

MEGA Database Builder

User Guide



MEGA HOPEX V1R2-V1R3
Second Edition (January 2016)

Information in this document is subject to change and does not represent a commitment on the part of MEGA International.

No part of this document is to be reproduced, transmitted, stored in a retrieval system, or translated into any language in any form by any means, without the prior written permission of MEGA International.

© MEGA International, Paris, 1996 - 2014

All rights reserved.

MEGA Database Builder and MEGA are registered trademarks of MEGA International.

Windows is a registered trademark of Microsoft Corporation.

The other trademarks mentioned in this document belong to their respective owners.

INTRODUCTION TO HOPEX DATABASE BUILDER



HOPEX Database Builder is used to design and model databases.

A database describes organization of a system in the form of data stored in a structured way. Database design is an analysis procedure covering different levels:

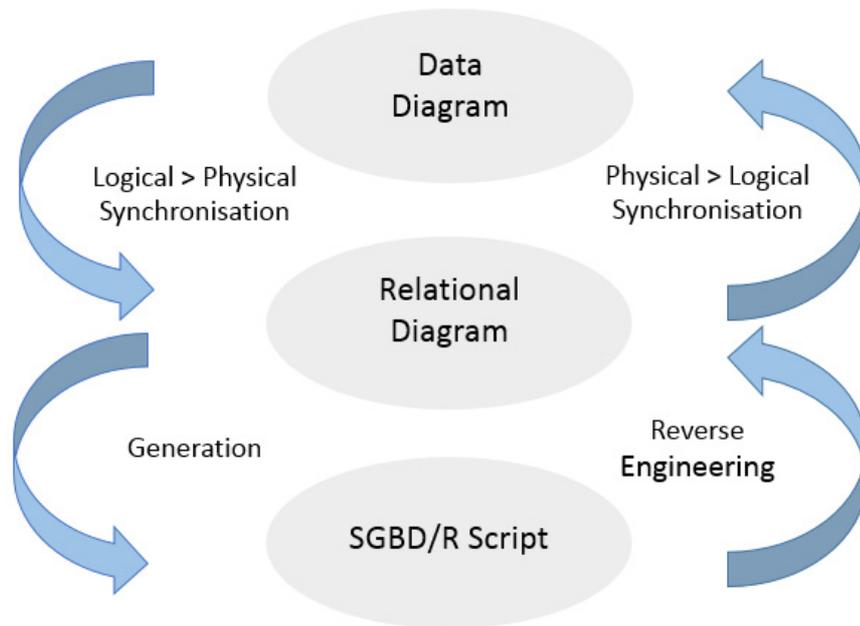
- ✓ Conceptual level: integration with **HOPEX Information Architecture** is used to define the business function architecture concepts. These concepts can be implemented by objects such as applications or processes and be described by data models.
- ✓ Logical level: intended for clients seeking to develop general business-oriented models. Here it consists of modeling data of a domain, application or business process. It represents what we wish to do and where we want to go, irrespective of technical questions related to implementation. Data is represented in a data model or a class diagram.
- ✓ Physical level: consists of defining models intended to persist in a DBMS. It comprises detailed specifications for production of the physical diagram of the repository. It is represented by the relational diagram. The physical level also defines the way in which data is stored and how it can be accessed. It enables use of data by DBMSs.

HOPEX Database Builder integrates by default the logical and physical modeling levels and is used to switch from one model to another. You can therefore:

- Create data diagrams.
- From these data diagrams, create database tables and their columns, indexes, and keys, as well as the drawings for the corresponding relational diagrams.
- Optimize the resulting relational model and generate SQL commands to define the tables. **HOPEX Database Builder** in particular takes account

of changes in the conceptual model without losing optimizations made to the relational model.

- Reverse generate a database definition using the ODBC protocol to create the corresponding tables and columns in **HOPEX Database Builder**, and obtain the corresponding data diagram.



To use the functionalities of **HOPEX Database Builder**, you must be in "Advanced" mode:

1. Click **Main Menu > Settings > Options**.
2. In the left pane of the window, click the Repository folder.
3. In the right pane, check that access to the repository is in "Advanced" mode.

CONNECTING TO HOPEX DATABASE BUILDER

The menus and commands available in **HOPEX Database Builder** depend on the profile with which you are connected.

Connecting to the solution

To connect to **HOPEX Database Builder**, see HOPEX Common Features, "HOPEX Desktop", "Accessing (HOPEX)".

Connection Profiles to HOPEX Database Builder

In **HOPEX Database Builder**, there are default connection profiles with which specific rights and accesses are associated. These profiles define the objects displayed on the desktop, according to each business.

The profiles available are:

- **Data architect** : he has read and write access to logical data. The data architect is responsible for defining the logical view (data models, packages, classes, etc.) of the system. During integration with **HOPEX Information Architecture**, the data architect has reading access to business information and ensures their connection with the objects in the logical view.
- **Database architect**: the database architect has access to logical and physical data, and synchronization and denormalization tools. The database architect defines the generation parameters for databases and generates SQL scripts that it sends to the DBA.
- **Database Administrator (DBA)** : the Database Administrator has read access to the description of databases; the Database Administrator can generate them or generate their physical components, tables, view, etc.
- **IA Functional Administrator**: the IA Functional Administrator is responsible for standard administration functions, such as user management. The IA Functional Administrator has rights to all **HOPEX Database Builder** objects.

INTERFACE PRESENTATION

The menus and commands available depend on the profile with which you are connected.

☛ *For more details on the use of analysis reports, see the **HOPEX Common Features**.*

Desktop Presentation

Once connected to **HOPEX Database Builder**, the "Information Architecture" desktop appears.

 *The "Information Architecture" desktop is the **HOPEX Database Builder** workspace, in common with the **HOPEX Information Architecture** and **HOPEX Logical Data** solutions.*

According to your profile, the navigation menu gives access to the navigation panes.

Presentation of space common to all roles

All users have access to the "Information Architecture" desktop and access to the following panes:

- **Home, Dashboard, List of tasks** and **Reports** that are common to all **HOPEX** solution users.
- **Environment**: pane used to organize and access all library elements easily,

Data architect space

In addition to the panes offered in standard mode to all "Information Architecture" desktop users, the data architect has access to the **Logical data** pane that gives access to the following folders:

- **Data Type Packages,**
- **Packages,**
- **Data areas,**
- **Data views.**

☛ *During integration with the **HOPEX Information Architecture solution**, the data architect also has access to the **Business information** pane.*

Database architect workspace

In addition to the panes offered in standard mode to all "Information Architecture" desktop users, the database architect has access to the following panes:

- **Logical data** that gives access to:
 - **DataType Package,**
 - **Package,**
 - **Data areas,**
 - **Data views,**
 - **Databases,**
 - **Tools** from the logical view.
- **Physical data** that gives access to:
 - **Databases**
 - **Physical data areas (RDB)**
 - **Physical views**
 - **Tools** from the physical view.

IA functional administrator workspace

In addition to the panes offered in standard mode to all "Information Architecture" desktop users, the IA Functional Administrator has access to the following navigation panes:

- **Logical data** that gives access to:
 - **DataType Package,**
 - **Package,**
 - **Data areas,**
 - **Data views,**
 - **Databases,**
 - **Tools** from the logical view.
- **Physical data** that gives access to:
 - **Databases**
 - **Physical data areas (RDB)**
 - **Physical views**
 - **Tools** from the physical view.
- **Administration,** particularly for user management.

☛ *During integration with the **HOPEX Information Architecture solution**, the data architect also has access to the **Business information** pane.*

Database administrator workspace

In addition to the panes offered in standard mode to all "Information Architecture" desktop users, the database administrator has access to the **Physical data** pane that gives access to the following folders:

- **Databases,**
- **Physical data areas (RDB),**
- **Physical views.**
- **Tools** (mapping and SQL generation).

Data Modeling Options

Formalisms

You can model logical data using two formalisms:

- the data package, to build class diagrams (UML notation)
- the data model, for data diagrams (standard notations, IDEF1X, I.E, Merise)

To display one of the formalisms:

1. On the desktop, click **Main Menu > Settings > Options.**
2. In the navigation tree, expand the **Data Modeling** folder.
3. Click **Data Formalism.**
4. In the right part of the window pane select the formalism(s) that you want to display.
5. Click **OK.**
6. The folders corresponding to the packages and data models appear in the **Logical data** navigation pane.

Notations

You have access to a standard data model notation, selected by default. To display another notation (DEF1X, I.E or Merise):

1. On the desktop, click **Main Menu > Settings > Options.**
2. In the navigation tree, expand the **Data Modeling** folder.
3. Click **Data Notation.**
4. In the right part of the window, select the notations that you want to display.
5. Click **OK.**

ABOUT THIS GUIDE

Guide Structure

The **HOPEX Database Builder** guide covers the following points:

- ["Conceptual Modeling", page 19](#), presents data diagrams used by **HOPEX Database Builder**.
- ["The Class Diagram", page 49](#), describes characteristics available on classes.
- ["Database and Physical Modeling", page 57](#) indicates how to create a database and its relational diagrams.
- ["Synchronizing logical and physical models", page 73](#), describes how to automatically synchronize a conceptual data model with a database described in relational formalism, and vice versa.
- ["Model Mapping", page 115](#), presents the application that enables mapping between different views of a database.
- ["Denormalizing logical and physical models", page 125](#), explains how to modify models to suit specific requirements.
- ["Attribute and Column Types", page 145](#), presents attribute datatypes and their mapping with column datatypes.
- ["Generating SQL scripts", page 163](#), presents table definition SQL orders from specifications created with **HOPEX Database Builder**.
- ["Advanced SQL Options", page 179](#), describes the use of views, triggers and stored procedures, and adding physical properties to database objects.
- ["Reverse engineer tables", page 199](#), presents how to create tables and columns in **HOPEX Database Builder** from databases accessible via the ODBC, protocol, and then obtain the corresponding class diagram in UML.
- ["ODBC Extraction Utility", page 205](#) describes the utility used to access databases.
- ["Glossary", page 217](#), presents definitions of objects handled in **HOPEX Database Builder**.
- ["Pivot Types and Datatypes Correspondence Tables", page 223](#), lists correspondence tables between pivot types and all DBMS datatypes.

Additional Resources

This guide is supplemented by:

- the **HOPEX Common Features Common Features** guide, which describes the Web interface and tools specific to MEGA solutions.
 - *It can be useful to consult this guide for a general presentation of the interface.*
- the **HOPEX Power Supervisor** administration guide.

Conventions Used in the Guide

Styles and formatting

-  *Remark on the preceding points.*
-  *Definition of terms used.*
-  *A tip that may simplify things.*
-  *Compatibility with previous versions.*
-  **Things you must not do.**



Very important remark to avoid errors during an operation.

Commands are presented as seen here: **File > Open.**

Names of products and technical modules are presented in bold as seen here:
HOPEX.

CONCEPTUAL MODELING



The first step in data modeling is at conceptual level. To help you describe business function data, **MEGA Database Builder** proposes simple notation enabling representation of all current cases and based on the data model.

A data model is used to represent the static structure of a system, particularly the types of objects handled in the system, their internal structure, and the relationships between them.

Other notations (IDEF1X, IE, Merise) are also available. For more information, see chapter "Other Notations" in the **MEGA Business Data** guide.

For modeling with UML notation, see ["The Class Diagram", page 49](#).

The following points are covered here:

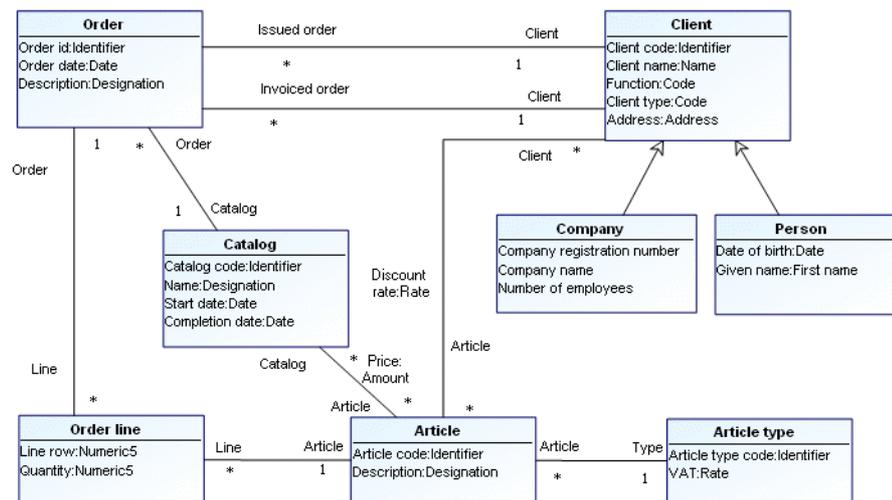
- ✓ ["Data Models", page 20](#)
- ✓ ["Building a Data Diagram", page 22](#)
- ✓ ["Normalization Rules", page 39](#)
- ✓ ["SQL Identifiers and Options", page 42](#)
- ✓ ["Data Modeling Rules", page 48](#)

DATA MODELS

The data model is the starting point of the data description process. It enables creation of business function oriented general models, independent of physical implementation questions.

A database can be connected to a data model from the time of its creation. It is on this database that the different data processing tools can then be run (generation, synchronization etc.).

Example of data model



Presentation

A data model is used to represent the static structure of a system, particularly the types of objects manipulated in the system, their internal structure, and the relationships between them.



An object is an entity with a well-defined boundary and identity that encapsulates state and behavior. Its state is represented by the values of its attributes and its relationships with other objects. An object is an instance of a class.

Examples of objects:

- Business objects:
 - John Williams, Elizabeth Davis and Paul Smith are instances of the "person" class.
 - Orders 10533 and 7322 are instances of the "order" class.
 - SPD-1730 Monitor is an instance of the "item" class.

Data modeling consists of identifying the entities representing the activity of the company, and defining the associations existing between them. The entities and associations in the data diagram associated with a sector of the company must be sufficient to provide a complete semantic description. In other words, one should be able to describe the activity of a company by using only these entities and associations.

This does not mean that each word or verb used in this explanation corresponds directly to an object in the data diagram. It means that one must be able to state what is to be expressed using these entities and associations.

Data diagram specification is often considered the most important element in modeling of an information system.

Creating a Data Model

The data model is designed to contain modeled elements. It is described by one or several data diagrams.

☛ *The data model is also owner of entities and associations derived from a synchronization (logical to conceptual). For more details on synchronization, see ["Synchronizing logical and physical models", page 73](#).*

To create a data model:

1. In **MEGA**, select the **Main Objects** navigation window.
2. Right-click the "Data Models" folder and select **New > Data Model**.
3. In the dialog box that opens, enter its name.

The data model created appears in the navigator.

☛ *You can also create a data model from a database. See ["Connecting a Data Model to a Database", page 59](#).*

BUILDING A DATA DIAGRAM

A data diagram is a graphical representation of a model or of part of a model.

In **MEGA Database Builder**, a data diagram is represented by:

- *Entities*, which represent the basic concepts (customer, account, product, etc.).
- *Associations*, which define the relationships between the different entities.
- *Attributes*, which describe the characteristics of entities and, in certain cases, of associations.

The data diagram also contains multiplicity definitions.

☛ For SQL generation of entities and attributes, see "[SQL Identifiers and Options](#)", page 42.

Creating a Data Diagram

A data diagram is a graphical representation of a model or of part of a model.

When you create a data model from a database, the corresponding data diagram is also created.

You can then connect new data diagrams to the data model.

To create a data diagram:

1. Right-click the data model.
 2. Select **New > Diagram**.
 3. In the dialog box that opens, select **Data Diagram**.
 4. Click the **Create** button.
- The diagram opens in the edit area.

Entities

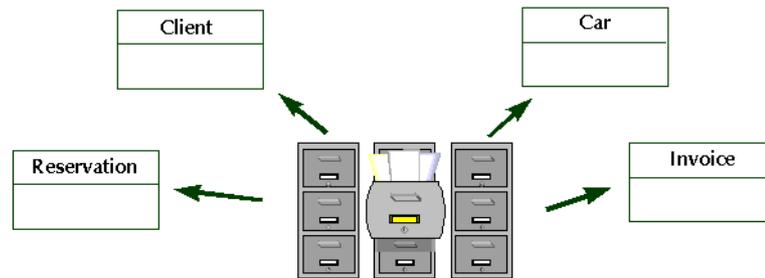


An entity groups objects that share the same characteristics and have similar behavior. Persistent entities are management elements considered useful for representing enterprise activity, and are therefore reserved for this purpose. They will have corresponding tables in the relational database.

An *entity* is described by a list of attributes.

An entity is linked to other entities via associations. The set of entities and associations forms the core of a data model.

We can illustrate the entity concept by comparing entities to index cards filed in drawers.



Entities can represent management objects.

Examples: Customer, Order, Product, Person, Company, etc.



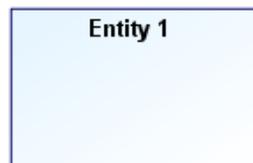
Entities can represent technical objects used in industry.

Examples: Alarm, Sensor, Zone

Creating an entity

To create an entity:

1. In the data diagram click **Entity (DM)**  in the objects toolbar.
2. Click in the diagram.
The **Add Entity (DM)** dialog box opens.
3. Enter the entity **Name**.
4. Click **Create**.
The entity appears in the diagram.

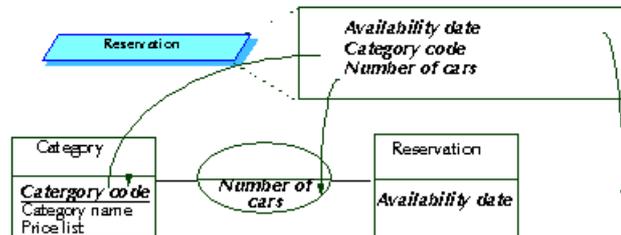


☛ On the form of an entity it is possible to display the attributes and characteristics of the entity. See ["Object Display Options \(Windows Front-End\)", page 78](#).

Attributes

Entities and associations can be characterized by *attributes*.

 An attribute is a named property of a class. This is the most basic data saved in the enterprise information system.



These attributes can be found by studying the content of messages circulating within the enterprise.

Examples:

- "Client Name" (property of the client entity).
- "Client No." (identifier of the client entity).
- "Account Balance" (property of the account entity).

An attribute characterizes an association when the attribute depends on all the entities participating in the association.

In the diagram below, the role that a "Consultant" plays in a "Contract" depends on the consultant and on the contract, and therefore on the "Intervene" association.



Defining attributes

To create an attribute on an entity:

1. In the data diagram, right-click the entity and select **Attributes**. The entity properties dialog box opens.
2. To add a new attribute to the entity, in the **Attributes** tab, click the  button. A default name is automatically proposed for the new attribute. You can modify this name.

You can specify its **Type**.

Example: Numeric value.

 A datatype is used to group characteristics shared by several attributes.

 See "Attribute and Column Types", page 145 for more details on types that can be assigned to an attribute.

See also "Attribute properties", page 54.

Inherited attributes

When a generalization exists between a general entity and a more specialized entity, the specialized entity inherits the attributes of the general entity.

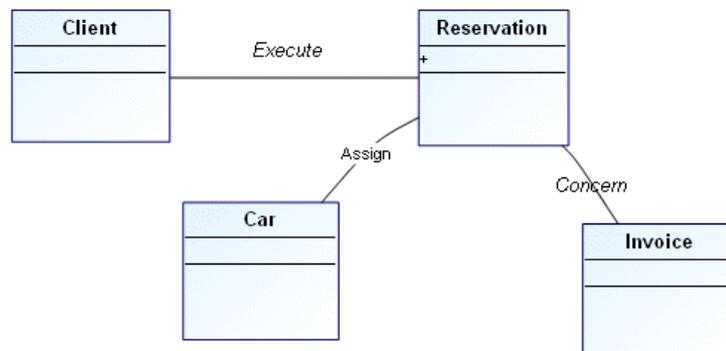
See "Generalizations", page 33.

Associations

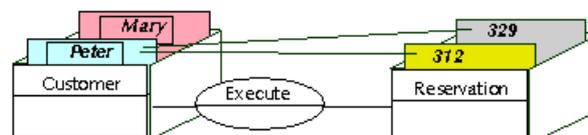
 An association is a relationship that exists between two or more entities. It can carry attributes that characterize the association between these entities

Examples of associations

Associations can be compared to links between index cards.



The following drawing provides a three-dimensional view of the situations a data diagram can store.



Peter and Mary are clients. Peter has made reservations numbers 312 and 329.

A data diagram should be able to store all situations in the context of the company, but these situations only.

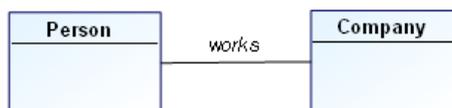
☛ *The diagram should not allow representing unrealistic or aberrant situations.*

Examples of associations:

- A client issues an order.
- An order includes several products.



- A person works for a company.



- An alarm is triggered by a sensor.
- A sensor covers a zone.
- A window displays a string of characters.

Creating an Association

To create an association:

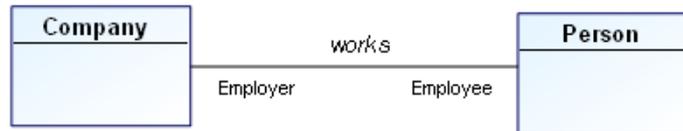
1. In the data diagram objects toolbar, click the **Association**  button.
2. Click one of the entities concerned, and holding the mouse button down, drag the mouse to the other entity, before releasing the button. A line appears in the diagram to indicate the association.
3. To indicate the association name, open its properties dialog box from its pop-up menu. You can enter its name in the **Local Name** field of the **Characteristics** tab.

Defining association roles (ends)

 *A role enables indication of one of the entities concerned by the association. Indication of roles is particularly important in the case of an association between an entity and itself.*

Each end of an association specifies the role played by the entity in the association.

The role name is distinguished from the association name in the drawing by its position at the link end. In addition, the role name appears in a normal font, while the association name is italicized.



☺ The status bar (located at the bottom of the window) also allows identification of the different zones: when you move your mouse along the association, it indicates if you are on an association or on a role.

When two entities are linked by only one association, the names of the entities are often sufficient to describe the role. Role names are useful when several associations link the same two entities.

Multiplicities

Each role in an association has an indicated multiplicity to specify how many objects in the entity can be linked to an object in the other entity. Multiplicity is information related to the role and is specified as a completely bounded expression. This is indicated in particular for each role that entities play in an association.

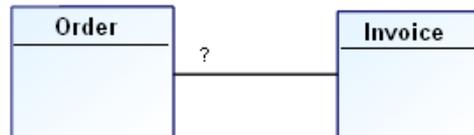
Multiplicity specifies the minimum and maximum number of instances of an entity that can be linked by the association to each instance of the other entity.

The usual multiplicities are "1", "0..1", "*" or "0..*", "1..*", and "M..N" where "M" and "N" are integers:

- The "1" multiplicity indicates that each object of the entity is linked by this association once and once only.
- The "0..1" multiplicity indicates that at most one instance of the entity can be linked by this association.
- The "*" or "0..*" multiplicity indicates that any number of instances of the entity can be linked by the association.
- The "1..*" multiplicity indicates that at least one instance of the entity is linked by the association.
- The "M..N" multiplicity indicates that at least M instances and at most N instances of the entity are linked by the association.

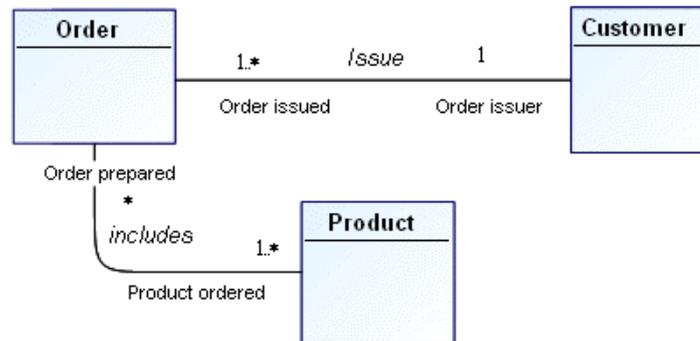
1	One and one only
0 / 1	Zero or one
M..N	From M to N (natural integer)
*	From zero to several
0..*	From zero to several
1..*	From one to several

Example:



- 0 / 1 An order corresponds to zero or at most one invoice.
- * No restriction is placed on the number of invoices corresponding to an order.
- 1 Each order has one and only one corresponding invoice.
- 1..* Each order has one or more corresponding invoices.

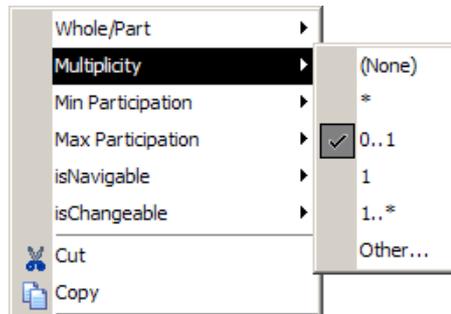
Other examples of multiplicity:



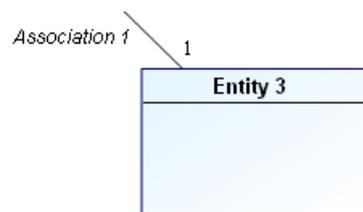
- 1..* A client can issue one or more orders.
- 1 An order is issued by one and only one client.
- 1..* An order contains one or more products.
- * A product can be contained in any number of orders, including no orders.
- 0 / 1 A person works for a company.
- 1..* An alarm is triggered by one or more sensors.
- 1 A sensor covers one and only one zone.
- 1..* A window displays one or more strings.

To specify role multiplicity:

- 1 Open the pop-up menu for the role by right-clicking the line between the association and the entity, then select **Multiplicity**.



The multiplicity now appears on the role.



☛ If the menu you see does not propose multiplicity, check that you clicked on that part of the line indicating the role and not the association.

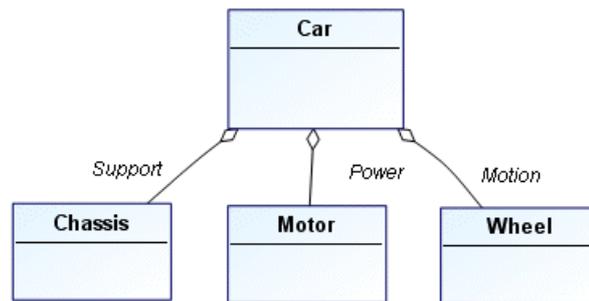
All the information specified with the pop-up menu can also be viewed and modified in the role properties dialog box.

Aggregation

📖 Aggregation is a special form of association, indicating that one of the entities contains the other.

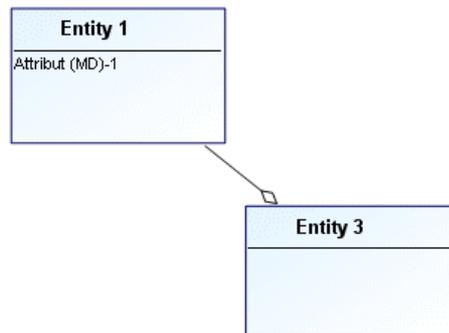
Example of *aggregation*:

A car includes a chassis, an engine, and wheels.



- › Select **Whole/Part > Aggregate** on the role played by "Entity 3" in its association with "Entity 1".

A diamond now appears on the role, representing the aggregation.



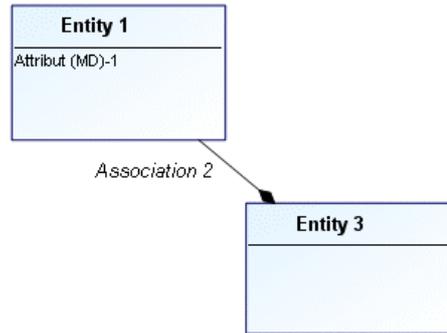
Composition

 A composition is a strong aggregation where the lifetime of the components coincides with that of the composite. A composition is a fixed aggregation with a multiplicity of 1.

Example of *composition*:

An order consists of several order lines that will no longer exist if the order is deleted.

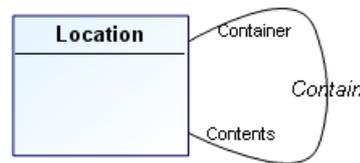
Composition is indicated by a black diamond.



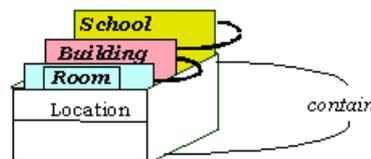
- » To specify the composition of a role, select **Whole/Part > Composite** from its pop-up menu.

Using reflexive associations

Certain associations use the same entity several times.



A classroom, a building, and a school are all locations.



A classroom is contained in a building, which is contained in a school.

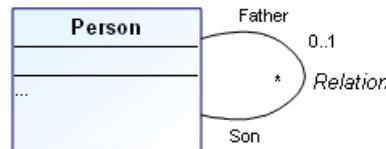
A reflexive association concerns the same entity at each end.

To create a reflexive association:

1. Click the **Association** button  in the objects toolbar.

2. Select the entity concerned and drag the mouse outside the entity, then return inside it and release the mouse button.
The reflexive association appears in the form of a half-circle in a broken line.

☛ *If there is association of an entity with itself, the roles need to be named in order to distinguish between the corresponding links in the drawing.*



“Father” and “Son” are the two roles played by the “Person” entity in the “Parent” association.

📖 *A role enables indication of one of the entities concerned by the association. Indication of roles is particularly important in the case of an association between an entity and itself.*

You can segment a line by adding joints to modify its path. You can in particular segment a role to avoid an obstacle for example. You can also change the line to a curve.

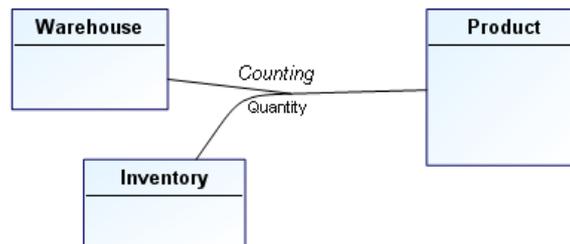
To produce a curved line:

1. Select the association.
2. In the toolbar "Edit", click **Curved line**. The link now curves at each joint.

Displaying an N-ary Association

Certain associations associate more than two entities. These associations are generally rare.

Example: When taking inventory, a certain quantity of product was counted in each warehouse.



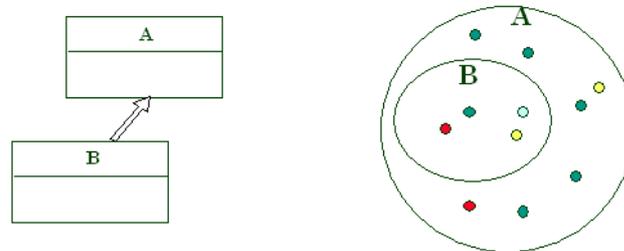
To create a ternary association:

1. Create the association between two entities.
2. Click the **Association Role** button  and connect the association to the third entity.

Generalizations

 A generalization represents an inheritance relationship between a general entity and a more specific entity. The specific entity is fully consistent with the general entity and inherits its characteristics and behavior. It can however include additional attributes or associations. Any object of the specific entity is also a component of the general entity.

What is a generalization?



Entity A is a *generalization* of entity B. This implies that all objects in entity B are also objects in entity A. In other words, B is a subset of A. B is then the sub-entity, and A the super-entity.

Example:

A: Person, B: Employee.

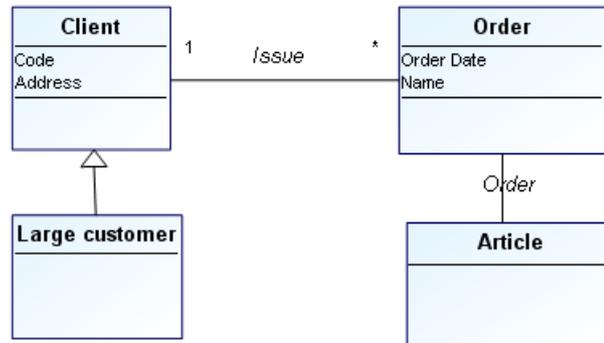
B being a subset of A, the instances of entity B "inherit" the characteristics of those in entity A.

It is therefore unnecessary to redescribe for entity B:

- Its attributes
- Its associations

Example:

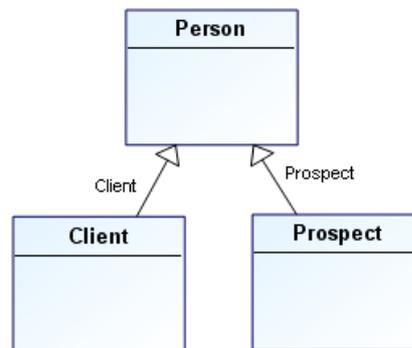
The "Large Client" entity, representing clients with a 12-month revenue exceeding \$1 million, can be a specialization of the Client entity (origin).



In the above example, the associations and attributes specified for "Client" are also valid for "Large client".

Other examples of generalizations:

"prospect" and "client" are two sub-entities of "person".

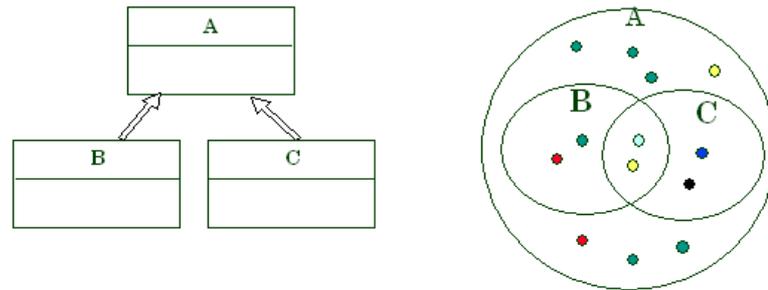


"export order" is a sub-entity of the "order" entity.

"Individual person" and "corporate person" are two sub-entities of the "person" entity.

"motor vehicle", "off-road vehicle" and "amphibious vehicle" are sub-entities of the "vehicle" entity.

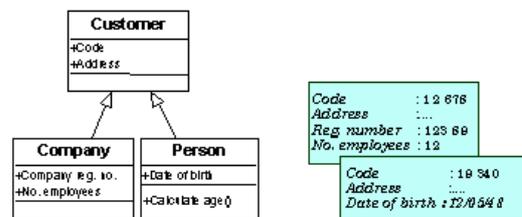
Multiple sub-entities



Several sub-entities of the same entity:

- are not necessarily exclusive.
- do not necessarily partition the set.

Advantages of sub-entities

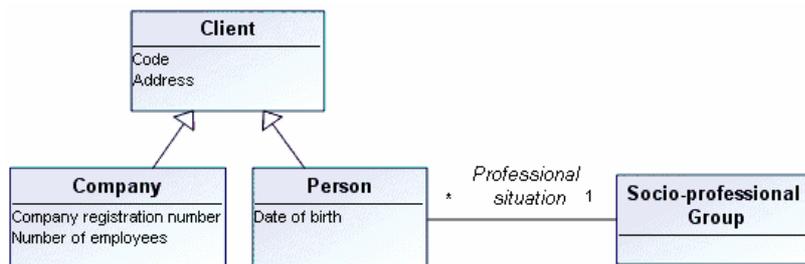


A sub-entity inherits all the attributes and associations of its super-entity, but can have attributes or associations that the super-entity does not have.

A sub-entity can also have specific attributes. These only have meaning for that particular sub-entity. In the above example:

- "Registry number" and "number of employees" only have meaning for a "company".
- "Date of birth" is a characteristic of a "person", not a "company".

A sub-entity can also have specific associations.



- A "person" falls into a "socio-professional group": "manager", "employee", "shopkeeper", "grower", etc. This classification makes no sense for a "company". There is also a classification for companies, but this differs from the one for persons.

Multiple inheritance

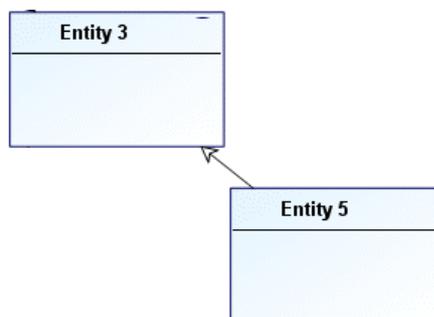
It is sometimes useful to specify that an entity has several super-entities. The sub-entity inherits all the characteristics of both super-entities. This possibility should be used carefully.

☛ *Multiple inheritance is not taken into account when generating tables.*

Creating a generalization

To create a generalization:

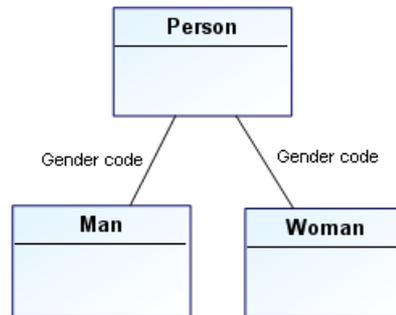
1. Click the **Generalization**  button in the objects toolbar.
2. Click the sub-entity, in this example "Entity 5", and drag the mouse to the general entity, in this example "Entity 3", then release the button. The **Add Generalization** dialog box opens.
3. Enter the name of the generalization, "Generalization 1", then click **Create**.
The generalization is now indicated in the diagram by an arrow.



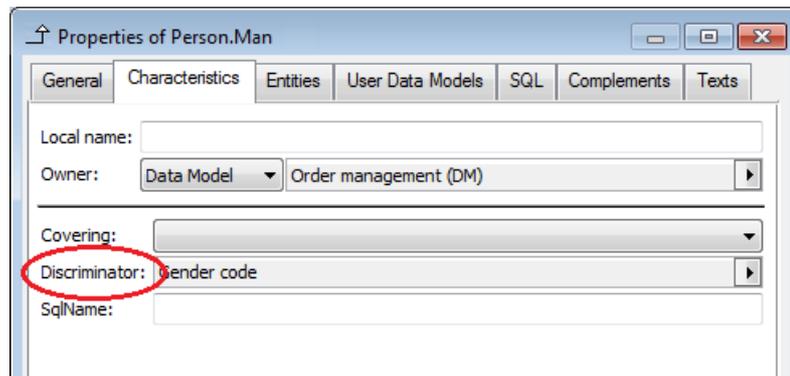
Discriminator

The discriminator is a general entity attribute whose value partitions the objects into the sub-entities associated with the generalization.

For example, the gender code attribute divides the objects in the person entity into the man and woman sub-entities.



You can define discriminator(s) in the generalization properties dialog box.



- It is also possible to specify whether a generalization:
- **IsDisjoint**: An instance cannot belong to two sub-entities of the generalization simultaneously.
 - **IsComplete**: All instances of the super-entity belong to at least one of the sub-entities of this generalization.

Constraints

 A constraint is a declaration that establishes a restriction or business rule that must be applied on execution of processing.

Most *constraints* involve associations between entities.

Examples of constraints:

The person in charge of a department must belong to the department.

Any invoiced order must already have been delivered.

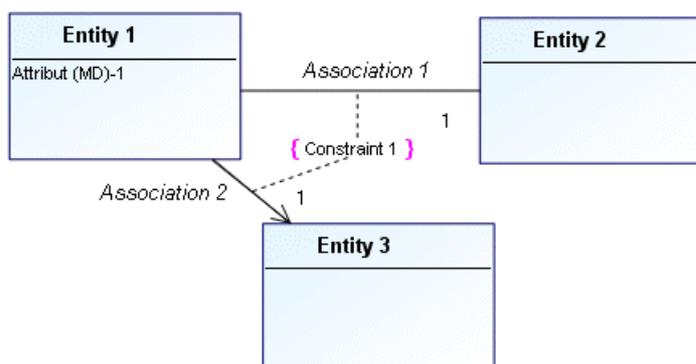
The delivery date must be later than the order date.

A sensor covering a zone can trigger an alarm for that zone only.

☛ *The aim of constraints specification is to enhance the model, but it is not taken into account by synchronization and table generation with **MEGA Database Builder**.*

To create a constraint:

1. Click the **Constraint**  button in the diagram objects toolbar.
2. Click one of the associations concerned by the constraint, for example "Association 1", then drag the mouse to the second association, for example "Association 2" before releasing the mouse button. The **Add constraint** dialog box opens.
3. Enter the name of the constraint, "Constraint 1", then click **Create**. The constraint now appears in the diagram.



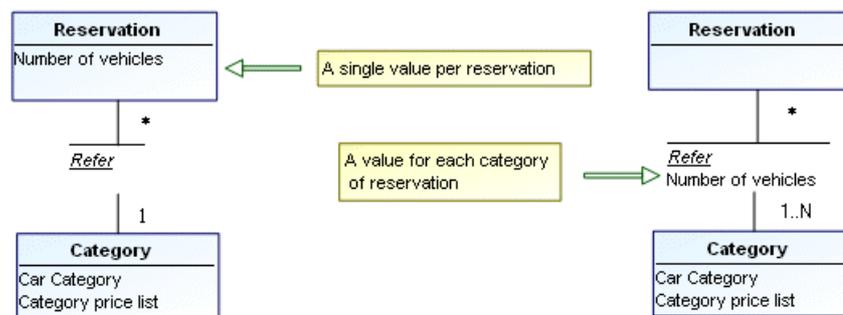
☺ Save your work regularly using the button 

NORMALIZATION RULES

Normal forms are rules that are designed to avoid modeling errors. Currently, there are six or seven normal forms. Here are the first three.

First Normal Form

Rule: The value of an attribute is uniquely set when the object(s) concerned are known.



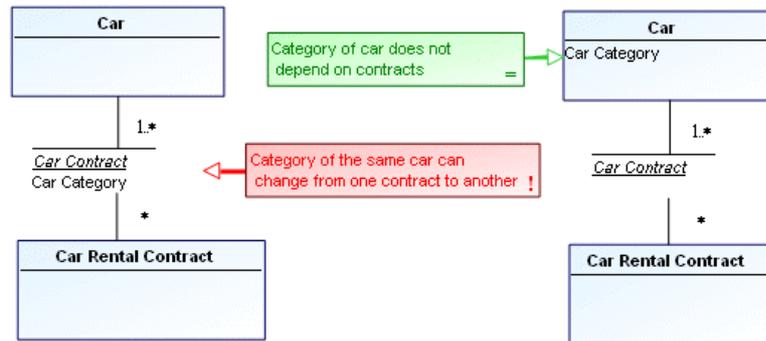
If the number of vehicles is an attribute of the “Reservation” entity, you can only indicate the total number of vehicles for a reservation. You must therefore make one reservation per category of rental vehicle (multiplicity of 1).

If the number of vehicles is an attribute of the association, you can specify the number of vehicles reserved for each category in the association. You can therefore make a single reservation for several categories of vehicles (multiplicity of 1..N).

See also:

Second Normal Form

Rule: The value of an association attribute is set only when all the entities concerned are known.



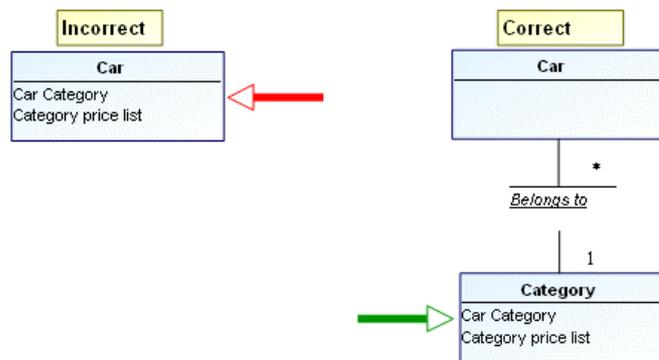
If the car category is an attribute of the "Car Contract" association, this assumes that the car category may change from one contract to the next, which would not be very honest.

If the car category is to be independent of the contract, it must be an attribute of the "Car" entity.

See also:

Third Normal Form

Rule: An attribute depends directly and uniquely on the entity it describes.



If the "Category Price List" is an attribute of the "Car" entity, this indicates that two cars in the same category can have a different "Category Price List".

See also:

SQL IDENTIFIERS AND OPTIONS

Defining an Entity Identifier

Each object has an identity that characterizes its existence. The *identifier* provides an unambiguous way to distinguish any object in an entity. It is one way to distinguish between two objects with identical attribute values.

 An identifier consists of one or several mandatory attributes or roles that enable unique identification of an entity.

Example



Customer number 2718 executes Reservation number 314159.

Each entity has a unique identifier whose value can be used to find each of its instances.

By default, the identifier is *implicit*. At synchronization, a primary key will be automatically generated from the entity name.

Implicit identifier

On the implicit identifier it is possible to define length, decimal and datatype.

 **The datatype of an implicit identifier cannot be composite (example: datatype in form of address).**

You can define identifier properties:

- In the identifier properties dialog box, **SQL** tab.
- In the database editor.

To define identifier properties from the database editor:

1. Right-click the database concerned.
2. Select **Edit**.
3. In the editor conceptual view, expand the entity that contains the identifier then select the identifier.

Its characteristics appear in the left pane of the editor. You can modify the values shown by clicking the corresponding box.

For more information on the database editor, see ["The Database Editor"](#), page 116.

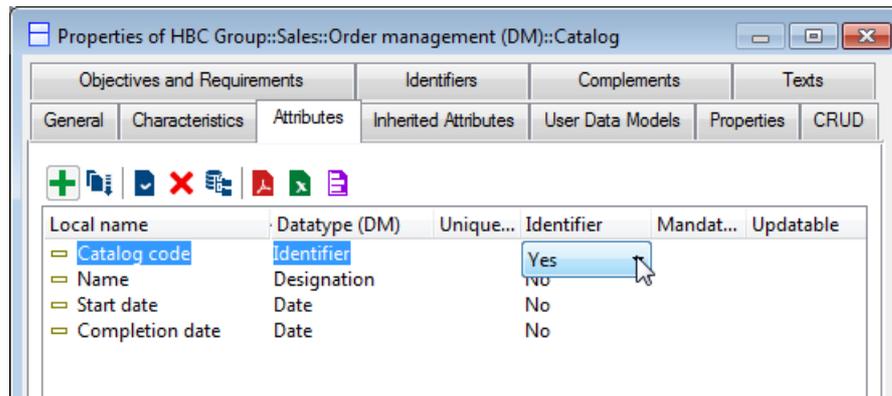
Configuring PK column names at synchronization

At synchronization in conceptual > logical mode, the entity identifier becomes the primary key of the table. If the identifier is implicit, a column is automatically created. It is possible to configure the PK column name construction rule. See "Configuring PK column names (implicit identifier)", page 109.

Identification by an attribute

It is also possible to select one of the attributes of the entity as its identifier.

