the **...** button in the **Query Expression** cell. In the element Selection dialog navigate to the abstraction relationship and select the expression inside of it.

This way to model view query expressions is rather tedious - so it is not recommended for modeling novices. But it has an advantage of capturing the real relationship in the model between the view and the constituent table(s). Also query expression can be shown on the abstraction relationship (using note mechanism) instead of showing expression on the view.

In the following figure, you can see a diagram that illustrates the alternative way of view modeling.

Alternative notation for modeling view derivation from tables.

Column

> **Note**
> SQL Column is modeled as UML Property with «Column» stereotype applied. For the sake of compactness, columns are displayed with the «col» keyword (instead of the long form - «Column») on the diagram.

Column model element describes one column of the table. In the most frequent case it's just a name of the column and a type. Additionally, column can carry default value specification, column constraints.

Column definition syntax in SQL (in CREATE TABLE, ADD COLUMN statements) is as follows.

```
<column name> [ <data type> ]
[ DEFAULT <value expression> |
GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
[ '(' <sequence options> ')' ] |
GENERATED ALWAYS AS <expression>]
[ <column constraint definition>... ]
```

Besides the standard SQL element properties, column has the following properties available in the Specification window.

Property name	Description
Type	Collectively these two fields describe the type of the column. Type could be any of the primitive types from the library or user defined type. Modifier provides additional parameters for the type - such as width of the character type (when type=varchar and modifier="(20)" - column is of varchar(20) type). See Type Usage section for details.
Type Modifier	
Nullable	Marks column as nullable or not. Basically this is an in-line nullability constraint. See Constraints section for details.
Default Value	Carries the default value of the column. This is normally an opaque expression, allowing to specify the value of the column. However it can be switched to Identity Specifier. In this case it describes the autoincrement options of the column. See Sequences and autoincrement columns section.

Is Derived	Standard UML field, used to mark the column as derived (GENERATED ALWAYS AS <expression>). It works together with Default Value field.
Scope Check	Marks this column as scope checked to a particular table and allows choosing particular referential integrity ensuring action (RESTRICT CASCADE, etc).
Scope Checked	
Implementation Dependent	Marks this column as implementation dependent.