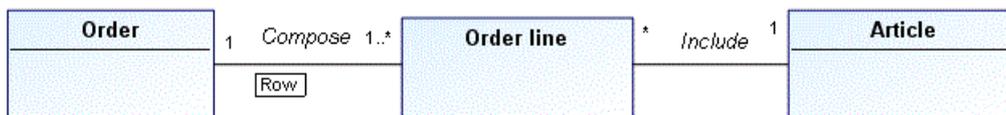
To do this, in the **Attributes** tab of the entity properties dialog box, select value "Yes" for the **Identifier** characteristic of the selected attribute.



A candidate key comprising this attribute is then automatically created for this entity. After synchronization to the logical model, the corresponding primary key will be created in the table generated for the entity.

Relative identification

In certain cases, it is useful to identify an entity relative to another entity.



In the above example, an order line is identified relative to the order it is from. Its identification could consist of the order number followed by its order line number (example: 3rd line in order A721).

To identify an entity relative to another entity:

1. Open the properties dialog box of the entity to be identified, for example "Order line".
2. Select the **Identifiers** tab.
3. Select the identifier and click the **Add elements**  button. A window lists elements to which additions can be made.

4. In our example, select the "Command" role and click **OK**.

Composite identifier

You can define an identifier by:

- attributes
- roles with a multiplicity of 1
- generalizations

When a role is added to the entity identifier, a foreign key and its columns are generated in the corresponding table at synchronization.

By default, associations do not have their own identifiers: an association is identified by the identifiers of the linked entities.

An identifier can comprise one or several elements.

To define identifier composition:

1. Open the properties dialog box of the entity.
2. Select the **Identifiers** tab.
3. Select the identifier and click the **Add elements** button.
A list of available elements appears.
4. Select an element and click **OK**.

The elements that comprise an identifier are displayed in the form of a list. You can specify the order of these elements by clicking the **Order** button  in the **Identifiers** tab.

Other buttons allow you to:

-  **Delete** an element
-  Access **Properties** of an element

Entity SQL Options

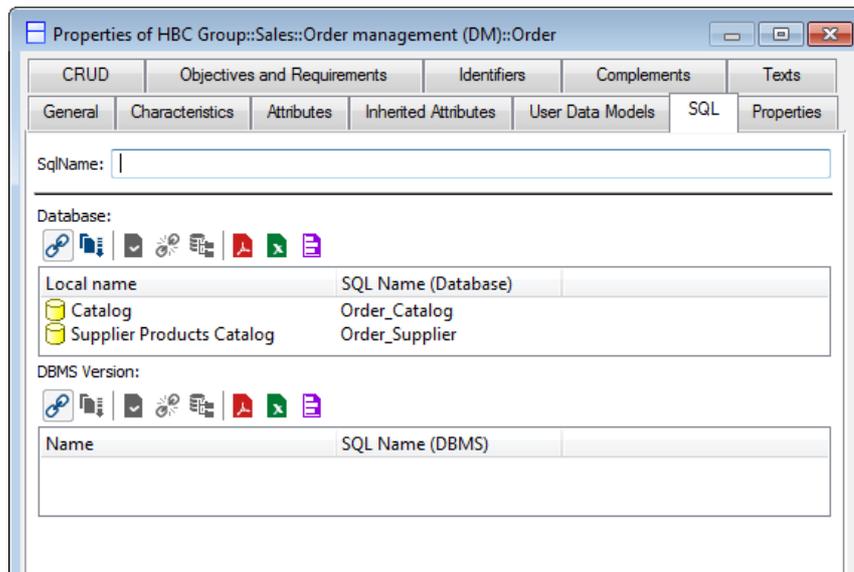
SQL options available in an entity properties dialog box concern synchronization, which enables translation of a data model expressed in entity/association formalism to a logical model expressed in relational formalism. See "[Synchronizing logical and physical models](#)", page 73.

At synchronization, an entity produces a table. On the entity you can define the name of the table that will be produced.

To indicate the SQL name of the table:

1. Open the properties dialog box of the entity.
2. Select the **SQL** tab.

When an entity is connected to several databases, you can assign to the table a name differing from one database to another.

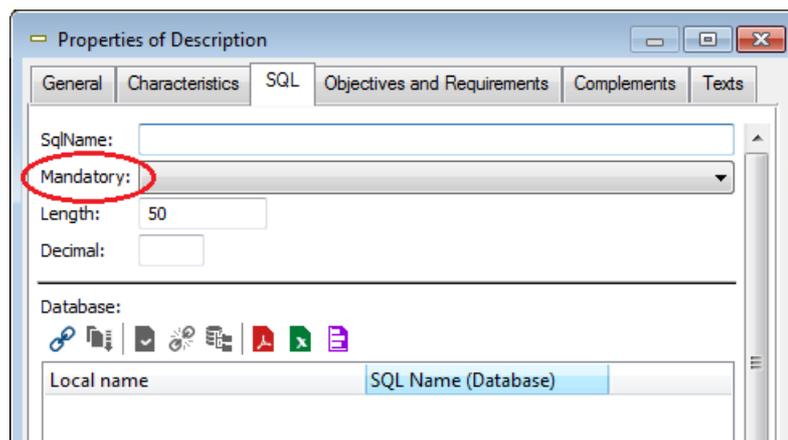


You can also define a default name as a function of the target DBMS.

When fields **SQL Name (Database)** and **SQL Name (DBMS)** indicate different names, the name defined at database level takes precedence at synchronization.

Attribute SQL Options

In the **SQL** tab of the attribute properties dialog box, the **Required** field is used to specify that an attribute is required when generating tables. In this case, a "Not Null" clause is generated on the corresponding column.

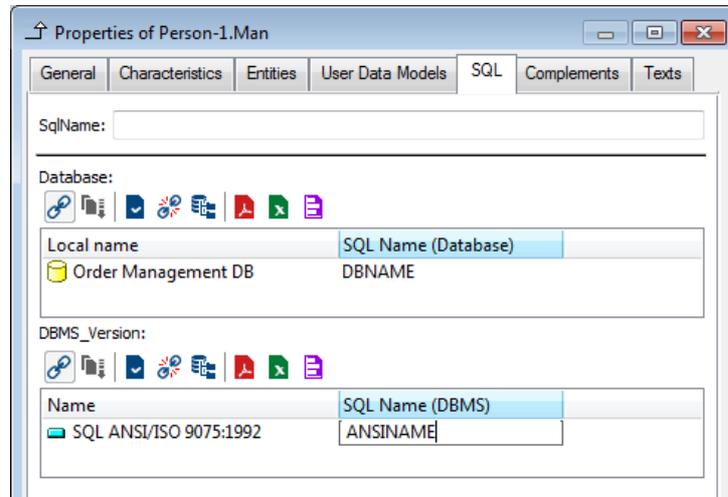


Generalization SQL Options

As for an entity, it is possible to give an SQL name to a generalization, according to a database or a target DBMS. See ["Entity SQL Options"](#), page 44.

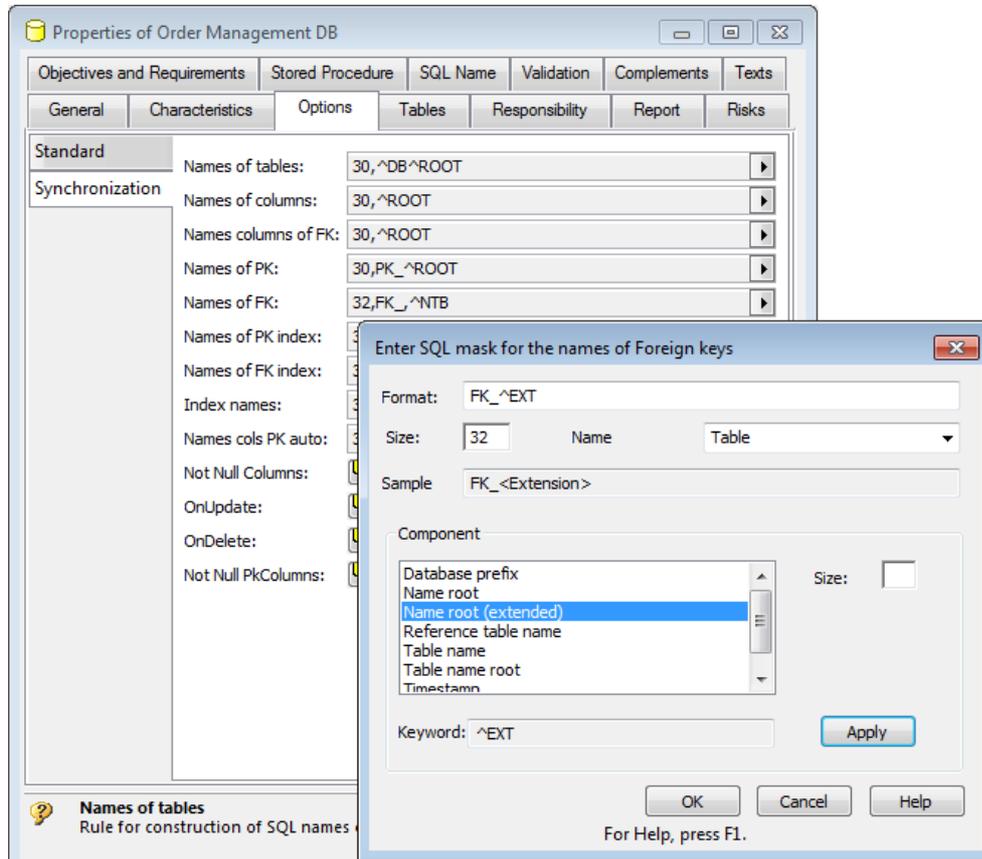
To indicate the SQL name of a generalization:

1. Open the properties dialog box of the generalization.
2. Select the **SQL** tab.



This SQL name can be used in naming foreign keys derived from synchronization, with key word ^EXT:

For more details on naming relational objects, see ["Configuring Name Generation"](#), page 105.



Properties of a database, Options tab, **Synchronization** subtab

DATA MODELING RULES

Repository build is subject to modeling rules. Depending on the product or products you have, **MEGA** provides a certain number of rules you can apply to objects you create so as to check their consistency.

In a data model creation context, **MEGA Database Builder** provides "Data modeling regulation". This is used to check business function data models and those to be transformed in relational formalism.

For more details, see the **MEGA Common Features** user guide, "Generating Documentation", "Checking Objects".

Checking a Data Model

To check a data model:

1. Right-click the data model name.
2. Select **Check > Regulation with Propagation**.
A dialog box appears displaying the list of regulations that can apply to the object.
3. Select a regulation and click **OK**.

The effect of this command is to check the object and its connected objects. For a data model you therefore obtain results on the objects it contains, such as entities, attributes and associations.

Check results appear in an HTML report.

THE CLASS DIAGRAM



In addition to data model formalism, **HOPEX** makes another notation available to describe logical data. This is UML (Unified Modeling Language) notation.

Data description in UML notation is carried out in a class diagram.

This chapter describes the different concepts handled in this diagram.

- ✓ ["Creating a Package", page 50](#)
- ✓ ["The Class Diagram", page 51](#)
- ✓ ["Classes", page 52](#)
- ✓ ["The attributes of a class", page 54](#)
- ✓ ["Association Roles", page 55](#)

CREATING A PACKAGE

A package partitions the domain and the associated work. It is designed to contain the modeled elements. Graphical representation of all or of certain of these elements is in a class diagram.

A database can be connected to a data model from the time of its creation. It is on this database that the different data processing tools can then be run (generation, synchronization etc.). The database package is the default owner of the classes and associations represented in the class diagram.

Creating a Package

To create a package:

1. On the desktop, click the navigation menu, then **Logical data**.
2. In the tree on the left, in the **Packages** folder, click **All Packages**.
The list of packages appears in the edit area.
3. Click the **New** button.
The dialog box for creating a package opens.
4. Enter the **Name** of the package.
5. If appropriate, enter the name of the **Owner**.
6. Click **OK**.
The package created is added to the list of packages.

☛ *When the **OK** or **Create** buttons are grayed, this is because the requirements for the dialog box in which they appear have not been completed.*

Connecting a Package to a Database

To create a package from a database:

1. Click on the navigation menu, then on **Logical Data**.
2. In the tree to the left, in the **Database** folder, click **All Databases**.
3. Click the database icon and select **New > Data Package**.

To connect an existing package to a database:

1. Right-click the database and select **New > Data Package**.

THE CLASS DIAGRAM

A class diagram is used to represent the static structure of a system, in particular the types of objects manipulated in the system, their internal structure, and the relationships between them.

A class diagram includes:

- Classes, which represent the basic concepts (client, account, product, etc.).
- Associations, which define the relationships between the different classes.
- Attributes, which describe the characteristics of classes and, in certain cases, of associations.
- Operations, which can be executed on objects of the class.

☛ *Operations are not taken into account by **HOPEX Database Builder** tools (synchronization, generation etc.).*

Creating a Class Diagram

To create the class diagram of a package:

1. On the desktop, click the navigation menu, then **Logical data**.
2. In the tree to the left, click in the packages list.
The list of packages appears in the edit area.
3. Click the icon of the package concerned and select **New > Class Diagram**.

The new class diagram opens.

Note that when you create a package from a database, a class diagram is automatically created at the same time.

CLASSES

A class groups objects that share the same characteristics and have similar behavior.

Class Properties

Different characteristics are available on classes.

To display the properties of a class:

1. On the desktop, click the navigation menu, then **Logical data**.
2. In the tree to the left, click the **Packages Hierarchy** folder.
The list of packages appears in the edit area.
3. Expand the folder of the class owner package.
4. Right-click the class concerned and select **Properties**.
Its properties dialog box appears.

Different pages are used to define the class properties.

Characteristics Page

The **Characteristics** page is used to enter different characteristics of the class.

- Its **Name**, which you can modify.
 - ☺ *You can also modify the name of a class by clicking directly on the name in the diagram drawing and pressing key F2.*
- The **Owner** package: in a class diagram that describes a package, any class you create is attached by default to this package.
- Its **Stereotype**: a stereotype is a type of modeling element that extends the semantics of the metamodel. Stereotypes of interest in **HOPEX Database Builder** are:
 - **Org-Unit**: an org-unit represents the role played by something or someone within the enterprise environment of the modeled system.
 - **Control**: this stereotype is used for classes that perform processing internal to the system. These generally require contributions from several classes.
 - **Entity**: this stereotype is used to describe classes that are passive; that is classes that do not initiate interactions on their own. An entity object may participate in many different use cases and usually outlives

any single interaction. They represent objects shared between the different actors that handle them.

- **Enumeration**: datatype containing a list of tabulated values.
 - ☛ *It is possible to enter this list of values in the **Characteristics** tab of the properties dialog box of the class.*
- **Expression**: expressions of complex datatypes based on elementary types.
 - ☛ *It is possible to enter the elementary type of an expression in the **Characteristics** tab of the properties dialog box of the class.*
- **Primitive Type**: this stereotype is used to describe the datatypes.
 - ☛ *Other stereotypes can be created by the user.*
- Another characteristic you can specify is the abstraction: an **Abstract** class does not have instances. It is only used to group operations or attributes common to its subclasses.
 - ☛ *For other characteristics related to UML notation, see the **HOPEX UML** guide.*

Text Pages

In the **Texts** page you can add a comment for the class.

Comments can be used to add key information to diagrams when certain details cannot be displayed in the drawing. These comments are included in the document describing the class diagram.

To add a comment:

1. In the **Texts** page, select the **Comment** tab.
2. Enter the comment for the class.
3. Click **OK** to exit the properties dialog box for the class.

THE ATTRIBUTES OF A CLASS

A class can be described by its attributes. An attribute is a named property of a class.

Attribute properties

To open the properties of an attribute:

- 】 Right-click the attribute and select **Properties**.
In the **Characteristics** page, you can specify:
 - Its **Type** described by an expression (see ["Types"](#), page 54).
 - ☛ *Only the specifications of name, type and persistence are taken into account in derivation and generation of tables with **HOPEX Database Builder**.*
 - The **Initial Value** of the attribute, assigned when an instance of the class is created.
 - Whether it is **Read Only**, that is if its value can be modified once it has been specified.
 - ☛ *For other characteristics related to UML notation, see the **HOPEX UML** guide.*

Types

A datatype is used to group characteristics shared by several attributes.

Any class can be used to *type* an attribute for example.

Example: Client, Order, Window, Table.

Classes of the "Primitive type" stereotype are created only for typing attributes or parameters. They are fixed.

Examples of primitive types:

String, Integer, Export address, Monetary amount.

- 】 Click the drop-down list to display the list of types.
 - ☛ *The listed types include the classes in the package, used by the package, and in or used by packages of which it is the client.*

To define the structure of a type:

1. Place the corresponding class in the same diagram or in another diagram.
2. Display its **Properties**.
 - ☛ *For more information on the use of primitive types, see ["Attribute and Column Types"](#), page 145.*

ASSOCIATION ROLES

An association is a relationship that exists between two or more classes. It is possible to describe the different roles played by the classes in associations

Order

It is possible to specify whether or not a role is **Ordered**. For example, for a client order, it can be useful to store the sequence in which its lines appear.

To order a role:

1. Right-click the role concerned and select **Properties**.
The properties window of the role appears.
2. Click the drop-down list then **Characteristics**.
3. Select the is **Ordered** check box.

For example, each square on a chessboard is identified by its row number and column number on the chessboard.

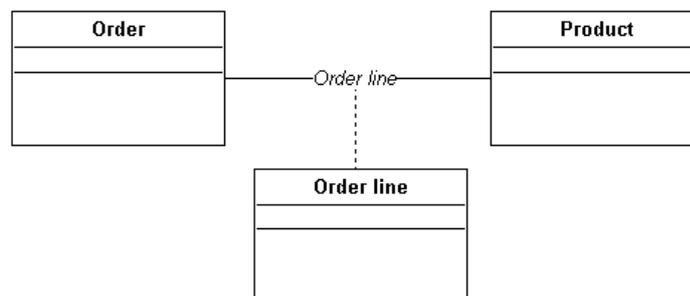


Using Association Classes

It is helpful to create an *association class* in order to specify the characteristics of an association.

 An association class is an association that also has class properties as attributes.

For example, the quantity of the requested product needs to be specified on each order line.



To create an association class:

- 1 In the class diagram, create the class and, using the **Link** button, connect the class to the association.
The association class now appears in the drawing, linked to the association by a dotted line.

DATABASE AND PHYSICAL MODELING



The data model enables representation of the static structure of data at business level. A database is the object that enables storage and organization of data for use by programs corresponding to distinct applications, to facilitate the independent evolution of the data and the application programs.

You can connect a data model and associated diagram to a database. You can then create the corresponding relational diagram.

The points covered here are:

- ✓ ["Database", page 58](#)
- ✓ ["Relational Diagram", page 60](#)
- ✓ ["Logical Modeling", page 62](#)
- ✓ ["Specifying Primary and Foreign Keys", page 69](#)
- ✓ ["Database Modeling Rules", page 72](#)

DATABASE

On a database, and depending on the target DBMS, control parameters of the various data processing tools (synchronization, generation, reverse generation etc.) will be defined.

Creating Databases

To create a *database*:

1. Click on the navigation menu, then on **Physical Data**.
2. In the left tree, expand the **Databases** folder.
3. Click **My databases..**
The list of databases appears in the edit window.
4. Click the **New** button.
The creation window of the database appears.
5. Enter the name of the database.



A database enables specification of data physical storage structure.

6. Select the owner library if required.
7. Click **OK**.
The database created appears in the list of databases.

Database Properties

To access the properties of a database:

1. Click on the navigation menu, then on **Physical Data**.
2. In the left tree, expand the **Databases** folder.
3. Click the folder containing the database concerned.
The list of databases appears in the edit window.
4. Select the database concerned and click **Properties**  in the edit window.
The properties window of the database appears.
5. Click on the drop-down list to access the different properties pages.

Properties pages are used to:

- Access the **Components** of the database (tables, physical views, data groups, etc.).
- Modify the **Characteristics** of the database (name, target DBMS, etc.).
- Define **Responsibilities**.
- Define the **Risks** associated, the **Standards** used, the **Objectives and Requirements**.
- To define the **Options** linked to:
 - the generation of tables. See ["Configuring Database Generation", page 170](#).
 - synchronization. See ["Configuring Synchronization", page 103](#).

Connecting a Data Model to a Database

You can create a data model from the database or connect an existing model to it. The data model enables representation of the structure of the database, the entities it contains and their associations.

The database data model is the default owner of entities and associations represented in the data diagram. These owned entities and associations can however be used in other data models.

To create a data model:

- Click the database icon and select **New > Data Model**. This command creates a new datamodel as well as its associated diagram. See ["Data Models", page 20](#).

To connect a data model:

1. Click the database icon and select **Connect > Data Model**. The query dialog box appears.
2. Click **Find**.
3. Select the desired data model and click **Connect**.

Importing a DBMS version

At creation of a new repository, only the **SQL ANSI/ISO 9075:1992** DBMS version is installed. If you choose another target DBMS for a database, you must import the solution pack for SQL datatypes that is compressed in the **DBMS SQL Type.exe** file.

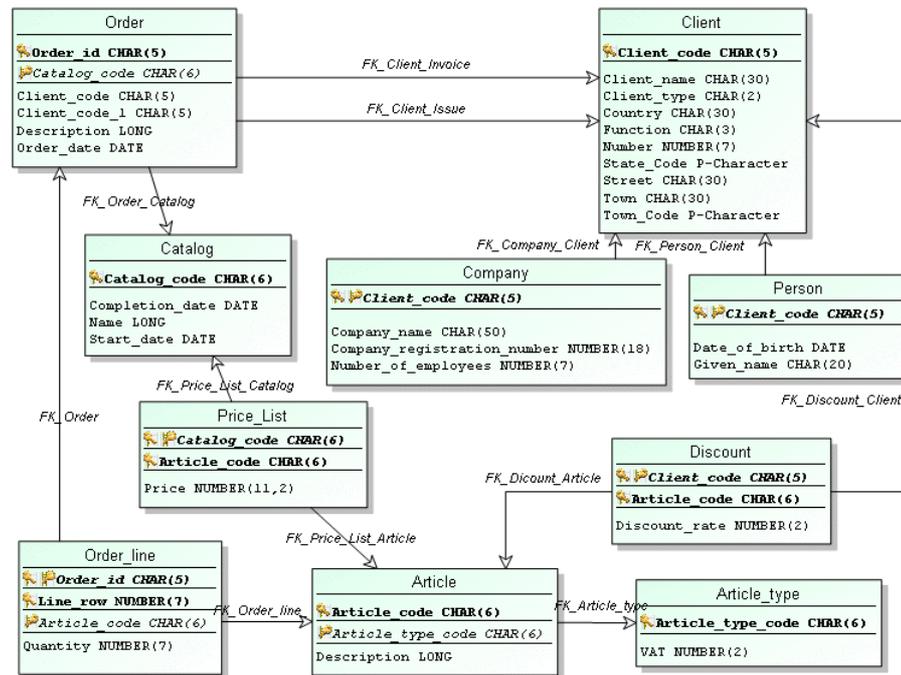
The **DBMS SQL Type.exe** file is available in the Utilities\Solution Pack of your HOPEX installation directory.

Once the solution pack is imported in **HOPEX Database Builder**, you can select the appropriate DBMS in the database properties dialog box.

See also ["Configuring Synchronization", page 103](#).

RELATIONAL DIAGRAM

The Relational Diagram (RD) describes a database: it represents the physical data structures used by application programs.



Description in **HOPEX** of relational diagrams makes it possible to interface with the selected DBMS, guaranteeing semantic consistency between design data and production data.

Creating the Relational Diagram

The relational diagram is generally built in two phases:

1. Automated synchronization of the data diagram or diagrams produces the "raw" diagram.
See ["Synchronizing logical and physical models", page 73.](#)
2. Optimizing the diagram, or denormalization, to take into account the data access requirements of the application and to fine-tune the database performance.
See ["Denormalizing logical and physical models", page 125.](#)

The key concept in a relational diagram is the table, which is derived from an entity or association.

 *Logical structure of data, used as the reference for the switch to production, the table is the central element of the database. A table is accessible by means of a primary key, and if necessary foreign keys; it is described by an ordered sequence of columns. A table is generally derived from an entity or association.*

A **table** is accessible by one or several keys, whose type indicates whether they are primary or foreign keys. It is possible to define indexes for a table, specifying their sort order (ascending or descending) and whether they are unique. Keys and indexes are connected to the columns that they contain.

Creating objects in the diagram

To create a key or an index in the relational diagram:

1. Click the table concerned.
The list of commands associated with the table appears.
2. Click **Key**  or **Index** 

 *Check that the columns used by the key or index already exist in the table.*

You can also use the **Components** page in the properties dialog box of the database to create these objects. See ["Creating a Key", page 66](#) and ["Creating an Index", page 67](#).

To create a foreign key:

1. Select the **Key**  button, click the first table, and then hold the button down while dragging the mouse to the second table.
The creation dialog box opens.
2. Specify the name of the key and click **Add**.
A second window asks you if you want to automatically create the columns of the foreign key from those of the primary key.
3. Click **Yes** to validate or **No** to create.

Configuring display of relational diagrams

As for data diagrams, you can specify which elements are to appear in the diagram:

- Either by using the **Views and Details** button to indicate globally the types of objects to be displayed in the diagram.
- Or by using display options that enable definition of which object characteristics should be presented.

To configure the display for a selected object:

1. Right-click the object and select **Shapes and Details**.

 *When configuring the display of an object, the **Display** dialog box first shows the shapes that can be used to represent the object. Selecting an element in the tree causes its content to appear.*

LOGICAL MODELING

A database is a set of data organized for use by distinct applications, to facilitate the independent evolution of the data and the application programs.

A database consists of tables, columns, keys and indexes:

- A **table** is the logical entity where columns are stored.
- A **column** is contained in a table.
- Just as an identifier uniquely identifies a class, the **primary key** for the table uniquely identifies a row in the table.
- A **foreign key** accesses another table, and imposes consistency between the corresponding columns in the tables concerned.
- An **index** accelerates access to data. It can be unique or not, and may be ascending or descending.

Database Tables

Database tables can be viewed or updated in two ways:

- In its relational diagram, that is the diagram of the tables in the database.
- In its properties, in the **Components** page.
The components page displays the database tables.
The name of the associated **Data Group** is indicated if applicable.

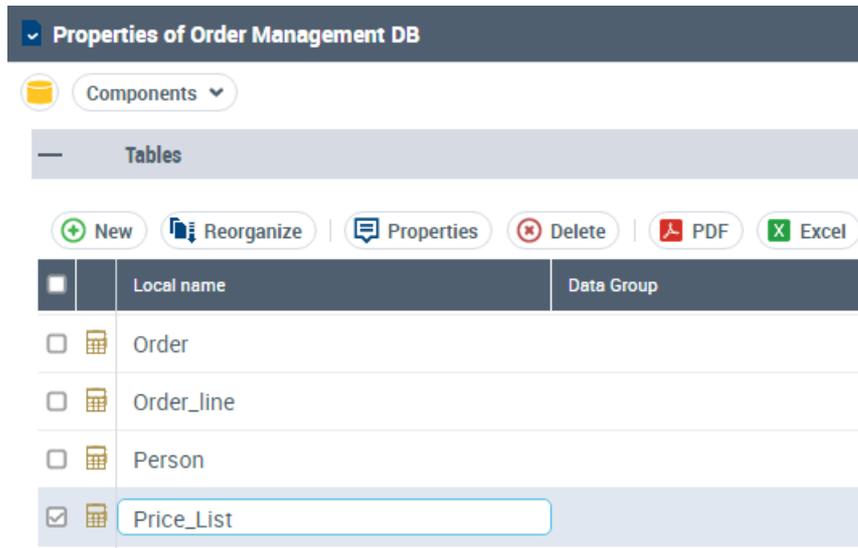
Creating a table

To create a table from the properties of the database:

1. Right-click the database and select **Properties**.
The properties window appears.
2. Click the drop-down list then **Components**.
3. In the **Tables** section, click **New**.
The new table appears. It is named "Table 1" by default.

To rename it:

- › Double-click the name of the table and enter the new name.



Deleting a table

To delete a table and its columns, keys and indexes:

- › Right-click the table and select **Delete**  .
A message requests confirmation.

☛ *The deleted table will not be automatically recreated at a new synchronization. To recreate a table, at the synchronization results validation step validate the creation action proposed for this table (select the corresponding check box). See "Step 4: Validating results", page 82.*

Table Columns

Viewing columns

To view the columns of a table:

- 1. Right-click the table concerned and select **Properties**. The properties window appears.
- 2. Click the drop-down list then **Components**. The component page displays the **Columns** section. Presented for each column are:

- Its **Local Name**
- Its **Datatype**

☛ *The administrator can add to the list of datatypes (see "Creating New Datatypes", page 158).*

- Its length **Ln** and its number of decimals **Dcml** where appropriate.
- The value of its **NotNull** attribute.
- Its **default value**: on generation of the table, the default value taken is that of the attribute from which it originated. If no initial value is specified for the attribute, or if you want to modify the value of a column, enter a value in this field.
- The fact that the column is connected or not connected to a primary key (PK) or foreign key (FK). This is indicated by **Y** ("Yes") or **N** ("No").

You can modify the **Local Name** for a column by clicking its name and then entering the new name. This local name will be used in the script generated for the table.

An **SQL Name** can be specified directly in the **SQL** page of the properties of an attribute in the data diagram. Then all columns created from this attribute will have the same local name. In addition, the name will be reused during successive synchronizations, including total or partial reinitializations.

It is also possible to modify the value of other column characteristics.

☛ *These modifications will be retained in subsequent synchronizations.*

It is possible to create columns not derived from attributes in the data diagram whether in a table generated or created by the user.

Creating a column

To create a column:

1. Open the properties of the table concerned.
2. In the **Components** tab, **Columns** section, click **New**.

☛ *When creation of a column is not carried out from the **Properties** of a table, but for example from the explorer, it is necessary to previously select the table that will contain it, otherwise a message will indicate that creation is impossible.*

Deleting a column

To delete a column:

1. Right-click the column and select **Delete**.

☛ *The deleted column will not be automatically recreated at a new synchronization. To recreate a column, at the synchronization results validation step validate the creation action proposed for this column (select the corresponding check box). See "Step 4: Validating results", page 82.*

The **Reorder** button  accesses the Modify Order dialog box.

Modifying Keys and Indexes

The automatic creation of primary and foreign keys, and of indexes on these keys, is indicated in the synchronization configuration.

When these creations are requested:

- The primary keys use the columns corresponding to the identifiers.
- The foreign keys use the columns that are included in the tables because of a constraint association.

An index is created for each key.

It is possible to add to, modify, or delete the keys and indexes proposed during generation. To do this:

1. Right-click the table concerned and select **Properties**.
2. Click the drop-down list then **Components**.
3. The page presents the **Keys** and **Index** of the table.

The following is specified in the **Keys** section:

- The type of key (**Key-Type**): Foreign or Primary.
- In the case of a foreign key:
 - The table referenced.
 - Management of repository integrity on update (**On Update**) and on deletion (**On Delete**); consult the target DBMS documentation for the order types managed.

☛ *When a "migratory" column is created in a table to reflect a constraint association, you can instruct the DBMS to verify the updated value in this column. The DBMS then verifies that this value still exists in the original table (database integrity).*

When an update (**On Update**) or delete (**On Delete**) command is applied to the original table, the DBMS may:

- Update the values in the tables concerned, with the **Cascade** option.
- Do nothing, with the **NoAction** option.
- Prohibit updates or deletes, with the **Restrict** option.
- Reset to the default value in the tables concerned, with the **Set Default** option.
- Set the value to **Null** in the tables concerned, with the **Set Null** option.

The following is specified in the **Index** section:

- Its **Type**: Bitmap, Standard, Unique, Unique where not null.
- Its **Sort Order** (**Ascending** or **Descending**).
- If a grouped index **Clustered**.

☛ *Creation of a column from a key or index is not possible. A column must first be created in the table, then connected to the key or index.*

Creating a Key

To create a key:

1. Right-click the table concerned and select **Properties**.
2. Click the drop-down list then **Components**.
3. In the **Keys** section, click **New**.
The key creation dialog box appears.
4. Select the type of key to be created: "foreign" or "primary". Creation of the key varies according to the type selected.

Primary key

When you select "Primary" type, the key appears in the properties of the table.

To define the properties of the key:

1. Right-click the key and select **Properties**.
In the **Columns** page, you can specify the columns concerned by the key.

It is also possible to specify the primary key of a table in the **Identifiers** page of the properties dialog box of the entity to which the table belongs. See "[Defining an Entity Identifier](#)", page 42.

It is also possible to manually specify the primary key by relating it to elements that can be attributes of the entity or the primary key of another table connected by a constraint association (multiplicity 1).

In all cases, the key specified will be created in the table on synchronization.

Foreign key

When the key created is a foreign key, a list of database tables is presented.

1. Select the reference table to which the foreign key relates.
If the table you select includes a primary key, a dialog box opens.



2. Select **Yes**.
the key appears in the properties of the table.

You can modify the **Local Name** of the key (the full name of the key is composed of the name of the database to which it belongs, followed by the name of the table then the local name: in the above example, "Exchange DB::Concern::Key1").

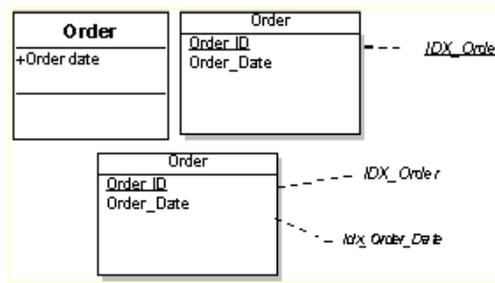
For a foreign key, as when editing a key, it is possible to specify repository integrity management on update (**On Update**) and on deletion (**On Delete**).

Creating an Index

Indexes are created automatically for primary and foreign keys. It is possible to add to the generated indexes columns used frequently in search criteria.

☺ *It is also possible to specify an index in the **Identifiers** page of the properties dialog box of the entity to which the table belongs. An index specified in this way will be created in the table during synchronization.*

Examples of indexes:



To create an index:

1. Right-click the table concerned and select **Properties**.
The properties window appears.
2. Click the drop-down list then **Components**.
3. In the **Index** section, click **New**.
The **Create Index** page appears.
4. Specify the **Local Name** and click **OK**.

Depending on the possibilities offered by the DBMS, you can specify the index **Type**, index **Sort direction** ("Ascending" or "Descending"), and if it is a grouped index (**Clustered**).

It is then possible to select the columns of the key (or index) in the **Columns** page of the properties dialog box of the index.

Adding a Column to a Key or Index

To add a column to a key (or to an index):

1. Right-click the table concerned and select **Properties**.
The table properties dialog box opens.
2. Click the drop-down list then **Components**.

3. In the **Index** section, right-click on the index and select **Properties**.
The properties dialog box of the index appears.
4. Click the drop-down list then **Columns**.
5. Click the **Connect** button.
The query dialog box appears.
6. Find and select the column to be added to the key (index).

It is possible to indicate the sort order of the key or index, which can be "Ascending" or "Descending".

SPECIFYING PRIMARY AND FOREIGN KEYS

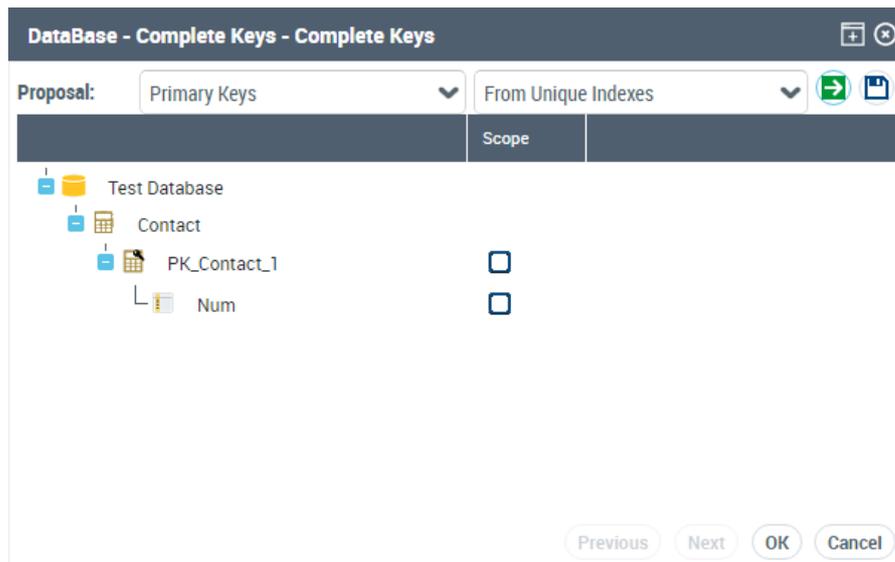
When the keys of a database are not completely specified, you must **Complete** them.

Specifying Primary Keys

To specify the primary keys of the database:

- 1 Right-click the database and select **Complete the table keys**. The **Complete the keys** window appears.

➤ When database specification is completed, the dialog box presents an empty list: no additional specification is required.



The **Proposition** list is used to complete the keys:

- **From unique indexes**: the columns that belong to a unique index are proposed as components of the primary key.
- **From mandatory columns**: these columns are proposed as components of a key.
- **Through name comparison**: if the same column name is found in several tables, the column is proposed as primary key.

Each key is proposed under the table to which it belongs.

To validate a primary key:

1. Select the check box of the **Scope** column corresponding to the key. The associated columns are automatically selected by default. You can eliminate those that correspond to search criteria but are not components of the key.
2. Click the **Apply** button. The **Apply** button removes from the list the propositions of keys explicitly accepted or rejected.

For foreign keys, two keys including the same column on the same table are incompatible: acceptance of one automatically results in rejection of the other.

It is not possible to select several primary keys on the same table: acceptance of one results in rejection of the others.

☺ *You can complete the specification of keys in several stages. This allows you to consult the contents of the database while making your selections. To do this:*

- Click the **Apply** button to save your modifications.
- Click the **Cancel** button to exit this dialog box without starting processing.

Specifying Foreign Keys

To specify the foreign keys of the database:

1. Right-click the database and select **Complete the table keys**. The **Complete the keys** window appears.

☛ *When database specification is completed, the dialog box presents an empty list: no additional specification is required.*

The **Proposition** list is used to complete the foreign keys:

- **From indexes**
- **From name comparison**

If the proposition is made from indexes, it is based on non-unique indexes of the table. The reference table is indicated after the name of the key.

To validate a foreign key:

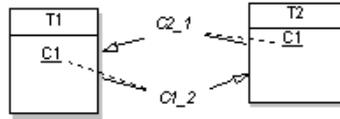
1. Select the check box of the **Scope** column corresponding to the key.
2. Click the **Apply** button. The **Apply** button removes from the list the propositions of keys explicitly accepted or rejected.

If no reference table is specified, the wizard automatically proposes the selection of possible tables. Keys that do not have a reference table cannot be accepted.

On proposition of keys, several tables may be found with an identical primary key. This could for example be the case for tables corresponding to different sub-types of the same entity.

Column Primary Key of Two Tables

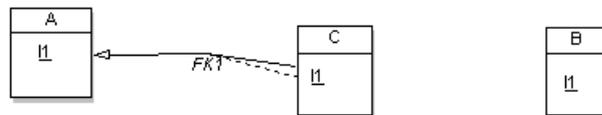
When the same column is the primary key of two tables, the foreign key proposition permits creation of each of these two keys.



A choice is then made of which of the two keys should effectively be taken into account.

Column Primary Key of Three Tables

When the same column is the primary key of three tables, the foreign key proposition permits creation of one foreign key from a column of a table.



The key proposition permits creation of only one foreign key from table C. The other should be added in data entry of tables in the database.

DATABASE MODELING RULES

HOPEX Database Builder provides rules that enable database modeling checks. The physical regulation contains the rules relating to the database relational diagram. It is used to check the corresponding relational diagram in a DBMS.

The physical regulation contains rules relating to technical specifications of the database DBMS. It is used to check consistency of physical parameters of the relational diagram specific to the DBMS.

Checking a database

You can run a check on the database or on a database object.

To check a database:

1. Right-click the name of the database.
2. Select **Administer > Check > Regulation with Propagation**.
When several regulations can apply to the check object, a dialog box asks you to select the required regulation.

The check applies to the database as well as the objects it owns.

Results appear in an HTML report.

For more details, see the **HOPEX Common Features** user guide, "Exploring the repository", "Tools for Checking Objects".

SYNCHRONIZING LOGICAL AND PHYSICAL MODELS



Synchronization is a process that translates a data model expressed in entity/association formalism to a physical model expressed in relational formalism, and vice-versa. It therefore ensures correspondence of objects in the two models. The procedure should be carried out periodically. Throughout the modeling project, these models are each subject to their particular changes. Synchronization intervenes when we wish to compare the two models and automatically restore canonical mappings that connect them.

At synchronization, mapping rules are applied by default. You can modify mappings and arrange that synchronization keeps any customizations that have been carried out over time. See ["Denormalizing logical and physical models", page 125](#).

Synchronization is carried out in the database editor. It is available with "Advanced" access to the **HOPEX** repository.

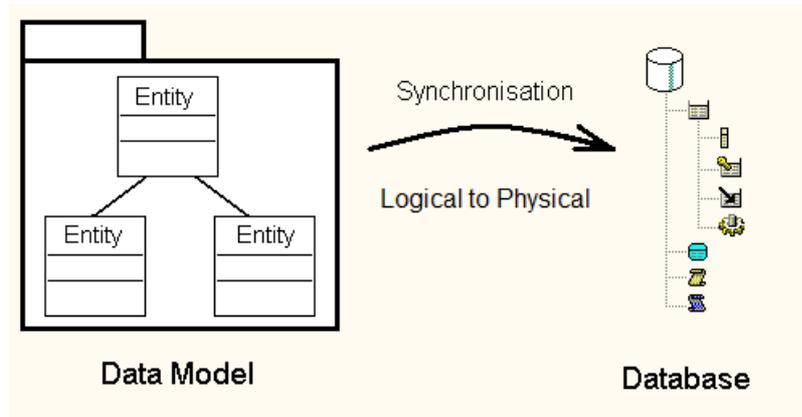
 **Synchronization of models of a database can be in one direction or another - either in physical > logical direction or in logical > physical direction, but not both at the same time. When synchronization direction has been determined, synchronization should not be reversed. Mapping justification between the logical and physical levels is not guaranteed if this rule is not followed.**

See also ["Model Mapping", page 115](#).

- ✓ [""Logical to Physical" Synchronization Rules", page 74](#)
- ✓ ["From the Logical Model to the Physical Model", page 80](#)
- ✓ ["Reduced Synchronization \(Logical to physical mode\)", page 88](#)
- ✓ ["Running Synchronization After Modifications", page 94](#)
- ✓ ["From the Physical Model to the Logical Model", page 97](#)
- ✓ ["Configuring Synchronization", page 103](#)
- ✓ ["Diagram Synchronization", page 111](#)

"LOGICAL TO PHYSICAL" SYNCHRONIZATION RULES

The following rules are applied for transforming data models into relational formalism.



➤ See also ["Configuring Name Generation"](#), page 105 and ["Attribute and Column Types"](#), page 145.

Entities

General rule

- Any non-abstract entity of the model becomes a table.
- The entity identifier becomes the primary key for the table. If the identifier is implicit, a column is automatically created. See ["Configuring Name Generation"](#), page 105.
- Entity attributes become columns in the table.
- Mapping rules are applied to determine the column datatypes from the datatype (DM) of each attribute. The possible configurations depend on the DBMS.

➤ For more detailed information, see ["Attribute and Column Types"](#), page 145.

Sub-entity

- The foreign key reflecting dependency between the sub-entity and its super-entity is created.

Abstract entity

An abstract entity does not produce a table at synchronization.

If constraint associations point to an abstract entity, the corresponding foreign keys are not created, but columns corresponding to the foreign keys are created to respect table integrity.

When a sub-entity is abstract, all columns and foreign keys of the corresponding table are taken by the table corresponding to the super-entity.

Conversely, when a super-entity is abstract, all columns and foreign keys of the corresponding table are taken by the table corresponding to the sub-entity.

To define an abstract entity:

1. Open the properties dialog box of the entity.
2. Click the **Characteristics** tab.
3. In the **Abstract** box, select "Yes".

Realized entity

An entity is said to be realized if it produces creation of a table at synchronization.

A "not realized" entity is treated as an abstract entity.

Unlike the "abstract" property which characterizes the entity in all use cases, the "realized" concept applies only in the context of database synchronization. See ["Realized mode", page 85](#).

Associations

Constraint associations (multiplicities: 0,1 or 1,1)

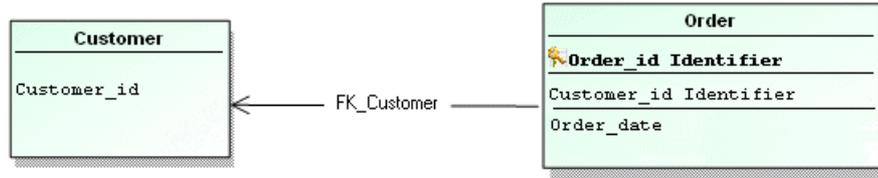
A constraint association is a binary association one role of which has maximum multiplicity 1. In this case, it is not necessary to create a table corresponding to the association. Simply add a column to the table that corresponds to the entity.

A constraint association (one of its maximum multiplicities is 1) does not result in a table. In the following example, an order has only one customer.



Synchronization of this data diagram produces one of the two following results:

The association does not result in a table.



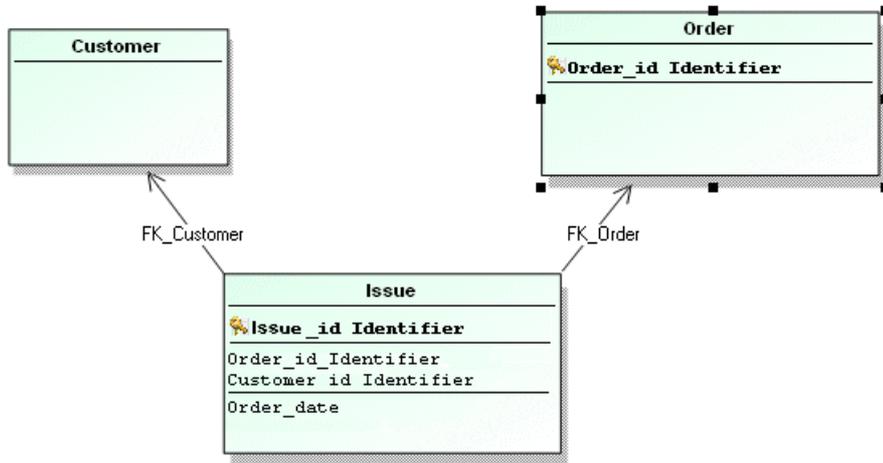
- A column corresponding to the key for the "Customer" entity is created in the "Order" table.
- A column is also created for each attribute of the association.
- A foreign key "FK_Customer" is added to check the "Customer_ID" column in the "Order" table. It indicates that the possible values for the "Customer_ID" column in the "Order" table are the values that already exist in the "Customer ID" column of the "Customer" table.

The foreign key is created from the entity identifier.

The association is transformed to a table

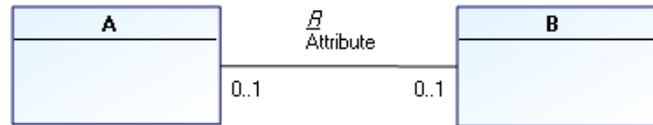
For the association to be transformed into a table:

1. Open the properties dialog box of the association.
2. Click the **Characteristics** tab.
3. In the **Potential Mapping** field, select "Table".



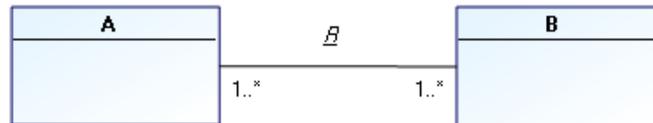
Constraint associations (multiplicities: 0,1 and 0,1)

In this particular case, the combination of multiplicities is ambiguous. There is nothing that can be used to decide which table should contain the column corresponding to the attribute.



Synchronization proposes a column in each table.

Deadlocks



The multiplicities 1..X, 1..X indicate that each of the two objects must be connected to at least one object of the other type in order for it to exist.

This poses problems when creating the first object of each type. In fact:

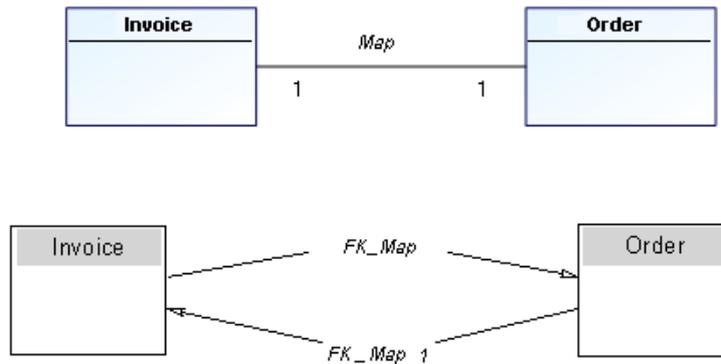
- An object of type A must exist in order to create an object of type B and then connect them.
- Conversely, an object of type B must exist in order to create an object of type A and then connect them.

This is the case for the following multiplicity combinations:

- Multiplicities of 1..*, 1..*
- Multiplicities of 1, 1..*

However, it is not physically impossible to resolve such cases, because the problem is limited to creating the first object of each type. In addition, no foreign key is generated for verifying data integrity in the first case, and only one in the second case, so there is no resulting deadlock situation.

Multiplicities 1, 1 generate several obligatory foreign keys that will become deadlocked:



This situation results in complete deadlock, because in order to meet the constraints, several tables must be created at the same time.

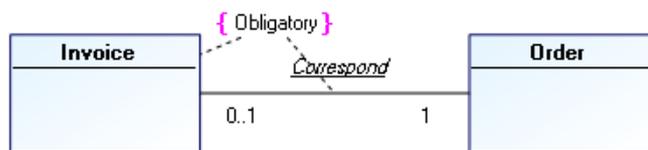
Certain DBMSs prohibit creation of tables of this type.

For correct synchronization, situations such as this should be avoided.

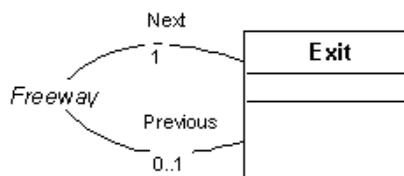
- ☛ **Select one of the foreign keys and assign it a multiplicity of 1, then assign 0..1 to the other. You can also pull the foreign key from the table after synchronization, but this is less convenient.**

You can still impose the minimum multiplicity of 1 by adding a constraint as shown below. This constraint will not be taken into account in the synchronization.

☛ See "Constraints", page 37 for further information.

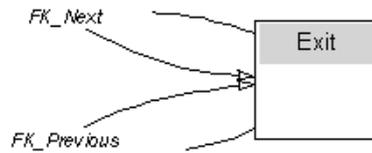


Here is another example using a reflexive link: Reflexive link 1, 1 or 0..1, 1



We want to show that each exit from a freeway comes before an exit and comes after an exit.

When modeled this way, all exits must be created at the same time, because each exit must have a previous exit already created.



To avoid this, set the multiplicities to 0..1 and 0..1.

Non-constraint association

An association where maximum multiplicities are not 1 will have a corresponding table:

- A column is created for each attribute of connected entities identifiers.
- The primary key for the table uses all these columns.
- A foreign key is also built for each connected entity.
- An additional column is created for each attribute of the association.

Before starting synchronization it is advisable to check validity of the data diagram and check that synchronization configuration is correct. See ["Data Modeling Rules", page 48](#) and ["Preparing Synchronization", page 103](#).

Associative class (UML notation)

Associations connecting associative classes are not included in synchronization.

FROM THE LOGICAL MODEL TO THE PHYSICAL MODEL

This section explains how to synchronize the logical model of a database (represented by a data diagram) with the corresponding physical (relational) model.

Although synchronization of the relational model from the data diagram essentially concerns entities, it can also be carried out more generally from classes of a class diagram.

Synchronization in the reverse direction, from a physical model to a logical model, is also possible but not on the same database. When synchronization direction has been selected for an object, synchronization in the other direction will no longer be possible.

See ["From the Physical Model to the Logical Model"](#), page 97.

The points that follow present synchronization of a database. You can also run reduced synchronization, in other words on a specific object of the database. See ["Reduced Synchronization \(Logical to physical mode\)"](#), page 88.

Running Synchronization

Logical to Physical synchronization consists of building the physical model from the logical model, in other words of creating tables and columns corresponding to data diagram entities and attributes:

The synchronization tool is available in the **Logical data** navigation pane. You can also open the synchronization tool directly from the database concerned.

To run a Logical to Physical synchronization on a database:

1. Click on the navigation menu, then on **Logical Data**.
2. In the navigation pane, click **All databases**.
The list of repository databases appears in the edit window.
3. Right-click the database and select **Synchronization (logical to physical)**.
The synchronization wizard opens.

Step 1: Selecting the source objects to be synchronized

To define synchronization scope:

1. In the logical view tree, expand the list of objects contained in the database.

- By default, all objects are selected and therefore included in synchronization. To exclude an object from the synchronization, clear it from the **Scope** column. When an object is excluded, its mapping is also excluded.

DataBase Selection

Define the scope of your synchronization:

	Scope	Not Realiz...	Name	ExpressionType
Order Management DB	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
Order management (D...	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
Data diagram	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Article	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Article type	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Catalog	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Client	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Company	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Order	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Order line	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Person	<input checked="" type="checkbox"/>		HBC Group::Sales::...	
+ Discount	<input checked="" type="checkbox"/>		Article Discount Cli...	

Previn

- By default, all the objects are "realized", in other words, they give way to the creation of an object during synchronization. To specify that an object is "not realized", select it in the **Not realized** column. For more detailed information, see ["Realized mode", page 85](#).
- When the list of objects has been defined, click the **Next** in the wizard.

Step 2: Synchronization options

From the synchronization options, you can:

- in a case where objects of the logical view have already been synchronized, synchronization will start from zero and will delete existing target objects.
 -  **When models have already been synchronized, so that mappings are taken into account at a new synchronization, make sure the "Target object reinitialization" option is cleared.**
- Recalculate target object names: names of physical objects are recalculated as a function of those of the source objects. This means that any manual modification of physical object names is canceled.
- Take account of optimizations: all optimizations - including those not selected in the validation step (see step 4) are proposed.
- Take account of deletions: entities, associations and diagrams that have been deleted are included in the scope. Consequently, deletion of corresponding target objects and links is proposed.
See ["Using Options", page 83](#).

Other options concern target object properties update. By default, synchronization updates all properties of each object concerned.

Scheduling

You can run synchronization:

- Immediately
 - As soon as possible (after publication of updates)
 - At a predefined date and time
- 】 When options have been specified, click **Next**.

Step 3: Protecting objects

Synchronization can impact all objects in an existing database.

- 】 To keep an object intact, select it in the **Frozen**.
- 】 Click **Next** to continue.

See ["Protecting Objects", page 85](#).

Step 4: Validating results

The wizard displays the results that will produce synchronization validation.

Objects that will be automatically modified are indicated by a tick.

Icons preceding object names indicate actions that will be executed on the objects.

Actions can be creation , deletion  or update .

An arrow  preceding an object indicates that synchronization has an impact on sub-objects of the object in question.

- » Expand the object to view the modifications concerned.

Validating optimizations

Optimizations are customizations on objects thus removed from automatic synchronization processing.

Optimization examples:

A tick indicates objects that will be modified. If you do not wish to validate modifications relating to certain objects, you must clear the corresponding boxes. This optimization is kept at subsequent synchronizations.

In addition, **HOPEX** deduces optimizations following actions you may have carried out manually. If you have added a table in the physical view without having created the corresponding object in the logical view, synchronization does not select

deletion of this table.  Table1 

So that the object will be deleted, you must select the corresponding box.

- » When actions on target objects have been defined, you can click **Next**.

A report shows actions carried out.

You can close the wizard and view the results in the editor.

Using Options

The combination of the "Take account of optimizations" and "Take account of deletions" options varies depending on the scope of objects you wish to update.

Take account of optimizations

When this option is selected, synchronization proposes all creations, deletions and modifications, including optimizations not selected by default at the validation step.

When the option is cleared, only the modifications selected by default are proposed at the validation step.

This option enables filtering of the synchronization result to present only those modifications that have a real impact on target data.

Take account of deletions

When this option is selected, synchronization includes entities, associations and diagrams that have been deleted. Consequently, deletion of corresponding target objects and links is proposed.

When this option is cleared, the entities, associations and diagrams that have been deleted are not included. Consequently, the corresponding target objects are not modified.

 **This option only applies to entities, associations and diagrams. For other deleted object types (attributes, identifiers, etc.), the impact on target objects and links is conditioned not by this option, but by the object that contains them.**

This option enables limitation of impact of a synchronization strictly to the source scope defined by the user, excluding any object not explicitly declared in the scope. This option can be associated with synchronization scope for use case types.

Possible option combinations

1. "Take account of deletions" option selected and complete synchronization scope

This is a use case that favors complete synchronization between source and target. In this case, all objects have a valid mapping on completion of each synchronization wizard operation. This mode should be used when source and target should be totally consistent.

2. "Take account of deletions" option selected and partial synchronization scope

This use case enables working on the selected scope, while including impact of deleted objects. In particular it enables confirmation of deletions of target objects following the deletion of source objects of entity, association or diagram types: as these source objects have been deleted, it is theoretically not possible to include them in the synchronization scope. Selecting this option makes this choice possible. This mode should be favored when the scope is wide, and the few objects excluded from the scope are only excluded temporarily (for example, new objects for which we wish to delay the impact on the target).

3. "Take account of deletions" option selected and empty synchronization scope

This is a special mode enabling "cleanup" of target objects whose mapping is no longer valid, with no other impact.

4. "Take account of deletions" option not selected and partial synchronization scope

This use case enables working strictly on the selected scope, excluding any impact outside this scope. In particular it avoids deletions of target objects following the deletion of source objects of entity, association or diagram types: as these source objects have been deleted, it is theoretically not possible to exclude them in the synchronization scope. Clearing this option makes this choice possible. This mode should be favored for a specific synchronization on a restricted scope, which does not include total consistency of source and target. In addition, it is the fastest mode.

5. "Take account of deletions" option not selected and complete synchronization scope

This combination is in principle an infrequent use case. It corresponds to a work mode in which deletion of source objects has no effect on the target; in other words no target object created is ever deleted when this mode is activated.

6. "Take account of deletions" option not selected and empty synchronization scope

This combination has no effect.

Protecting Objects

You can protect an object so that synchronization will have no impact on it. This excludes the object from synchronization without it disappearing.

There are two object protection modes; one upstream and the other downstream.

Frozen mode

"Frozen" mode concerns the target object, that which results from synchronization.

When you freeze a relational model table, you also freeze all child objects of this table: no child object is created, modified or deleted by synchronization.

You can freeze objects:

- Before running synchronization, in the database editor.
- When running synchronization, in the options presented in the wizard.
See ["Step 3: Protecting objects", page 82](#).

Realized mode

"Realized" mode concerns the source object of synchronization.

An object is said to be realized if it produces object creation at synchronization.

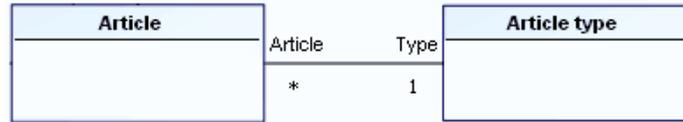
An object not realized does not produce object creation at synchronization but is treated as an abstract object. See ["Abstract entity", page 74](#).

By default, all objects are realized.

You can exclude a source object from synchronization selecting the "Not realized" column on the object in question in the synchronization wizard. This action is available on entities, attributes and associations. The "realized" or "not realized" action is propagated to child objects.

Not realized entity example

The "Article" entity has an association to the "Article Type" entity.



The "Article Type" entity is said to be "not realized".

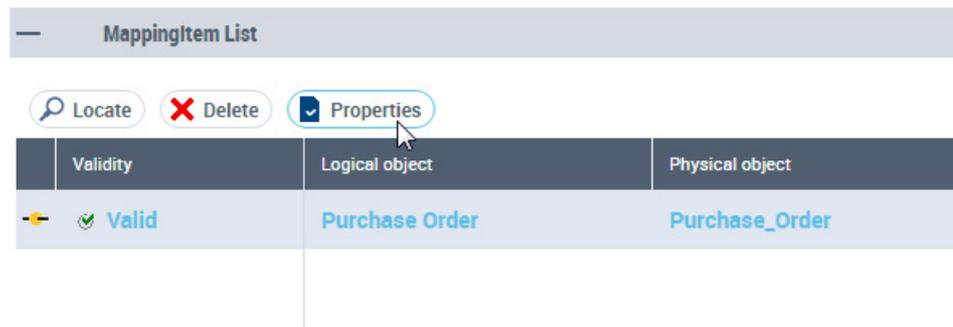
At synchronization, the "Article Type" entity does not produce creation of a table. In the "Article" table, the foreign key to "ArticleType" is not created; however the "Code_Type_Article" column is created. **Synchronization Results: Correspondences**

When synchronization is completed, the tables, columns, keys, and indexes of the physical diagram have been synchronized with the data diagram. They can now be viewed and the desired optimizations made.

Mapping characteristics

For more details on mapping:

- 1 In the mapping editor, select the object mapping element and click **Properties**.
- 2 In the window that opens, click the drop down list and select



Characteristics.

Synchronization scope

By default, all objects in synchronized models are included in the synchronization. You can however exclude an object from the synchronization.

See "Step 1: Selecting the source objects to be synchronized", page 80.

Synchronization state

You can protect an object so that it will not be modified at synchronization by specifying that it is "Frozen".

See also ["Protecting Objects"](#), page 85.

Synchronization direction

The synchronization direction of a mapping indicates which object is updated related to the other.

In certain cases, synchronization is possible in both directions (for example, when two objects that can be synchronized do not yet have a mapping). In other cases, it is only possible in one direction (for example, if one of the two objects is already synchronized) or impossible in both (because each object already has a mapping or because the object types concerned cannot be subject to synchronization). This indication is given in the **Synchronization** box.

Summing up:

- **Bidirectional: synchronization in both directions.**
- **Left to right:** synchronization is from left to right (the object on the right is synchronized with the object on the left).
- **Right to left:** synchronization is from right to left.
- **Never** no mapping is possible between the two objects.

For more details on mapping, see ["The Database Editor"](#), page 116.

REDUCED SYNCHRONIZATION (LOGICAL TO PHYSICAL MODE)

The synchronization function enables synchronization of a logical model and a physical model in the database. In design phase, it is often useful to synchronize part of the current model without having to consider the complete database which can be extremely large. **HOPEX Database Builder** enables limitation of synchronization scope to a set of objects, thus reducing synchronization processing time.

The points that follow detail "Reduced synchronization" mode in direction Logical > Physical directions, but it is also available in Physical > Logical direction.

Reduced Synchronization Source Objects

Reduced synchronization is synchronization applied to an object other than to the database. Reduced synchronization applies only to an object of which the database has already been synchronized.

Objects on which you can run reduced synchronization are:

- Class
- Association
- Package
- Table
- Table file
- Entity (DM)
- Association (DM)
- Data model

Reduced synchronization scope is determined by the object on which you run reduced synchronization.

The following cases illustrate reduced synchronization in the logical to physical direction.

Running from a data model

When you run a synchronization on a data model, by default all objects of the model are selected in the scope of the synchronization; they are all selected in the editor.

Running from a data model entity

When you run a synchronization from an entity (or another object) belonging to a data model, only those objects linked to this entity within the same model are selected by default.

Objects linked to the entity but belonging to another model are displayed in the editor (when they are connected to the target database) but not selected by default. You must select the associated check boxes to take them into account in the synchronization.

The data model of the synchronized entity is taken as scope only if the ownership link between data model and entity is clearly identified: this is the case when you select the entity in the navigation window, but not when you select it in a diagram.

Running on an entity outside context

When you run synchronization on an entity outside context, for example in an explorer window, all objects depending on the modified entity, whether or not included in different models (on condition that these models are connected to the target database) are selected by default in synchronization scope, since no particular model context is identified.

Reduced Synchronization Strategies

At synchronization from an object, the three strategies below can be applied.

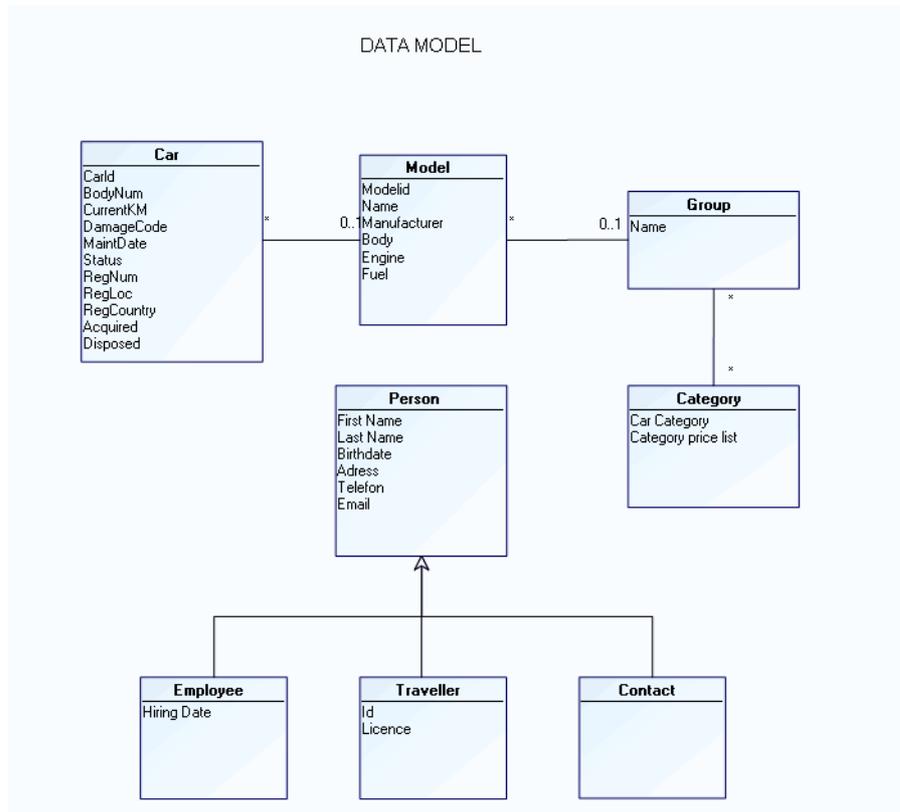
Impact of synchronized object on other objects

This strategy enables definition of synchronization scope from the source object, with the possibility of extending this to all objects dependent on the source object and likely to be affected by its modification.

Example

With this strategy in the model below, reduced synchronization of the "Model" entity allows inclusion of the "Car" entity, which has a constraint association to the "Model" entity.

Logical model



Reduced synchronization results

Indicate objects to dismiss:

	Name	ExpressionType		Scope	Name
	Car Rental				Car Rental
	Car Model				Car
	Car	Car Model::Car			Model
	Model	Car Model::Model			
	Car Rental	Car Rental			

Impact of other objects on synchronized object

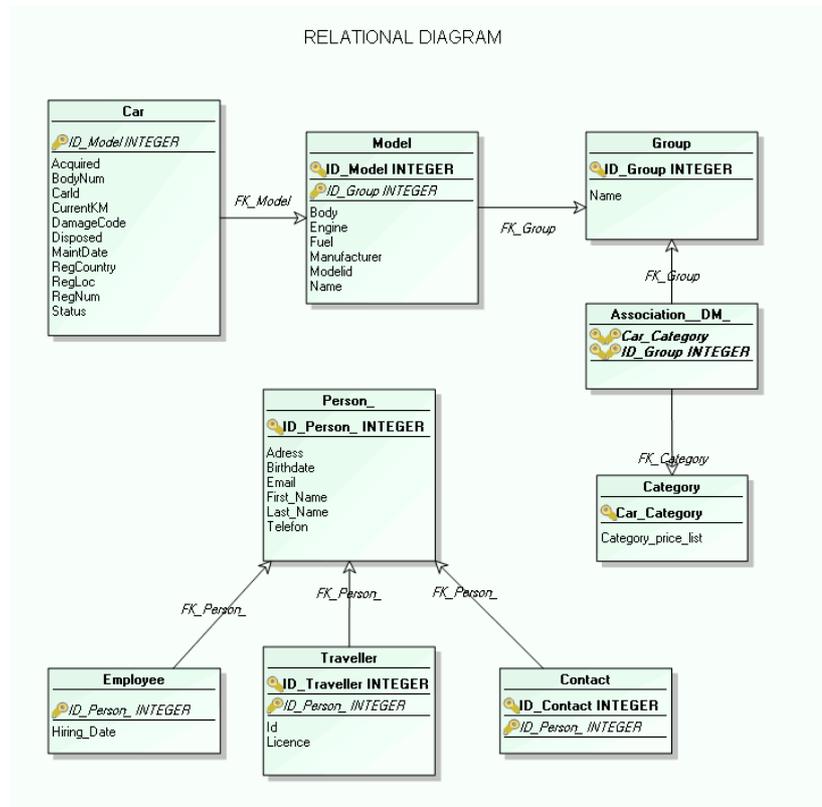
This strategy enables integration in reduced synchronization scope of objects on which the source object directly depends, for example all objects associated with the source entity which are necessary for update of the corresponding table.

Example

In the same example as before, taking the "Model" entity as source of reduced synchronization, the scope extends to the "Group" entity since tables corresponding to these two entities are linked by a foreign key: the "Group" entity can modify the "Model" table associated with the "Model" entity via the intermediary foreign key "FK_Group" (see diagram below).

The "Car" entity however is not taken into account in the scope since it cannot act on the "Model" table.

Physical model



Reduced synchronization results

Indicate objects to dismiss:

	Name	ExpressionType		Scope	Name
  Car Rental 	Car Rental		  Car Rental 		
  Car Model	Car Model		  Group  	<input checked="" type="checkbox"/>	
  Group	Car Model::Group		  Model  	<input checked="" type="checkbox"/>	
  Model	Car Model::Model				
  Car Rental	Car Rental				

All impacts

This strategy allows a combination of the two strategies described above. Scope of reduced synchronization is extended to objects required by the source object, and to all objects likely to be affected.

Example

Reduced synchronization results

Indicate objects to dismiss:

	Name	ExpressionType		Scope	Name
  Car Rental 	Car Rental		  Car Rental 		
  Car Model	Car Model		  Car  	<input checked="" type="checkbox"/>	
  Car	Car Model::Car		  Group  	<input checked="" type="checkbox"/>	
  Group	Car Model::Group		  Model  	<input checked="" type="checkbox"/>	
  Model	Car Model::Model				
  Car Rental	Car Rental				

Running Reduced Synchronization

Before starting, check that option "Reduced synchronization (logical > physical)" is active:

1. On the desktop, click **Main Menu > Settings > Options**.
The options window appears.
2. In the left pane of the window, expand the "Data Modeling" folder.
3. Click "Database Synchronization".
4. In the right pane of the window, select "Activate reduced synchronization (logical > physical)".

To start reduced synchronization:

1. Right-click the object to be synchronized.
2. Select **Synchronize (Logical to physical)**.
The synchronization wizard opens.

An entity can be used by several data models, therefore by several databases. When this is the case, you must select the database concerned.

3. Select the **Strategy**.
4. Click **Next**.
5. Select the objects to be synchronized
The scope selected by default depends on the context in which you select the object to be synchronized: if reduced synchronization is initialized from an entity in a diagram, the diagram model in question is selected. If the entity is selected outside its context, all models in which it appears are displayed in the editor.

☛ *Selected objects are not memorized and at a new synchronization default scope is again displayed.*

6. Click **Next**.
7. Define the **Synchronization Options**. All standard synchronization options are available with the exception of the "Reinitialize target objects" option.
8. Click **Next**.
The target objects protection option is displayed, you can view frozen objects. Protection of objects cannot be modified.
9. Validate results by clicking **Next**.
The synchronization report appears.
10. Click **OK** to close the synchronization wizard.
The mapping editor appears (unless the mapping option was cleared from the synchronization wizard).

Reduced synchronization options

Reduced synchronization presents the same options as total synchronization, with the exception of:

- Reinitialization of target objects
- Order

The reduced scope of reduced synchronization does not give a valid result for these two options

RUNNING SYNCHRONIZATION AFTER MODIFICATIONS

When a database has been synchronized and then manually modified, any additional specifications made directly in the database are retained, unless:

- Reinitialization is requested.
- Changes made in the data diagrams prevent this (addition or deletion of objects or links).

These changes include:

- Creation of entities, associations, attributes in the data diagram
- Deletion of entities, associations, attributes in the data diagram
- Modifying the characteristics of an attribute
- Modifying the name of an attribute, entity, or association
- Modifying the maximum multiplicity of an association
- Modifying the links of an association

Additional specifications made to the relational diagram may include:

- Deleting objects created by the synchronization
- Creating objects
- Modifying object characteristics created by the synchronization
- Modifying the order of columns in the tables, keys, or indexes

Synchronization after Modification of the Data Diagram

Newly created entities, associations, and attributes in the data diagram

The corresponding elements are created in the relational diagram, according to the rules used in the first synchronization.

Entities, associations, or attributes deleted from the data diagram

The corresponding elements are deleted in the database. For example, when an attribute is deleted in the data diagram, the corresponding column(s) are also deleted.

Modified attribute characteristics

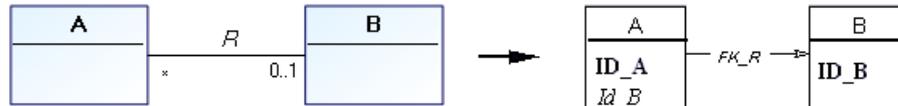
Modifications made to the characteristics of an attribute (type, length, decimal places, etc.) are reflected in the corresponding column in the relational diagram.

If the value of a characteristic of a column has been changed directly in the relational diagram, it will be preserved.

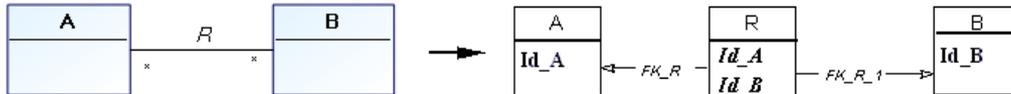
Modified name of an attribute, entity, or association

Modifications made to the name of an attribute, entity, or association are not reflected in the corresponding object in the relational diagram.

Modified maximum multiplicity of an association



If the maximum multiplicity of an association was 1, resulting in the creation of a migratory column, and has been changed to N, the migratory column is deleted and the table mapped by the maximum multiplicity of N is created.



Modified association links



Association R no longer concerns entity B, but does concern entity C.

In this case, the migratory column for B in A is no longer mapped.

- It is deleted.
- A migratory column from C is created.

Synchronization after Modifications to the Physical Diagram

Deleted table or column

If you delete objects from the database that were created by the synchronization (table, column, key, index...), these deletions are memorized and retained.

As long as the entity, association, or attribute that maps the table or column exists, the table or column is no longer recreated.

To recreate a column created by the synchronization and subsequently deleted:

1. Run database synchronization
2. At the results validation step, confirm the creation action (select the corresponding check box) proposed for this column.

Created objects

Objects (table, column, key, index) created in the relational diagram are retained.

However, deleting the objects they depend on may result in their deletion.

For example, a column is created in a table mapped by an entity. If the entity is then configured as "No table" or if the entity is disconnected from the datamodel, the corresponding table will disappear and the column with it.

☛ *Objects created in the relational diagram can be mapped manually. Objects created at synchronization are mapped automatically.*

Modified characteristics of objects created by synchronization

Modifications to the characteristics (SQL name, length, not null, datatypes) of objects created by synchronization are retained.

Modified order

Concerning modifications of order, processing depends on options defined in the synchronization wizard (see "[Step 2: Synchronization options](#)", page 100).

FROM THE PHYSICAL MODEL TO THE LOGICAL MODEL

This section describes how to synchronize the physical model model of a database with the corresponding conceptual model.

"Physical to Logical" Synchronization Rules

Synchronizing the logical model from the physical model enables creation of the database data diagram from its tables.

Rules used for this transformation are:

- A table of which the primary key is composed of a foreign key relating to the same columns becomes an entity. A generalization is created between this entity and the entity corresponding to the table to which the foreign key is pointed.

Physical level example:

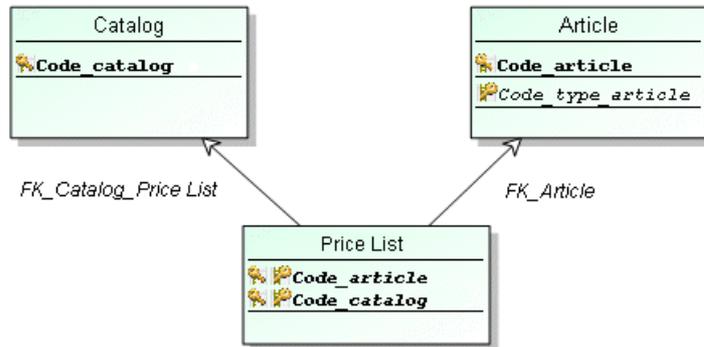


Result at the logical level:



- A table of which the primary key is composed of foreign keys only becomes an association of multiplicities (*..*). If columns do not belong to the primary key, an attribute connected to the association will be created for each of these columns.

Physical level example:



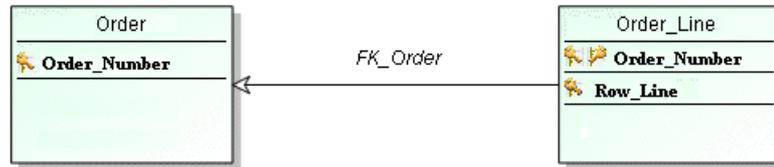
Result at the logical level:



- A table of which the primary key contains foreign keys and at least one column that is not a foreign key becomes an entity. An aggregated association is created between this entity and the entity corresponding to the table to which each of the foreign keys is pointed.

The candidate key of the entity is composed of the roles of aggregated associations and of the attribute(s) corresponding to the other columns of the primary key of the table.

Physical level example:



Logical level result:

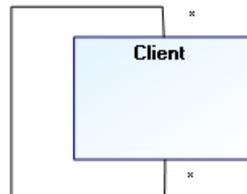


- A table of which the primary key is composed of foreign keys only pointing to the same table becomes a reflexive association of multiplicities (*..*).

Physical level example:



Logical level result:



- In other cases, each table becomes an entity and its columns the attributes of the entity.

- A foreign key becomes an association (0..1, *). If all the columns of the key are mandatory, its cardinalities become (1, *).

- Types of attributes are recalculated with the help of the conversion table specific to the target DBMS (see ["Attribute and Column Types"](#), page 145).

Running Synchronization

To start the synchronization:

1. Select the database concerned (in the list of databases or in a diagram for example).
2. Right-click the database and select **Synchronization (Physical to Logical)**.
The synchronization wizard opens.

Step 1: Selecting objects to be synchronized

To define synchronization scope:

Scroll the list of objects contained in the database.

1. By default, all objects are selected and therefore included in synchronization. To exclude an object from the synchronization, clear it from the **Scope** column. When an object is excluded, its mapping is also excluded.
2. By default, all the objects are "realized", in other words, they give way to the creation of an object during synchronization. To specify that an object is "not realized", select it in the **Not realized** column. For more detailed information, see ["Realized mode"](#), page 85.
3. When the list of objects has been defined, click the **Next** in the wizard.

Step 2: Synchronization options

From the synchronization options, you can:

- Reinitialize target objects: synchronization starts from zero and deletes existing target objects.
- Recalculate target object names: names of data diagram objects are recalculated as a function of those of the relational diagram. This means that any manual modification of the data diagram is canceled.
- Take account of optimizations: all optimizations - including those not selected in the validation step (see ["Validating optimizations"](#), page 83) are proposed.
- Take account of deletions: entities, associations and diagrams that have been deleted are included in the scope. Consequently, deletion of corresponding target objects and links is proposed.
See ["Using Options"](#), page 83.

You must also indicate the data model that will own the set of objects created by the synchronization.

Other options concern target object properties update. By default, synchronization updates all properties of each object concerned.

Scheduling

You can run synchronization:

- Immediately
- As soon as possible (after publication of updates)
- At a predefined date and time

› When options have been specified, click **Next**.

Step 3: Protecting objects

Synchronization can impact all target objects.

- › To keep an object intact, select it in the **Frozen**.
- › Click **Next** to continue.

See ["Protecting Objects", page 85](#).

Step 4: Validating results

The wizard displays the results that will produce synchronization validation.

- › To validate these results, click **Next**.

A report presents a list of processes that have been carried out.

Reduced synchronization

The above synchronization applies to a database but you can also run a Physical > Logical synchronization on a specific object of the database to reduce synchronization scope and processing time. See ["Reduced Synchronization \(Logical to physical mode\)", page 88](#).

"Physical to Logical" Synchronization Results

Owner data model

A default data model owning the entities is created at the time of the Physical to Logical synchronization. Synchronized classes and associations are automatically connected to it. You can then distribute these entities and associations between the various data models of your study.

Data diagrams

A data diagram is created for each of the relational diagrams of the database. Classes and associations resulting from the tables of the corresponding relational diagram are connected to it.

Mappings

See "At synchronization, the "Article Type" entity does not produce creation of a table. In the "Article" table, the foreign key to "ArticleType" is not created; however the "Code_Type_Article" column is created. Synchronization Results: Correspondences", page 86.

CONFIGURING SYNCHRONIZATION

This section describes the default options and parameters taken into account at synchronization.

Preparing Synchronization

To prepare synchronization:

1. Right-click the database and select **Properties**.
The properties window appears.
2. Click the drop-down list then **Options > Standard**.
3. If you want the physical database name used at SQL script generation to be different from the database name, specify the target database name in the **SQL Name** text box.
4. If required, indicate a **Prefix** for the SQL Tables. This prefix will be added to the beginning of the name for each table generated.
 - ☛ *Additional parameters for configuration of synchronization and generation can be indicated in **Options**. These parameters vary as a function of the DBMS selected.*
5. Click again on the scroll-down list of the properties window and click **Characteristics**.
6. Select the **Target DBMS** and its version.
 - ☛ *The type of the target DBMS determines:*
 - *For synchronization, the generation of column datatypes based on the type and length of attributes (see "[Determining Column Datatypes from Attribute Types](#)", page 149).*
 - *In generation, the syntax of the generated SQL commands.*
7. Click **OK** to close the dialog box, saving the modifications.

Creation Options

On a database

It is possible to configure synchronization for each database in order to modify:

- Its creation options
- Processing of repository integrity (keys OnDelete or OnUpdate as a function of the possibilities offered by the DBMS target)

This configuration also concerns processing of the Not Null columns and the automatic creation of indexes on primary keys.

To configure the creation options for the database:

1. Right-click the database and select **Properties**.
The properties window appears.

- Click the drop-down list then **Options > Synchronization**. The corresponding options appear.

You can specify the following parameters:

Parameter	Value
Noms de Table:	128,*DB*ROOT
Noms de colonnes:	128,*ROOT
Noms colonnes de FK:	128,*ROOT
Noms de PK:	128,PK_*ROOT
Noms de FK:	128,FK_*ROOT
Noms des Index de PK:	128,IDX_*ROOT
Noms des Index de FK:	128,IDX_*ROOT
Noms des index:	128,IDX_*ROOT
Noms cols PK auto:	128,ID_*TBL
Colonnes Not Null:	Not Null
OnUpdate:	
OnDelete:	
Not Null PkColumns:	Not Null

- **Columns Not Null** activates/deactivates the WITH DEFAULT option for Not Null columns.
- **OnDelete**: key deletion default strategy. Possible values are:
 - **Restrict**: deletion is refused.
 - **Cascade**: deletion of a column is reflected in dependent tables.
 - **Set Null**: indicates "Null".
 - **Set Default**: gives the default value if this is specified. If no default value is specified, nothing occurs ("No action").
 - **No Action**: nothing occurs.
- **OnUpdate**: key update default strategy.
- **Names cols PK auto**: columns derived from the implicit identifier. See "General rule", page 74.

Parameters whose names begin with **Names of** indicate rules applied in name generation (see "Configuring Name Generation", page 105).

On the DBMS

The default values for database synchronization and generation parameters are accessible in the properties dialog box of the DBMS used.

To display DBMS synchronization parameters:

1. In the **HOPEX** workspace, open the **Utilities** navigation window.
2. Right-click the target DBMS name and open its **Properties** dialog box.
3. Click the drop-down list then **Options** > **Synchronization**.

By default, the parameters specified at the DBMS levels are valid for all new databases. When you modify synchronization parameters on a database, this database no longer takes account of DBMS parameters.

☛ *For more information on synchronization configuration, see also "Running Synchronization", page 80.*

Configuring Name Generation

Naming rules

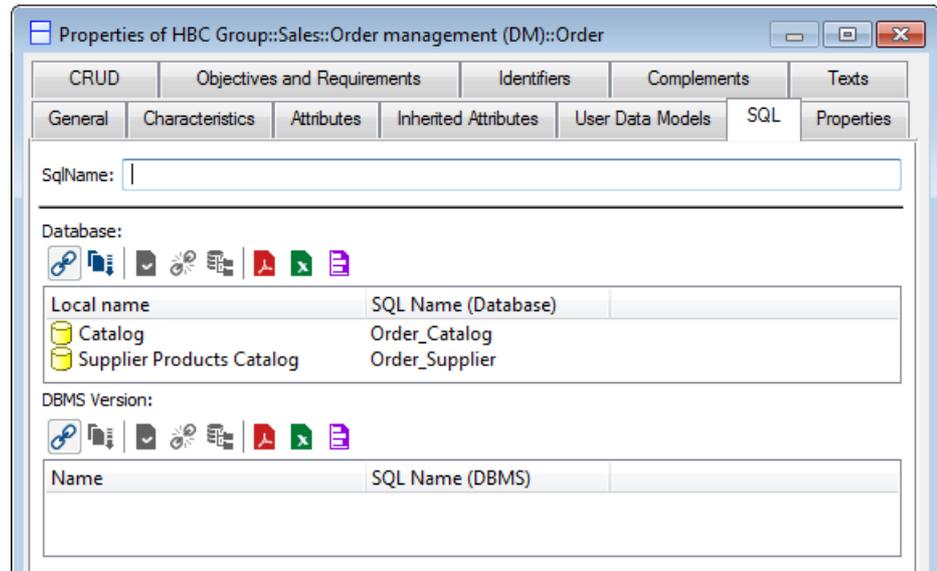
The names of physical objects created at "logical to physical" synchronization are deduced from the **Local Name** of the logical objects from which they are derived.

As logical object names are not subject to any particular restrictions, transformation rules apply by default at their synchronization. These rules are accessible locally in the **Options** > **Standard** page of synchronized database properties, or globally in the target DBMS properties:

- **Identifier size**: maximum size of SQL identifier for this target DBMS
- **First character**: character set authorized for first character of SQL identifier
- **Authorized characters**: character set authorized for SQL identifier characters
- **Replacement character**: replacement character for unauthorized characters
- **Converted characters**: SQL identifier character set to be converted
- **Conversion characters** character set corresponding to characters to be converted
- **Upper-case conversion**: conversion to upper-case of SQL identifiers

It is possible to indicate another name for each synchronized object using its **SQL Name**. The **SQL Name** replaces the **Local Name** at synchronization, while taking account of default transformation rules.

The **SQL Name** of logical objects is accessible in the **SQL** page of their properties.



You can give a different name depending on the database and DBMS.

When fields **SQL Name (Database)** and **SQL Name (DBMS)** indicate different names, the name defined at database level takes precedence at synchronization.

For more details on SQL options, see ["SQL Identifiers and Options", page 42](#).

By default, names of relational objects are generated according to the following masks:

Table	Database prefix + name* of entity or association
Column	name* of attribute
Primary key	"PK_" + name* of entity or association
Foreign key	"FK_" + name* of target entity or role if specified
Primary key index	"IDX_" + name* of entity or association
Foreign key index	"IDX_" + name* of target entity or role if specified

**Name calculated according to previously explained naming rules.*

These masks can be modified locally in each database, or globally for a given target DBMS.

Modifying a naming rule

To modify the mask of a naming rule:

1. Right-click the database and select **Properties**.
2. Click the drop-down list then **Options > Synchronization**.
3. In the field of the rule in question, click the arrow.



4. Click **Modify**.
The **Enter SQL Mask** dialog box opens.

Entering the SQL mask

SQL masks define relational object naming rules at synchronization.

Example: In the DB_EMPLOYEES database, which has the prefix EMP, the mask ^DB_^ROOT generates the following for the table derived from the Customer entity: EMP_CUSTOMER

In the SQL mask entry dialog box, you can directly enter the **Mask** using syntax indicated below, but you can also use entry help proposed in the Component frame.

- In the list in the **Component** box, select the elements that are to prefix the names. These elements are:

^ROOT	<ul style="list-style-type: none"> • For a table: name* of the entity or association from which it is derived. • For a column: name* of the attribute. • For a primary key: "PK_" + name* of entity or association • For a foreign key: "FK_" + name* of target entity or role if specified • For an FK column: name* of the attribute. • For an auto PK column: name* of the entity identifier. • For an index on primary key: "PK_" + name* of entity or association • For an index on foreign key: "FK_" + name* of target entity or role if specified • For an index: name* of the attribute.
^DB	Database prefix
^TBL	Table name

^TBR	Reference table name
^KEY	Key name
^CPT	Timestamp

**Name calculated according to previously explained naming rules.*

Size indicates total length limit of the generated name. It also applies to each of the elements used, which will be shortened to the number of characters indicated between brackets alongside the element concerned.

Definition of a Timestamp ("^CPT") enables automatic generation of an order number and indication of its length, (for example, ^CPT[^1^] will generate "1", "2", "3"; ^CPT[^3^] will generate "001", "002", "003").

The **Always** option indicates that timestamping begins from the first occurrence (CLI00, CLI01, etc., instead of CLI, CLI01).

You can also specify characters used as prefix and suffix of this timestamp.

Using the **Name Unicity** option ensures that the name of an object is not repeated in the database, repository or table in which this object appears. As such, if you apply the Name unicity option to a table, different objects of this table cannot have the same name.

Configuring PK column names (implicit identifier)

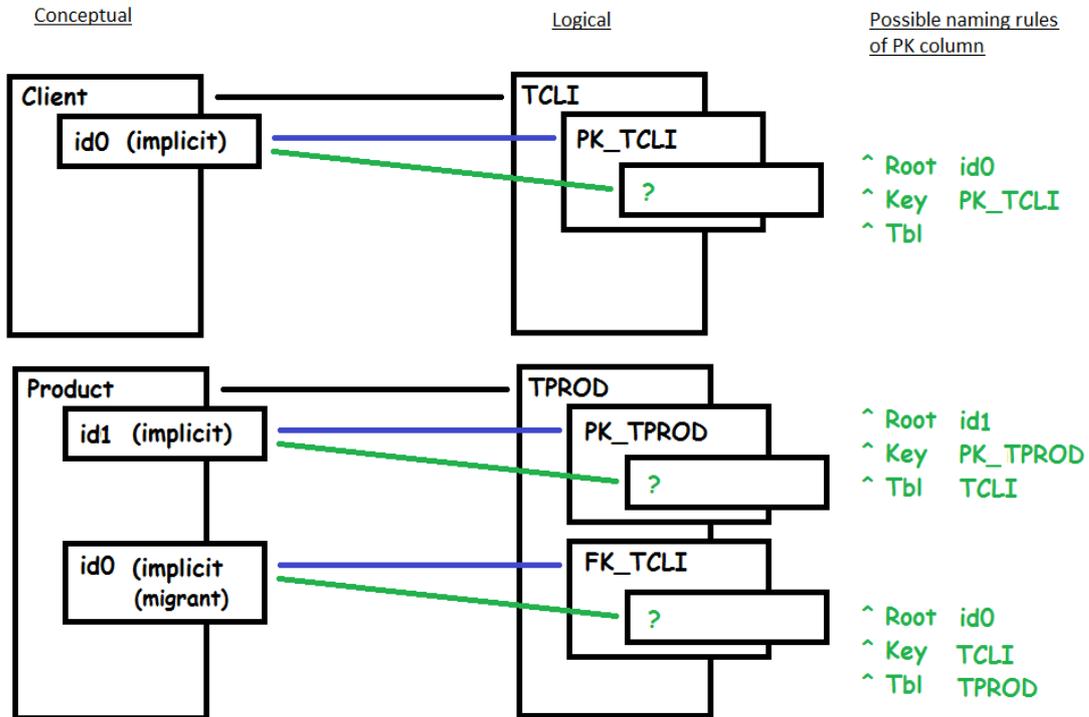
At synchronization in Logical > Physical mode, the entity identifier becomes the primary key of the table. If the identifier is implicit, a column is automatically created. For more details, see [""Logical to Physical" Synchronization Rules", page 74.](#)

By default, the name of a column derived from an implicit identifier is built using the ^TBL keyword which corresponds:

- to the name of the migrating table (in other words derived from a foreign key) if the identifier is migrating
- to the name of the table if the identifier is not migrating

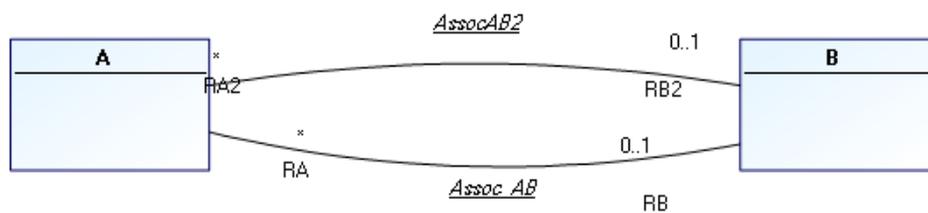
You can modify construction rules and build the name of these columns with the ^KEY keyword corresponding to the name of the foreign key (without " FK_ ") if the

identifier is migrating, as well as with the ^ROOT keyword corresponding to the name of the identifier (ID). See "Modifying a naming rule", page 107.

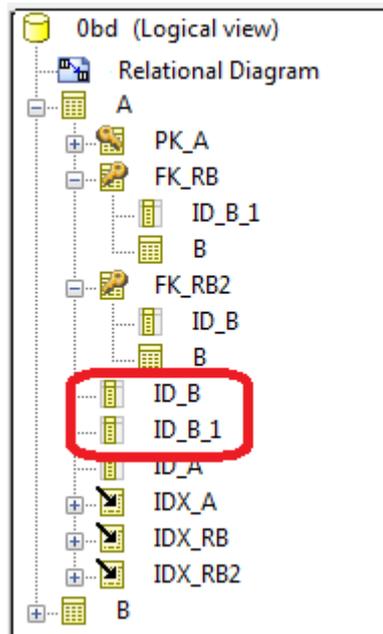


Example

When there are two constraint associations between two entities as below:



By default after synchronization, you obtain two columns with identical names, differentiated only by prefix "1".



You can modify the naming rule and build the name of these columns with the ^KEY keyword corresponding to the name of the foreign key (without "FK_").

The name of the foreign key being calculated on the name of the Role when it is specified, the names obtained for these two columns will be different.

In our example, if you replace "ID^TBL" par "ID^KEY" after synchronization you obtain:

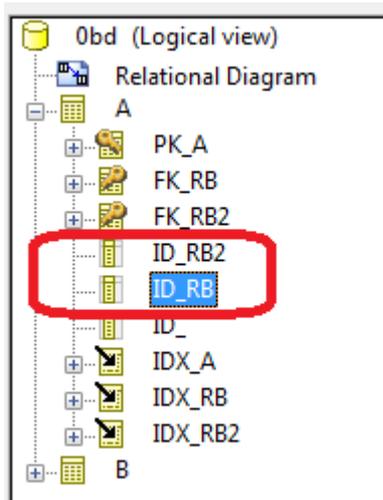


DIAGRAM SYNCHRONIZATION

Synchronization enables translation of a logical data model to a physical model, and vice versa. Before running synchronization, source diagrams (data diagrams and physical diagrams) must be saved and closed.

A first synchronization automatically creates the target model diagram. A new synchronization on previously synchronized models does not include automatic update of diagrams, in other words of graphical representation of models. Depending on the changes you have made, the synchronization wizard may propose update of the target diagram.

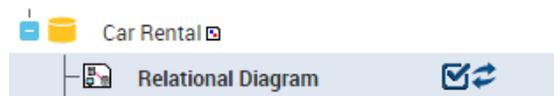
Case of Diagram Update at Synchronization

The synchronization wizard proposes diagram update in the following cases.

After source diagram modification

When you run synchronization after source diagram modification, by default the wizard activates target diagram update. Updating is indicated by display of two small blue arrows.

Below, at synchronization in logical to physical mode, modification of the logical diagram automatically produces a physical diagram update.



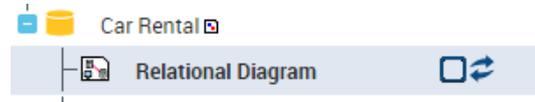
After target diagram modification

If you have modified the target diagram, for example the physical diagram, by default the synchronization does not propose update of this target diagram.

To display the case of target diagram update, you must select the synchronization option "Take account of optimizations". So that the update will be activated, you must select the check box in question.

After modification of both diagrams

If both diagrams - logical and physical - have been modified, target diagram update is proposed but not selected by default.



No modification detected

If neither of the two diagrams has been modified, the wizard does not propose update. It should be noted that any diagram modification must be saved before closing the diagram so that it will be taken into account by the synchronization wizard.

Particular case: an entity mapping with two tables.

When an entity of the logical model is associated with two tables in the physical model (following merging of entities for example), the updated physical model displays only one of the tables.

MODEL MAPPING



In many modeling projects the problem arises of communication between teams of analysts and architects and the database development teams.

HOPEX Database Builder offers two modeling levels:

- ✓ The logical level which describes data modeling in terms of entities and relationships and is intended for analysts and developers.
- ✓ The physical (or relational) level, which describes the database in terms of tables and interfaces with the DBMS. This level is intended for the designer and the database administrator.

By enabling change from one data model to another, the database editor favors consistency between the architecture of data and its support systems.

The following points are covered:

- ✓ ["The Database Editor", page 116](#)
- ✓ ["Mapping Details", page 119](#)

THE DATABASE EDITOR

The database editor allows you to synchronize the different views of a database manually; the logical model and the physical (or relational) model.

The synchronization wizard automatically maps the the two views. See ["Synchronizing logical and physical models", page 73](#).

After synchronization, you can create or modify mappings in the editor manually, but this method no longer guarantees consistency of the two models. The denormalization wizard maintains this consistency. See ["Denormalizing logical and physical models", page 125](#).

Run the editor on a database

To open the editor on a database:

- 1. Click the database icon and select **Mapping Editor**.
The mapping editor juxtaposes the logical view and the physical view of the database. When a mapping tree exists, it is automatically displayed. When a tree has not been created for the database, a window prompts you to create it.

Creating a Logical/Physical Mapping Tree

To create a mapping tree:

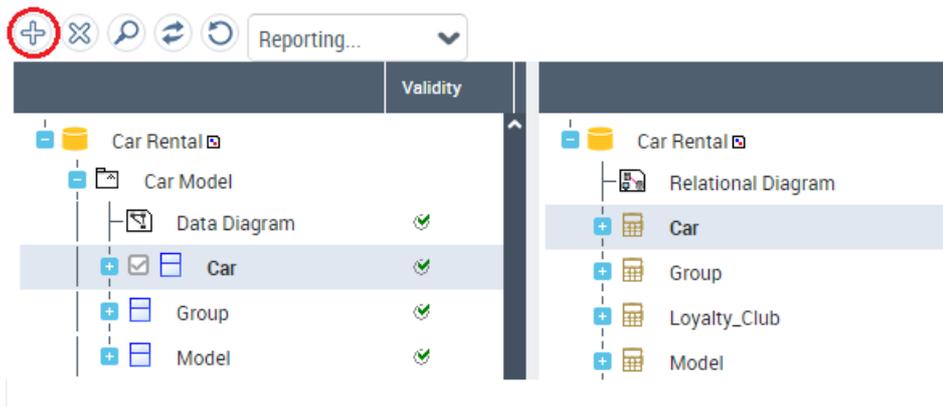
1. In the creation dialog box that opens, indicate the name of the new mapping tree.
2. In the **Nature** list box, select the nature of the tree: "Logical/Physical".
3. In the **Left Object** and **Right Object** frames, select the logical and physical models that you wish to align.
4. Click **OK**.
The editor displays the mapping tree juxtaposing the two models.

Creating a Mapping

To create a mapping between an entity and a table:

1. In the database editor, select the entity then the table.

2. Click **Create mapping item**.

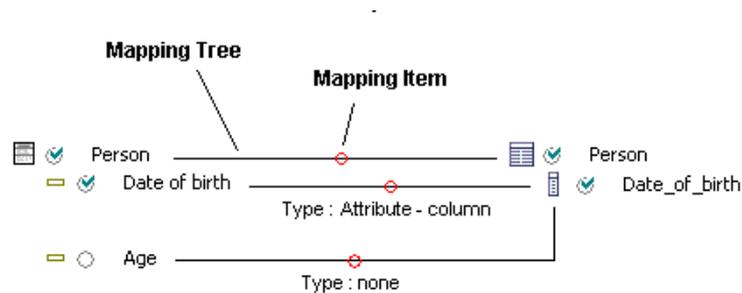


The mapping is created from the last object selected. Therefore, in order to create a mapping from the logical model to physical, in other words to define an object of the physical model from an object of the logical model, you must select the logical model object then the physical model object and create the mapping from the latter. If a mapping cannot be created, an error message appears (see "[Synchronization direction](#)", page 87).

New mapping example

Consider the "Person" entity that contains the "Birth Date" attribute. In physical formalism it has as mapping the "Person" table which contains the "Birth_Date" column.

Suppose we add the "Age" attribute on the entity. This can be calculated from the birth date. So as not to create a column corresponding to this new attribute at synchronization, you can directly connect it to the "Birth_Date" column.



To create a mapping between the "Age" attribute and the "Birth_Date" column:

1. In the editor, on one side select the "Birth_Date" column, and on the other the "Age" attribute.
2. Click **Create mapping item**.

When the mapping has been created, a tick appears in front of the "Age" attribute.

Deleting a mapping

To delete a mapping on an object:

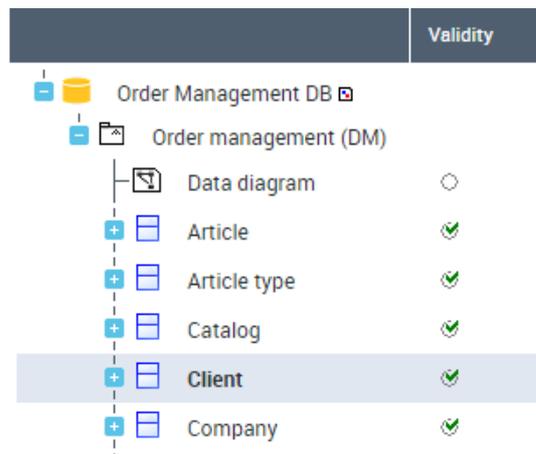
- 1 Select the object in question and click the **Delete mapping item** button.

MAPPING DETAILS

Objects with mappings are ticked green. When you select one of these objects, its mapping appears in the mapping properties dialog box located by default at the bottom of the database editor. It groups the names of objects connected in the two formalisms, the object types and comments where applicable.

Mapping example

The "Customer" table is selected in the logical view tree:



The mapping displays the following objects:

Validity	Logical object	Physical object	Type
Valid	Client	Client	Entity - Table

This means that the "Customer" table is derived from the entity of the same name.

Mapping Properties

To view mapping properties:

1. In the mapping editor, select the mapping item and click **Properties**.
2. In the window that opens, click the drop down list and select **Characteristics**.

See also "Mapping characteristics", page 86.

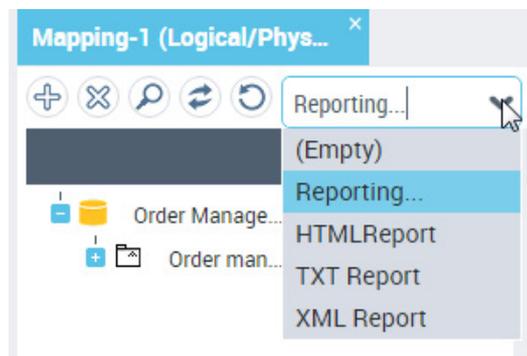
Mapping Report

In a document you can generate detail of mappings between the two database models. This can be an HTML, text or XML file.

Generating an HTML report

To generate the HTML report of a mapping tree:

- 1 In the database editor toolbar, click the drop-down list and select **HTML Report**.



The corresponding file opens. It presents detail of mappings of the physical view and the logical view in the form of a table.

Each row of the table presents an object of the model and its equivalent in the other model. In the middle of each row appears the status of the mapping between the two objects.

Description of Mapping View "Logical view"	
Object	Mapping
Order Management DB	
Relational Diagram	Data diagram
Article	Article
FK_Article_type	ID Article type3
	Type
Article_type_code	Article type code
Article_type_code_1	Article type code

At the end of the document, you will also find a list of invalid mappings.

Object status

Indicators enable indication of status of synchronized objects. A filter bar allows you to show all or only certain of these indicators. This bar is available by selecting **View > Toolbar** in the editor.

Object status can be characterized as:

-  Valid
-  Invalid (when an object has kept a mapping to an object that no longer exists)
-  No mapping
-  Frozen (Protected)
-  Standard

Saving display of editor indicators

An option allows you to save the status of indicators in the mapping editor. It is specific to the user and the current mapping tree. A user who has selected the indicators display option will automatically find the status of objects in the previously created mapping tree.

The indicators display option of the editor is cleared by default. To activate it:

1. On the desktop, click **Main Menu > Settings > Options**.
2. In the left pane of the options window, select **Mapping Editor**.
3. In the right pane, select option **Save display of editor indicators**.
4. Click **OK**.

Mapping Source

When you select an object in the editor in one of the formalisms, you can display its mapping in the other formalism.

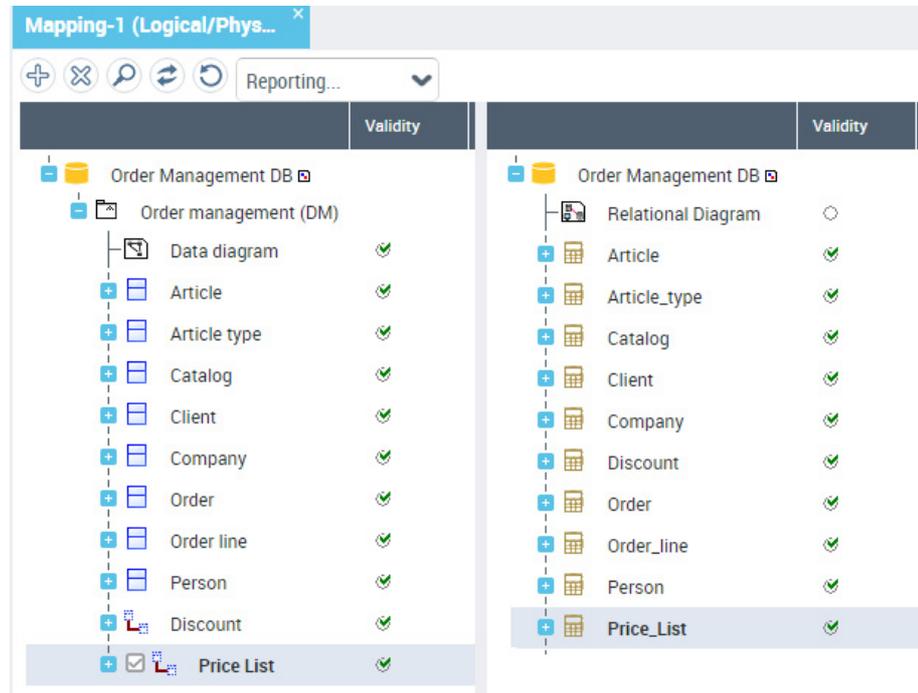
To display an object mapping:

1. In the mapping editor, select the object concerned and click **Find**.

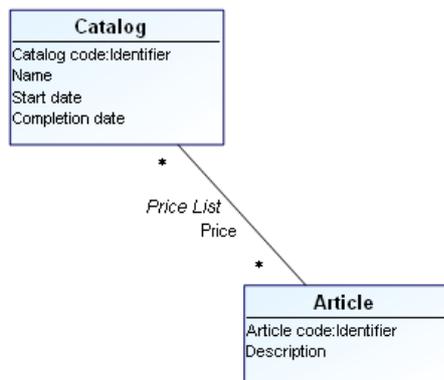
The editor displays the source object.

Mapping example

Consider the "Price List" table created in the logical model. In the pop-up menu of the table, select **Locate**. You will see that it corresponds to an association.



Opening the logical model diagram, you can see that this is the "Price List" association that connects the "Catalog" and "Article" entities.



At synchronization, this non-constraint association produces a table in which:

- A column is created for each connected entity.
- The primary key for the table uses all these columns.
- A foreign key is also built for each connected entity.

For more information on synchronization of associations, see ["Associations"](#), page 75.

Mapping Drawing

To view the mapping drawing:

- ▶ In the mapping tree, select the objects concerned.

A window appears at the bottom of the editor showing a drawing of objects and their connecting links.



DENORMALIZING LOGICAL AND PHYSICAL MODELS



After having synchronized your models, you can develop them by modification in their diagrams. This method does not however guarantee consistency between the two models.

Denormalization wizards have been created to ensure such consistency. They allow you to modify definition of one model while maintaining consistency with the other.

The wizards also allow you to quickly carry out duplication or merging operations without mapping transfer and/or without source object deletion when you wish to transfer modifications in the relational model.

The following points are covered here:

- ✓ ["Denormalization Principles", page 126](#)
- ✓ ["Logical Denormalization", page 130](#)
- ✓ ["Physical Denormalization", page 138](#)

DENORMALIZATION PRINCIPLES

Denormalization enables transformation or detailing of models as a function of specific requirements: choice of modeling, performance optimization, physical implementation level, redundancies, etc.

The denormalization tool is presented in the form of wizards enabling execution of these transformations.

A wizard applies to an object or group of objects. Depending on optimization type requested, denormalization creates new objects in a model starting from initial objects (source objects).

Denormalization: consistency of models

Denormalization changes the object of a model. When this model has been mapped with another (see "[Synchronizing logical and physical models](#)", page 73), you can manage impact of this change on the other model.

At denormalization, you can therefore:

- Transfer or not transfer mappings with synchronized objects.
- Delete or keep source objects.

Transferring mappings

Transfer of mappings guarantees stability and consistency between two models. When denormalization creates a new object at the logical level, mapping with the physical object is transferred to the new logical object. When you clear this option, mappings with new objects are not created and the two models must therefore be resynchronized.

 **So that synchronization can validate changes resulting from denormalization, make sure the "Reinitialize target objects" option is cleared.**

Deleting source objects

Source objects are deleted by default. Non-deletion of source objects allows you to keep initial objects after denormalization.

Synchronization and Denormalization

In data modeling, synchronization and denormalization are often combined to respond to particular use cases.

Example

A PAYMENT entity that you wish to represent in the database by three tables TRANSFER, CHECK and OTHER. To produce this modeling:

1. Create the PAYMENT entity.
2. Run synchronization (logical to physical) to obtain the PAYMENT table.
3. Run the physical wizard to horizontally partition the PAYMENT table.
4. Rename the three duplicates TRANSFER, CHECK and OTHER.

The three tables obtained in this way are now connected to the PAYMENT entity and will follow the developments of future synchronizations.

Combining denormalization and synchronization options

Impact of a denormalization on two synchronized models varies depending on the options selected.

Consider the example of a logical denormalization. Possible combinations are:

- Deletion of source objects + transfer of mappings: logical model source objects are deleted. The physical level is unchanged; mapping with logical objects resulting from denormalization is assured.
- Deletion of source objects + non-transfer of mappings: physical objects corresponding to logical objects resulting from denormalization are created. The physical objects corresponding to deleted logical objects are deleted.
- Non-deletion of source objects + transfer of mappings: logical model source objects are kept. The physical level is unchanged; mapping with objects resulting from denormalization is assured.
- Non-deletion of source objects + non-transfer of mappings: physical objects corresponding to objects resulting from denormalization are created. Existing physical objects are kept.



Particular cases: ascending and descending merges:

In the case of an ascending merge, the supertype entity plays a particular role. Denormalization keeps its mapping in all cases. Similarly, in the case of a descending merge, subtype entities play a particular role; their mapping is kept in all cases.

Denormalization: Use Case

Combination of denormalization options varies depending on design mode of your models.

1. Maintaining stability at the physical level when a modification is applied at the logical level.

Context: a synchronization has already been established between the logical and physical levels. The physical level is in production. A modification must be applied at the logical level, without impact on the physical level.

Recommended denormalization options: transfer of mappings, deletion of source objects.

This use case corresponds to a preventive maintenance-oriented work mode: modifications are carried out by anticipation on the logical level, knowing that the physical level must not be modified until further notice.

Result: after denormalization, mappings are re-established between target logical objects and physical objects. After synchronization, nothing changes at the physical level.

2. Developing the physical level when denormalization is applied at the logical level.

Context: a synchronization has already been established between the logical and physical levels. The physical level is not frozen and must develop as a function of the logical level.

Recommended denormalization options: non-transfer of mappings, deletion of source objects.

This use case favors developments at conceptual level, ignoring impact at the physical level.

Result: after denormalization, target logical objects are without mappings and physical objects corresponding to source logical objects are no longer synchronized. After synchronization, the physical level is updated: the physical objects corresponding to source logical objects (objects existing before denormalization are deleted; new physical objects corresponding to target logical objects (objects created by denormalization) are created.

3. Simplifying logical level development during the development phase.

Context: a synchronization has already been established between the logical and physical levels, or the new physical level has not yet been implemented.

Recommended denormalization options: non-transfer of mappings, non-deletion of source objects.

This use case corresponds to an "incremental" work mode: logical level source objects are unchanged. The model is supplemented by target objects resulting from denormalization. These target objects produce a new section at the physical level and the existing physical section remains stable.

Result: after denormalization, mappings are unchanged. After synchronization, new physical objects corresponding to new logical objects are created; physical objects corresponding to source logical objects are unchanged.

4. Favoring installation of multiple scenarios in development phase.

Context: a synchronization has already been established between the logical and physical levels, several modeling options temporarily coexist for a single physical level.

Recommended denormalization options: transfer of mappings, non-deletion of source objects.

This use case, to be used with care, enables keeping two modeling options at conceptual level that produce a common result at the physical level.

Result: after denormalization, physical objects remain connected to source logical objects and are also connected to target logical objects. After synchronization, objects at the physical level are unchanged.

LOGICAL DENORMALIZATION

Logical denormalization applies to data model entities (or classes) and attributes.

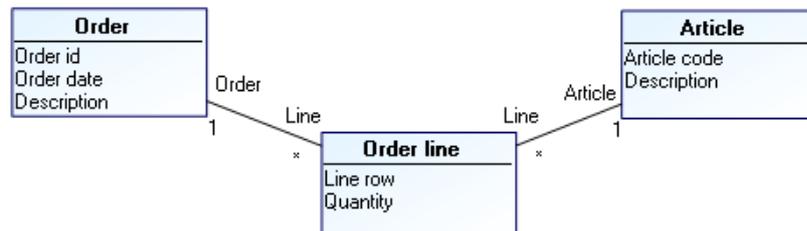
Running Logical Denormalization

To denormalize logical formalism:

1. Right-click the database with which the data model is associated and select **Logical denormalization..**
A wizard opens.
2. In the **Select denormalization** field, select the type of denormalization concerned and follow the instructions of the wizard. See "[Logical Denormalization Wizards](#)", page 132.

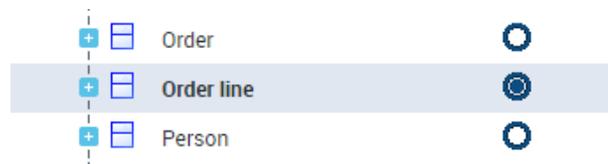
Logical denormalization example

Suppose that you wish to transform the "Order line" entity to an association.



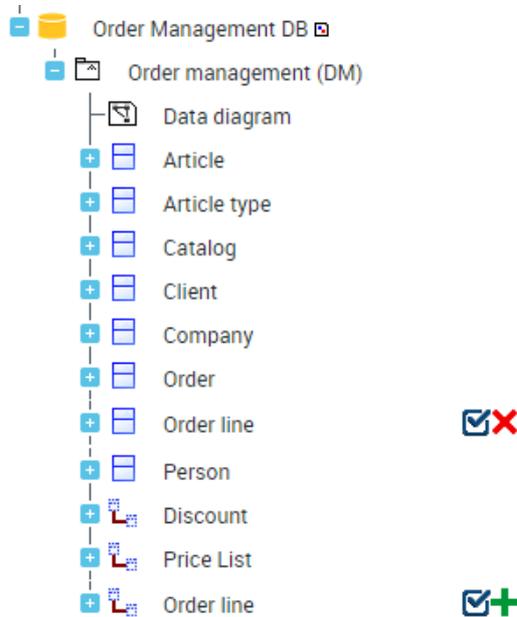
To transform this entity to an association:

1. Right-click the "Order management" database which contains this entity and select **Logical denormalization** .
A wizard opens.
2. In the **Select the denormalization type field**, select "Transform an entity to an association".
3. Click **Next**.
4. In the editor tree, select the **Scope** column opposite the "Order line" entity check box.



☛ You can select several entities at denormalization.

5. Click **Next**.
 Denormalization options appear. Mapping transfer and source object deletion are activated by default. This means that the "Order line" entity will be deleted and the mapping link with the "Order line" table will be transferred to the association which replaces it.
6. Click **Next**.
 The editor displays changes produced by this denormalization. You can see that the "Order line" entity will be deleted and the "Order line" association will be created.



☛ When a selected entity cannot be transformed, the editor will indicate the reason.

You can refuse a modification by clearing the corresponding box.

7. Validate results by clicking **Next**.
 This transformation is definitive and will be taken into account by the next synchronization.
 On completion of denormalization, you can see that the "Order line" association that replaces the entity is now mapped with the "Order line" table.

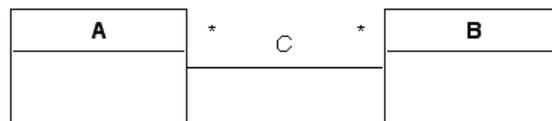
☛ If an object is protected, it is not possible to select it during denormalization.

Logical Denormalization Wizards

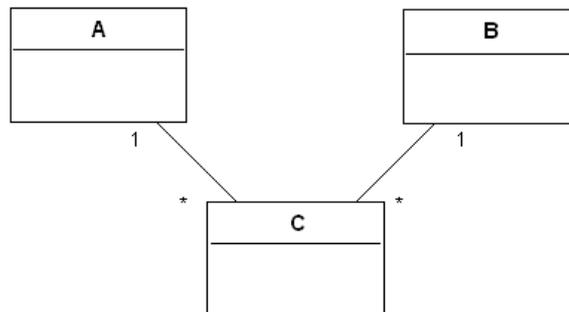
Transform association to entity

This denormalization enables transformation of an n-ary association, whatever its multiplicities, to an entity. An association of multiplicity '*,1' is created between this entity and the entities of the association.

Before



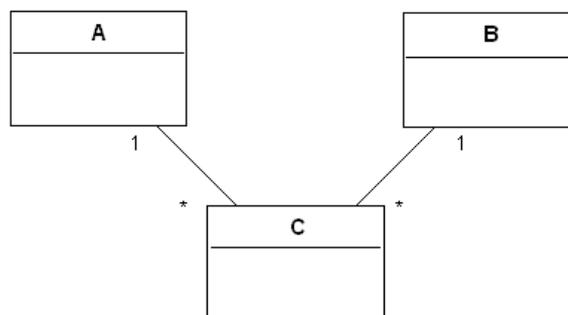
After



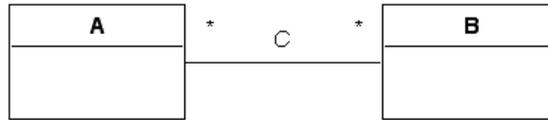
Transform entity to association

This denormalization enables transformation of an entity with n binary associations, of which opposite roles are of multiplicity '1', to an association. A new n-ary association of multiplicity '*' is created between these entities.

Before



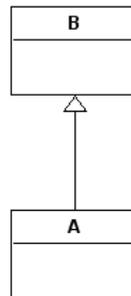
After



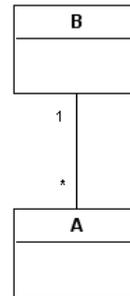
Transform generalization to association

This denormalization enables transformation of a generalization between two entities to an association. An association of multiplicity *,1 is created between the two entities and the generalization is deleted.

Before



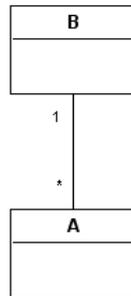
After



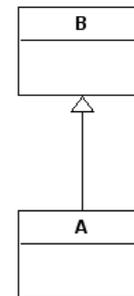
Transform association to generalization

This denormalization enables transformation of an association 1,* to a generalization. A generalization is created between the two entities and the association is deleted.

Before



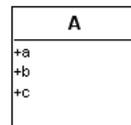
After



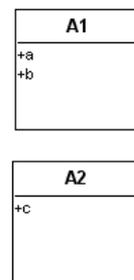
Vertical partition of entities

This denormalization enables division of an entity into several entities. The attributes, associations and generalizations are shared between the entities.

Before



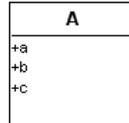
After



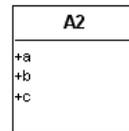
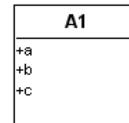
Horizontal partition of an entity

This denormalization enables duplication of an entity.

Before



After



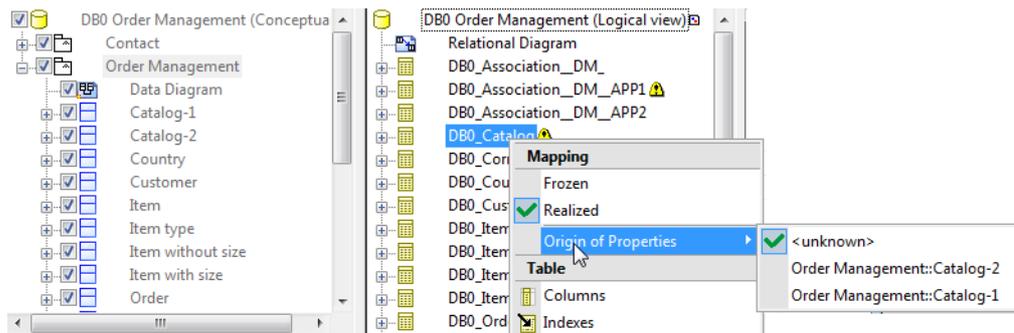
Horizontal partition and synchronization in Logical > Physical mode

Consider a "Catalog" entity. After horizontal partition, this entity gives two entities "Catalog-1" and "Catalog-2".

After synchronization of the Logical > Physical mode, the two entities have a table as mapping.

If you display properties of mappings, you will note that both are bidirectional, meaning that entities and table are updated in both directions of synchronization.

If you carry out logical modifications on these entities and then re-run synchronization, the editor displays a signal on the target table; it does not know which entity to take to carry out updates.

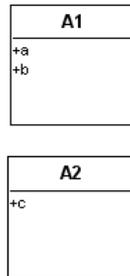


When you select an entity, this is kept as reference entity (for example Catalog-1). The other entity will be kept in one direction only, in other words Catalog-2 could be updated in the logical model but will have no impact on the table.

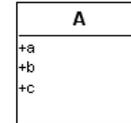
Merging of entities

This denormalization enables merging of entities.

Before



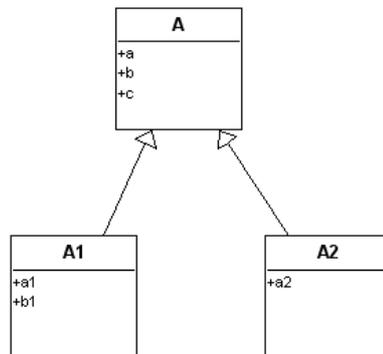
After



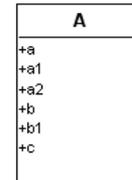
Merging of ascending entities

This denormalization enables merging of an entity with its parent entity: all attributes and links are transferred to the parent entity and the child entity is deleted.

Before



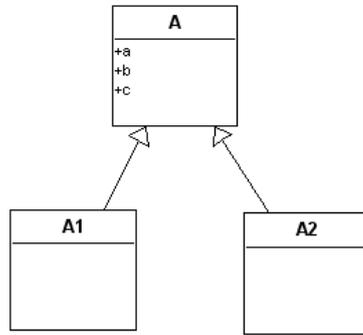
After



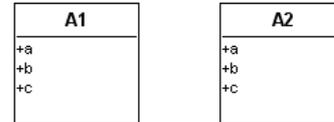
Merging of descending entities

This denormalization enables merging of an entity with its child entity: all attributes and links are transferred to the child entity and the parent entity is deleted.

Before



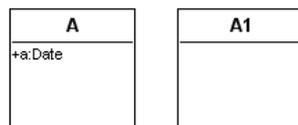
After



Copy/paste of attributes

This denormalization enables transfer of attributes of an entity or association to other entities or associations.

Before



After



PHYSICAL DENORMALIZATION

Physical denormalization applies to database objects represented by tables, columns, keys and indexes.

Running Physical Denormalization

To denormalize the physical formalism:

1. Right-click the database and select **Physical denormalization**.

A wizard presents all customizations possible on database objects.

Physical denormalization example

It is possible to arrange that a table is partitioned into two separate tables, either to separate columns of the two tables (vertical partition), or to duplicate information in a table in two others (horizontal partition).

Consider the example of an order entry. This order entry is represented by an entity at the logical level, and gives a table at the physical level. Suppose that order entry has to be managed differently at technical level, depending on whether it is a telephone or Internet order. This change can be integrated in the physical model without having to modify the logical model in parallel. To do this, we arrange that the entity representing order entry is partitioned into two tables; one for telephone orders, the other for Internet orders. By integrating this modification from the wizard, the partition becomes automatic at each synchronization, and the two models remain consistent.

To create a horizontal partition such as that described above:

1. Right-click the "Order management" database that contains this entity and select **Physical denormalization**.
A wizard opens.
2. In the **Select the denormalization type field**, select "Horizontal partition of table".
3. Click **Next**.
4. In the editor tree, select the table you wish to duplicate, in this case "Order".
5. Click **Next**.

Denormalization options appear. Mapping transfer and source object deletion are activated by default. This means that the "Order" table will be deleted and the link with the "Order" entity will be transferred to the two tables.

Specify the number of partitions, in other words the number of tables created. The tool creates two tables by default.

- Click **Next**.
The editor displays changes produced by this denormalization. You can see that the "Order" table will be deleted and that two new tables will be created.

	Scope	Name
Order Management DB		HBC Group::Sales::...
Relational Diagram		Order Management...
Article		HBC Group::Sales::...
Article_type		HBC Group::Sales::...
Catalog		HBC Group::Sales::...
Client		HBC Group::Sales::...
Company		HBC Group::Sales::...
Discount		HBC Group::Sales::...
Order		HBC Group::Sales::...
Order_1		HBC Group::Sales::...
Order_2		HBC Group::Sales::...
Order_line		HBC Group::Sales::...

- Validate results by clicking **Next**.
This transformation is definitive and will be taken into account by the next synchronization.
On completion of denormalization, you can see that the two new tables are now mapped with the "Order" entity.

Denormalization here applies to the physical model. The synchronization that will take this customization into account must therefore be run in the same direction, in other words from logical model to the physical. It is not valid in the other direction.

List of Physical Denormalization Wizards

Vertical partition of a table

This denormalization enables division of a table into several tables. Columns are shared between the tables obtained.

Only columns that are not part of a key can be distributed between tables.

Before

A
a
b
c

After

A1
a
b

A2
c

Horizontal partition of a table

This denormalization enables duplication of a table. The two tables obtained contain all columns of the original table.

Before

A
a
b
c

After

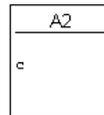
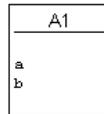
A1
a
b
c

A2
a
b
c

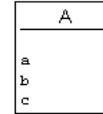
Merging of tables

This denormalization enables merging of tables.

Before



After

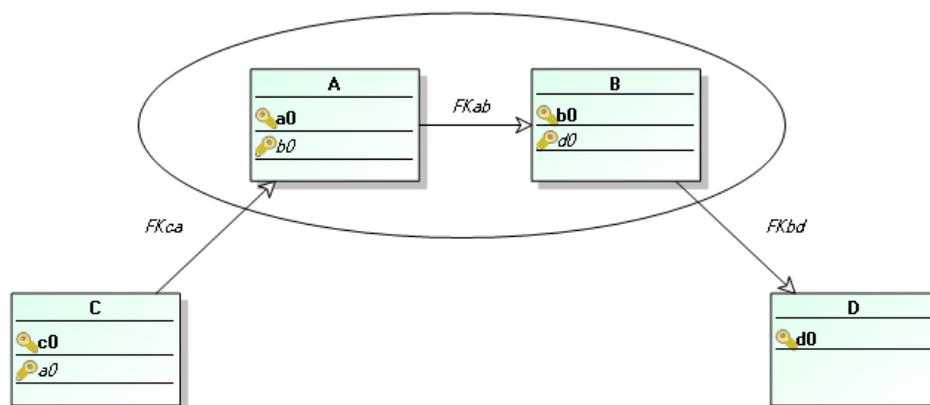


Primary keys option

When you run merging of tables, an option allows you to determine the primary key of the merge table: you can select one of the primary keys of the source tables or merge all primary keys of the source tables.

When you select a primary key for the merge table, only those foreign keys that reference this primary key are transferred. Foreign keys that reference primary keys that are not transferred are not taken into account.

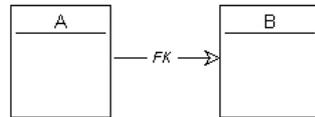
Therefore in the following example, if you merge tables A and B and keep the primary key of table A, the primary key of B disappears at merge. Nor is foreign key FK_{ab} transferred since it references primary key B. The other foreign keys, FK_{ca} and FK_{bd}, are transferred in the relational diagram.



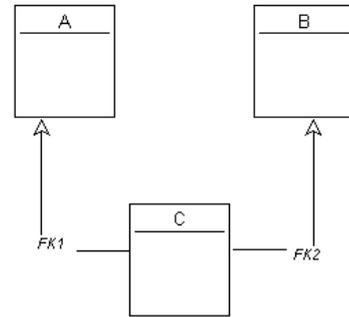
Transform foreign key to table

This denormalization enables transformation of a foreign key to a table. A new table and two new foreign keys are created. The original foreign key is deleted.

Before



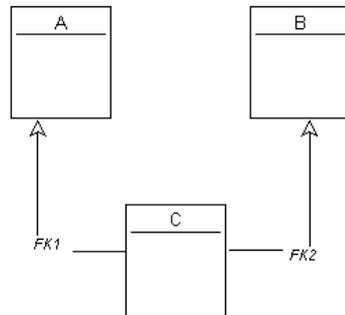
After



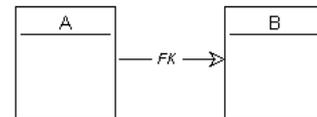
Transform table to foreign key

This denormalization enables transformation of a table to a foreign key. The table and its two foreign keys are deleted and a new foreign key is created.

Before



After



Copy/paste of columns

This denormalization enables transfer of columns of one table to another.

Before

A
a DATE

A1

After

A
a DATE

A1
a DATE



ATTRIBUTE AND COLUMN TYPES



Not all data has the same value type. Datatype determination enables indication of format and therefore facilitates handling by the different data processing tools.

HOPEX manages datatypes at different modeling levels, assuring correspondence of datatypes at the logical level with datatypes handled by the different supported DBMSs.

The following points are covered here:

- ✓ ["Attribute Datatypes", page 146](#)
- ✓ ["Determining Column Datatypes from Attribute Types", page 149](#)
- ✓ ["Mappings Between Pivot Types and Datatypes", page 155](#)
- ✓ ["Creating New Datatypes", page 158](#)

ATTRIBUTE DATATYPES

A type is used to group characteristics shared by several attributes.

To type attributes of an entity, only those datatypes defined for the *data model* that contains this entity are proposed.

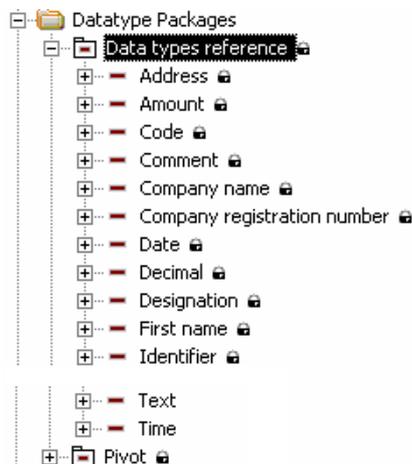
 A data model is used to represent the static structure of a system, particularly the types of objects manipulated in the system, their internal structure, and the relationships between them. A data model is a set of entities with their attributes, the associations existing between these entities, the constraints bearing on these entities and associations, etc.

Datatype Packages

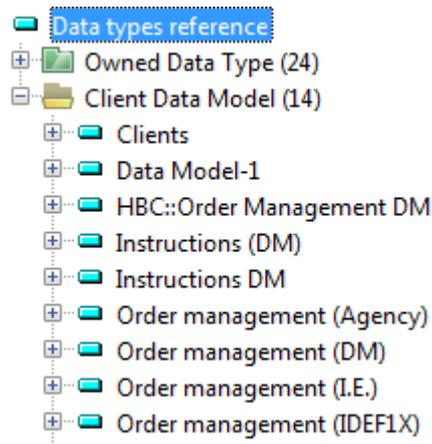
 A datatype package is a reference package owning all or a proportion of the datatypes used in the enterprise. All the other packages are declared as clients of the reference package of datatypes.

When you create a data model, the "Datatype Reference" *datatype package* is automatically associated with it by default.

This "Datatype Reference" package owns standard datatypes "Address", "Code", "Date", etc.



Opening the explorer on this datatype package, you can see that it is referenced by several data models.



The attributes of entities of these models can therefore be typed using the datatypes "Address", "Code", "Date", etc.

Creating a New Datatype Package

You can define a new reference datatype package owning the datatypes used by the enterprise.

To create your own datatype package:

1. On the desktop, click the navigation menu, then **Logical data**.
2. In the navigation pane, click the **Datatype packages**.
The list of datatype packages in the repository appears in the edit window.
3. Click **New**.
The datatype package creation dialog box opens.
4. Enter its name and an owner if appropriate.
5. Click **OK**.

You can then add types to this package.

Creating a datatype

To create a datatype:

1. Right-click the datatype package and select **Properties**.
The properties dialog box of the package appears.
2. Click the drop-down list then **Data Types**.
3. Click **New**.
The datatype creation dialog box opens.
4. Enter the name of the datatype and click **OK**.

Compound datatype

You can create compound datatypes by adding to them a list of attributes, for example an "Address" type comprising number, street postal code, city and country.

Literal value

You can allocate to a datatype literal values that define the values it can take. Attributes based on such a datatype can take only those values defined by the datatype.

When the new datatype package has been created, it should be referenced on the client data model.

Referencing Datatype Packages

When you define a new datatype package, you must connect it to the data model concerned.

To connect a datatype package to a data model:

1. On the desktop, click the navigation menu, then **Logical data**.
2. In the navigation pane, click the **All data models** folder.
The list of data models appears in the edit area.
3. Right-click the data model concerned and select **Properties**.
The data model properties dialog box appears.
4. Click the drop-down list then **Characteristics**.
5. In the **Reference** field, click the arrow and select **Connect Package**.
The query dialog box appears.
6. Click **Find**.
The list of datatype packages appears.
7. Select the desired package and click **OK**.

Assigning Types to Attributes

When the datatype package has been referenced for the data model, the list of types it contains is available on each attribute of entities of the model. All that is required is to select the one that is suitable.

To define the type of an attribute:

1. Right-click the entity that contains the attribute and select **Properties**.
The properties dialog box of the entity appears.
2. Click the drop-down list then **Attributes**.
3. In the **Datatype (DM)** column that corresponds to the attribute, select the desired type in the list.
4. Click **Apply**.

DETERMINING COLUMN DATATYPES FROM ATTRIBUTE TYPES

Datatypes defined at the logical level level are not always comprehensible for the target DBMS. In this case, they need to be converted to datatypes corresponding to the target DBMS.

This conversion intervenes notably at synchronization. Datatypes of attributes defined in the logical model are translated to datatypes for the generated columns.

Conversion is assured by an equivalence link with pivot types. The pivot types are an intermediary between logical datatypes and generated datatypes.

Pivot Types

Pivot types are datatypes defined independently of the target DBMS, which you can use when you do not yet know the system in which the database will be hosted, or when several systems may be used.

Pivot types have an equivalent datatype in each supported DBMS. They therefore enable you to define the attribute types just once, then to reinterpret them later as a function of the target DBMS.

To use the datatypes of a DBMS, you must import the corresponding solution pack. See ["Importing a DBMS version", page 59](#).

List of pivot types

Once imported, the pivot types are available in the **Logical data** navigation tab, in the "Pivot" datatype package.

Alphanumeric types	Other Information	
P-String	Alphanumeric character chain	
P-Text	Alphanumeric character chain	
P-Character	Alphanumeric string of fixed length	Length
P-Varchar	Alphanumeric string of variable length	
 Numeric types		
P-Decimal	Decimal	
P-Double		
P-Float		

P-Integer	Short	
P-Long Integer		
P-Long Real		
P-Real		
P-Smallint		
P-Tinyint		
P-Numeric	Number	Length, decimal places
P-Currency	Amount expressed as currency	Length, decimal places

Date types

P-Date	Date
P-Time	Time
P-DateTime	Date and time

Binary types

P-Binary	Binary string
P-Byte	Binary string
P-Timestamp	Identification automatically generated from the date and time, expressed in thousandths of seconds since January 1, 1970
P-Boolean	Boolean, equals 0 or 1
P-Multimedia	Binary string
P-Varbinary	Binary string

Connecting a Datatype to a Pivot Type

Datatypes contained in the "Datatypes Reference" package and associated by default with all new data models are connected to these pivot types. Therefore when you create new datatypes, these must be connected to the corresponding pivot types so that they can subsequently be used at physical level.

To connect a datatype to a pivot type:

1. Right-click the datatype and select **Properties**.
The properties dialog box of the datatype properties dialog box appears.
2. Click the drop-down list then **Characteristics**.
3. In the **SQL Datatype** field, select the pivot type.

Take the "Code" datatype. Open its properties dialog box and click the **Characteristics** page. In the **SQL Datatype** field, you can see that it is connected to the "P-Character" pivot type .

At synchronization of a logical model to a physical model, this pivot type "P-Character" will give a datatype CHAR, VARCHAR, LONG or TEXT depending on the DBMS concerned by synchronization. You can modify the target DBMS without having to modify the datatype, **HOPEX** assuring automatic conversion. See "Mappings Between Pivot Types and Datatypes", page 155.

Connecting a Datatype to a Pivot Type in UML Notation

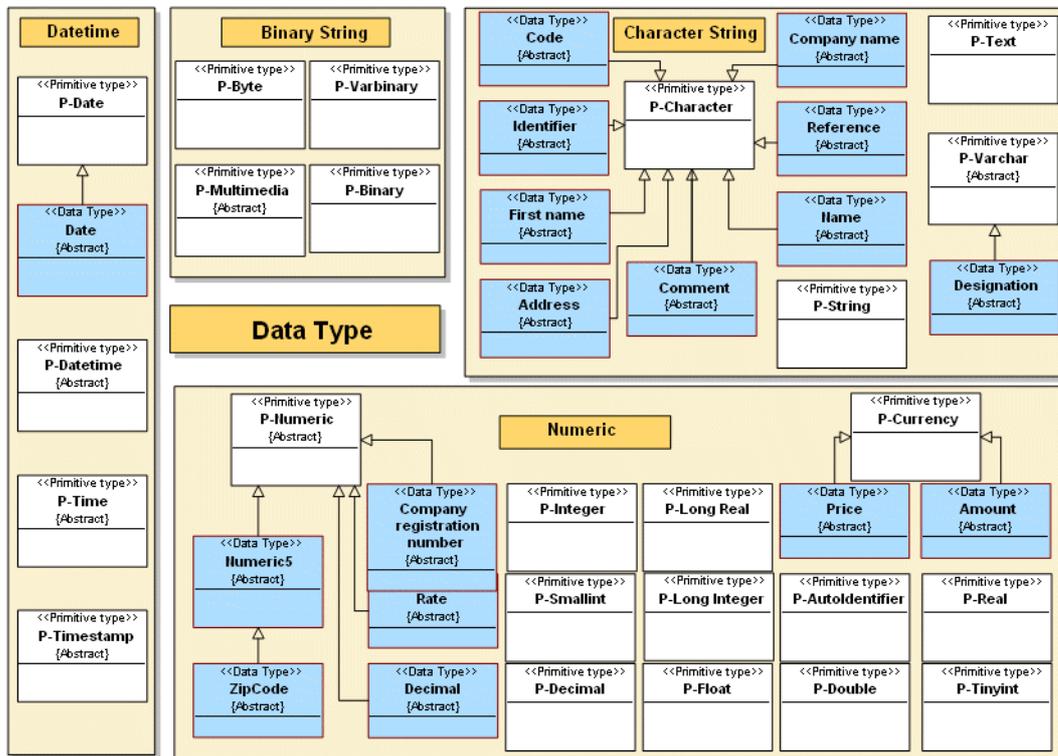
If you use UML notation and class diagrams to modify your data - and for reasons of compatibility with earlier versions of **HOPEX Database Builder** - other methods of referencing pivot types are possible.

You can create new datatypes and connect them to pivot types:

- By inheritance
- By a correspondence link
- By an equivalence link
- By creating a compound datatype

By inheritance

You can define your own datatypes by declaring them as subclasses of the pivot types, as shown in the example below.



The datatypes defined as subclasses will automatically inherit the characteristics of their superclass. In particular, the datatype conversion rule for the superclass is applied to the subclass.

It is possible to specify a length and a number of decimal places for the subclass. These will be taken into account when generating the data types if they were not already defined for the superclass.

By a correspondence link

To create this link:

1. Open the properties dialog box of the class.
2. Click the drop-down list then **Generation > SQL**.
3. Indicate the **SQL Type** associated with the class.
☛ Only pivot types of the Standard::Types::Pivot package are proposed in the list.
4. You can also indicate the length and the number of decimal places to be applied.

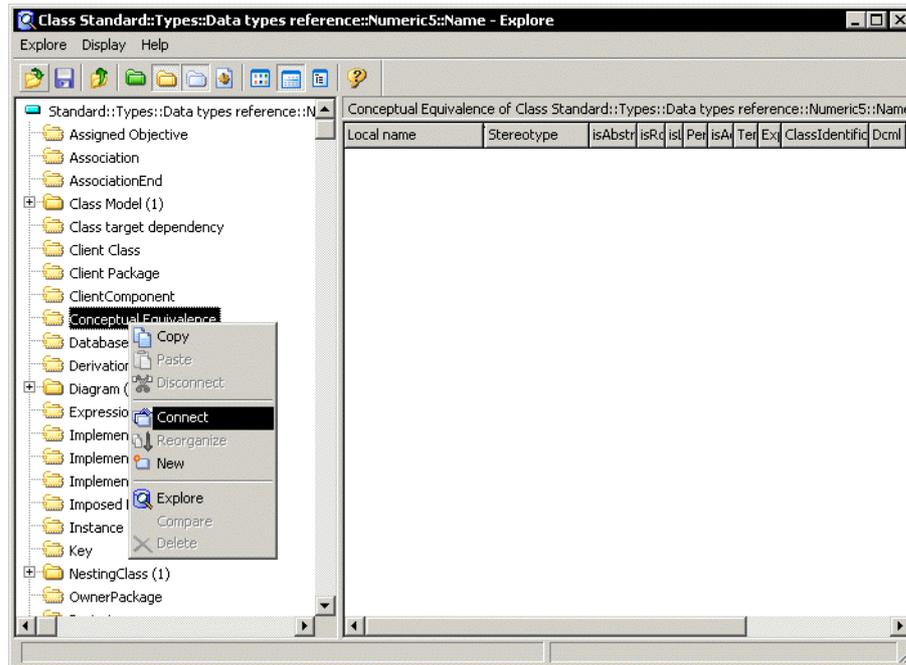
By an equivalence link

It is possible to define a new datatype by creating an equivalence link with a pivot type. To do this:

1. Create a new datatype
2. Right-click the type and select **Explorer**.
3. In the dialog box that appears, click the **Empty Collections** button.

☛ If necessary, click  to display the hidden commands.

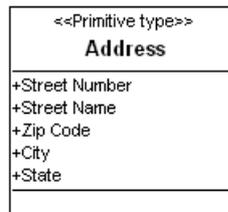
- Right-click the "Conceptual Equivalence" folder and select **Connect**.



- In the standard query dialog box, select the pivot type you wish to connect to your type.

By creating a compound datatype

You can define a compound datatype by assigning to it a list of attributes.



Here the "Address" type is composed of the number, street, zip code, city, and country.

The derivation of the "Address" attribute will produce these five columns.

It is possible to have several levels of compound types by assigning a compound type to an attribute of a compound type.

For example, the zip code can be broken down into the main five digits and the four-digit extension:

<<Primitive type>> Address
+Street Number +Street Name +Zip Code +City +State

<<Primitive type>> Zip code
+State code +Town code

MAPPINGS BETWEEN PIVOT TYPES AND DATATYPES

Pivot types establish correspondence between the logical datatypes to which they are connected and the datatypes for which they have an equivalent in each target DBMS.

The equivalence links carry conditions that enable them to be distinguished one from the other.

To visualize correspondences between pivot types and the different DBMS datatypes, see ["Pivot Types and Datatypes Correspondence Tables", page 223](#).

Example of correspondence between pivot types and Oracle 8 datatypes

Pivot to Datatype

Pivot	Condition	Datatype
P-AutoIdentifier		NUMBER
P-Binary		RAW(@L)
P-Boolean	L=2 or L \emptyset	RAW(1)
	L>1	RAW(@L)
P-Byte		RAW(1)
P-Character	L=256 or L \emptyset	CHAR(@L)
	L>2000	LONG
	255<L<2001	VARCHAR2(@L)
P-Currency		NUMBER(@L,@D)
P-Date		DATE
P-Datetime		DATE
P-Decimal		NUMBER(@L,@D)
P-Double		NUMBER(@L,@D)
P-Float		NUMBER(@L,@D)
P-Integer		NUMBER(@L)
P-Long Integer		NUMBER(@L)
P-Long Real		NUMBER(@L,@D)

Pivot	Condition	Datatype
P-Multimedia		LONG RAW
P-Numeric	L=0 or L \emptyset	NUMBER
	L>0 and D \emptyset	NUMBER(@L)
	L>0 and D not \emptyset	NUMBER(@L,@D)
P-Real		NUMBER(@L,@D)
P-Smallint		NUMBER(@L)
P-String		LONG
P-Text		VARCHAR2(@L)
P-Time		DATE
P-Timestamp		ROWID
P-Tinyint		NUMBER(@L)
P-Varbinary		LONG RAW
P-Varchar	L>2000 or L=0 or L \emptyset	LONG
	0<L<2001	VARCHAR2(@L)

Datatype to Pivot

Datatype	Condition	Pivot
CHAR(L)		P-Character
DATE		P-Date
LONG		P-String
LONG RAW		P-Multimedia
NUMBER		P-Numeric
NUMBER(L)		P-Numeric
NUMBER(L,D)		P-Numeric
RAW(1)		P-Boolean
RAW(L)		P-Boolean
ROWID		P-Timestamp
VARCHAR2(L)		P-Varchar

In this table, we can see that the "P-Numeric" type has three correspondences for type classes using three different conditions on the equivalence links.

if P_Numeric is assigned to an attribute and the length of this attribute is 10, then the column justified by this attribute via the synchronization will give Number(10).

The condition is written in VB Script language. The main elements of the condition are:

- **Sub ConditionInvoke (Column, ByRef bValid)**: the first line constitutes the signature of the function.
- **Column**: the column is given as an input parameter.
- **bValid** : is the return parameter. Its value is "True" if the condition is verified, "False" if not.

Example:

```
Sub ConditionInvoke (Column, ByRef bValid)
    bValid = False
    If (IsNumeric(Column.Length)) Then bValid = True
End Sub
```

The following can be specified in the condition:

- Presence of a number
- Presence of a decimal
- Range concerned (example: between 0 and 150 inclusive)

CREATING NEW DATATYPES

Each datatype is implemented in the form of a class; it is specific to a DBMS version. It is possible to use masks with datatypes.

Example for Oracle 10

Objective

In the ORACLE scripts, create a numeric datatype called Data8 with a length and a specified number of decimal places.

Steps

Steps are as follows:

1. Create a new datatype in **HOPEX**. If necessary, you can create a mask.
2. Connect the datatype to the target DBMS (in this case Oracle 10).
3. Connect the datatype to the corresponding type in the "Pivot" package.
4. Configure the conditions on each link in both directions (from datatype to pivot type and vice-versa).

☛ *For more information on equivalence links and conditions, see "Determining Column Datatypes from Attribute Types", page 149 .*

Prerequisite Conditions

To see packages containing DBMS datatypes, you must import the corresponding solution pack. See ["Importing a DBMS version", page 59](#).

☛ *You can connect the datatype to a generation target that you created. To create a generation target, see ["", page 170](#) and ["", page 170](#).*

🔒 **It is recommended that a datatype be defined in only one DBMS version.**

In addition, certain data is protected in **HOPEX**. To be able to modify objects contained in DBMS packages:

1. On the desktop, click **Main Menu > Settings > Options**.
2. In the tree on the left, click **Repository**.
The list of options linked to the repository appears in the right pane of the window.
3. In the "Authorize MEGA data modification" box, select "Authorize".
4. Click **OK**.

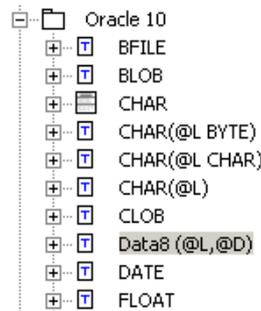
Creating a new datatype

To create a new datatype:

1. On the desktop, click the navigation menu, then **Logical data**.

2. Select the **All Packages** folder.
3. Right-click the "Oracle 10" folder and select **New > Class**. The **Creation of Class** dialog box opens.
4. Name your class "Data8 (@L,@D)".
5. Open the properties dialog box of this new class.
6. In the **Characteristics** page, select "Expression" in the **Stereotype** drop-down list, then click **Apply**.
7. The **Expression Type** field appears. In this field, select "Data8 (@L,@D)". Also delete the quotation marks around this value.

You will see in the navigator that a new class "Data 8" has been automatically created.



➤ This new class is automatically created for UML operating requirements..

Connecting the datatype to the pivot type

If you wish to obtain this datatype after synchronization, you must give it an equivalence at the logical level.

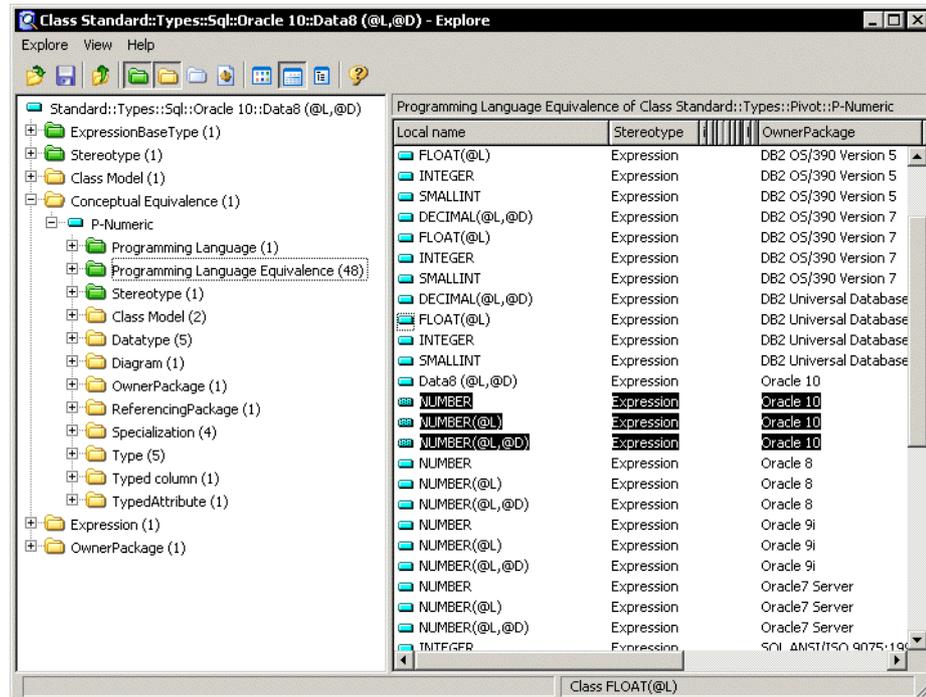
1. Open the properties dialog box of datatype "DATA8 (@L,@D)".
2. Click the drop-down list then click **Complements**.
3. Right-click the "Conceptual Equivalence" folder and select **Connect**.
4. In the query dialog box, select the class "P-Numeric".

Configuring conditions on links

To configure a condition on links:

1. Right-click the "Data8" class and select **Explorer**.
2. Expand the "Conceptual Equivalence" folder.

3. Select the green folder "Programming Language Equivalence".
You will see that there are three other cases of correspondence for Oracle 10.



Conditions on these correspondences must therefore be modified so that they will be coherent with the conditions placed on the new datatype.

4. Open the properties dialog box of datatype "NUMBER(@L,@D)".
5. In the **Texts** page, select "Language equivalence condition", and modify the text as follows:

```

Sub ConditionInvoke (Column, ByRef bValid)
    bValid = False
    Dim IsNumericLength
    IsNumericLength = IsNumeric(Column.Length)
    Dim IsNumericDecimal
    IsNumericDecimal = IsNumeric(Column.Decimal)
    If (IsNumericLength and IsNumericDecimal) Then
        If (Column.Length <> 8) Then
            bValid = True
        End If
    End If
End Sub

```

6. In the same way, add the following text in the properties dialog box of new datatype "Data8".

```
Sub ConditionInvoke (Column, ByRef bValid)
    bValid = False
    Dim IsNumericLength
    IsNumericLength = IsNumeric(Column.Length)
    Dim IsNumericDecimal
    IsNumericDecimal = IsNumeric(Column.Decimal)
    If (IsNumericLength and IsNumericDecimal) Then
        If (Column.Length = 8) Then
            bValid = True
        End If
    End If
End Sub
```

Verifying datatypes

To verify datatypes:

1. Right-click a **HOPEX Database Builder** navigator database and select **Tables**.
2. Display the properties of a column concerned by conditions placed on the datatype.
3. Verify that the mask displayed in the **Datatype** column is "Data8 (%l, %d)" for this column.

Example for SQL Server 7

Objective

Reread SQL Server 7 columns containing a non-standard datatype.

Manipulations are the same as for Oracle (see ["Example for Oracle 10", page 158](#)). On this occasion we shall not create a mask.

Creating a new datatype

To create a new datatype:

1. On the desktop, click the navigation menu, then **Logical data**.
2. Select the **All Packages** folder.
3. Right-click the "SQL Server 7" package and select **New > Class**. The **Creation of Class** dialog box opens.
4. Name your class "TLongName".
5. Open the properties dialog box of this new class.
6. In the **Characteristics** page, select "Expression" in the **Stereotype** drop-down list, then click **OK**.

Connecting the datatype to the pivot type

To connect the datatype to the primitive type:

1. Open the properties dialog box of the "TLibelleLong" datatype.
2. Select the **Complements** page.
3. Right-click the "Conceptual Equivalence" folder and select **Connect**.
4. In the query dialog box, select the "P-Text" class.

Configuring conditions on links

To configure a condition on links:

1. Right-click the "TLibelleLong" class and select **Explorer**.
2. Select the green folder "Programming Language Equivalence".
You will see that there is another correspondence for SQL Server 7.
3. Open the properties dialog box of datatype "text".
4. In the **Texts** page, select "Language equivalence condition", and modify the text as follows:

```
Sub ConditionInvoke (Column, ByRef bValid)
    bValid = False
    Dim IsNumericLength
    IsNumericLength = IsNumeric(Column.Length)
    If (IsNumericLength) Then
        If (Column.Length > 255) Then
            bValid = True
        End If
    End If
End Sub
```

5. In the same way, add the following text in the properties dialog box of new datatype "TLongName".

```
Sub ConditionInvoke (Column, ByRef bValid)
    bValid = False
    Dim IsNumericLength
    IsNumericLength = IsNumeric(Column.Length)
    If (IsNumericLength) Then
        If (Column.Length <= 255) Then
            bValid = True
        End If
    End If
End Sub
```

GENERATING SQL SCRIPTS



The SQL generation function produces SQL script files, which, from logical objects of your **HOPEX** repository (database, table, column, etc.) allow you to create, modify or update the corresponding objects in the target DBMS of your choice.

Generation takes into account parameters inherited from the target DBMS (specified for the database), parameters that you can customize at a global level (see ["Configuring Database Generation", page 170](#)) or at a more detailed level, on a column or primary key for example.

For the main target DBMSs on the market, the database editor makes accessible a "physical view" that allows you to optimize the SQL grammar of generated scripts in order to integrate technical options specific to the selected DBMS, such as partitioning. See ["Adding Physical Properties to Database Objects", page 191](#).

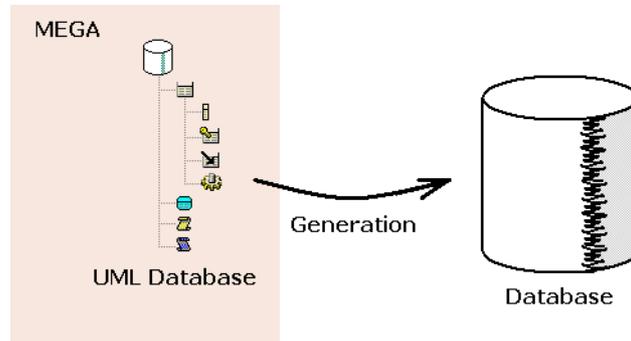
Finally, logical level can be completed by generation of physical objects specific to each database for a DBMS, such as logical views, stored procedures and triggers. See ["Advanced SQL Options", page 179](#).

The different generation modes presented below take into account the constraints linked to database administration under different systems with maximum flexibility.

The points covered here are:

- ✓ ["Running SQL Generation", page 164](#)
- ✓ ["Incremental Generation", page 166](#)
- ✓ ["Configuring SQL generation", page 169](#)
- ✓ ["Supported Syntax", page 173](#)

RUNNING SQL GENERATION



SQL Generation Objects

Objects taken into account in generation are:

- Table
- Column
- Primary key
- Foreign key
- Indexes
- Data group
- Logical view
- Material view
- Trigger
- Stored procedure
- Job Title
- Synonym
- Sequence
- Cluster
- Partition

Scripts generated by **HOPEX** manage only the structure of relational objects, their content is not covered.

Start the generation wizard

Display of certain generation targets can be filtered. Before starting generation, check that the selected generation target is activated:

1. On the desktop, click **Main Menu > Settings > Options**.

2. In the left pane of the options window, expand the **Data Modeling** folder then double-click the **SQL Generation** folder. This folder contains all supported SQL generators.
3. In the right part of the window pane select those that you want to display in **HOPEX Database Builder**

For generation targets, see "[Supported DBMS versions](#)", page 169.

To start an SQL generation:

1. On the desktop, click the navigation menu, then **Physical data**.
2. In the navigation pane, select **SQL Code Generation**. A wizard opens.
3. Define the **Generation scope**:
 - database for a complete generation
 - another SQL object for a partial generation.
4. Select the database concerned and click **Next**.
- 5.
6. Select the **Generation Mode**. Four generation modes are possible:
 - "Creation": generates creation orders for all objects.
 - "Deletion": generates object deletion orders only.
 - "Replacement": starts deletion of objects, then recreation (to avoid creation of duplicates for example). Assumes that the target DBMS supports this generation mode.
 - "Modification": modifications only are taken into account. Unlike the other modes that act without taking account of what possibly exists, this mode enables connection to the DBMS server to obtain read-only access to the database already created. The wizard compares the **HOPEX** data file and the database information. After analysis of the two structures, the wizard generates the corresponding modification SQL orders. See "[Incremental Generation](#)", page 166.
 - ☛ *This mode is only available for the main DBMSs. See "[Supported DBMS versions](#)", page 169.*
7. Click **Next**. A dialog box then presents the objects generated.
8. Click **Next**. Generation starts. A dialog box presents progress of the operation indicating the file(s) containing the result. Click **Open** to see the list of the files generated.

When the extension used for the generated files is recognized by Windows, you need only double-click the file name to view it using the editor associated with the extension.

INCREMENTAL GENERATION

When a database has already been generated, you can subsequently reflect only changes to the database using "Modification" mode of SQL generation.

For a database, incremental generation allows you to:

- consult in an HTML report the differences between the database and its representation in **HOPEX**.
- produce SQL scripts enabling update of the target database from its description in **HOPEX**.

Incremental Generation Objects

Objects managed by incremental generation are the same as those of generation in "Creation" mode: table, column, primary key, foreign key, index, data group, logical view, material view, trigger, stored procedure, function, synonym, cluster, partition.

Scripts generated by **HOPEX** manage only the structure of relational objects, their content is not covered. Incremental generation options enable isolation of SQL orders that require particular precautions or additional processing.

Running Incremental Generation

Generation options

Incremental generation is done in a global file; it is carried out from the database and not from a particular modified object.

Before starting generation:

1. Right-click the database and select **Properties**.
The properties window of the database appears.
2. Click the drop-down list then **Options > Generation**.
3. In **Script Distribution**, select "A global file".
4. Click **OK**.

In options, you must also indicate incremental generation mode, which authorizes object deletion or not.

For more details on generation options, see ["Configuring Database Generation"](#), page 170.

Start the generation wizard

To run incremental generation:

1. Right-click the database and select **Generate the code**.

2. In the **Generation mode** field, select "Modification".
 - ☛ This "Modification" command is only available for the main DBMSs. See "Supported DBMS versions", page 169.
3. Select the **Data source**. Incremental generation can be carried out:
 - From an ODBC connection.
 - From an extraction file See "ODBC Extraction Utility", page 205.
4. Click **Next**.
5. When connected to the target DBMS, enter the name of the owner. This will enable you to filter the tables to be taken into account in the generation.
6. Click **Next**.

The result window presents two files, the SQL file and a "Report.htm" file. The latter file is a report of the generation. It is a dynamic file that presents initial content of the database and modifications carried out.

Origin (MEGA)		Target (Oracle 10)	Updating ac
TBSTABLE		TBSTABLE	
TBSPARTITION		TBSPARTITION	
TBSOVERFLOW		TBSOVERFLOW	
DEPARTEMENT_TEST5	⬇ ⚠	DEPARTEMENT_TEST5	Insight
DEPARTEMENT_TEST4		DEPARTEMENT_TEST4	
DEPARTEMENT_TEST1	⬇	DEPARTEMENT_TEST1	Insight
DEPTNO		DEPTNO	
DOM		DOM	
NAME		NAME	
PAYS		PAYS	
REGION		REGION	
PARAMETERS		PARAMETRES	

Each row of the list describes:

- The **HOPEX** object. This can be empty if it has been deleted from the **HOPEX** repository.
- The DBMS update reference as compared to **HOPEX**. The various actions possible on objects are:
 - Creation ✳
 - Modification ⬆
 - Deletion ✖
 - Replacement 📄
- A warning symbol ⚠ when the update of a DBMS object is not complete, or when this must be handled with care. When this icon is present, a block in the generated script details what cannot be updated.
- Object on DBMS side. This can be empty in the context of creation on the **HOPEX** side.
- The link to the object update script.

From each object you can access all its sub-objects by expanding the corresponding tree. You can also view the physical parameters. In a context of object modification, only the modified physical parameters are displayed.

There is also a report of the generation (.txt) in the properties dialog box of the generated database.

CONFIGURING SQL GENERATION

Configuring the DBMS Version

Supported DBMS versions

This list is given as a guide only. It is not intended to be an exhaustive list, and may change as new DBMS versions are released.

Product	Editor	Supported Versions
SQL ANSI	ISO 9075	1992
DB2	IBM	OS 390 V5 / OS 390 V7 / OS 390 V8 UDB V5 / UDB V7 / UDB V8
Ingres II	Computer Associates	2.0
Dynamic Server	Informix	7.3
Oracle	Oracle	8 / 9i / 10 / 11
SQL Server	Microsoft	7 / 2000 / 2005 / 2008
Adaptative Server	Sybase	11 / 12.5
Teradata Database	Teradata	14
PostgreSQL	PostgreSQL Global Development Group	9.3

Modifying DBMS version properties

Generation configuration is carried out to check the size of the identifiers generated, the characters authorized, etc.

You can access a DBMS from a database linked to this DBMS.

To configure generation options of a DBMS version:

1. Right-click the database concerned and select **Properties**.
The properties window of the database appears.
2. Click the drop-down list then **Characteristics**.
3. In the **Target DBMS** field, click the arrow at the extreme right to access the properties of the DBMS.
4. In the properties window of the DBMS, click the drop down list and select **Options > Generation**.

5. Modify the parameters you wish to change.

☛ *The parameters available vary as a function of the target DBMS. If the **Generation** subtab does not appear, select **Tools > Options** in the HOPEX bar, expand "Data Modeling, SQL Generation" and check that the generation option linked to the DBMS concerned is selected.*

Configuring Database Generation

To configure generation of a database:

1. Open the database properties dialog box.
2. Click the drop-down list then **Options > Generation**.

As for configuration of a target, the parameters proposed vary as a function of the DBMS and an explanatory message indicates the use of each parameter.

You can specify the following parameters:

- The **Trigger Name** parameters define the names of three types of trigger.
- **Error Ref Value**: user error number for the current DBMS.
- **Default Value**: activates/deactivates generation of DEFAULT orders for columns.
- **Quoted Identifier**: activates/deactivates generation of quotes around SQL identifiers (SQL name).
- **Qualifier** : enables prefix of object names. See "[Prefixing Object Names](#)", page 172.
- **Mode Generation Inc**: this parameter applies to incremental generation and can take values "Alter" and "Drop/Create".
 - "Alter" does not authorize deletion of objects (tables, indexes, etc.) at the level of generated scripts. Only those instructions that can be executed using the ALTER command are generated. For physical parameters, deletion is still authorized (this is the case notably for partitions).
 - "Drop/Create" authorizes object deletion. If an update cannot execute via the ALTER command, the object is deleted then recreated. By default, the parameter takes value "Alter".
- **Script Distribution**: indicates if the result of generation should be created in a unique file or in one file per object or in one file per object type.
- **SQL Script**: name of the file generated when this is a unique file. By default, this sub-folder is called REFEXT. You can customize at DBMS level. The arrow at extreme right of the field allows you to reinitialize the parameter.
 -  **You can also reinitialize all parameters of the object concerned. This action should be carried out with care.**
- **Script Directory**: relative generation folder.
- The various **Ext** parameters allow you to specify extensions of each file generated for tables, datagroups, views, etc.

- **Conversion:** format of generated files (ANSI Windows or ASCII MS-DOS).
- **CREATE CLUSTER:** activates/deactivates generation of CREATE CLUSTER orders.
- **CREATE TABLE:** activates/deactivates generation of CREATE TABLE orders.
- **CREATE TABLESPACE:** activates/deactivates generation of CREATE TABLESPACE orders.
- **PRIMARY KEY:** activates/deactivates generation of PRIMARY KEY orders.
- **FOREIGN KEY:** activates/deactivates generation of FOREIGN KEY orders.
- **CREATE INDEX:** activates/deactivates generation of CREATE INDEX orders.
- **CREATE PROCEDURE:** activates/deactivates generation of stored procedures.
- **CREATE INDEX PK:** activates/deactivates generation of CREATE INDEX orders for primary key indexes.
- **CREATE INDEX[UNIQUE]:** activates/deactivates generation of CREATE INDEX orders for unique indexes.
- **CREATE VIEW:** activates/deactivates generation of logical views.
- **CREATE SEQUENCE:** activates/deactivates generation of CREATE SEQUENCE orders.
- **CREATE SYNONYM:** activates/deactivates generation of CREATE SYNONYM orders.
- **CREATE TRIGGER:** activates/deactivates generation of triggers.
- **Comments:** activates/deactivates generation of **HOPEX** comments in SQL script.
- **UNIQUE:** activates/deactivates generation of UNIQUE orders.
- **UNIQUE[PK]:** activates/deactivates generation of UNIQUE orders for primary keys.
- **PRIMARY KEY syntax:** PRIMARY KEY orders are generated in CREATE TABLE order or in an ALTER TABLE order.
- **Position FOREIGN KEY:** generation of FOREIGN KEY orders after each CREATE TABLE or grouped at end of script.
- **COMMENT ON TABLE:** comments on tables (0:no comment, 1:one line, Total:all text)
- **COMMENT ON COLUMN:** comments on columns (0:no comment, 1:one line, Total:all text)
 - ☛ *Generation of comments is only possible for target systems that accept these (Oracle, DB2,...).*
- The various **Add-Ons** parameters activate/deactivate generation of add-ons on tables, datagroups, etc.
- **Tbspace of Tables:** by default, tables are generated in the SYSTEM tablespace.
- **Tbspace of Indexes:** by default, tables are generated in the SYSTEM tablespace.

Prefixing Object Names

There is a schema concept for most DBMSs, which enables definition of a logical grouping for objects.

Therefore at creation of a table for example, it can be automatically stored in a schema, and if this is not specified, a default schema can be automatically assigned.

In **HOPEX** there is not a schema concept, but a specific concept - the Qualifier - which enables prefix of database object names at generation. If for example you want objects to have names prefixed "MEGA", you must enter this value in the Qualifier field of the objects in question.

Inheritance

The Qualifier property can be defined at database level and on all other object types. There is an Inheritance system: if the Qualifier is not specified at the level of a table, by default it is the value entered on the database that is taken into account.

To prefix the name of an object at generation:

1. Open the properties dialog box of the object in question.
2. Click the drop-down list then **Options > Generation**.
3. In the **Qualifier** field, enter the value that will prefix the name of the object.

DBMSs concerned

The Qualifier is available for the following DBMSs:

- Oracle
- SQL Server
- DB2
- MySQL

SUPPORTED SYNTAX

CREATE TABLE Instruction

The CREATE TABLE instruction defines a table. The definition includes:

- Table name
- The names and attributes of its columns.
- The attributes of the table such as its primary and foreign keys.

The syntax is as follows:

```
CREATE TABLE table-name (col1-name col1-type [NOT NULL]
...
name-coln type-coln [NOT NULL])
```

For DB2, the syntax is as follows:

```
CREATE TABLE table-name (col1-name col1-type [NOT NULL]
...
name-coln type-coln [NOT NULL])
[in Tablespace <Name>]
```

For Oracle, the syntax is as follows:

```
CREATE TABLE table-name (col1-name col1-type [NOT NULL]
...
name-coln type-coln [NOT NULL])
[Tablespace <Name>]
```

- **table name** : "SQL" value for the table, or else defaults to the name of the table; unrecognized characters are replaced by "_"
- **col-name**: Value of the SQL Name attribute for the column, or by default the name of the column; unrecognized characters are replaced by "_"
- **col-type**
- **NOT NULL**: See ["Managing NOT NULL", page 173](#).
- **Tablespace**: DB2 and Oracle: Name of the target tablespace for the tables

The PRIMARY KEY clause is generated within the CREATE TABLE command (see ["PRIMARY KEY clause", page 174](#)).

Managing NOT NULL

Clauses NULL, NOT NULL and NOT NULL WITH DEFAULT are generated automatically on the columns of primary keys and on columns derived from obligatory attributes at time of synchronization.

These values can be initialized as "Null", "Not Null" or "Not Null with Default" as a function of configuration defined in the database Properties dialog box for **Not Null Columns**, in the **Synchronization** subtab of the **Options** tab.

The values proposed can then be modified on each column.

PRIMARY KEY clause

Defining a primary key

One or more of the columns in a table can be used to uniquely identify each row in the table. Values in these columns must be specified. They act as the primary key for the table.

A table must have only one primary key or none.

Each column name must identify a column in the table, and the column cannot be identified more than once.

Processing and generating SQL commands

After declaring the names of the columns in the table, if the **PRIMARY KEY** option is enabled, the name(s) of the columns in the primary key are declared as follows:

```
PRIMARY KEY (list of columns in the primary key)
```

The PRIMARY KEY clause is generated within the **CREATE TABLE** command.

Example 1	Example 2
<p>The primary key "PK" has only one column, "PK-col".</p> <pre>CREATE TABLE table-name (PK-col CHAR(9) NOT NULL, info1 CHAR(7), info2 CHAR(7), PRIMARY KEY (PK-col)</pre>	<p>The primary key "PK1" has columns "PK11" and "PK12".</p> <pre>CREATE TABLE table-name PK11 CHAR(9) NOT NULL, PK12 CHAR(9) NOT NULL, info1 CHAR(7), info2 CHAR(7), PRIMARY KEY (PK11, PK12)</pre>

For Oracle, the complete PRIMARY KEY clause is as follows:

```
CONSTRAINT PK_<key name> (list of columns in the primary
key)
```

FOREIGN KEY clause

Database integrity can be ensured either by FOREIGN KEY clauses or by generated triggers, depending on the target DBMS. (In Oracle, it is ensured either with triggers, or with FOREIGN KEY clauses, depending on the database configuration.)

One or more columns in a table can refer to a primary key in this table or in another table. These columns form the foreign key. These columns do not need to have a value in each row.

The table containing the referenced primary key is a parent table. The table containing the foreign key is a dependent table.

Each column name must identify a single column in the table, and the same column cannot be identified more than once. If the same list of column names is specified in more than one FOREIGN KEY clause, these clauses cannot identify the same table.

The table name specified in the FOREIGN KEY clause must identify a parent table. A foreign key in a dependent table must have the same number of columns as the primary key for the parent table.

The number of foreign keys is unlimited.

Processing and generating SQL commands

After declaring the primary keys (PRIMARY KEY), the column name(s) for the foreign key(s) are declared for a table using FOREIGN KEY:

```
FOREIGN KEY (list of columns in the foreign key) REFERENCES
<Parent table name> [ON DELETE <Action>] [ON UPDATE
<Action>]
```

or:

```
ALTER TABLE tablename [ADD] FOREIGN KEY (list of columns for
the foreign key) REFERENCES <Parent table name > [ON DELETE
<Action>] [ON UPDATE <Action>]
```

For Oracle, the syntax is as follows:

```
CONSTRAINT FK_<name of the foreign key> (list of columns for
the foreign key) REFERENCES <Parent table name > [ON DELETE
<Action>] [ON UPDATE <Action>]
```

or:

```
ALTER TABLE...
```

Examples

The table "table1-name" has two foreign keys. These keys have no components.

```
CREATE TABLE table1-name
pk1 CHAR(9) NOT NULL,
pk2-rel12 CHAR(7) NOT NULL,
pk3-rel13 CHAR(7) NOT NULL,
info1 CHAR(7),
info2 CHAR(7),
PRIMARY KEY (pk1))
ALTER TABLE table1-name ADD FOREIGN KEY(cp2-rel12)
REFERENCES table2-name
ALTER TABLE table2-name ADD FOREIGN KEY(cp3-rel13)
REFERENCES table3-name
```

The table "table1-name" has a foreign key "fk2" which has two components, "fk21" and "fk22". The foreign key "fk2" has no reference (it is therefore a component of the primary key of another table).

```
CREATE TABLE table1-name
pk1 CHAR(9) NOT NULL,
fk21 CHAR(7) NOT NULL,
fk22 CHAR(7) NOT NULL,
info1 CHAR(7),
PRIMARY KEY (pk1)
ALTER TABLE table1-name ADD FOREIGN KEY (fk21, fk22)
REFERENCES table2-name
```

The table "table1-name" has a foreign key, "fk2". The foreign key "fk2" is equivalent to the primary key "pk2" which has two components, "pk21" and "pk22". The columns identified by "pk21" and "pk22" are "NOT NULL".

```
CREATE TABLE table1-name
pk1 CHAR(9) NOT NULL,
pk21 CHAR(7) NOT NULL,
pk22 CHAR(7) NOT NULL,
info1 CHAR(7),
info2 (CHAR7),
PRIMARY KEY (pk1)
ALTER TABLE table1-name ADD FOREIGN KEY (pk21, pk22)
REFERENCES table2-name
```

UNIQUE clause

A UNIQUE clause is generated for each unique index in the table, unless this index corresponds to the primary key.

Processing and generating SQL commands

For each unique index, the following clause is generated:

```
UNIQUE (col1, ..., coln)
```

(col1,...n,coln) represent the columns used by the index.

CREATE INDEX Instruction (Oracle, Sybase, SQL Server)

Definition of an index

An index is a set of columns in a table for which a direct access is defined.

For Sybase and SQL Server, the value of the index-type attribute for the index determines what type of index is generated: UNIQUE, CLUSTERED, or UNIQUE CLUSTERED.

Processing and generating SQL commands

For each index a clause is generated as a function of the target DBMS.

For Oracle:

```
CREATE INDEX (column1, ..., columnN) [TABLESPACE  
(TbSpaceName)
```

(column1, ..., columnN) represent the columns in the index; TbSpaceName is the name of the tablespace for the indexes (see ["Configuring SQL generation", page 169](#)).

For Sybase and SQL Server:

```
CREATE [UNIQUE] [CLUSTERED] INDEX (IndexName) (TableName)  
(column1, ..., columnN)
```

(column1, ..., columnN) represent the columns in the index; TbSpaceName is the name of the tablespace for the indexes (see ["Configuring SQL generation", page 169](#)).

CREATE VIEW Clause

A view is defined for a database. It can include one or more tables.

```
CREATE VIEW view-name  
AS  
SELECT  
(column-name, column-name, ...)
```

You can enhance this definition in the view specification (see ["Defining Database Views", page 180](#)).



ADVANCED SQL OPTIONS



You can supplement relational object definition with definitions of technical objects such as views, triggers and stored procedures, specifications of which are specific to each target DBMS and can be generated separately or in the SQL file of the database.

You can therefore:

- ✓ ["Defining Database Views", page 180](#)
- ✓ ["Defining Triggers for a Database", page 186](#)
- ✓ ["Using Stored Procedures", page 188](#)
- ✓ ["Adding Physical Properties to Database Objects", page 190](#)

Tools presented in the pages that follow are available with "Advanced" access to the repository.

DEFINING DATABASE VIEWS

A physical view is a virtual table, of which structure and content are deduced from one or several other tables with an SQL query.

Database *views* are created in a tree format, which automatically generates part of the view definition. The user can then add to it as desired.

Creating Database Views

To create a physical view from a database:

1. On the desktop, click the navigation menu, then **Physical data**.
2. In the navigation pane, select **Physical views**.
The list of physical views appears in the edit area.
3. Click **New**.
The view creation wizard opens.
4. In the **Owner** field, select the database concerned.
5. In the **Physical View Component** field, click **New**.
6. Select the tables concerned by the view
7. Click **OK**.
The physical view editor appears.

The left tree displays tables to which the physical view relates, with their columns. The right tree displays the tables and columns that constitutes the view. By default these ones have the same names as the source tables and columns. You can rename them.



Add a table or a column to a view

To add a table to a view:

1. In the right-hand part of the editor, right-click the **Table** folder and select **Table view**.
2. Select the desired table and click **OK**.

To add a column to a view:

1. In the right-hand part of the editor, right-click the **Column** folder and select **Column view**.
2. Select the desired column and click **OK**.

SQL Definition

The right side of the dialog box, labeled **SQL Definition**, shows the SQL code that would be generated to define the view. The code is initially calculated based on the definition indicated in the tree.

You can modify this code, in particular by using joints. You can also directly enter modifications in the SQL frame.

View joints

By default, the edition of logical views window proposes the foreign keys of the selected tables where these exist.

It is thus possible to complete specification of a view by associating with it foreign keys, potential sources of joints.

To associate a foreign key with the view:

- 1. Select the foreign key in the tree on the left and drag it into the SQL definition field.

User mode

You can modify the view code by typing directly in the SQL definition field:

- 1. Click the **Save** button so that the **SQL Definition** will be saved in the repository as is.

After you have modified the definition, you can restore the definition as determined by the tree:

- 1. Click the **Initialize the SQL definition**  button.
A message warns you that the previously saved definition will be reinitialized. In other words, any user additions made to the definition will be lost.
- 2. Click **OK** to confirm.

Fields

Field categories correspond to object types used in the declarative tree: table, view, column and foreign key column. Fields displayed in the SQL definition correspond to elements declared in the tree.

The foreign key type does not produce a field category: usable fields are derived from key columns and not from the keys themselves.

The **Field properties**  button displays properties of the object corresponding to the selected field.

If an object is added to the tree, a corresponding field becomes available for insertion.

If an object is renamed in the tree or in the repository, its references remain valid and the fields are displayed in the text with the new name.

If an object is deleted in the tree or in the repository, its references become invalid and are indicated as such in the fields.

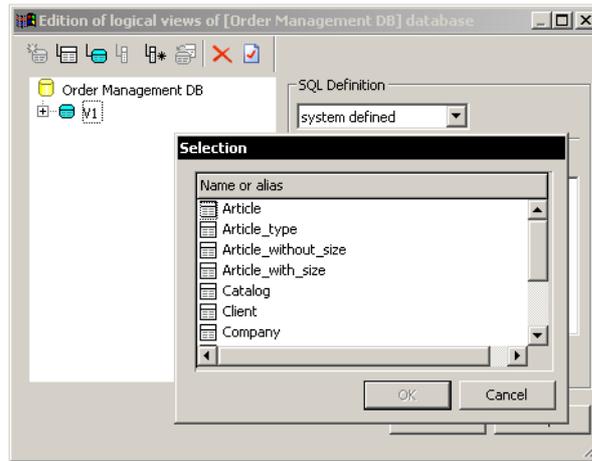
View example

Suppose you want to create a view for the "Order Management" database. In this view, you wish to see the code of the various articles and their sales tax rate.

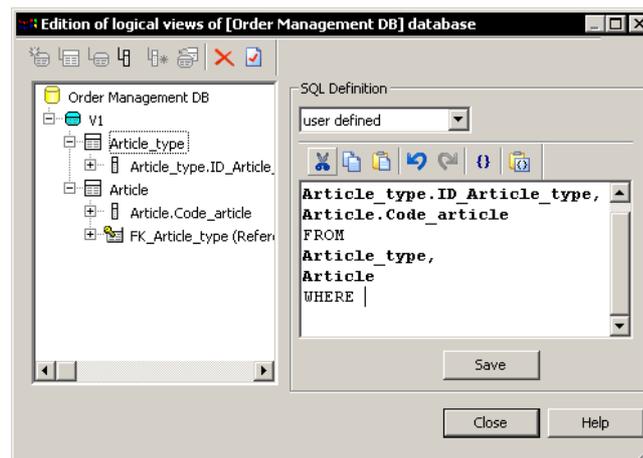
To do this:

1. Right-click the "Order Management" database.
2. Select **Views**.
3. In the dialog box that appears, select "Order Management" then click the **Create a View**  button.
4. Enter the name of the view, "V1" by default, then click **OK**.
To specify tables taken into account in the view:

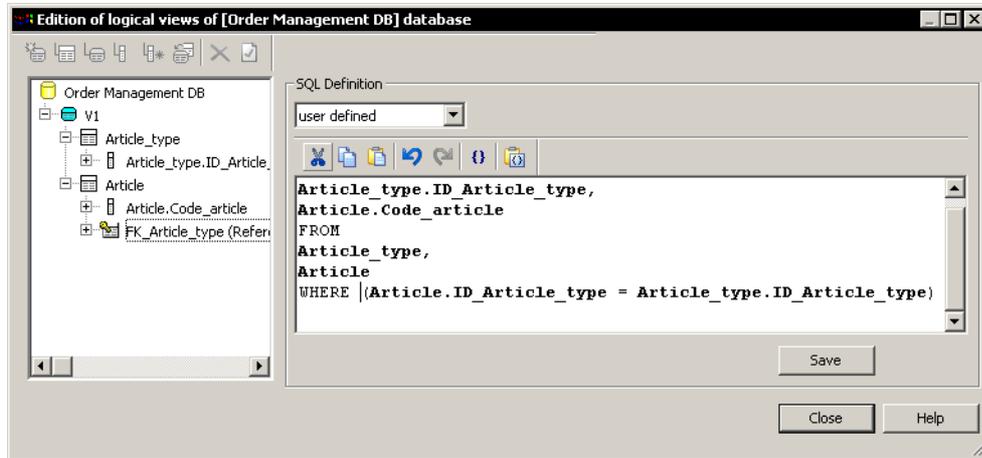
5. Select "V1", then click the **Add Tables**  button. The tables selection dialog box opens.



6. Select the tables that will enable you to define the view. In this case, these are "Article" and "Article_type".
7. Click **OK**. You will see that the corresponding codes appear in the right part of the Edition of logical views dialog box. To specify the columns that you wish to display:
8. Right-click the tables and select **Add Columns**. In our example, in the "Article_type" table, select the "Article_type.Tax_rate" column, and in the "Article" table select the "Article.Code_article" column.
9. In the SQL definition frame, type WHERE + space. You then pass automatically to user mode.



10. Click **Save**.
Under the "Article" table you will see the "FK_Article_type" foreign key. This key is proposed by default, and enables you to make the joint between the two tables.
11. Select the key and drag it into the code after the WHERE command.
The columns to which the key is linked appear automatically in the code.



12. Click **Save**.

Defining a Data Group

A data group - or tablespace - is a group, in the same physical pages of the database, of the rows of several tables to optimize queries, joints in particular. Example: Tablespace in DB2, Cluster in Oracle etc.

To define *data groups* in the database:

1. Open the database properties dialog box.
2. Click the drop-down list then **Components**.
3. The **Data groups** section displays the list of data groups
4. Click the **New** button.
5. In the dialog box that opens, indicate the **Name** of the data group.
6. Click **OK**.

Then open the properties window of the data group to define the tables and indexes that it includes.

To specify the tables and indexes included in the data group:

1. Select the data group and click **Properties**.
The data group properties dialog box appears.
2. Click the drop-down list then **Tables**.
3. Click the **Connect** button.
4. In the list of database tables presented, select the tables to be included.

➡ Click **Disconnect** to remove a table from the list in the case of error.

5. Carry out the same operations for the indexes of the data group.
6. Click **OK**.

DEFINING TRIGGERS FOR A DATABASE

A trigger is processing recorded in a database, which automatically triggers on updating a table.

Creating Triggers

Triggers are defined at the level of database tables.

☛ *It should be noted that triggers are defined as a function of the target DBMS; this is why it is important to check that the target DBMS is correct before creating triggers.*

If the target DBMS is later modified, triggers created for the DBMS are not deleted but deactivated.

To create a trigger:

1. On the desktop, click the navigation menu, then **Physical data**.
2. In the navigation pane, in the **Database** folder, click **Database hierarchy**.

The list of databases appears in the edit area.

3. Expand the folder of the database then the table concerned.
4. Right-click the **Trigger** folder and select **New > Trigger**. The dialog box for creating a trigger opens.
5. Specify the name of the trigger and actions triggered. See ["Trigger triggering", page 186](#).

Trigger triggering

Triggering can occur following one of the three following actions:

- At **Creation** of a row in the table.
- At **Deletion** of a row.
- At **Modification** of the table or of a particular column.

In addition, you can choose to run it **Before** or **After** these actions, on the entire table, or on each row concerned.

References

The "Reference of old rows" and "Reference of new rows" fields create in the trigger code references to lines inserted, deleted or updated.

The name indicated in the "Reference of old rows" field corresponds to the line that existed before the update.

The name in the "Reference of new rows" field indicates the line after the update.

In the case of insertion, only the new line is valid.

In the case of deletion, only the old line is valid.

SQL Definition

The **SQL Definition** option in the properties window of the trigger presents the trigger code.

To access this option you must have "Expert" access to the metamodel.

To display Expert mode:

1. On the **HOPEX** desktop, click **Main Menu > Settings > Options**.
2. In the left pane of the window, click the **Repository** folder.
3. In the right pane of the window, for **Repository Access** select "Expert" mode.

To display the trigger code:

1. Right-click the trigger and select **Properties**.
The properties window of the trigger appears.
2. Click the drop-down list then click **Texts**.
3. Select the **SQL Definition** tab.

Repository Integrity

Repository integrity is managed by creation of foreign keys on a database.

It groups all constraints allowing a check of the impact of modification of a table in tables connected to it.

It could be that the existence of keys in certain DBMSs does not involve a systematic check. It could also be that you wish to customize constraints to be applied on a particular table.

This is why you can generate in triggers the code that corresponds to repository integrity management.

To generate repository integrity of a table:

1. Right-click the database and select **Generate triggers**.
The trigger generation dialog box opens.
2. Select the tables of the database.
3. Click **Next**.

Triggers are automatically created for the selected tables.

When generation has been completed, the triggers appear under the **Trigger** folder available under each table. There are three trigger types:

- An update trigger (U_ followed by table name), which enables specification of the action to be carried out in case of modification of a line of the table that is part of the foreign key.
- A delete trigger (D_), which specifies the action to be carried out in case of deletion.
- An insert trigger (I_), which specifies the action to be carried out in case of insertion.

These triggers are only valid for a target DBMS. When you change DBMS, you must regenerate the triggers.

USING STORED PROCEDURES

HOPEX Database Builder allows you to create *stored procedures*.

A stored procedure combines a procedural language and SQL requests within a program. It enables execution of a particular task on a database. It is recorded in a database and can be called from a program external to the database or from a trigger.

A stored procedure can be implemented in two ways; either as a procedure or as a function.

 *A procedure is a set of instructions executing a sub-program.*

 *A function is a procedure returning a value on completion of execution.*

To create a procedure stored on a database:

1. Open the database properties dialog box.
2. Click the drop-down list then **Components**.
The **Stored Procedures** section displays the list of stored procedures.
3. Click the **New** button.
4. In the window that opens, specify the name of the procedure and its nature (Procedure or Function).
5. Click **OK**.
The stored procedure appears. Open its properties to define its code.

Example of stored procedure for Oracle

This is an example of a stored procedure updating the unit price of a part as a function of the part identifier:

```
CREATE PROCEDURE update_part_unitprice (part_id IN INTEGER,
new_price IN NUMBER)
IS
  Invalid_part EXCEPTION;
BEGIN
  -- HERE'S AN UPDATE STATEMENT TO UPDATE A DATABASE RECORD
  UPDATE sales.parts
    SET unit_price = new_price
    WHERE id = part-id;
  -- HERE'S AN ERROR-CHECKING STATEMENT
  If SQL%NOTFOUND THEN
    RAISE invalid_part;
  END IF;
EXCEPTION
  -- HERE'S AN ERROR-HANDLING ROUTINE
  WHEN invalid_part THEN
    raise_application_error(-20000, 'Invalid Part ID');
```

```
END update_part_unitprice;
```

ADDING PHYSICAL PROPERTIES TO DATABASE OBJECTS

When your database has been defined in a relational diagram, you can generate the corresponding SQL scripts for the different DBMSs.

The physical data navigation pane allows you to complete database physical modeling by specifying parameters specific to each DBMS and therefore to produce complete SQL scripts.

In **HOPEX**, you can also import physical parameters defined on reverse engineered objects. See: "[Physical Properties Reverse Engineering](#)", page 202.

You can adapt the same logical model to several DBMSs. It is not necessary to duplicate objects.

Target DBMSs

To define a target DBMS on a database:

1. Open the properties dialog box of the database concerned.
2. Select the **Characteristics** page.
3. Specify the **target DBMS** field in the corresponding field.

See also "[Importing a DBMS version](#)", page 59.

Creating Physical Properties

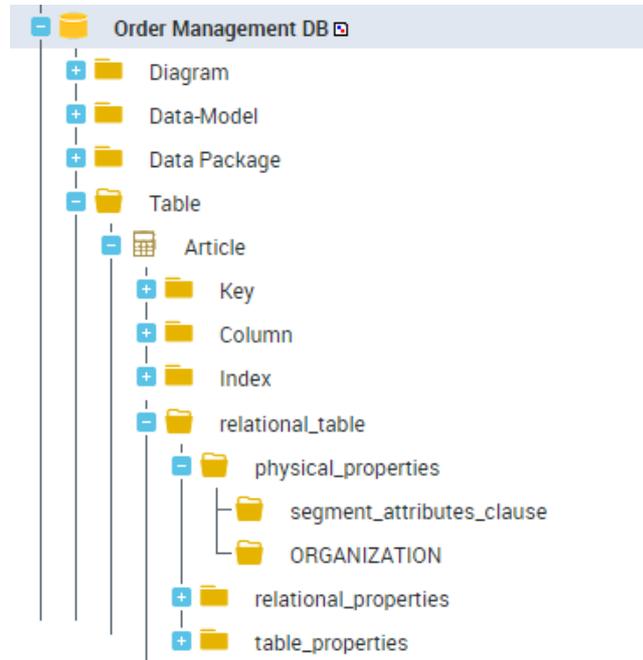
To create physical properties on objects of a database:

1. On the desktop, click the navigation menu, then **Physical data**.
2. Expand the **Database** folder and click **Database hierarchy**.
The list of repository databases appears in the edit area.
3. Expand the folder and the sub-folders of the databases concerned.

Parameters are presented in tree form, conforming to SQL grammar of the DBMS considered (refer to DBMS SQL documentation).

Two folder types are presented in the tree:

- Navigation folders.
- Parameter groups that you must instance.



Each parameter group, represented by an "SQL clause" object, has a properties page enabling value definition.

SQL clauses defined in this way are accessible just like repository standard objects. For example it is possible to query SQL objects that have a given parameter value.

By default, clauses cannot be reused from one object to another. It is however possible to define a clause for one object and connect it to other objects. In this case, any modification of the clause affects all objects that use it.

Objects containing physical parameters

Not all objects in **HOPEX** support physical parameters. These concern only:

- Data groups
- Tables
- Indexes
- Clusters

Creating a new clause

To define object parameters:

1. Right-click the corresponding parameter group and select **New > SQL clause**.

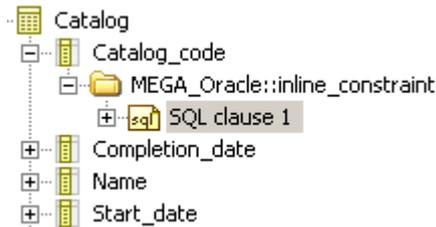
2. Open the properties window of the clause and specify the value of the parameter to be defined.

Connecting a clause

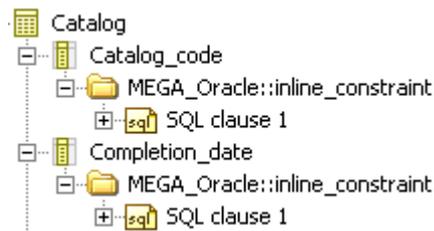
You can assign the same clause to several objects, on condition that you connect the correct clause type.

Consider the "Order Management" database with Oracle 9i as DBMS.

On the column "Code_catalogue", create "Clause 1" of type "inline_constraint".



You can connect "Clause 1" to another column. Being the same type of clause, this is copied on the new column with no problem.



On the other hand, if you connect "Clause 1" to an object of type different from that initially defined on "Clause 1" - for example "Storage_clause" - then "Clause 1" changes type to take that of the last element connected. In other words, "Clause 1" that was type "inline_constraint" takes type "storage_clause". This change is reflected on the start columns to which "Clause 1" was connected.

Naming clauses

Standard case

By default, the clause takes the name of the clause type to which it is attached. When you attach a clause to another type, the name automatically adapts.

Specific naming

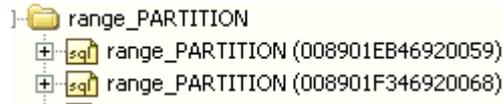
A specific name can be given to a clause. In this case, the clause name becomes static and will not be modified at change of clause type.

☛ You can return to dynamic mode by overloading the name empty.

Specific naming is essential when a clause is used in different contexts (generic clause).

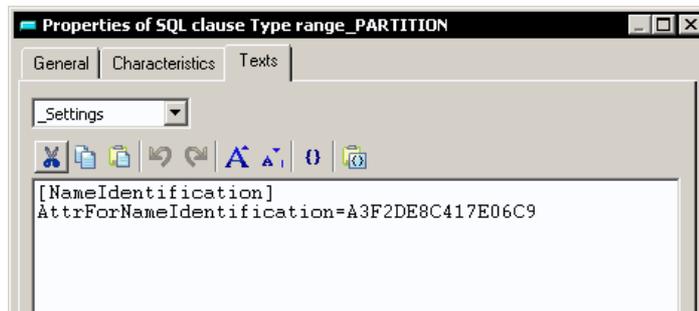
Multiple clauses

For a given level, several clauses can be attached to the same clause type. To distinguish different clauses, the clause name comprises the name of the clause type followed by its hexaIdAbs.



Naming from a property

It is possible to modify standard behavior by defining an automatic naming rule for an SQL clause type. This configuration is carried out at the clause type _settings property level. In the example below, the configuration on clause type "range_PARTITION" for Oracle 9i indicates that the name of SQL clauses of this type will be built from the value of the PARTITION property.



When the parameter has been executed, names of SQL clauses are automatically calculated from values of the PARTITION property.

Property	Value
Key compression	
PARTITION	P1
Value list	

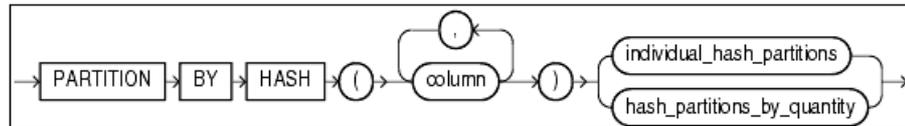


The name of SQL clauses is not taken into account in SQL generation. In the example provided, it is the value of the PARTITION property that feeds the generated SQL scripts.

Physical Model Customization Example

You can partition a table to simplify data access or to manage the information blocks differently.

Suppose you wish to partition the "Order Line" table of the "Order Management" database using the Oracle by hash method. This method enables dynamic calculation of table partitioning.



Hash partitioning instruction syntax

Check that the database has Oracle 9i as target DBMS.

- 1. Open its properties dialog box and click the **Characteristics** page. The DBMS name is indicated in the **Target DBMS** box.

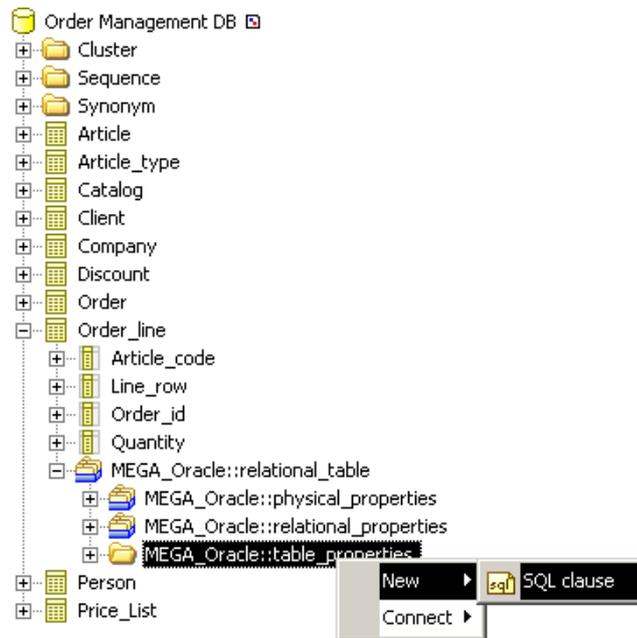
Display the physical properties of the "Order Management" database:

1. On the desktop, click the navigation menu, then **Physical data**.
2. Expand the **Database** folder and click **Database hierarchy**. The list of repository databases appears in the edit area.
3. Expand the folder and the sub-folders of the "Order Management" database to display the parameters linked to the Oracle grammar.

To partition the "Order line" table:

1. Expand the "Order_line" table.
2. Expand the "MEGA_Oracle::relational_table" parameter group.

3. Right-click the "MEGA_Oracle::table_properties" clause type and select **New > SQL clause**.



4. Name the clause "Order_line/Table_properties". It appears in the navigation tree.
5. Under this clause, expand the "MEGA_Oracle::table_partitioning_clauses" parameter group. It contains the different partitioning types that can be produced in Oracle.
6. On the "MEGA_Oracle::hash_partitioning" folder, create the clause "Order_line/hash_partitioning".
7. Open its properties page.
8. In the **PARTITION BY HASH** page, indicate the columns on which breakdown applies. To do this, connect the columns concerned by partitioning.
9. Close the properties dialog box.
10. Under the clause "Order_line/hash_partitioning" two clause types are available:
 - **individual_hash_partitions**: enables naming of each partition.
 - **hash_partitions_by_quantity**: enables definition of the number of partitions you wish to create.
11. Create the clause "Order_line/Hash_partition_by_quantity".
12. Open its properties page.
13. Select the **PARTITIONS** page.
14. In the **Hash partition quantity** field, indicate the number of partitions. These partitions are represented by data groups.
15. Under box **STORE IN**, connect the data groups.

To obtain the script corresponding to this partitioning, right-click the "Order line" table and select **Generate the code**.

```
SPOOL \_MEGASQL.LST;
PROMPT ----- ;
PROMPT  Compte-Rendu de génération ;
PROMPT ----- ;
/* -----
/* Begin of generation Oracle 9i for database Order_Management_DB the June 14, 2006 at 14: 0:49 */
/* -----
CREATE TABLE "Order_line"
(
  "Article_code" CHAR(6),
  "Line_row" NUMBER(7),
  "Order_id" CHAR(5),
  "Quantity" NUMBER(7),
  CONSTRAINT "FK_Order_line" PRIMARY KEY
  (
    "Line_row",
    "Order_id"
  )
)
PARTITION BY HASH ("Line_row","No_commande")
PARTITIONS 5 STORE IN ("tbs_1","tbs_2","tbs_3","tbs_4") TABLESPACE "SYSTEM";
/* -----
/* Foreign Key FK_Order */
/* -----
ALTER TABLE "Order_line"
ADD CONSTRAINT "FK_Order" FOREIGN KEY
(
  "Order_id"
) REFERENCES "Order";
/* -----
/* Foreign Key FK_Order_line */
/* -----
ALTER TABLE "Order_line"
ADD CONSTRAINT "FK_Order_line_" FOREIGN KEY
(
  "Article_code"
) REFERENCES "Article";
SPOOL OFF;
DEFINE _EDITOR = "notepad.exe";
EDIT \_MEGASQL.LST
EXIT;
/* -----
/* End of generation Oracle 9i for database Order_Management_DB the June 14, 2006 at 14: 0:51 */
/* -----
```

Generating the SQL File

When object customization has been completed, you can generate the corresponding script file to consult the results, without having to regenerate the entire database.

For example, to generate the SQL file of an index:

- Right-click the index and select **Generate the code**.

See also "Generating SQL scripts", page 163.

REVERSE ENGINEER TABLES



Reverse engineering enables you to take existing databases and create the corresponding tables and columns in the **HOPEX** repository.

Reverse engineering can be done from a previous database extraction. See ["ODBC Extraction Utility", page 205](#).

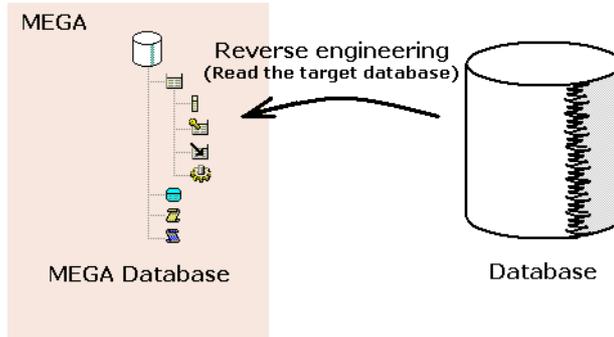
The tables and columns are integrated in a database where they can be easily maintained and documented.

The following points are covered here:

- ✓ ["Running Reverse Engineering", page 200](#)
- ✓ ["Physical Properties Reverse Engineering", page 202](#)

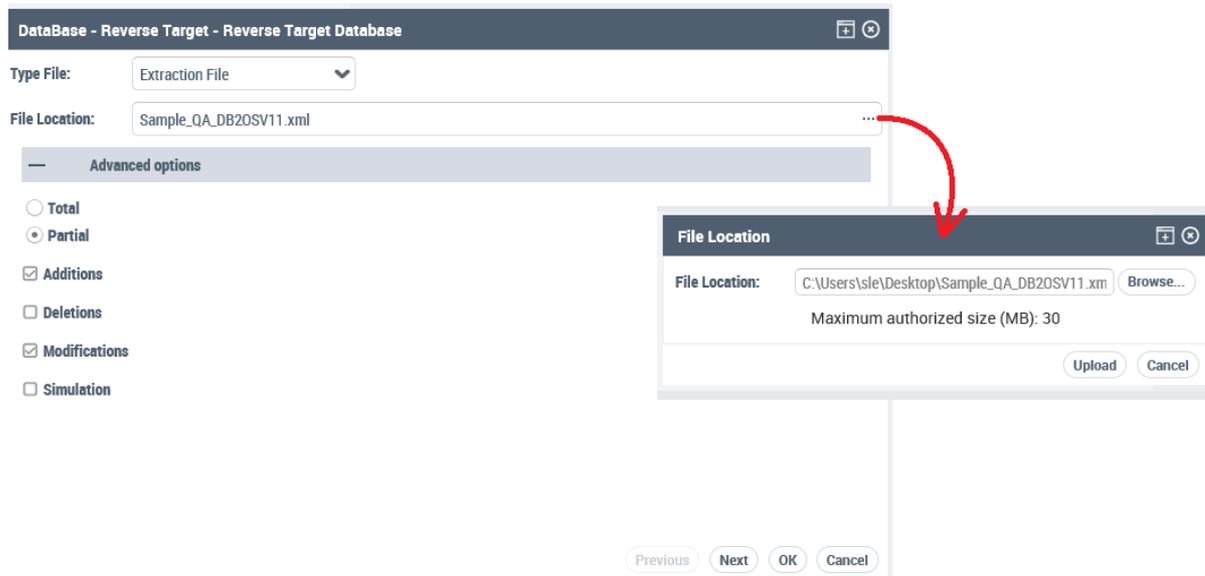
Tools presented in the pages that follow are available with "Advanced" access to the repository.

RUNNING REVERSE ENGINEERING



To run reverse engineering:

1. Right-click the database and select **Reverse target database**. The **Reverse a target database** dialog box opens.
2. Select the **Type** of data source: Extraction file obtained using the data extraction utility (see "[ODBC Extraction Utility](#)", page 205).
3. Indicate the location of the file.



4. Define the **Options**:
If the database has already been reverse engineered, it is possible to specify the extent of the **Reinitialization**, which can be:
 - **Total**: all existing tables are deleted.
 - **Partial** concerns only **Additions**, **Deletions** or **Modifications**.
The **Simulation** option is provided so you can simulate the operation and generate a report indicating the impact on the repository.
5. Click the **Next** button to run reverse engineering.
Messages inform you of progress.
6. On completion of processing, a report displays the details of the execution.
At the end of reverse engineering, the database tables and columns are created. However, the relational diagram must be created in order to view the database in a graphical model.

☺ *If non-standard datatypes were created in the DBMS, they must be added to the configuration if they are to be recognized by HOPEX..*

PHYSICAL PROPERTIES REVERSE ENGINEERING

Reverse engineering also enables creation in **HOPEX** of physical properties on objects of a database.

Physical properties are parameters enabling expression, for a relational object (table, index, etc.), of the way in which information will be stored within a database. These parameters are specific to each DBMS and can evolve depending on versions of the same DBMS.

See also ["Adding Physical Properties to Database Objects"](#), page 191.

Default Values

Certain DBMS properties are automatically reversed in **HOPEX** even if they have not been explicitly specified.

So as not to recover these values by default, **HOPEX** provides for each DBMS and for each DBMS version a generic clause that contains the list of these default values and treating them specifically.

At reverse engineering, DBMS properties with values equal to values defined in the generic clause are not imported.

You can activate the generic clause by importing into **HOPEX** the .mol file associated with each DBMS in the Mega_Std folder of your installation.

Eliminating Redundant and Transverse Values

At reverse engineering of objects in **HOPEX**, certain physical properties that have not been clearly determined are automatically regenerated by the DBMS via an inheritance mechanism.

With Oracle for example, the value of property PCTFREE in a table, if it has not been specified, is directly inherited from that of its attached tablespace. Such a value is called transverse, since it is derived from an inheritance between two distinct object types. A value is said to be redundant if inheritance is derived from objects of the same type.

At reverse engineering, **HOPEX** does not recover transverse and redundant values.

Only the management of redundant values can be customized.

Specific Cases

Physical properties of tablespaces

In certain cases, reverse engineering of object physical properties requires an ODBC connection with DBMS Administrator rights. For example with Oracle, reverse engineering of the physical properties of tablespaces requires that you use a "System" account.

Clusters Reverse Engineering

Reverse engineering of a cluster in Oracle is carried out correctly if the connecting user verifies one of the two following conditions:

- The user is owner of the cluster
- The user has Administrator profile

If the cluster is not accessible, it is not reversed.

When the user sees the cluster but is neither owner nor administrator, the cluster is reversed, but the link between columns of the cluster and columns of the attached tables is not reversed in **HOPEX**.

☛ *From a technical viewpoint, for an administrator user, reverse engineering depends on the view oracle sys.all_clu_columns (relationship between cluster columns and table columns). This view enables reversing only of information relating to objects of which the user is owner.*



ODBC EXTRACTION UTILITY



The ODBC extraction utility is used to extract the description of a database accessed with the ODBC protocol. This description, obtained in structured format, can then be used for reverse engineering purposes in **HOPEX** or for generation in modification mode.

The following points are covered here:

- ✓ ["Extracting the Description of a Database", page 206](#)
- ✓ ["Customizing ODBC Extraction", page 210](#)
- ✓ ["Select Clause Formats", page 212](#)

EXTRACTING THE DESCRIPTION OF A DATABASE

The extraction utility is designed to run on a workstation that does not have **HOPEX** installed. You can then transfer the results to a **HOPEX** workstation for reverse engineering. It can also be used for generation in modification mode.

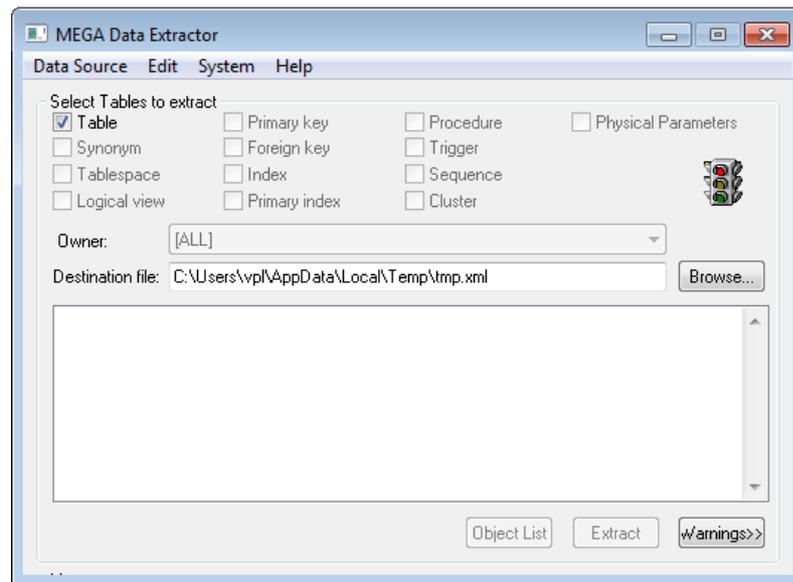
In order to extract the data, the workstation must be connected to the database using ODBC protocol (see the installation documentation for this product). The database driver must have a conformance level of 1 or higher.

To install this utility, simply copy to a directory the files mgwdbx32.exe and mg_dbex.dll. If it does not exist, the file ODWDBEX.INI must be created.

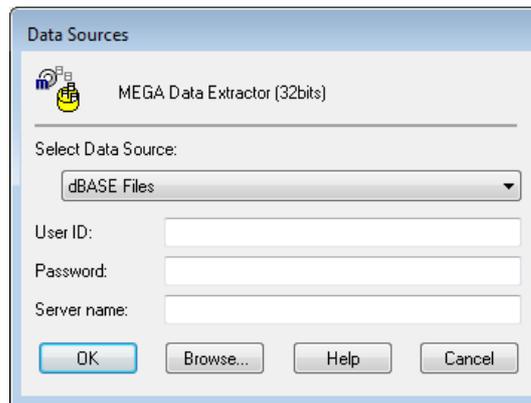
The extraction utility can be installed independently of **HOPEX Database Builder**.

To extract the description of a database:

1. In the **HOPEX** installation folder, under the "Utilities\MEGA Data Extractor" folder, run the mgwdbx32.exe extraction utility.



2. Connect to a data source by clicking **Data Source > Connect**. The **Data Sources** dialog box opens.



The drop-down list shows the ODBC connections.

☛ (The list is empty if these connections are not defined or could not be established.)

3. Use the **Browse** button to access the ODBC manager to define a new data source or install a new driver.
4. Depending on the connection, you may need to specify a **User ID**, a **Password**, and a **Server Name**. If other parameters are required by the ODBC manager, you will be prompted for them when the connection is established.
5. Confirm the information entered by clicking **OK**.

A green light indicates that the connection has been established. Otherwise a message appears if the connection has not been correctly established. In this case, check the data source definition.

Once the connection has been established, select the desired extraction options.

☛ If some of the options remain disabled, this is because the driver does not support them.

☺ To obtain information on the ODBC protocol used, select ODBC information in the **System** menu.

6. Select the elements to be extracted in addition to the tables and columns. By default, all of these elements are selected. All the accessible tables are displayed, whether or not you are the owner. **Synonym** tables can also appear if you select the corresponding check box.

📖 A synonym is an alternative name given to an object (table, view, stored procedure, synonym and sequence). A synonym can be defined to indicate an object in another database.

To view only those tables belonging to a specific **Owner**, select the appropriate owner from the drop-down list. It may take a few seconds for the list of owners and their tables to appear. Table extraction takes a few minutes.

The following elements are included in the extraction:

- Primary keys (**Primary keys**).
- Foreign keys (**Foreign keys**).
- Index (**Index**): these are indexes that do not use primary keys.
- Primary index (**Primary index**): these are indexes that do not use primary keys.

 **Not all drivers support the ODBC primitives used to extract these elements; if this is your case, a message will indicate this in the report file. In addition, some DBMS do not handle the corresponding concepts, which are then ignored.**

The **Destination file** field specifies the name and path of the extraction file; use the **Browse** button to specify its location.

7. After selecting the extraction options, click the **Extract** button to begin the extraction.

A message reports the number of tables extracted. You can select the **Warnings** button to view the report file.

 *During the extraction, the **Extract** button becomes **Cancel** and can be used to stop the extraction.*

You can view the list of accessible tables by clicking the **List Tables** button, and then select specific tables from the list for extraction (all tables are selected by default).

8. On completion of extraction, select **Edit > Report file** to consult the report, or **Edit > Extraction file** to consult the result.

The results file can then be transferred to a workstation that has **HOPEX Database Builder** where reverse engineering can be performed (see "[Reverse engineer tables](#)", page 199). It contains a database description in the form of **HOPEX** objects.

When the extraction operation has been completed:

- 】 Select **Disconnect** in the **Data Source** menu to disconnect from the data source.
- 】 Reconnect to another data source if needed, or exit the dialog box by selecting **Exit** in the **Data Source** menu.

Extraction Report File

The file that reports on the table extraction is created by the ODBC extraction utility. It is called <FIC>_CRD.TXT where <FIC> represents the first three characters of the name of the results file.

It contains a list of the tables that were reread.

Example:

```
=====
Data Source Extracting: DATASOURCE
=====
Table: OWNER.TABLENAME1
Table: OWNER.TABLENAME2
(cont.)
=====
End of extraction
=====
```

Extraction Results File

The extraction results file contains the description of the tables and columns that results from the read. This file has extension ".xml".

Example of extraction file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <XMDB Name="Xmdb" Version="2.0">
- <DATABASE Name="Database">
- <TABLE Name="TABLE.AURORA036IOP036SYSTEM036PROPERTIES" DBBName="AURORA$IOP$SYSTEM$PROPERTIES">
- <TBLCOL Name="TBLCOL.AURORA036IOP036SYSTEM036PROPERTIES.KEY" Order="1" DBBName="KEY" Decimale="0"
  Length="200" NotNull="N">
  <COLDATATYPE Name="COLDATATYPE.AURORA036IOP036SYSTEM036PROPERTIES.KEY"
    DataType="SQL_VARCHAR" TypeTech="SQL_Oracle8VARCHAR2" />
  </TBLCOL>
- <TBLCOL Name="TBLCOL.AURORA036IOP036SYSTEM036PROPERTIES.VALUE" Order="2" DBBName="VALUE"
  Decimale="0" Length="1024" NotNull="N">
  <COLDATATYPE Name="COLDATATYPE.AURORA036IOP036SYSTEM036PROPERTIES.VALUE"
    DataType="SQL_VARCHAR" TypeTech="SQL_Oracle8VARCHAR2" />
  </TBLCOL>
- <INDEX Name="INDEX.AURORA036IOP036SYSTEM036PROPERTIES.KEY095C" DBBName="KEY_C" Type="U">
  <IDXCOL Name="IDXCOL.AURORA036IOP036SYSTEM036PROPERTIES.KEY095C.KEY"
    Reference="TBLCOL.AURORA036IOP036SYSTEM036PROPERTIES.KEY" />
  </INDEX>
</TABLE>
- <TABLE Name="TABLE.JAVA036HTTP036DEPLOYMENT036DIGEST036"
  DBBName="JAVA$HTTP$DEPLOYMENT$DIGEST$">
- <TBLCOL Name="TBLCOL.JAVA036HTTP036DEPLOYMENT036DIGEST036.USERNAME" Order="1"
  DBBName="USERNAME" Decimale="0" Length="30" NotNull="M">
  <COLDATATYPE Name="COLDATATYPE.JAVA036HTTP036DEPLOYMENT036DIGEST036.USERNAME"
    DataType="SQL_VARCHAR" TypeTech="SQL_Oracle8VARCHAR2" />
  </TBLCOL>
- <TBLCOL Name="TBLCOL.JAVA036HTTP036DEPLOYMENT036DIGEST036.IS095DOC095ROOT" Order="4"
  DBBName="IS_DOC_ROOT" Decimale="0" Length="1" NotNull="N">
  <COLDATATYPE Name="COLDATATYPE.JAVA036HTTP036DEPLOYMENT036DIGEST036.IS095DOC095ROOT"
    DataType="SQL_DECIMAL" TypeTech="SQL_Oracle8DECIMAL" />
  </TBLCOL>
</TABLE>
</DATABASE>
</XMDB>
```

CUSTOMIZING ODBC EXTRACTION

When the extraction is incomplete or does not correspond to your needs, you can customize the extraction with the Odwdbex.ini file. This configuration depends on the ODBC driver you are using.

You can customize the extraction in a number of different ways by using:

- ODBC standard APIs available for the main concepts (Table, Column, Key, Index, etc.)
- **HOPEX** queries delivered as a replacement for ODBC standard APIs
- customized queries.

By default, ODBC standard APIs are used for the main concepts and **HOPEX** queries for the other concepts. For some ODBC drivers, **HOPEX** queries are used for the main concepts as the result obtained by the ODBC standard is incomplete.

Using the Odwdbex.ini file and customized queries

To customize extraction:

1. Contact your data administrator to obtain the customized queries corresponding to your ODBC driver used to select objects (Eg: primary keys, foreign keys, sequences, etc).
2. In the "All users" folder in Windows, create a file named Odwdbex.ini (example: C:\Documents and Settings\All Users\ApplicationData\Mega\Odwdbex.ini).
3. Edit the file and add the queries for the concepts whose behavior you want to manage. The concepts that are not cited here remain unchanged.

```
[<DBMS Name>]
PRIMARY KEYS="Custom query"
FOREIGN KEYS="Custom query"
TBLCOLUMNS="Custom query"
...
```

The <DBMS Name> value depends on the ODBC utility. To obtain the appropriate value:

1. Run the **HOPEX** extraction tool (mgwdbx32.exe).
2. In the **Data Source** menu, select the data source.
3. Then click **System > ODBC Information**.
4. Read the "DBMS Name".

You can edit the Odwdbex.ini file by selecting **System > Edit Odwdbex.ini** in the **HOPEX** extraction tool. Check that the file is archived.

For more the format of queries, see ["Select Clause Formats"](#), page 212.

Using ODBC standard APIs

To force the use of ODBC APIs:

1. Edit the Odwdbex.ini file.
2. At the level of each concept concerned, modify the extraction strategy using the keyword: USE_DRIVER_ODBC.

Example in the ODWDBEX.INI file:

```
[<DBMS Name>]
```

```
INDEXES=USE_DRIVER_ODBC
```

SELECT CLAUSE FORMATS

☛ It is important to use the indicated syntax and in particular not to omit any of the "1"s. Note that the clauses must fit on a single line in the ODWDBEX.INI file.

Primary Keys

```
SELECT
1,
TABLE_OWNER,
TABLE_NAME,
COLUMN_NAME,
KEY_SEQUENCE,
PK_NAME
FROM ...
WHERE ...
```

- TABLE_OWNER: owner of the primary key table
- TABLE_NAME: name of the primary key table
- COLUMN_NAME: name of the primary key column
- KEY_SEQUENCE: Number of the column in the key sequence (starts at 1)
- PK_NAME: Name of the primary key; "1" if this name is not supported by the DBMS.

Foreign Keys

```
SELECT
  1,
  PKTABLE_OWNER,
  PKTABLE_NAME,
  1,
  1,
  FKTABLE_OWNER,
  FKTABLE_NAME,
  FKCOLUMN_NAME,
  KEY_SEQ,
  UPDATE_RULE,
  DELETE_RULE,
  FK_NAME
FROM ...
WHERE ...
```

- PKTABLE_OWNER: name of the owner of the primary key table (reference table)
- PKTABLE_NAME: name of the primary key table
- FKTABLE_OWNER: name of the owner of the foreign key table
- FKTABLE_NAME: name of the foreign key table
- FKCOLUMN_NAME: name of the foreign key column
- KEY_SEQ: number of the column in the key sequence (starts at 1)
- UPDATE_RULE : R: Restrict, C: Cascade
- DELETE_RULE : R: Restrict, C: Cascade
- FK_NAME: name of the foreign key; "1" if this name is not supported by the DBMS.

Indexes

```
SELECT
  1,
  TABLE_OWNER,
  TABLE_NAME,
  NON_UNIQUE,
  1,
  INDEX_NAME,
  TYPE,
  SEQ_IN_INDEX,
  COLUMN_NAME,
  COLLATION
FROM ...
WHERE ...
```

- TABLE_OWNER: name of the owner of the table concerned by the statistic or the index
- TABLE_NAME: name of the index table
- NON_UNIQUE: the indexes must have a unique value
- INDEX_NAME: name of the index
- TYPE : Index type
- SEQ_IN_INDEX: number of the column in the key sequence (starts at 1)
- COLUMN_NAME: name of the column
- COLLATION: column sort; "A" increasing order, "D" decreasing order

Columns

```
SELECT
1,
COLUMN_OWNER,
TABLE_NAME,
COLUMN_NAME,
DataType ODBC,
DataType Name,
Detail,
Length,
Scale,
1,
NULLABLE,
COMMENT,
DEFAULT_VALUE,
1,
1,
1,
Order
WHERE [Joint between <MEGA:OWNER><MEGA:OBJECT_NAME>]
```

- <MEGA:OWNER> is replaced by the user, the Schema or "".
- <MEGA:OBJECT_NAME>] is replaced by the name of the table.
- COLUMN_OWNER: name of the column, string with 128 characters.
- TABLE_NAME: name of the table, string with 128 characters.
- DataType ODBC: data type in the form of an integer. This value is the value of ODBC data types therefore comprised of the following:
 - # -1 (SQL_LONGVARCHAR)
 - # -2 (SQL_BINARY)
 - # -3 (SQL_VARBINARY)
 - # -4 (SQL_LONGVARBINARY)
 - # -5 (SQL_BIGINT)
 - # -6 (SQL_TINYINT)
 - # -7 (SQL_BIT)
 - # 0 (SQL_UNKNOWN_TYPE)
 - # 1 (SQL_CHAR)
 - # 2 (SQL_NUMERIC)
 - # 3 (SQL_DECIMAL)
 - # 4 (SQL_INTEGER)

```
# 5 (SQL_SMALLINT)
# 6 (SQL_FLOAT)
# 7 (SQL_REAL)
# 8 (SQL_DOUBLE)
# 9 (SQL_DATE)
# 10 (SQL_TIME)
# 11 (SQL_TIMESTAMP)
# 12 (SQL_VARCHAR)
```

- **DataType Name:** name of the data type, string of 128 characters. It is built as follows: "SQL_<DbmsName><String>"
- **Precision:** length in MEGA if "Length" is empty.
- **Length:** length in MEGA if greater than 0.
- **Scale:** integer
- **NULLABLE:** integer specifying if the column can be NULL. ODBC values possible: 0 (SQL_NO_NULLS), 1 (SQL_NULLABLE) or 3 (SQL_NULL_WITH_DEFAULT).
- **COMMENT:** column comments, string with 1257 characters.
- **DEFAULT_VALUE:** default value of the column, string with 1257 characters.

PIVOT TYPES AND DATATYPES CORRESPONDENCE TABLES



The following tables show correspondence between pivot types and the different supported DBMSs and their versions.

- ✓ "DB2 OS/390 Version 5", page 224
- ✓ "DB2 OS/390 Version 7", page 227
- ✓ "DB2 for z/OS Version 8", page 230
- ✓ "DB2 Universal Database Version 5", page 233
- ✓ "DB2 Universal Database Version 7", page 236
- ✓ "DB2 Universal Database Version 8", page 239
- ✓ "DB2 Universal Database Version 9", page 242
- ✓ "DB2 Version 9 For OS", page 245
- ✓ "Informix Dynamic Server 7.3", page 249
- ✓ "Ingres II 2.0", page 251
- ✓ "MySQL 4.1", page 253
- ✓ "MySQL 5.0", page 264
- ✓ "Oracle 10", page 275
- ✓ "Oracle 11", page 278
- ✓ "Oracle 8", page 281
- ✓ "Oracle 9i", page 283
- ✓ "PostgreSQL9.3", page 286
- ✓ "SQL ANSI/ISO 9075:1992", page 289
- ✓ "SQL Server 2000", page 291
- ✓ "SQL Server 2005", page 294
- ✓ "SQL Server 2008", page 297
- ✓ "SQL Server 7", page 300
- ✓ "Sybase Adaptive Server 11", page 302
- ✓ "Sybase Adaptive Server 12.5", page 305
- ✓ "Teradata Database", page 308

DB2 OS/390 VERSION 5

Pivot --> Datatype (DB2 OS/390 Version 5)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L ø)	CHAR
	Not Unicode and 0<L<256	CHAR(@L)
	Not Unicode and (L=256 or L ø)	GRAPHIC(@L)
	Not Unicode and L>255	VARCHAR(@L)
	Not Unicode and L>255	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L ø	DECIMAL
	L>0 and D ø	DECIMAL(@L)
	L>0 and D not ø	DECIMAL(@L,@D)
P-Double		DOUBLE
P-Float	L=0 or L ø	FLOAT
	L<>0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		INTEGER
P-Long Real		REAL

Pivot --> Datatype (DB2 OS/390 Version 5)

Pivot	Condition	Datatype
P-Multimedia		LONG VARCHAR FOR BIT DATA
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L,@D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	L=5 or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 OS/390 Version 5)

Datatype	Condition	Pivot
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
DATE		P-Date
DECIMAL		P-Decimal

Datatype --> Pivot (DB2 OS/390 Version 5)

Datatype	Condition	Pivot
DECIMAL(L)		P-Decimal
DECIMAL(L,D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
LONG VARCHAR FOR BIT DATA		P-Multimedia
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
TIMESTAMP		P-Timestamp
VARCHAR		P-Varchar
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 OS/390 VERSION 7

Pivot --> Datatype (DB2 OS/390 Version 7)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L \emptyset)	CHAR
	Not Unicode and 0<L<256	CHAR(@L)
	Not Unicode and (L=256 or L \emptyset)	GRAPHIC(@L)
	Not Unicode and L>255	VARCHAR(@L)
	Not Unicode and L>255	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L, @D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L \emptyset	DECIMAL
	L>0 and D \emptyset	DECIMAL(@L)
	L>0 and D not \emptyset	DECIMAL(@L, @D)
P-Double		DOUBLE
P-Float	L=0 or L \emptyset	FLOAT
	L<>0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		INTEGER
P-Long Real		REAL

Pivot --> Datatype (DB2 OS/390 Version 7)

Pivot	Condition	Datatype
P-Multimedia		LONG VARCHAR FOR BIT DATA
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L, @D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	(L<5 and D \emptyset) or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 OS/390 Version 7)

Datatype	Condition	Pivot
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
DATE		P-Date
DECIMAL		P-Decimal

Datatype --> Pivot (DB2 OS/390 Version 7)

Datatype	Condition	Pivot
DECIMAL(L)		P-Decimal
DECIMAL(L, D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
LONG VARCHAR FOR BIT DATA		P-Multimedia
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
TIMESTAMP		P-Datetime
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 FOR z/OS VERSION 8

Pivot --> Datatype (DB2 for z/OS Version 8)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L \emptyset)	CHAR
	Not Unicode and $0 < L < 256$	CHAR(@L)
	Not Unicode and (L=256 or L \emptyset)	GRAPHIC(@L)
	Not Unicode and $L > 255$	VARCHAR(@L)
	Not Unicode and $L > 255$	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L, @D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L \emptyset	DECIMAL
	L>0 and D \emptyset	DECIMAL(@L)
	L>0 and D not \emptyset	DECIMAL(@L, @D)
P-Double		DOUBLE
P-Float	L=0 or L \emptyset	FLOAT
	L \diamond 0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		INTEGER
P-Long Real		REAL

Pivot --> Datatype (DB2 for z/OS Version 8)

Pivot	Condition	Datatype
P-Multimedia		LONG VARCHAR FOR BIT DATA
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L, @D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	(L<5 and D \emptyset) or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 for z/OS Version 8)

Datatype	Condition	Pivot
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
DATE		P-Date
DECIMAL		P-Decimal

Datatype --> Pivot (DB2 for z/OS Version 8)

Datatype	Condition	Pivot
DECIMAL(L)		P-Decimal
DECIMAL(L, D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
LONG VARCHAR FOR BIT DATA		P-Multimedia
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
TIMESTAMP		P-Datetime
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 UNIVERSAL DATABASE VERSION 5

Pivot --> Datatype (DB2 Universal Database Version 5)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L \emptyset)	CHAR
	Not Unicode and 0<L<255	CHAR(@L)
	Not Unicode and (L=256 or L \emptyset)	GRAPHIC(@L)
	Not Unicode and L>254	VARCHAR(@L)
	Not Unicode and L>255	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L \emptyset	DECIMAL
	L>0 and D \emptyset	DECIMAL(@L)
	L>0 and D not \emptyset	DECIMAL(@L,@D)
P-Double		DOUBLE
P-Float	L=0 or L \emptyset	FLOAT
	L<>0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		BIGINT
P-Long Real		REAL

Pivot --> Datatype (DB2 Universal Database Version 5)

Pivot	Condition	Datatype
P-Multimedia		BLOB(@L)
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L,@D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	(L<5 and D not \emptyset) or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 Universal Database Version 5)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
BLOB(L)		P-Multimedia
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
DATE		P-Date

Datatype --> Pivot (DB2 Universal Database Version 5)

Datatype	Condition	Pivot
DECIMAL		P-Decimal
DECIMAL(L)		P-Decimal
DECIMAL(L,D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
TIMESTAMP		P-Timestamp
VARCHAR		P-Varchar
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 UNIVERSAL DATABASE VERSION 7

Pivot --> Datatype (DB2 Universal Database Version 7)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L ø)	CHAR
	Not Unicode and 0<L<255	CHAR(@L)
	Not Unicode and (L=256 or L ø)	GRAPHIC(@L)
	Not Unicode and L>254	VARCHAR(@L)
	Not Unicode and L>255	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L ø	DECIMAL
	L>0 and D ø	DECIMAL(@L)
	L>0 and L not ø	DECIMAL(@L,@D)
P-Double		DOUBLE
P-Float	L=0 or L ø	FLOAT
	L<>0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		BIGINT
P-Long Real		REAL

Pivot --> Datatype (DB2 Universal Database Version 7)

Pivot	Condition	Datatype
P-Multimedia		BLOB(@L)
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L,@D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	L=5 or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 Universal Database Version 7)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
BLOB(L)		P-Multimedia
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
DATE		P-Date

Datatype --> Pivot (DB2 Universal Database Version 7)

Datatype	Condition	Pivot
DECIMAL		P-Decimal
DECIMAL(L)		P-Decimal
DECIMAL(L,D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
TIMESTAMP		P-Timestamp
VARCHAR		P-Varchar
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 UNIVERSAL DATABASE VERSION 8

Pivot --> Datatype (DB2 Universal Database Version 8)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L \emptyset)	CHAR
	Not Unicode and $0 < L < 255$	CHAR(@L)
	Not Unicode and (L=256 or L \emptyset)	GRAPHIC(@L)
	Not Unicode and $L > 254$	VARCHAR(@L)
	Not Unicode and $L > 255$	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	$L = 0$ or $L \emptyset$	DECIMAL
	$L > 0$ and $D \emptyset$	DECIMAL(@L)
	$L > 0$ and L not \emptyset	DECIMAL(@L,@D)
P-Double		DOUBLE
P-Float	$L = 0$ or $L \emptyset$	FLOAT
	$L \diamond 0$	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		BIGINT
P-Long Real		REAL

Pivot --> Datatype (DB2 Universal Database Version 8)

Pivot	Condition	Datatype
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L,@D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	L=5 or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 Universal Database Version 8)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
DATE		P-Date
DECIMAL		P-Decimal
DECIMAL(L)		P-Decimal
DECIMAL(L,D)		P-Decimal

Datatype --> Pivot (DB2 Universal Database Version 8)

Datatype	Condition	Pivot
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
REAL		P-Real
SMALLINT		P-Smallint
SMALLINT		P-Tinyint
TIME		P-Time
TIMESTAMP		P-Timestamp
VARCHAR		P-Varchar
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 UNIVERSAL DATABASE VERSION 9

Pivot --> Datatype (DB2 Universal Database Version 9)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L \emptyset)	CHAR
	Not Unicode and 0<L<255	CHAR(@L)
	Not Unicode and (L=256 or L \emptyset)	GRAPHIC(@L)
	Not Unicode and L>254	VARCHAR(@L)
	Not Unicode and L>255	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L \emptyset	DECIMAL
	L>0 and D \emptyset	DECIMAL(@L)
	L>0 and L not \emptyset	DECIMAL(@L,@D)
P-Double		DOUBLE
P-Float	L=0 or L \emptyset	FLOAT
	L<>0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		BIGINT
P-Long Real		REAL

Pivot --> Datatype (DB2 Universal Database Version 9)

Pivot	Condition	Datatype
P-Multimedia		BLOB(@L)
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L,@D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	L=5 or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary		VARCHAR(@L) FOR BIT DATA
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 Universal Database Version 9)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
BLOB(L)		P-Multimedia
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
CLOB(L)		CLOB

Datatype --> Pivot (DB2 Universal Database Version 9)

Datatype	Condition	Pivot
DATE		P-Date
DECIMAL		P-Decimal
DECIMAL(L)		P-Decimal
DECIMAL(L,D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
REAL		P-Real
SMALLINT		P-Smallint
SMALLINT		P-Tinyint
TIME		P-Time
TIMESTAMP		P-Timestamp
VARCHAR		P-Varchar
VARCHAR(L)		P-Varchar
VARCHAR(L) FOR BIT DATA		P-Varbinary

DB2 VERSION 9 FOR OS

Pivot --> Datatype (DB2 Version 9 For OS)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		CHAR(@L) FOR BIT DATA
P-Boolean		CHAR(@L) FOR BIT DATA
P-Byte		CHAR (1) FOR BIT DATA
P-Character	Not Unicode and (L=0 or L \emptyset)	CHAR
	Not Unicode and 0<L<256	CHAR(@L)
	Not Unicode and (L=256 or L \emptyset)	GRAPHIC(@L)
	Not Unicode and L>255	VARCHAR(@L)
	Not Unicode and L>255	VARGRAPHIC(@L)
P-Currency		DECIMAL(@L, @D)
P-Date		DATE
P-Datetime		TIMESTAMP
P-Decimal	L=0 or L \emptyset	DECIMAL
	L>0 and D \emptyset	DECIMAL(@L)
	L>0 and D not \emptyset	DECIMAL(@L, @D)
P-Double		DOUBLE
P-Float	L=0 or L \emptyset	FLOAT
	L<>0	FLOAT(@L)
P-Integer		INTEGER
P-Long Integer		INTEGER
P-Long Real		REAL

Pivot --> Datatype (DB2 Version 9 For OS)

Pivot	Condition	Datatype
P-Multimedia		BLOB
		BLOB(@L)
		CLOB
		CLOB FOR MIXED DATA
		CLOB(@L)
		CLOB(@L) FOR MIXED DATA
		LONG VARCHAR FOR BIT DATA
P-Numeric	L not \emptyset and D not \emptyset	DECIMAL(@L, @D)
	L>9 and D \emptyset	FLOAT(@L)
	4<L<10 and D \emptyset	INTEGER
	(L<5 and D \emptyset) or L \emptyset	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		LONG VARCHAR
P-Text		LONG VARCHAR
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyint		SMALLINT
P-Varbinary	L <=1024	VARCHAR(@L) FOR BIT DATA
	L>1024	XML
P-Varchar	Not Unicode and (L=0 or L \emptyset)	VARCHAR
	Not Unicode and L<>0	VARCHAR(@L)
	Unicode	VARGRAPHIC(@L)

Datatype --> Pivot (DB2 Version 9 For OS)

Datatype	Condition	Pivot
BLOB		P-Multimedia
BLOB(L)		P-Multimedia
CHAR		P-Character
CHAR (1) FOR BIT DATA		P-Byte
CHAR(L)		P-Character
CHAR(L) FOR BIT DATA		P-Boolean
CLOB		P-Multimedia
CLOB FOR MIXED DATA		P-Multimedia
CLOB(L)		P-Multimedia
CLOB(L) FOR MIXED DATA		P-Multimedia
DATE		P-Date
DECIMAL		P-Decimal
DECIMAL(L)		P-Decimal
DECIMAL(L, D)		P-Decimal
DOUBLE		P-Double
FLOAT		P-Float
FLOAT(L)		P-Float
INTEGER		P-Integer
LONG VARCHAR		P-Text
LONG VARCHAR FOR BIT DATA		P-Multimedia
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
TIMESTAMP		P-Datetime
VARCHAR(L)		P-Varchar

Datatype --> Pivot (DB2 Version 9 For OS)

Datatype	Condition	Pivot
VARCHAR(L) FOR BIT DATA		P-Varbinary
XML		P-Varbinary

INFORMIX DYNAMIC SERVER 7.3

Pivot --> Datatype (Informix Dynamic Server 7.3)

Pivot	Condition	Datatype
P-AutoIdentifier		SERIAL
P-Binary		BYTE
P-Boolean		SMALLINT
P-Byte		BYTE
P-Character		CHAR(@L)
P-Currency		MONEY(@L,@D)
P-Date		DATE
P-Datetime		DATETIME
P-Decimal	D \emptyset	DECIMAL(@L)
	D not \emptyset	DECIMAL(@L,@D)
P-Double		FLOAT
P-Float	L=0 or L \emptyset	FLOAT
	L < 0	SMALLFLOAT(@L)
P-Integer		INTEGER
P-Long Integer		INTEGER
P-Long Real		FLOAT
P-Multimedia		BYTE
P-Numeric	L > 9	DECIMAL(@L)
	4 < L < 10	INTEGER
	L = 5 or L \emptyset	SMALLINT
P-Real		FLOAT
P-Smallint		SMALLINT
P-String		VARCHAR(@L)
P-Text		VARCHAR(@L)

Pivot --> Datatype (Informix Dynamic Server 7.3)

Pivot	Condition	Datatype
P-Time		DATETIME
P-Timestamp		SERIAL
P-Tinyint		SMALLINT
P-Varchar	D ø	VARCHAR(@L)
	D not ø	VARCHAR(@L,@D)

Datatype --> Pivot (Informix Dynamic Server 7.3)

Datatype	Condition	Pivot
BYTE		P-Multimedia
CHAR(L)		P-Character
DATE		P-Date
DATETIME		P-Datetime
DECIMAL(L)		P-Decimal
DECIMAL(L,D)		P-Decimal
FLOAT		P-Float
INTEGER		P-Integer
MONEY(L,D)		P-Currency
SERIAL		P-Timestamp
SMALLFLOAT(L)		P-Float
SMALLINT		P-Boolean
VARCHAR(L)		P-Varchar
VARCHAR(L,D)		P-Varchar

INGRES II 2.0

Pivot --> Datatype (Ingres II 2.0)

Pivot	Condition	Datatype
P-AutoIdentifier		integer
P-Binary		long byte(@L)
P-Boolean		integer1
P-Byte		byte(@L)
P-Character		char(@L)
P-Currency		money(@L,@D)
P-Date		date
P-Datetime		date
P-Decimal		decimal(@L,@D)
P-Double		integer1
P-Float		float4
P-Integer		integer
P-Long Integer		integer
P-Long Real		float
P-Multimedia	L=2001 or L \emptyset	byte(@L)
	L>2000	long byte(@L)
P-Numeric	L>9	float
	4<L<10	integer
	L=3 or L \emptyset	integer1
	2<L<5	smallint
P-Real		float
P-Smallint		smallint
P-String		long varchar(@L)
P-Text		text(@L)

Pivot --> Datatype (Ingres II 2.0)

Pivot	Condition	Datatype
P-Time		date
P-Timestamp		date
P-Tinyint		integer1
P-Varbinary		byte varying(@L)
P-Varchar		varchar(@L)

Datatype --> Pivot (Ingres II 2.0)

Datatype	Condition	Pivot
byte varying(L)		P-Varbinary
byte(L)		P-Multimedia
char(L)		P-Character
date		P-Datetime
decimal(L,D)		P-Decimal
float		P-Real
float4		P-Float
integer		P-Integer
integer1		P-Tinyint
long byte(L)		P-Binary
long varchar(L)		P-String
money(L,D)		P-Currency
smallint		P-Smallint
text(L)		P-Text
varchar(L)		P-Varchar

MySQL 4.1

Pivot --> Datatype (MySQL 4.1)

Pivot	Condition	Datatype
	L>0 and D not \emptyset	REAL (@L,@D) UNSIGNED
	L=0 or L \emptyset	REAL UNSIGNED
	L>0 and D not \emptyset	REAL (@L,@D) UNSIGNED ZEROFILL
	L=0 or L \emptyset	REAL UNSIGNED ZEROFILL
P-AutoIdentifier		INTEGER
P-Binary	L \emptyset or L<=0	BINARY
	L is numeric and L<>0	BINARY (@L)
P-Boolean		BOOLEAN
P-Byte	L \emptyset or L<=0	BIT
	L is numeric and L<>0	BIT (@L)
P-Character	Not Unicode and L=0	CHAR
	Not Unicode and L<>0	CHAR (@L)
	Unicode and L<>0	CHAR (@L) UNICODE
	Unicode and L=0	CHAR UNICODE
P-Character Ascii	L \emptyset or L<=0	CHAR ASCII
	L is numeric and L<>0	CHAR(@L) ASCII
P-Character Binary	Not Unicode and L<>0	CHAR (@L) BINARY
	Unicode and L<>0	CHAR (@L) UNICODE BINARY
	Not Unicode and L=0	CHAR BINARY
	Unicode and L=0	CHAR UNICODE BINARY

Pivot --> Datatype (MySQL 4.1)

Pivot	Condition	Datatype
P-Character Unicode	L is numeric and L > 0	CHAR (@L) UNICODE
	L = 0 or L <= 0	CHAR UNICODE
P-Character Unicode Binary	L is numeric and L > 0	CHAR (@L) UNICODE BINARY
	L = 0 or L <= 0	CHAR UNICODE BINARY
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		DATETIME
P-Decimal	L = 0 or L = 0	DECIMAL
	L > 0 and D = 0	DECIMAL (@L)
	L > 0 and D not = 0	DECIMAL(@L,@D)
P-Decimal Unsigned	L > 0 and D = 0	DECIMAL (@L) UNSIGNED
	L > 0 and D not = 0	DECIMAL (@L,@D) UNSIGNED
	L = 0 or L = 0	DECIMAL UNSIGNED
P-Decimal Unsigned Zero-fill	L > 0 and D = 0	DECIMAL (@L) UNSIGNED ZEROFILL
	L > 0 and D not = 0	DECIMAL (@L,@D) UNSIGNED ZEROFILL
	L = 0 or L = 0	DECIMAL UNSIGNED ZEROFILL
P-Double	L = 0 or L <= 0	DOUBLE PRECISION
	L > 0 or D > 0	DOUBLE PRECISION (@L,@D)
P-Double Unsigned	L > 0 and D not = 0	DOUBLE PRECISION (@L,@D) UNSIGNED
	L = 0 or L = 0	DOUBLE PRECISION UNSIGNED

Pivot --> Datatype (MySQL 4.1)

Pivot	Condition	Datatype
P-Double Unsigned Zero-fill	$L > 0$ and $D \neq \emptyset$	DOUBLE PRECISION (@L,@D) UNSIGNED ZEROFILL
	$L = 0$ or $L \emptyset$	DOUBLE PRECISION UNSIGNED ZEROFILL
P-Float	$L = 0$ or $L \emptyset$	FLOAT
	$L > 0$ and $D \emptyset$	FLOAT (@L)
	$L > 0$ and $D \neq \emptyset$	FLOAT (@L,@D)
P-Float Unsigned	$L > 0$ and $D \emptyset$	FLOAT (@L) UNSIGNED
	$L > 0$ and $D \neq \emptyset$	FLOAT (@L,@D) UNSIGNED
	$L = 0$ or $L \emptyset$	FLOAT UNSIGNED
P-Float Unsigned Zerofill	$L > 0$ and $D \emptyset$	FLOAT (@L) UNSIGNED ZEROFILL
	$L > 0$ and $D \neq \emptyset$	FLOAT (@L,@D) UNSIGNED ZEROFILL
	$L = 0$ or $L \emptyset$	FLOAT UNSIGNED ZEROFILL
P-Integer	$L \emptyset$ or $L \leq 0$	INTEGER
	L is numeric and $L > 0$	INTEGER (@L)
P-Integer Unsigned	L is numeric and $L > 0$	INTEGER (@L) UNSIGNED
	$L \emptyset$ or $L \leq 0$	INTEGER UNSIGNED
P-Integer Unsigned Zerofill	L is numeric and $L > 0$	INTEGER (@L) UNSIGNED ZEROFILL
	$L \emptyset$ or $L \leq 0$	INTEGER UNSIGNED ZEROFILL
P-Long Integer	$L \emptyset$ or $L \leq 0$	BIGINT
	L is numeric and $L > 0$	BIGINT (@L)

Pivot --> Datatype (MySQL 4.1)

Pivot	Condition	Datatype
P-Long Integer Unsigned	L is numeric and L > 0	BIGINT (@L) UNSIGNED
	L = 0 or L <= 0	BIGINT UNSIGNED
P-Long Integer Unsigned Zerofill	L is numeric and L > 0	BIGINT (@L) UNSIGNED ZEROFILL
	L = 0 or L <= 0	BIGINT UNSIGNED ZEROFILL
P-Longblob		LONGBLOB
P-Longtext		LONGTEXT
P-Mediumblob		MEDIUMBLOB
P-Mediumint	L = 0 or L <= 0	MEDIUMINT
	L is numeric and L > 0	MEDIUMINT (@L)
P-Mediumint Unsigned	L is numeric and L > 0	MEDIUMINT (@L) UNSIGNED
	L = 0 or L <= 0	MEDIUMINT UNSIGNED
P-Mediumint Unsigned Zerofill	L is numeric and L > 0	MEDIUMINT (@L) UNSIGNED ZEROFILL
	L = 0 or L <= 0	MEDIUMINT UNSIGNED ZEROFILL
P-Mediumtext		MEDIUMTEXT
P-Multimedia		BLOB
P-National Varchar	L = 0 or L < 0	NATIONAL VARCHAR
	L is numeric and L >= 0	NATIONAL VARCHAR (@L)
P-National Varchar Binary	L is numeric and L >= 0	NATIONAL VARCHAR (@L) BINARY
	L = 0 or L < 0	NATIONAL VARCHAR BINARY

Pivot --> Datatype (MySQL 4.1)

Pivot	Condition	Datatype
P-Numeric	$L=0$ or $L \emptyset$	NUMERIC
	$L>0$ and $D \emptyset$	NUMERIC (@L)
	$L>0$ and D not \emptyset	NUMERIC (@L,@D)
P-Real	$L \emptyset$ or $L \leq 0$	REAL
	$L>0$ and D not \emptyset	REAL (@L,@D)
P-Smallint		SMALLINT
	L is numeric and $L <> 0$	SMALLINT (@L)
P-Smallint Unsigned	L is numeric and $L <> 0$	SMALLINT (@L) UNSIGNED
	$L \emptyset$ or $L \leq 0$	SMALLINT UNSIGNED
P-Smallint Unsigned Zero-fill	L is numeric and $L <> 0$	SMALLINT (@L) UNSIGNED ZEROFILL
	$L \emptyset$ or $L \leq 0$	SMALLINT UNSIGNED ZEROFILL
P-String		VARCHAR(@L)
P-Text		TEXT
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyblob		TINYBLOB
P-Tinyint	$L \emptyset$ or $L \leq 0$	TINYINT
	L is numeric and $L <> 0$	TINYINT (@L)
P-Tinyint Unsigned	L is numeric and $L <> 0$	TINYINT (@L) UNSIGNED
	$L \emptyset$ or $L \leq 0$	TINYINT UNSIGNED
P-Tinyint Unsigned Zero-fill	L is numeric and $L <> 0$	TINYINT (@L) UNSIGNED ZEROFILL
	$L \emptyset$ or $L \leq 0$	TINYINT UNSIGNED ZEROFILL
P-Tinytext		TINYTEXT

Pivot --> Datatype (MySQL 4.1)

Pivot	Condition	Datatype
P-Varbinary	L \emptyset or L < 0	VARBINARY
	L is numeric and L \geq 0	VARBINARY (@L)
P-Varchar	L \emptyset or L < 0	VARCHAR
	L is numeric and L \geq 0	VARCHAR(@L)
P-Varchar Binary	L is numeric and L \geq 0	VARCHAR (@L) BINARY
	L \emptyset or L < 0	VARCHAR BINARY
P-Wide Character	L \emptyset or L \leq 0	NATIONAL CHAR
	L is numeric and L \geq 0	NATIONAL CHAR (@L)
P-Wide Character Binary	L is numeric and L \geq 0	NATIONAL CHAR (@L) BINARY
	L \emptyset or L \leq 0	NATIONAL CHAR BINARY
P-Year	L \emptyset or L \leq 0	YEAR
	L is numeric and L \geq 0	YEAR (@L)

Datatype --> Pivot (MySQL 4.1)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
BIGINT (L)		P-Long Integer
BIGINT (L) UNSIGNED		P-Long Integer Unsigned
BIGINT (L) UNSIGNED ZEROFILL		P-Long Integer Unsigned Zerofill
BIGINT UNSIGNED		P-Long Integer Unsigned
BIGINT UNSIGNED ZEROFILL		P-Long Integer Unsigned Zerofill
BINARY		P-Binary
BINARY (L)		P-Binary
BLOB		P-Multimedia

Datatype --> Pivot (MySQL 4.1)

Datatype	Condition	Pivot
BOOLEAN		P-Boolean
CHAR (L)		P-Character
CHAR (L) BINARY		P-Character Binary
CHAR (L) UNICODE		P-Character Unicode
CHAR (L) UNICODE BINARY		P-Character Unicode Binary
CHAR ASCII		P-Character Ascii
CHAR BINARY		P-Character Binary
CHAR UNICODE		P-Character Unicode
CHAR UNICODE BINARY		P-Character Unicode Binary
CHAR(L) ASCII		P-Character Ascii
DATE		P-Date
DATETIME		P-Datetime
DECIMAL (L)		P-Decimal
DECIMAL (L) UNSIGNED		P-Decimal Unsigned
DECIMAL (L) UNSIGNED ZEROFILL		P-Decimal Unsigned Zero-fill
DECIMAL (L,D) UNSIGNED		P-Decimal Unsigned
DECIMAL (L,D) UNSIGNED ZEROFILL		P-Decimal Unsigned Zero-fill
DECIMAL UNSIGNED		P-Decimal Unsigned
DECIMAL UNSIGNED ZEROFILL		P-Decimal Unsigned Zero-fill
DOUBLE PRECISION		P-Double
DOUBLE PRECISION (L,D)		P-Double

Datatype --> Pivot (MySQL 4.1)

Datatype	Condition	Pivot
DOUBLE PRECISION (L,D) UNSIGNED		P-Double Unsigned
DOUBLE PRECISION (L,D) UNSIGNED ZERO-FILL		P-Double Unsigned Zero-fill
DOUBLE PRECISION UNSIGNED		P-Double Unsigned
DOUBLE PRECISION UNSIGNED ZEROFILL		P-Double Unsigned Zero-fill
FLOAT		P-Float
FLOAT (L)		P-Float
FLOAT (L) UNSIGNED		P-Float Unsigned
FLOAT (L) UNSIGNED ZEROFILL		P-Float Unsigned Zerofill
FLOAT (L,D)		P-Float
FLOAT (L,D) UNSIGNED		P-Float Unsigned
FLOAT (L,D) UNSIGNED ZEROFILL		P-Float Unsigned Zerofill
FLOAT UNSIGNED		P-Float Unsigned
FLOAT UNSIGNED ZERO-FILL		P-Float Unsigned Zerofill
INTEGER		P-Integer
INTEGER (L)		P-Integer
INTEGER (L) UNSIGNED		P-Integer Unsigned
INTEGER (L) UNSIGNED ZEROFILL		P-Integer Unsigned Zerofill
INTEGER UNSIGNED		P-Integer Unsigned
INTEGER UNSIGNED ZEROFILL		P-Integer Unsigned Zerofill
LOB		P-Longblob

Datatype --> Pivot (MySQL 4.1)

Datatype	Condition	Pivot
LONGTEXT		P-Longtext
MEDIUMBLOB		P-Mediumblob
MEDIUMINT		P-Mediumint
MEDIUMINT (L)		P-Mediumint
MEDIUMINT (L) UNSIGNED		P-Mediumint Unsigned
MEDIUMINT (L) UNSIGNED ZEROFILL		P-Mediumint Unsigned Zerofill
MEDIUMINT UNSIGNED		P-Mediumint Unsigned
MEDIUMINT UNSIGNED ZEROFILL		P-Mediumint Unsigned Zerofill
MEDIUMTEXT		P-Mediumtext
NATIONAL CHAR		P-Wide Character
NATIONAL CHAR (L)		P-Wide Character
NATIONAL CHAR (L) BINARY		P-Wide Character Binary
NATIONAL CHAR BINARY		P-Wide Character Binary
NATIONAL VARCHAR		P-National Varchar
NATIONAL VARCHAR (L)		P-National Varchar
NATIONAL VARCHAR (L) BINARY		P-National Varchar Binary
NATIONAL VARCHAR BINARY		P-National Varchar Binary
NUMERIC		P-Numeric
NUMERIC (L)		P-Numeric
NUMERIC (L,D)		P-Numeric
REAL		P-Real

Datatype --> Pivot (MySQL 4.1)

Datatype	Condition	Pivot
REAL (L,D)		P-Real
REAL (L,D) UNSIGNED		
REAL (L,D) UNSIGNED ZEROFILL		
REAL UNSIGNED		
REAL UNSIGNED ZEROFILL		
SMALLINT		P-Smallint
SMALLINT (L)		P-Smallint
SMALLINT (L) UNSIGNED		P-Smallint Unsigned
SMALLINT (L) UNSIGNED ZEROFILL		P-Smallint Unsigned Zero-fill
SMALLINT UNSIGNED		P-Smallint Unsigned
SMALLINT UNSIGNED ZEROFILL		P-Smallint Unsigned Zero-fill
TEXT		P-Text
TIME		P-Time
TIMESTAMP		P-Timestamp
TINYBLOB		P-Tinyblob
TINYINT		P-Tinyint
TINYINT (L)		P-Tinyint
TINYINT (L) UNSIGNED		P-Tinyint Unsigned
TINYINT (L) UNSIGNED ZEROFILL		P-Tinyint Unsigned Zero-fill
TINYINT UNSIGNED		P-Tinyint Unsigned
TINYINT UNSIGNED ZEROFILL		P-Tinyint Unsigned Zero-fill
TINYTEXT		P-Tinytext
VARBINARY		P-Varbinary

Datatype --> Pivot (MySQL 4.1)

Datatype	Condition	Pivot
VARBINARY (L)		P-Varbinary
VARCHAR		P-Varchar
VARCHAR (L) BINARY		P-Varchar Binary
VARCHAR BINARY		P-Varchar Binary
VARCHAR(L)		P-Varchar
YEAR		P-Year
YEAR (L)		P-Year

MySQL 5.0

Pivot --> Datatype (MySQL 5.0)

Pivot	Condition	Datatype
	L>0 and D not \emptyset	REAL (@L,@D) UNSIGNED
	L=0 or L \emptyset	REAL UNSIGNED
	L>0 and D not \emptyset	REAL (@L,@D) UNSIGNED ZEROFILL
	L=0 or L \emptyset	REAL UNSIGNED ZEROFILL
P-AutoIdentifier		INTEGER
P-Binary	L \emptyset or L<=0	BINARY
	L is numeric and L<>0	BINARY (@L)
P-Boolean		BOOLEAN
P-Byte	L \emptyset or L<=0	BIT
	L is numeric and L<>0	BIT (@L)
P-Character	Not Unicode and L=0	CHAR
	Unicode and L<>0	CHAR (@L) UNICODE
	Unicode and L=0	CHAR UNICODE
	Not Unicode and L<>0	CHAR(@L)
P-Character Ascii	L \emptyset or L<=0	CHAR ASCII
	L is numeric and L<>0	CHAR(@L) ASCII
P-Character Binary	Not Unicode and L<>0	CHAR (@L) BINARY
	Unicode and L<>0	CHAR (@L) UNICODE BINARY
	Not Unicode and L=0	CHAR BINARY
	Unicode and L=0	CHAR UNICODE BINARY

Pivot --> Datatype (MySQL 5.0)

Pivot	Condition	Datatype
P-Character Unicode	L is numeric and L > 0	CHAR (@L) UNICODE
	L = 0 or L <= 0	CHAR UNICODE
P-Character Unicode Binary	L is numeric and L > 0	CHAR (@L) UNICODE BINARY
	L = 0 or L <= 0	CHAR UNICODE BINARY
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		DATETIME
P-Decimal	L=0 or L = 0	DECIMAL
	L>0 and D = 0	DECIMAL (@L)
	L>0 and D not = 0	DECIMAL(@L,@D)
P-Decimal Unsigned	L>0 and D = 0	DECIMAL (@L) UNSIGNED
	L>0 and D not = 0	DECIMAL (@L,@D) UNSIGNED
	L=0 or L = 0	DECIMAL UNSIGNED
P-Decimal Unsigned Zero-fill	L>0 and D = 0	DECIMAL (@L) UNSIGNED ZEROFILL
	L>0 and D not = 0	DECIMAL (@L,@D) UNSIGNED ZEROFILL
	L=0 or L = 0	DECIMAL UNSIGNED ZEROFILL
P-Double	L=0 or L = 0	DOUBLE PRECISION
	L>0 and D not = 0	DOUBLE PRECISION (@L,@D)
P-Double Unsigned	L>0 and D not = 0	DOUBLE PRECISION (@L,@D) UNSIGNED
	L=0 or L = 0	DOUBLE PRECISION UNSIGNED

Pivot --> Datatype (MySQL 5.0)

Pivot	Condition	Datatype
P-Double Unsigned Zero-fill	$L > 0$ and $D \neq \emptyset$	DOUBLE PRECISION (@L,@D) UNSIGNED ZEROFILL
	$L = 0$ or $L \emptyset$	DOUBLE PRECISION UNSIGNED ZEROFILL
P-Float	$L = 0$ or $L \emptyset$	FLOAT
	$L > 0$ and $D \emptyset$	FLOAT (@L)
	$L > 0$ and $D \neq \emptyset$	FLOAT (@L,@D)
P-Float Unsigned	$L > 0$ and $D \emptyset$	FLOAT (@L) UNSIGNED
	$L > 0$ and $D \neq \emptyset$	FLOAT (@L,@D) UNSIGNED
	$L = 0$ or $L \emptyset$	FLOAT UNSIGNED
P-Float Unsigned Zerofill	$L > 0$ and $D \emptyset$	FLOAT (@L) UNSIGNED ZEROFILL
	$L > 0$ and $D \neq \emptyset$	FLOAT (@L,@D) UNSIGNED ZEROFILL
	$L = 0$ or $L \emptyset$	FLOAT UNSIGNED ZEROFILL
P-Integer	$L \emptyset$ or $L \leq 0$	INTEGER
	L is numeric and $L > 0$	INTEGER (@L)
P-Integer Unsigned	L is numeric and $L > 0$	INTEGER (@L) UNSIGNED
	$L \emptyset$ or $L \leq 0$	INTEGER UNSIGNED
P-Integer Unsigned Zerofill	L is numeric and $L > 0$	INTEGER (@L) UNSIGNED ZEROFILL
	$L \emptyset$ or $L \leq 0$	INTEGER UNSIGNED ZEROFILL
P-Long Integer	$L \emptyset$ or $L \leq 0$	BIGINT
	L is numeric and $L > 0$	BIGINT (@L)

Pivot --> Datatype (MySQL 5.0)

Pivot	Condition	Datatype
P-Long Integer Unsigned	L is numeric and L <> 0	BIGINT (@L) UNSIGNED
	L \emptyset or L <= 0	BIGINT UNSIGNED
P-Long Integer Unsigned Zerofill	L is numeric and L <> 0	BIGINT (@L) UNSIGNED ZEROFILL
	L \emptyset or L <= 0	BIGINT UNSIGNED ZEROFILL
P-Longblob		LONGBLOB
P-Longtext		LONGTEXT
P-Mediumblob		MEDIUMBLOB
P-Mediumint	L \emptyset or L <= 0	MEDIUMINT
	L is numeric and L <> 0	MEDIUMINT (@L)
P-Mediumint Unsigned	L is numeric and L <> 0	MEDIUMINT (@L) UNSIGNED
	L \emptyset or L <= 0	MEDIUMINT UNSIGNED
P-Mediumint Unsigned Zerofill	L is numeric and L <> 0	MEDIUMINT (@L) UNSIGNED ZEROFILL
	L \emptyset or L <= 0	MEDIUMINT UNSIGNED ZEROFILL
P-Mediumtext		MEDIUMTEXT
P-Multimedia		BLOB
P-National Varchar	L \emptyset or L < 0	NATIONAL VARCHAR
	L is numeric and L <> 0	NATIONAL VARCHAR (@L)
P-National Varchar Binary	L is numeric and L <> 0	NATIONAL VARCHAR (@L) BINARY
	L \emptyset or L < 0	NATIONAL VARCHAR BINARY

Pivot --> Datatype (MySQL 5.0)

Pivot	Condition	Datatype
P-Numeric	$L=0$ or $L \emptyset$	NUMERIC
	$L>0$ and $D \emptyset$	NUMERIC (@L)
	$L>0$ and D not \emptyset	NUMERIC (@L,@D)
P-Real	$L=0$ or $L \emptyset$	REAL
	$L>0$ and D not \emptyset	REAL (@L,@D)
P-Smallint	$L \emptyset$ or $L \leq 0$	SMALLINT
	L is numeric and $L > 0$	SMALLINT (@L)
P-Smallint Unsigned	L is numeric and $L > 0$	SMALLINT (@L) UNSIGNED
	$L \emptyset$ or $L \leq 0$	SMALLINT UNSIGNED
P-Smallint Unsigned Zero-fill	L is numeric and $L > 0$	SMALLINT (@L) UNSIGNED ZEROFILL
	$L \emptyset$ or $L \leq 0$	SMALLINT UNSIGNED ZEROFILL
P-String		VARCHAR(@L)
P-Text		TEXT
P-Time		TIME
P-Timestamp		TIMESTAMP
P-Tinyblob		TINYBLOB
P-Tinyint	$L \emptyset$ or $L \leq 0$	TINYINT
	L is numeric and $L > 0$	TINYINT (@L)
P-Tinyint Unsigned	L is numeric and $L > 0$	TINYINT (@L) UNSIGNED
	$L \emptyset$ or $L \leq 0$	TINYINT UNSIGNED
P-Tinyint Unsigned Zero-fill	L is numeric and $L > 0$	TINYINT (@L) UNSIGNED ZEROFILL
	$L \emptyset$ or $L \leq 0$	TINYINT UNSIGNED ZEROFILL
P-Tinytext		TINYTEXT

Pivot --> Datatype (MySQL 5.0)

Pivot	Condition	Datatype
P-Varbinary	L \emptyset or L < 0	VARBINARY
	L is numeric and L \geq 0	VARBINARY (@L)
P-Varchar	L \emptyset or L \leq 0	VARCHAR
	L is numeric and L = 0	VARCHAR(@L)
P-Varchar Binary	L is numeric and L \neq 0	VARCHAR (@L) BINARY
	L \emptyset or L < 0	VARCHAR BINARY
P-Wide Character	L \emptyset or L \leq 0	NATIONAL CHAR
	L is numeric and L \neq 0	NATIONAL CHAR (@L)
P-Wide Character Binary	L is numeric and L \neq 0	NATIONAL CHAR (@L) BINARY
	L \emptyset or L \leq 0	NATIONAL CHAR BINARY
P-Year	L \emptyset or L \leq 0	YEAR
	L is numeric and L \neq 0	YEAR(@L)

Datatype --> Pivot (MySQL 5.0)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
BIGINT (L)		P-Long Integer
BIGINT (L) UNSIGNED		P-Long Integer Unsigned
BIGINT (L) UNSIGNED ZEROFILL		P-Long Integer Unsigned Zerofill
BIGINT UNSIGNED		P-Long Integer Unsigned
BIGINT UNSIGNED ZEROFILL		P-Long Integer Unsigned Zerofill
BINARY		P-Binary
BINARY (L)		P-Binary
BIT		P-Byte

Datatype --> Pivot (MySQL 5.0)

Datatype	Condition	Pivot
BIT (L)		P-Byte
BLOB		P-Multimedia
BOOLEAN		P-Boolean
CHAR		P-Character
CHAR (L) BINARY		P-Character Binary
CHAR (L) UNICODE		P-Character Unicode
CHAR (L) UNICODE BINARY		P-Character Unicode Binary
CHAR ASCII		P-Character Ascii
CHAR BINARY		P-Character Binary
CHAR UNICODE		P-Character Unicode
CHAR UNICODE BINARY		P-Character Unicode Binary
CHAR(L)		P-Character
CHAR(L) ASCII		P-Character Ascii
DATE		P-Date
DATETIME		P-Datetime
DECIMAL		P-Decimal
DECIMAL (L)		P-Decimal
DECIMAL (L) UNSIGNED		P-Decimal Unsigned
DECIMAL (L) UNSIGNED ZEROFILL		P-Decimal Unsigned Zero-fill
DECIMAL (L,D) UNSIGNED		P-Decimal Unsigned
DECIMAL (L,D) UNSIGNED ZEROFILL		P-Decimal Unsigned Zero-fill
DECIMAL UNSIGNED		P-Decimal Unsigned
DECIMAL UNSIGNED ZEROFILL		P-Decimal Unsigned Zero-fill

Datatype --> Pivot (MySQL 5.0)

Datatype	Condition	Pivot
DECIMAL(L,D)		P-Decimal
DOUBLE PRECISION		P-Double
DOUBLE PRECISION (L,D)		P-Double
DOUBLE PRECISION (L,D) UNSIGNED		P-Double Unsigned
DOUBLE PRECISION (L,D) UNSIGNED ZERO-FILL		P-Double Unsigned Zero-fill
DOUBLE PRECISION UNSIGNED		P-Double Unsigned
DOUBLE PRECISION UNSIGNED ZEROFILL		P-Double Unsigned Zero-fill
FLOAT		P-Float
FLOAT (L)		P-Float
FLOAT (L) UNSIGNED		P-Float Unsigned
FLOAT (L) UNSIGNED ZEROFILL		P-Float Unsigned Zerofill
FLOAT (L,D)		P-Float
FLOAT (L,D) UNSIGNED		P-Float Unsigned
FLOAT (L,D) UNSIGNED ZEROFILL		P-Float Unsigned Zerofill
FLOAT UNSIGNED		P-Float Unsigned
FLOAT UNSIGNED ZEROFILL		P-Float Unsigned Zerofill
INTEGER		P-Integer
INTEGER (L)		P-Integer
INTEGER (L) UNSIGNED		P-Integer Unsigned
INTEGER (L) UNSIGNED ZEROFILL		P-Integer Unsigned Zerofill

Datatype --> Pivot (MySQL 5.0)

Datatype	Condition	Pivot
INTEGER UNSIGNED		P-Integer Unsigned
INTEGER UNSIGNED ZEROFILL		P-Integer Unsigned Zerofill
LOBLOB		P-Longblob
LONGTEXT		P-Longtext
MEDIUMBLOB		P-Mediumblob
MEDIUMINT		P-Mediumint
MEDIUMINT (L)		P-Mediumint
MEDIUMINT (L) UNSIGNED		P-Mediumint Unsigned
MEDIUMINT (L) UNSIGNED ZEROFILL		P-Mediumint Unsigned Zerofill
MEDIUMINT UNSIGNED		P-Mediumint Unsigned
MEDIUMINT UNSIGNED ZEROFILL		P-Mediumint Unsigned Zerofill
MEDIUMTEXT		P-Mediumtext
NATIONAL CHAR		P-Wide Character
NATIONAL CHAR (L)		P-Wide Character
NATIONAL CHAR (L) BINARY		P-Wide Character Binary
NATIONAL CHAR BINARY		P-Wide Character Binary
NATIONAL VARCHAR		P-National Varchar
NATIONAL VARCHAR (L)		P-National Varchar
NATIONAL VARCHAR (L) BINARY		P-National Varchar Binary
NATIONAL VARCHAR BINARY		P-National Varchar Binary
NUMERIC		P-Numeric

Datatype --> Pivot (MySQL 5.0)

Datatype	Condition	Pivot
NUMERIC (L)		P-Numeric
NUMERIC (L,D)		P-Numeric
REAL		P-Real
REAL (L,D)		P-Real
REAL (L,D) UNSIGNED		
REAL (L,D) UNSIGNED ZEROFILL		
REAL UNSIGNED		
REAL UNSIGNED ZEROFILL		
SMALLINT		P-Smallint
SMALLINT (L)		P-Smallint
SMALLINT (L) UNSIGNED		P-Smallint Unsigned
SMALLINT (L) UNSIGNED ZEROFILL		P-Smallint Unsigned Zero-fill
SMALLINT UNSIGNED		P-Smallint Unsigned
SMALLINT UNSIGNED ZEROFILL		P-Smallint Unsigned Zero-fill
TEXT		P-Text
TIME		P-Time
TIMESTAMP		P-Timestamp
TINYBLOB		P-Tinyblob
TINYINT		P-Tinyint
TINYINT (L)		P-Tinyint
TINYINT (L) UNSIGNED		P-Tinyint Unsigned
TINYINT (L) UNSIGNED ZEROFILL		P-Tinyint Unsigned Zero-fill
TINYINT UNSIGNED		P-Tinyint Unsigned

Datatype --> Pivot (MySQL 5.0)

Datatype	Condition	Pivot
TINYINT UNSIGNED ZEROFILL		P-Tinyint Unsigned Zero-fill
TINYTEXT		P-Tinytext
VARBINARY		P-Varbinary
VARBINARY (L)		P-Varbinary
VARCHAR		P-Varchar
VARCHAR (L) BINARY		P-Varchar Binary
VARCHAR BINARY		P-Varchar Binary
VARCHAR(L)		P-Varchar
YEAR		P-Year
YEAR(L)		P-Year

ORACLE 10

Pivot --> Datatype (Oracle 10)

Pivot	Condition	Datatype
P-AutoIdentifier		NUMBER
P-Binary		RAW(@L)
P-Boolean	L=2 or L ø	RAW(1)
	L>1	RAW(@L)
P-Byte		RAW(1)
P-Character	Not Unicode and (L<2001 or L ø)	CHAR(@L)
	L>4000	LONG
	Not Unicode and (L=2001 or L ø)	NCHAR(@L)
	Unicode and 2000<L<4001	NVARCHAR2(@L)
	Not Unicode and 2000<L<4001	VARCHAR2(@L)
P-Currency		NUMBER(@L,@D)
P-Date		DATE
P-Datetime		DATE
P-Decimal		NUMBER(@L,@D)
P-Double		NUMBER(@L,@D)
P-Float	L=0 or L>126 or L ø	FLOAT
	0<L<127	FLOAT(@L)
P-Integer		NUMBER(@L)
P-Long Integer		NUMBER(@L)
P-Long Real		NUMBER(@L,@D)
P-Multimedia		LONG RAW

Pivot --> Datatype (Oracle 10)

Pivot	Condition	Datatype
P-Numeric	L=0 or L \emptyset	NUMBER
	L>0 and D \emptyset	NUMBER(@L)
	L>0 and D not \emptyset	NUMBER(@L,@D)
P-Real		NUMBER(@L,@D)
P-Smallint		NUMBER(@L)
P-String		LONG
P-Text	Unicode	NVARCHAR2(@L)
	Not Unicode	VARCHAR2(@L)
P-Time		DATE
P-Timestamp	L>9 or L \emptyset	TIMESTAMP
	L<10	TIMESTAMP(@L)
P-Tinyint		NUMBER(@L)
P-Varbinary		LONG RAW
P-Varchar	L>4000 or L=0 or L \emptyset	LONG
	Unicode and 0<L<4001	NVARCHAR2(@L)
	Not Unicode and 0<L<4001	VARCHAR2(@L)

Datatype --> Pivot (Oracle 10)

Datatype	Condition	Pivot
CHAR(L)		P-Character
DATE		P-Date
FLOAT		P-Float
FLOAT(L)		P-Float
LONG		P-String
LONG RAW		P-Multimedia
NUMBER		P-Numeric

Datatype --> Pivot (Oracle 10)

Datatype	Condition	Pivot
NUMBER(L)		P-Numeric
NUMBER(L,D)		P-Numeric
RAW(1)		P-Boolean
RAW(L)		P-Boolean
TIMESTAMP		P-Timestamp
TIMESTAMP(L)		P-Timestamp
VARCHAR2(L)		P-Varchar

ORACLE 11

Pivot --> Datatype (Oracle 11)

Pivot	Condition	Datatype
P-AutoIdentifier		NUMBER
P-Binary		RAW(@L)
P-Boolean	L=2 or L ø	RAW(1)
	L>1	RAW(@L)
P-Byte		RAW(1)
P-Character	Not Unicode and (L<2001 or L ø)	CHAR(@L)
	L>4000	LONG
	Not Unicode and (L=2001 or L ø)	NCHAR(@L)
	Unicode and 2000<L<4001	NVARCHAR2(@L)
	Not Unicode and 2000<L<4001	VARCHAR2(@L)
P-Currency		NUMBER(@L,@D)
P-Date		DATE
P-Datetime		DATE
P-Decimal		NUMBER(@L,@D)
P-Double		NUMBER(@L,@D)
P-Float	0<L<127	FLOAT(@L)
	L=0 or L>126 or L ø	FLOAT
P-Integer		NUMBER(@L)
P-Long Integer		NUMBER(@L)
P-Long Real		NUMBER(@L,@D)

Pivot --> Datatype (Oracle 11)

Pivot	Condition	Datatype
P-Numeric	L=0 or L \emptyset	NUMBER
	L>0 and D \emptyset	NUMBER(@L)
	L>0 and D not \emptyset	NUMBER(@L,@D)
P-Real		NUMBER(@L,@D)
P-Smallint		NUMBER(@L)
P-String		LONG
P-Text	Unicode	NVARCHAR2(@L)
	Not Unicode	VARCHAR2(@L)
P-Time		DATE
P-Timestamp	L<10	TIMESTAMP(@L)
	L>9 or L \emptyset	TIMESTAMP
P-Tinyint		NUMBER(@L)
P-Varchar	L>4000 or L=0 or L \emptyset	LONG
	Unicode and 0<L<4001	NVARCHAR2(@L)
	Not Unicode and 0<L<4001	VARCHAR2(@L)

Datatype --> Pivot (Oracle 11)

Datatype	Condition	Pivot
CHAR(L)		P-Character
DATE		P-Date
FLOAT		P-Float
FLOAT(L)		P-Float
LONG		P-String
NUMBER		P-Numeric
NUMBER(L)		P-Numeric
NUMBER(L,D)		P-Numeric

Datatype --> Pivot (Oracle 11)

Datatype	Condition	Pivot
RAW(1)		P-Boolean
RAW(L)		P-Boolean
TIMESTAMP		P-Timestamp
TIMESTAMP(L)		P-Timestamp
VARCHAR2(L)		P-Varchar

ORACLE 8

Pivot --> Datatype (Oracle 8)

Pivot	Condition	Datatype
P-AutoIdentifier		NUMBER
P-Binary		RAW(@L)
P-Boolean	L=2 or L \emptyset	RAW(1)
	L>1	RAW(@L)
P-Byte		RAW(1)
P-Character	L=2001 or L \emptyset	CHAR(@L)
	L>4000	LONG
	2000<L<4001	VARCHAR2(@L)
P-Currency		NUMBER(@L,@D)
P-Date		DATE
P-Datetime		DATE
P-Decimal		NUMBER(@L,@D)
P-Double		NUMBER(@L,@D)
P-Float		NUMBER(@L,@D)
P-Integer		NUMBER(@L)
P-Long Integer		NUMBER(@L)
P-Long Real		NUMBER(@L,@D)
P-Multimedia		LONG RAW
P-Numeric	L=0 or L \emptyset	NUMBER
	L>0 and D \emptyset	NUMBER(@L)
	L>0 and D not \emptyset	NUMBER(@L,@D)
P-Real		NUMBER(@L,@D)
P-Smallint		NUMBER(@L)
P-String		LONG

Pivot --> Datatype (Oracle 8)

Pivot	Condition	Datatype
P-Text		VARCHAR2(@L)
P-Time		DATE
P-Timestamp		ROWID
P-Tinyint		NUMBER(@L)
P-Varbinary		LONG RAW
P-Varchar	L>4000 or L=0 or L ø	LONG
	0<L<4001	VARCHAR2(@L)

Datatype --> Pivot (Oracle 8)

Datatype	Condition	Pivot
CHAR(L)		P-Character
DATE		P-Date
LONG		P-String
LONG RAW		P-Multimedia
NUMBER		P-Numeric
NUMBER(L)		P-Numeric
NUMBER(L,D)		P-Numeric
RAW(1)		P-Boolean
RAW(L)		P-Boolean
ROWID		P-Timestamp
VARCHAR2(L)		P-Varchar

ORACLE 9i

Pivot --> Datatype (Oracle 9i)

Pivot	Condition	Datatype
P-AutoIdentifier		NUMBER
P-Binary		RAW(@L)
P-Boolean	L=2 or L \emptyset	RAW(1)
	L>1	RAW(@L)
P-Byte		RAW(1)
P-Character	Not Unicode and (L<2001 or L \emptyset)	CHAR(@L)
	L>4000	LONG
	Not Unicode and (L=2001 or L \emptyset)	NCHAR(@L)
	Unicode and 2000<L<4001	NVARCHAR2(@L)
	Not Unicode and 2000<L<4001	VARCHAR2(@L)
P-Currency		NUMBER(@L,@D)
P-Date		DATE
P-Datetime		DATE
P-Decimal		NUMBER(@L,@D)
P-Double		NUMBER(@L,@D)
P-Float	L=0 or L>126 or L \emptyset	FLOAT
	0<L<127	FLOAT(@L)
P-Integer		NUMBER(@L)
P-Long Integer		NUMBER(@L)
P-Long Real		NUMBER(@L,@D)
P-Multimedia		LONG RAW
P-Numeric	L=0 or L \emptyset	NUMBER
	L>0 and D \emptyset	NUMBER(@L)
	L>0 and D not \emptyset	NUMBER(@L,@D)

Pivot --> Datatype (Oracle 9i)

Pivot	Condition	Datatype
P-Real		NUMBER(@L,@D)
P-Smallint		NUMBER(@L)
P-String		LONG
P-Text	Unicode	NVARCHAR2(@L)
	Not Unicode	VARCHAR2(@L)
P-Time		DATE
P-Timestamp	L>9 or L ø	TIMESTAMP
	L<10	TIMESTAMP(@L)
P-Tinyint		NUMBER(@L)
P-Varbinary		LONG RAW
P-Varchar	L>4000 or L=0 or L ø	LONG
	Unicode and 0<L<4001	NVARCHAR2(@L)
	Not Unicode and 0<L<4001	VARCHAR2(@L)

Datatype --> Pivot (Oracle 9i)

Datatype	Condition	Pivot
CHAR(L)		P-Character
DATE		P-Date
FLOAT		P-Float
FLOAT(L)		P-Float
LONG		P-String
LONG RAW		P-Multimedia
NUMBER		P-Numeric
NUMBER(L)		P-Numeric
NUMBER(L,D)		P-Numeric
RAW(1)		P-Boolean
RAW(L)		P-Boolean

Datatype --> Pivot (Oracle 9i)

Datatype	Condition	Pivot
TIMESTAMP		P-Timestamp
VARCHAR2(L)		P-Varchar

POSTGRESQL9.3

Pivot --> Datatype (PostgreSQL9.3)

Pivot	Condition	Datatype
P-Boolean		boolean
P-Byte	L=0 or L \emptyset	bit
	Valid	bit(@L)
P-Character	L=0 or L \emptyset	char
	Valid	char(@L)
P-Currency		money
P-Date		date
P-Decimal	L=0 or L \emptyset	decimal
	L \geq 1 and D \emptyset	decimal(@L)
	L \geq 1	decimal(@L,@D)
P-Double		double precision
P-Integer		integer
P-Long Integer		bigint
P-Numeric	L=0 or L \emptyset	numeric
	L \geq 1 and D \emptyset	numeric(@L)
	L \geq 1	numeric(@L,@D)
P-Real		real
P-Smallint		smallint
P-Text		text
P-Time	L=0 or L \emptyset	time
	Valid	time(@L)
P-Timestamp	L=0 or L \emptyset	timestamp
	L \diamond 0	timestamp(@L)

Pivot --> Datatype (PostgreSQL9.3)

Pivot	Condition	Datatype
P-Varchar	L=0 or L ∅	varchar
	Valid	varchar(@L)

Datatype --> Pivot (PostgreSQL9.3)

Datatype	Condition	Pivot
bigint		P-Long Integer
bit		P-Byte
bit(L)		P-Byte
boolean		P-Boolean
char		P-Character
char(L)		P-Character
date		P-Date
decimal		P-Decimal
decimal(L)		P-Decimal
decimal(L,D)		P-Decimal
double precision		P-Double
integer		P-Integer
money		P-Currency
numeric		P-Numeric
numeric(L)		P-Numeric
numeric(L,D)		P-Numeric
real		P-Real
smallint		P-Smallint
text		P-Text
time		P-Time
time(L)		P-Time
timestamp		P-Timestamp

Datatype --> Pivot (PostgreSQL9.3)

Datatype	Condition	Pivot
timestamp(L)		P-Timestamp
varchar		P-Varchar
varchar(L)		P-Varchar

SQL ANSI/ISO 9075:1992

Pivot --> Datatype (SQL ANSI/ISO 9075:1992)

Pivot	Condition	Datatype
P-AutoIdentifier		INTEGER
P-Binary		BIT VARYING(@L)
P-Boolean		BIT(@L)
P-Byte		BIT(@L)
P-Character		CHAR(@L)
P-Currency		DECIMAL(@L,@D)
P-Date		DATE
P-Datetime		DATETIME
P-Decimal		DECIMAL(@L,@D)
P-Double		DOUBLE PRECISION
P-Float		FLOAT
P-Integer		INTEGER
P-Long Integer		INTEGER
P-Long Real		REAL
P-Multimedia		BIT VARYING(@L)
P-Numeric	L>4	INTEGER
	L=5 or L ø	SMALLINT
P-Real		REAL
P-Smallint		SMALLINT
P-String		VARCHAR(@L)
P-Text		VARCHAR(@L)
P-Time		TIME
P-Timestamp		DATETIME
P-Tinyint		SMALLINT

Pivot --> Datatype (SQL ANSI/ISO 9075:1992)

Pivot	Condition	Datatype
P-Varbinary		BIT VARYING(@L)
P-Varchar		VARCHAR(@L)

Datatype --> Pivot (SQL ANSI/ISO 9075:1992)

Datatype	Condition	Pivot
BIT VARYING(L)		P-Multimedia
BIT(L)		P-Boolean
CHAR(L)		P-Character
DATE		P-Date
DATETIME		P-Datetime
DECIMAL(L,D)		P-Currency
DOUBLE PRECISION		P-Double
FLOAT		P-Float
INTEGER		P-Integer
REAL		P-Real
SMALLINT		P-Smallint
TIME		P-Time
VARCHAR(L)		P-Varchar

SQL SERVER 2000

Pivot --> Datatype (SQL Server 2000)

Pivot	Condition	Datatype
P-AutoIdentifier		uniqueidentifier
P-Binary		binary(@L)
P-Boolean		bit
P-Byte		bit
P-Character	Not Unicode and (L<8001 or L ø)	char(@L)
	Not Unicode and (L=8001 or L ø)	nchar(@L)
	Not Unicode and L>8000	ntext
	Not Unicode and L>8000	text
P-Currency		money
P-Date		smalldatetime
P-Datetime		datetime
P-Decimal		decimal(@L,@D)
P-Double		numeric(@L,@D)
P-Float		float
P-Integer		int
P-Long Integer		bigint
P-Long Real		real
P-Multimedia		image
P-Numeric		numeric(@L,@D)
P-Real		real
P-Smallint		smallint

Pivot --> Datatype (SQL Server 2000)

Pivot	Condition	Datatype
P-String	Unicode	ntext
	Not Unicode	text
P-Text	Unicode	ntext
	Not Unicode	text
P-Time		datetime
P-Timestamp		timestamp
P-Tinyint		tinyint
P-Varbinary		varbinary(@L)
P-Varchar	Not Unicode and L>8000	ntext
	Not Unicode and (L=8001 or L ø)	nvarchar(@L)
	Not Unicode and L>8000	text
	Not Unicode and (L<8001 or L ø)	varchar(@L)
P-Wide Character		nchar(@L)
P-Wide String		nvarchar(@L)

Datatype --> Pivot (SQL Server 2000)

Datatype	Condition	Pivot
bigint		P-Long Integer
binary(L)		P-Binary
bit		P-Boolean
char(L)		P-Character
datetime		P-Datetime
decimal(L,D)		P-Decimal
float		P-Float
image		P-Multimedia
int		P-Integer

Datatype --> Pivot (SQL Server 2000)

Datatype	Condition	Pivot
money		P-Currency
nchar(L)		P-Wide Character
numeric(L,D)		P-Numeric
nvarchar(L)		P-Wide String
real		P-Real
smalldatetime		P-Date
smallint		P-Smallint
text		P-Text
timestamp		P-Timestamp
tinyint		P-Tinyint
uniqueidentifier		P-AutoIdentifier
varbinary(L)		P-Varbinary
varchar(L)		P-Varchar

SQL SERVER 2005

Pivot --> Datatype (SQL Server 2005)

Pivot	Condition	Datatype
P-AutoIdentifier		uniqueidentifier
P-Binary		binary(@L)
P-Boolean		bit
P-Byte		bit
P-Character	Not Unicode and (L<8001 or L ø)	char(@L)
	Not Unicode and (L=8001 or L ø)	nchar(@L)
	Not Unicode and L>8000	ntext
	Not Unicode and L>8000	text
P-Currency		money
		smallmoney
P-Date		smalldatetime
P-Datetime		datetime
P-Decimal		decimal(@L,@D)
P-Double		numeric(@L,@D)
P-Float		float
P-Integer		int
P-Long Integer		bigint
P-Long Real		real
P-Multimedia		image
P-Numeric		numeric(@L,@D)
P-Real		real
P-Smallint		smallint

Pivot --> Datatype (SQL Server 2005)

Pivot	Condition	Datatype
P-String	Unicode	ntext
	Not Unicode	text
P-Text	Unicode	ntext
	Not Unicode	text
P-Time		datetime
P-Timestamp		timestamp
P-Tinyint		tinyint
P-Varbinary		varbinary(@L)
P-Varchar	Not Unicode and L>8000	ntext
	Not Unicode and (L=8001 or L ∅)	nvarchar(@L)
	Not Unicode and L>8000	text
	Not Unicode and (L<8001 or L ∅)	varchar(@L)
P-Wide Character		nchar(@L)
P-Wide String		nvarchar(@L)

Datatype --> Pivot (SQL Server 2005)

Datatype	Condition	Pivot
bigint		P-Long Integer
binary(L)		P-Binary
bit		P-Boolean
char(L)		P-Character
datetime		P-Datetime
decimal(L,D)		P-Decimal
float		P-Float
image		P-Multimedia
int		P-Integer

Datatype --> Pivot (SQL Server 2005)

Datatype	Condition	Pivot
money		P-Currency
nchar(L)		P-Wide Character
numeric(L,D)		P-Numeric
nvarchar(L)		P-Wide String
real		P-Real
smalldatetime		P-Date
smallint		P-Smallint
smallmoney		P-Currency
text		P-Text
timestamp		P-Timestamp
tinyint		P-Tinyint
uniqueidentifier		P-AutoIdentifier
varbinary(L)		P-Varbinary
varchar(L)		P-Varchar

SQL SERVER 2008

Pivot --> Datatype (SQL Server 2008)

Pivot	Condition	Datatype
P-AutoIdentifier		uniqueidentifier
P-Binary		binary(@L)
P-Boolean		bit
P-Byte		bit
P-Character	Not Unicode and (L<8001 or L \emptyset)	char(@L)
	Not Unicode and (L=8001 or L \emptyset)	nchar(@L)
	Not Unicode and L>8000	ntext
	Not Unicode and L>8000	text
P-Currency	L not empty or L > 10	money
	L empty or L < 11	smallmoney
P-Date		smalldatetime
P-Datetime		datetime
P-Decimal		decimal(@L,@D)
P-Double		numeric(@L,@D)
P-Float	L empty	float
	L not empty	float(@L)
P-Integer		int
P-Long Integer		bigint
P-Long Real		real
P-Multimedia		image
P-Numeric		numeric(@L,@D)
P-Real		real

Pivot --> Datatype (SQL Server 2008)

Pivot	Condition	Datatype
P-Smallint		smallint
P-String	Unicode	ntext
	Not Unicode	text
P-Text	Unicode	ntext
	Not Unicode	text
P-Time		time
P-Timestamp		timestamp
P-Tinyint		tinyint
P-Varbinary		varbinary(@L)
P-Varchar	Not Unicode and L>8000	ntext
	Not Unicode and (L=8001 or L ø)	nvarchar(@L)
	Not Unicode and L>8000	text
	Not Unicode and (L<8001 or L ø)	varchar(@L)
P-Wide Character		nchar(@L)
P-Wide String		nvarchar(@L)

Datatype --> Pivot (SQL Server 2008)

Datatype	Condition	Pivot
bigint		P-Long Integer
binary(L)		P-Binary
bit		P-Boolean
char(L)		P-Character
datetime		P-Datetime
decimal(L,D)		P-Decimal
float		P-Float
float(L)		P-Float

Datatype --> Pivot (SQL Server 2008)

Datatype	Condition	Pivot
image		P-Multimedia
int		P-Integer
money		P-Currency
nchar(L)		P-Wide Character
numeric(L,D)		P-Numeric
nvarchar(L)		P-Wide String
real		P-Real
smalldatetime		P-Date
smallint		P-Smallint
smallmoney		P-Currency
text		P-Text
time		P-Time
timestamp		P-Timestamp
tinyint		P-Tinyint
uniqueidentifier		P-AutoIdentifier
varbinary(L)		P-Varbinary
varchar(L)		P-Varchar

SQL SERVER 7

Pivot --> Datatype (SQL Server 7)

Pivot	Condition	Datatype
P-AutoIdentifier		timestamp
P-Binary		binary(@L)
P-Boolean		bit
P-Byte		bit
P-Character	0<L<251	char(@L)
	L>250 or L=0 or L ø	text
P-Currency		money
P-Date		smalldatetime
P-Datetime		datetime
P-Decimal		decimal(@L,@D)
P-Double		numeric(@L,@D)
P-Float		float
P-Integer		int
P-Long Integer		int
P-Long Real		real
P-Multimedia		image
P-Numeric	L>9 and D ø	float
	4<L<10 and D ø	int
	L not ø and D not ø	numeric(@L,@D)
	2<L<5 and D ø	smallint
	(L<3 and D ø) or L ø	tinyint
P-Real		real
P-Smallint		smallint
P-String		char(@L)

Pivot --> Datatype (SQL Server 7)

Pivot	Condition	Datatype
P-Text		text
P-Time		datetime
P-Timestamp		timestamp
P-Tinyint		tinyint
P-Varbinary		varbinary(@L)
P-Varchar		varchar(@L)

Datatype --> Pivot (SQL Server 7)

Datatype	Condition	Pivot
binary(L)		P-Binary
bit		P-Boolean
char(L)		P-String
datetime		P-Datetime
decimal(L,D)		P-Decimal
float		P-Float
image		P-Multimedia
int		P-Integer
money		P-Currency
numeric(L,D)		P-Double
real		P-Real
smalldatetime		P-Date
smallint		P-Smallint
text		P-Character
timestamp		P-Timestamp
tinyint		P-Tinyint
varbinary(L)		P-Varbinary
varchar(L)		P-Varchar

SYBASE ADAPTIVE SERVER 11

Pivot --> Datatype (Sybase Adaptive Server 11)

Pivot	Condition	Datatype
P-AutoIdentifier		timestamp
P-Binary		binary(@L)
P-Boolean		bit
P-Byte		bit
P-Character	Not Unicode and L<256	char(@L)
	Unicode and L<256	nChar(@L)
	L>255 or L ø	text
P-Currency		money
P-Date		smalldatetime
P-Datetime		datetime
P-Decimal	0<L<39 and D not ø	decimal(@L,@D)
	(L>9 and D ø) or (1<L<38 and D not ø)	float
	4<L<10 and D ø	int
	2<L<5 and D ø	smallint
	(L<3 and D ø) or L ø	tinyint
P-Double		double precision
P-Float		float
P-Integer		int
P-Long Integer		int
P-Long Real		real
P-Multimedia		image

Pivot --> Datatype (Sybase Adaptive Server 11)

Pivot	Condition	Datatype
P-Numeric	(L>9 and D \emptyset) or (1<L<38 and D not \emptyset)	float
	4<L<10 and D \emptyset	int
	0<L<39 and D not \emptyset	numeric(@L,@D)
	2<L<5 and D \emptyset	smallint
	(L<3 and D \emptyset) or L \emptyset	tinyint
P-Real		real
P-Smallint		smallint
P-String	Not Unicode	char(@L)
	Unicode	nChar(@L)
P-Text		text
P-Time		datetime
P-Timestamp		timestamp
P-Tinyint		tinyint
P-Varbinary		varbinary(@L)
P-Varchar	Unicode	nVarChar(@L)
	Not Unicode	varchar(@L)

Datatype --> Pivot (Sybase Adaptive Server 11)

Datatype	Condition	Pivot
binary(L)		P-Binary
bit		P-Boolean
char(L)		P-String
datetime		P-Datetime
decimal(L,D)		P-Decimal
double precision		P-Double
float		P-Float

Datatype --> Pivot (Sybase Adaptive Server 11)

Datatype	Condition	Pivot
image		P-Multimedia
int		P-Integer
money		P-Currency
numeric(L,D)		P-Numeric
real		P-Real
smalldatetime		P-Date
smallint		P-Smallint
text		P-Character
timestamp		P-Timestamp
tinyint		P-Tinyint
varbinary(L)		P-Varbinary
varchar(L)		P-Varchar

SYBASE ADAPTIVE SERVER 12.5

Pivot --> Datatype (Sybase Adaptive Server 12.5)

Pivot	Condition	Datatype
P-AutoIdentifier		timestamp
P-Binary	$L=0$ or $L \emptyset$	binary
	$L > 0$	binary(@L)
P-Boolean		bit
P-Byte		bit
P-Character	Not Unicode and ($L=0$ or $L \emptyset$)	char
	Not Unicode and $0 < L < 256$	char(@L)
	$L > 255$	text
	Not Unicode and ($L=256$ or $L \emptyset$)	unichar(@L)
P-Currency		money
P-Date		smalldatetime
P-Datetime		datetime
P-Decimal	$0 < L < 39$ and D not \emptyset	decimal(@L,@D)
	($L > 9$ and $D \emptyset$) or (($L < 1$ or $L > 38$) and D not \emptyset)	float
	$4 < L < 10$ and $D \emptyset$	int
	$2 < L < 5$ and $D \emptyset$	smallint
	$L \emptyset$ or ($L < 3$ and $D \emptyset$)	tinyint
P-Double		double precision
P-Float		float
P-Integer		int
P-Long Integer		int
P-Long Real		real

Pivot --> Datatype (Sybase Adaptive Server 12.5)

Pivot	Condition	Datatype
P-Multimedia		image
P-Numeric	(L>9 and D \emptyset) or ((L<1 or L>38) and D not \emptyset)	float
	4<L<10 and D \emptyset	int
	0<L<39 and D not \emptyset	numeric(@L,@D)
	2<L<5 and D \emptyset	smallint
	L \emptyset or (L<3 and D \emptyset)	tinyint
P-Real		real
P-Smallint		smallint
P-String	Not Unicode and (L=0 or L \emptyset)	char
	Not Unicode and (0<L<256)	char(@L)
	L>255	text
	Not Unicode and (L=256 or L \emptyset)	unichar(@L)
P-Text		text
P-Time		datetime
P-Timestamp		timestamp
P-Tinyint		tinyint
P-Varbinary	L=0 or L \emptyset	varbinary
	L > 0	varbinary(@L)
P-Varchar	L>255	text
	Not Unicode and (L=256 or L \emptyset)	univarchar(@L)
	Not Unicode and (L=0 or L \emptyset)	varchar
	Not Unicode and 0<L<256	varchar(@L)

Datatype --> Pivot (Sybase Adaptive Server 12.5)

Datatype	Condition	Pivot
binary		P-Binary
binary(L)		P-Binary
bit		P-Boolean
char		P-Character
char(L)		P-Character
datetime		P-Datetime
decimal(L,D)		P-Decimal
double precision		P-Double
float		P-Float
image		P-Multimedia
int		P-Integer
money		P-Currency
numeric(L,D)		P-Numeric
real		P-Real
smalldatetime		P-Date
smallint		P-Smallint
text		P-Text
timestamp		P-Timestamp
tinyint		P-Tinyint
varbinary		P-Varbinary
varbinary(L)		P-Varbinary
varchar		P-Varchar
varchar(L)		P-Varchar

TERADATA DATABASE

Pivot --> Datatype (Teradata Database 14)

Pivot	Condition	Datatype
P-AutoIdentifier		NUMBER
P-Boolean		BYTEINT
P-Byte	L=0 or L ø	BYTE
	L>0	BYTE(@L)
P-Character	L=0 or L ø	CHAR
	L>0	CHAR(@L)
P-Date		DATE
P-Datetime		DATE
P-Decimal	L=0 or L ø and D=0 Or D ø	DECIMAL
	L>0 and D=0 Or D ø	DECIMAL(@L)
P-Double		NUMBER(@L,@D)
P-Float		FLOAT
P-Integer		INTEGER
P-Long Integer		BIGINT
P-Long Real		FLOAT
P-Multimedia	L=0 or L ø	BLOB
	L>0	BLOB(@L)
P-Numeric	L=0 or L ø and D=0 Or D ø	NUMBER
	L ø and D > 0	NUMBER(*,@D)
	L>0 and D=0 Or D ø	NUMBER(@L)
	L>0 and D>0	NUMBER(@L,@D)
P-Real		FLOAT
P-Smallint		SMALLINT
P-String		VARCHAR(@L)

Pivot --> Datatype (Teradata Database 14)

Pivot	Condition	Datatype
P-Text		VARCHAR(@L)
P-Time		TIME
	D > 0	TIME(@D)
P-Timestamp		TIMESTAMP
	D > 0	TIMESTAMP(@D)
P-Tinyint		SMALLINT
P-Varbinary		VARBYTE(@L)
P-Varchar		VARCHAR(@L)

Datatype --> Pivot (Teradata Database 14)

Datatype	Condition	Pivot
BIGINT		P-Long Integer
BLOB		P-Multimedia
BLOB(L)		P-Multimedia
BYTE		P-Byte
BYTE(L)		P-Byte
BYTEINT		P-Boolean
CHAR		P-Character
CHAR(L)		P-Character
DATE		P-Date
DECIMAL		P-Decimal
DECIMAL(L)		P-Decimal
FLOAT		P-Real
INTEGER		P-Integer

Datatype --> Pivot (Teradata Database 14)

Datatype	Condition	Pivot
NUMBER		P-Numeric
NUMBER(*,D)		P-Numeric
NUMBER(L)		P-Numeric
NUMBER(L,D)		P-Numeric
SMALLINT		P-Smallint
TIME		P-Time
TIME(D)		P-Time
TIMESTAMP		P-Timestamp
TIMESTAMP(D)		P-Timestamp
VARBYTE(L)		P-Varbinary
VARCHAR(L)		P-Varchar





PostgreSQL 9.3: Syntax supported by HOPEX

This document details concepts of PostgreSQL 9.3 recognized by MEGA.

Detail of Concepts Managed by PostgreSQL

All SQL commands are available at the following address:

<http://www.postgresql.org/docs/9.3/static/sql-commands.html>

Tables

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [
  { column_name data type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
  | table_constraint
  | LIKE parent_table [ like_option ... ] }
  [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name
  OF type_name [ (
    { column_name WITH OPTIONS [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
  ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]
```

Indexes

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
  ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ... ] )
  [ WITH ( storage_parameter = value [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
  [ WHERE predicate ]
```

Tablespaces

```
CREATE TABLESPACE tablespace_name [ OWNER user_name ] LOCATION 'directory'
```

Views

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
  [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
  AS query
```

```
CREATE VIEW v_example_character_type AS
SELECT col_char1, col_char2
  FROM example_character_type
 WHERE col_char1 = 'A';
```

Triggers

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
  ON table_name
  [ FROM referenced_table_name ]
  { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  [ WHEN ( condition ) ]
  EXECUTE PROCEDURE function_name ( arguments )
```

where *event* can be one of:

```
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
```

Example:

```
CREATE OR REPLACE FUNCTION test() RETURNS trigger AS $$
  BEGIN
    RETURN 0;
  END IF;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON example_character_type
  FOR EACH ROW
  EXECUTE PROCEDURE test();
```

Sequences

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment ]
  [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
  [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
  [ OWNED BY { table_name.column_name | NONE } ]
```

Example:

```
CREATE SEQUENCE example_sequence
  INCREMENT 1
  MINVALUE 1
  START 101 ;
```

Functions

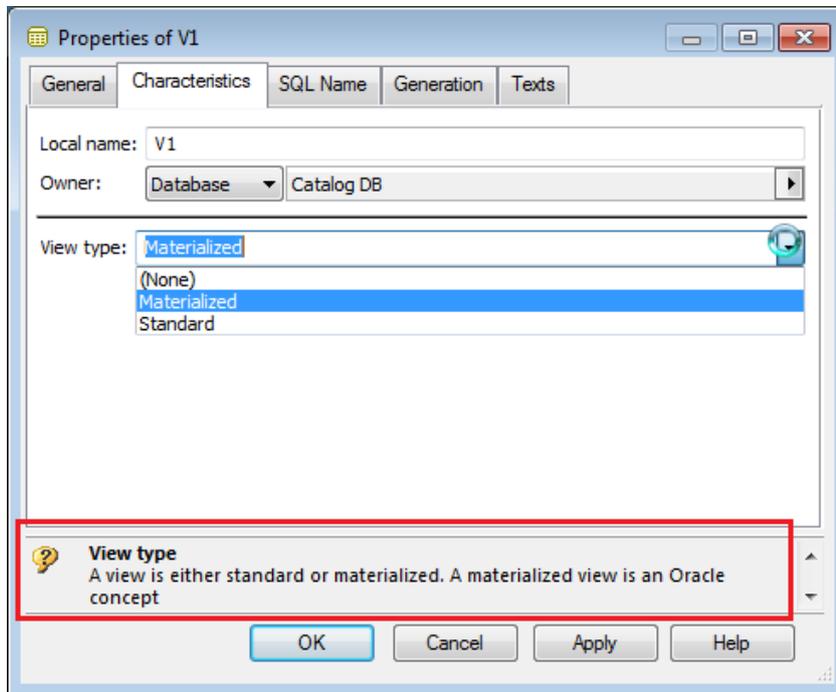
```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  [ RETURNS rettype
    | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
  [ WITH ( attribute [, ...] ) ]
```

Example:

```
CREATE OR REPLACE FUNCTION test() RETURNS trigger AS $$
  BEGIN
    RETURN i + 1;
  END IF;
$$ LANGUAGE plpgsql;
```

Materialized views

This concept is managed in MEGA by specifying the appropriate type to a view.



However, MEGA presents two restrictions:

- It is not possible to specify a tablespace for storage of the view.
- More generally, it is not possible to specify physical parameters.

Domains and types

These are two concepts that enable management of structures of elementary datatypes. These concepts exist for other DBMSs.

These concepts are not initially managed. A detailed study is necessary.

"Trigger events"

This concept enables extension of the trigger concept to a database.

MEGA does not currently offer suitable management of this concept.

Creating Datatype Packages

PostgreSQL datatypes

A complete description of datatypes is accessible from the following link:

<http://www.postgresql.org/docs/9.3/static/sql-commands.html>

 Complete list

The following sections provide additional information.

Name	Aliases	Description
<code>bigint</code>	<code>int8</code>	signed eight-byte integer
<code>bigserial</code>	<code>serial8</code>	autoincrementing eight-byte integer
<code>bit [(n)]</code>		fixed-length bit string
<code>bit varying [(n)]</code>	<code>varbit</code>	variable-length bit string
<code>boolean</code>	<code>bool</code>	logical Boolean (true/false)
<code>box</code>		rectangular box on a plane
<code>bytea</code>		binary data ("byte array")
<code>character [(n)]</code>	<code>char [(n)]</code>	fixed-length character string
<code>character varying [(n)]</code>	<code>varchar [(n)]</code>	variable-length character string
<code>cidr</code>		IPv4 or IPv6 network address

Name	Aliases	Description
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [<i>fields</i>] [(<i>p</i>)]		time span
json		JSON data
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
money		currency amount
numeric [(<i>p</i> , <i>s</i>)]	decimal [(<i>p</i> , <i>s</i>)]	exact numeric of selectable precision
path		geometric path on a plane
point		geometric point on a plane

Name	Aliases	Description
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(p)] [without time zone]		time of day (no time zone)
time [(p)] with time zone	timetz	time of day, including time zone
timestamp [(p)] [without time zone]		date and time (no time zone)
timestamp [(p)] with time zone	timestampz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot

Name	Aliases	Description
uuid		universally unique identifier
xml		XML data

Numeric types

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double_precision	8 bytes	variable-precision, inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

```
CREATE TABLE example_numeric_type
(
  col_bigint          bigint,
  col_decimal1       decimal,
  col_decimal2       decimal(8),
  col_decimal3       decimal(8,2),
  col_double_precision double precision,
  col_integer        integer,
  col_numeric1       numeric,
  col_numeric2       numeric(8),
  col_numeric3       numeric(8,2),
  col_real           real,
  col_serial         serial,
  col_bigserial      bigserial,
  col_smallserial    smallserial,
  col_smallint       smallint
);
```

Monetary types

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

```
CREATE TABLE example_monetary_type
(
  col_money    money
)
```

Character strings

Name	Description
character varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
text	variable unlimited length

```
CREATE TABLE example_character_type
(
  col_char1 char,
  col_char2 char(12),
  col_varchar1 varchar,
  col_varchar2 varchar(15),
  col_text text
)
```

Binary types

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

```
CREATE TABLE example_binary_type
(
  col_bytea bytea
)
```

Time types

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
timestamp [(p)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits
interval [fields] [(p)]	12 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits

```
CREATE TABLE example_datetime_type
(
  col_date          date,
  col_time1         time,
  col_time2         time(6),
)
```

```

col_time3          time with time zone,
col_time4          time(6) with time zone,
col_timestamp1     timestamp,
col_timestamp2     timestamp(6),
col_timestamp3     timestamp with time zone,
col_timestamp4     timestamp(6) with time zone,
col_interval1      interval,
col_interval2      interval(6),
col_interval3      interval day,
col_interval4      interval day(6)
)

```

Boolean types

Name	Storage Size	Description
boolean	1 byte	state of true or false

```

CREATE TABLE example_boolean_type
(
  col_boolean boolean
)

```

Geometric types

Name	Storage Size	Representation	Description
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line (not fully implemented)	((x1,y1),(x2,y2))
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
box	32 bytes	Rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

```

CREATE TABLE example_geometric_type
(
  col_point point,
  col_line line,
  col_lseg lseg,
  col_box box,
  col_path path,
  col_polygon polygon,
  col_circle circle
)

```

Network types

Name	Storage Size	Description
cidr	7 or 19 bytes	IPv4 and IPv6 networks
inet	7 or 19 bytes	IPv4 and IPv6 hosts and networks
macaddr	6 bytes	MAC addresses

```
CREATE TABLE example_network_type
(
  col_cidr cidr,
  col_inet inet,
  col_macaddr macaddr
)
```

Type : Bit String

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: `bit(n)` and `bit varying(n)`, where *n* is a positive integer. `bit` type data must match the length *n* exactly; it is an error to attempt to store shorter or longer bit strings. `bit varying` data is of variable length up to the maximum length *n*; longer strings will be rejected. Writing `bit` without a length is equivalent to `bit(1)`, while `bit varying` without a length specification means unlimited length.

```
CREATE TABLE example_bit_type
(
  col_bit1 bit,
  col_bit2 bit(3),
  col_bit_varying1 bit varying,
  col_bit_varying2 bit varying(5)
)
```

Type : Other

```
CREATE TABLE example_other_type
(
  col_json json,
  col_uuid uuid,
  col_xml xml
)
```

Physical Parameters

Colors used:

Gray: parameters where reverse engineering is non-directional.

Red: parameters not processed.

Green: parameters reverse engineered.

Tables

- ✚ Clause type: table properties
 - `POSTGRESQL_TBL_TEMPORARY` (boolean)
 - `POSTGRESQL_TBL_LOG` (boolean)
 - `POSTGRESQL_TBL_NOT_EXISTS` (boolean)
 - `POSTGRESQL_TBL_COMMIT` (PRESERVE ROWS, DELETE ROWS, DROP)
 - `POSTGRESQL_TBL_OIDS` (yes, no)
 - [Link to a data group](#)
 - [Link to a clause type storage parameter](#)

Example:

```
CREATE TABLE example_with_storage_parameter (code char(5)) WITH (fillfactor=60,
autovacuum_enabled);
```

- ✚ Clause type: table constraint
 - `POSTGRESQL_CONSTRAINT_NAME` (String)
 - `POSTGRESQL_CONSTRAINT_TYPE` (UNIQUE, PRIMARY KEY, EXCLUDE, FOREIGN KEY)
 - `POSTGRESQL_EXPRESSION` (String)
 - `POSTGRESQL_DEFERRABLE` (yes, no)
 - `POSTGRESQL_DEFERRED` (yes, no)
 - `POSTGRESQL_MATCH_TYPE` (Full, partial, simple)
 - `POSTGRESQL_UPDATE_ACTION` (String)
 - `POSTGRESQL_DELETE_ACTION` (String)
 - `POSTGRESQL_CONDITION` (String)
 - [Link to a clause type storage parameter](#)
 - [Link to a clause type column option \(multiple clause\)](#)
 - [Link to a data group](#)
 - [Link to a table](#)
 - [Link to a column](#)

```
CREATE TABLE table_with_constraint (
  did integer CHECK (did > 100),
  name varchar(40)
);
```

Columns

- ✚ Clause type: column properties

- **POSTGRESQL_COLLATE** (String)

Example:

```
CREATE TABLE table_with_collation (c1 serial, c2 text COLLATE "C");
```

✚ Clause type: column constraint

- **POSTGRESQL_CONSTRAINT_NAME** (String)
- **POSTGRESQL_CONSTRAINT_TYPE** (UNIQUE, PRIMARY KEY, FOREIGN KEY)
- **POSTGRESQL_CONSTRAINT_EXPRESSION** (String)
- **POSTGRESQL_DEFERRABLE** (yes, no)
- **POSTGRESQL_DEFERRED** (yes, no)
- **POSTGRESQL_MATCH_TYPE** (Full, partial, simple)
- **POSTGRESQL_ACTION** (String)
- Link to a clause type storage parameter
- Link to a data group
- Link to a table
- Link to a column

Indexes

✚ Clause type: index properties

- **POSTGRESQL_IDX_CONCURRENTLY** (boolean)
- **POSTGRESQL_IDX_METHOD** (btree,hash,gist,spgist,gin)
- **POSTGRESQL_PREDICATE** (String)
- **POSTGRESQL_EXPRESSION** (String)
- Link to a clause type storage parameter
- Link to a data group

✚ Clause type: column option

Examples:

```
CREATE INDEX example_character_idx ON example_character_type USING btree (col_char2) WITH (fillfactor=60)
```

```
CREATE INDEX index_with_tbs ON example_character_type (col_varchar1) TABLESPACE dbspace;
```

Tablespaces

✚ Clause type: tablespace properties

- **POSTGRESQL_TBS_LOCATION** (String)

```
CREATE TABLESPACE dbspace LOCATION 'C:\Travail\2014\Database\PostGreSql';
```

```
CREATE TABLE table_with_tablespace (
    id serial,
    name text,
    location text
) TABLESPACE dbspace;
```

Generic clause types

🔧 Clause type: storage parameter

- `POSTGRESQL_FILLFACTOR` (integer)
- `POSTGRESQL_VACUUM` (boolean)
- `POSTGRESQL_THRESHOLD` (integer)
- `POSTGRESQL_SCALE_FACTOR` (float)
- `POSTGRESQL_ANALYSE_THRESHOLD` (integer)
- `POSTGRESQL_ANALYSE_SCALE_FACTOR` (double)
- `POSTGRESQL_COST_DELAY` (integer)
- `POSTGRESQL_COST_LIMIT` (integer)
- `POSTGRESQL_FREEZE_MIN_AGE` (integer)
- `POSTGRESQL_FREEZE_MAX_AGE` (integer)
- `POSTGRESQL_FREEZE_TABLE_AGE` (integer)
- `POSTGRESQL_MULTIXACT_FREEZE_MIN_AGE` (integer)
- `POSTGRESQL_MULTIXACT_FREEZE_MAX_AGE` (integer)

🔧 Clause type: column option

- [Link to a column](#)
- `POSTGRESQL_EXPRESSION` (String)
- `POSTGRESQL_COLLATE` (String)
- `POSTGRESQL_CLASS` (String)
- `POSTGRESQL_SORT` (ASK, DESK)
- `POSTGRESQL_NULL` (NULLS, NULLS FIRST, NULLS LAST)

Unmanaged physical parameters

🔧 INHERITS

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. Use of `INHERITS` creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

LIKE

The `LIKE` clause specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints. Unlike `INHERITS`, the new table and original table are completely decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table. Default expressions for the copied column definitions will only be copied if `INCLUDING DEFAULTS` is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults.

TYPE_NAME

Creates a *typed table*, which takes its structure from the specified composite type (name optionally schema-qualified). A typed table is tied to its type; for example the table will be dropped if the type is dropped (with `DROP TYPE ... CASCADE`). When a typed table is created, then the data types of the columns are determined by the underlying composite type and are not specified by the `CREATE TABLE` command. But the `CREATE TABLE` command can add defaults and constraints to the table and can specify storage parameters.

Properties pages

An Sql Clause contains the definition of all physical parameters of all DBMSs managed by MEGA. To display the correct properties page according to the DBMS and the clause type, a specific properties page must be generated for each DBMS version.

Create a MetaAttributGroup "PostgreSQL_93_Parameters"

This should be connected to the SQL Clause MetaClass

Create a MetaPropertyPage "PostgreSQL_93_Parameters"

Setup as follows:

[Page]

Order=10

[Filter]

Condition = (ItemCount(~ISI(gRH(HTB8[IsClauseForPostgreSQL93]))>0)

[Template]

ExtraParam=IncludeProfile(~ESI(CMH(HHX7[GeneratePropertyPageForPostgreSQL93]),Origin(Macro)

Where `IsClauseForPostgreSQL93` is a query and `GeneratePropertyPageForPostgreSQL93` is a macro. We can use as models the query `IsClauseForPostgre93` and the macro `GeneratePropertyPageForPostgreSQL93`.

Clause type: table constraint

DB2 Version 10.5: Syntax supported by HOPEX

This document details concepts of DB2 V10.5 recognized by MEGA.

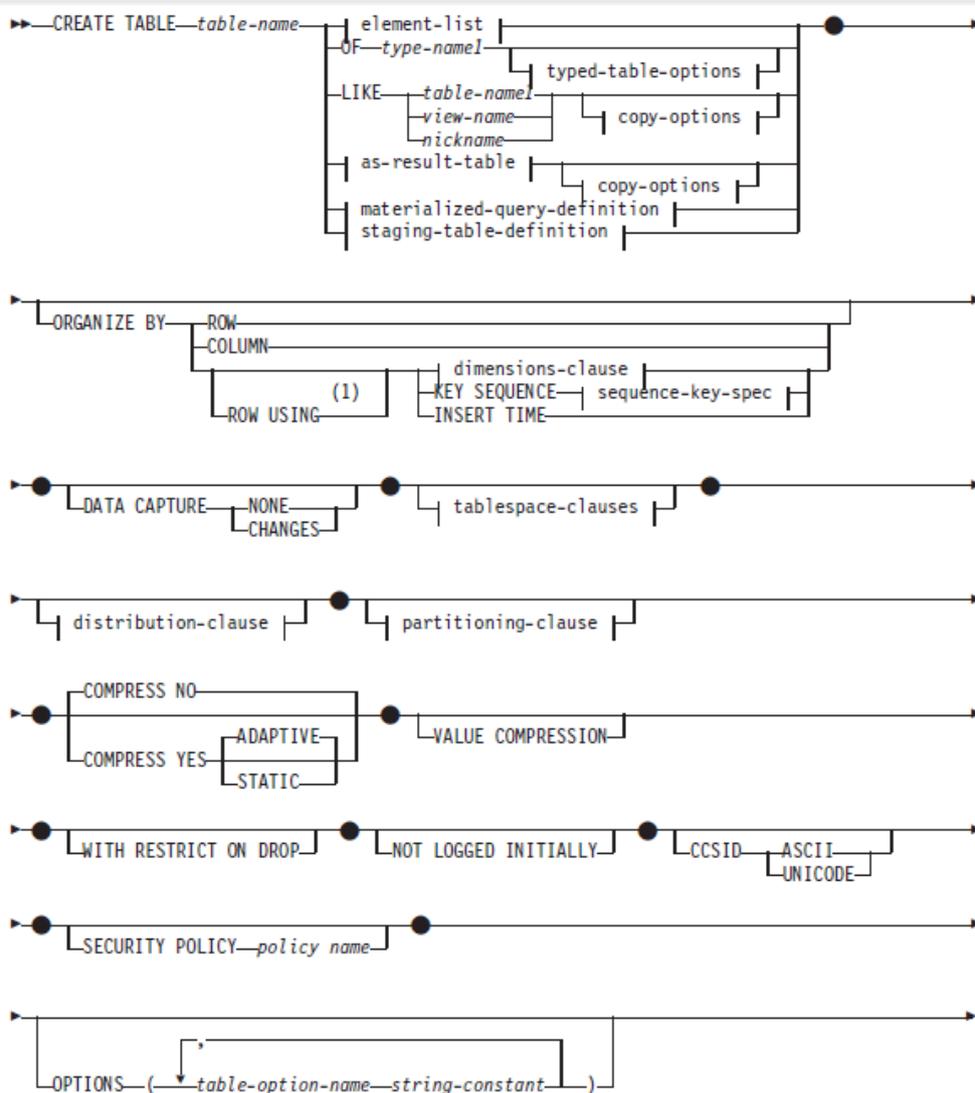
Detail of Concepts Managed by DB2 V10.5

All SQL commands are available at the following address:

http://public.dhe.ibm.com/ps/products/db2/info/vr105/pdf/en_US/DB2SQLRefVol1-db2s1e1051.pdf

The logo for MEGA, featuring the word "mega" in a bold, blue, lowercase sans-serif font. To the right of the text is a stylized circular graphic composed of two concentric, light-colored arcs that form a partial circle, resembling a stylized 'D' or a ring.

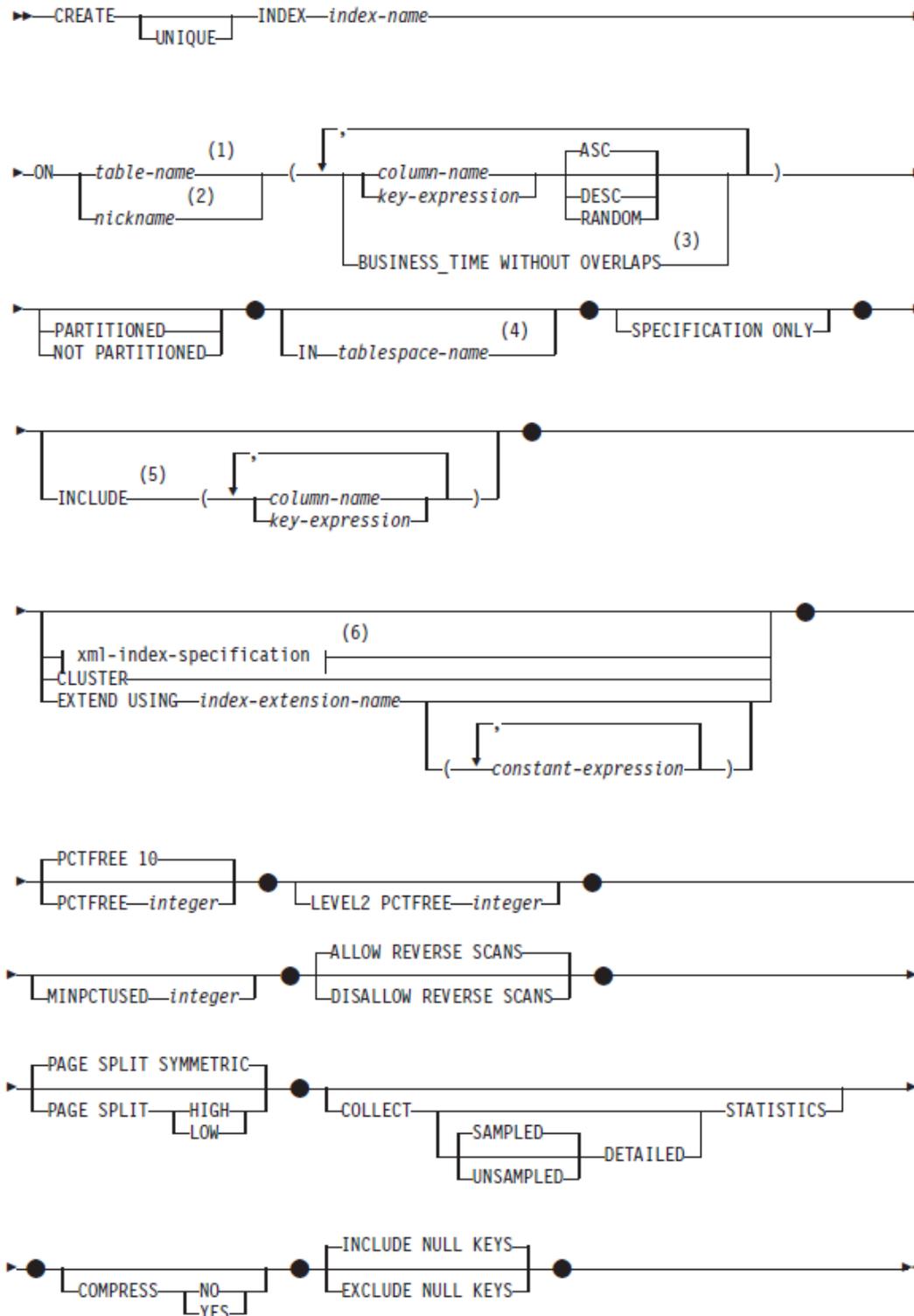
Tables



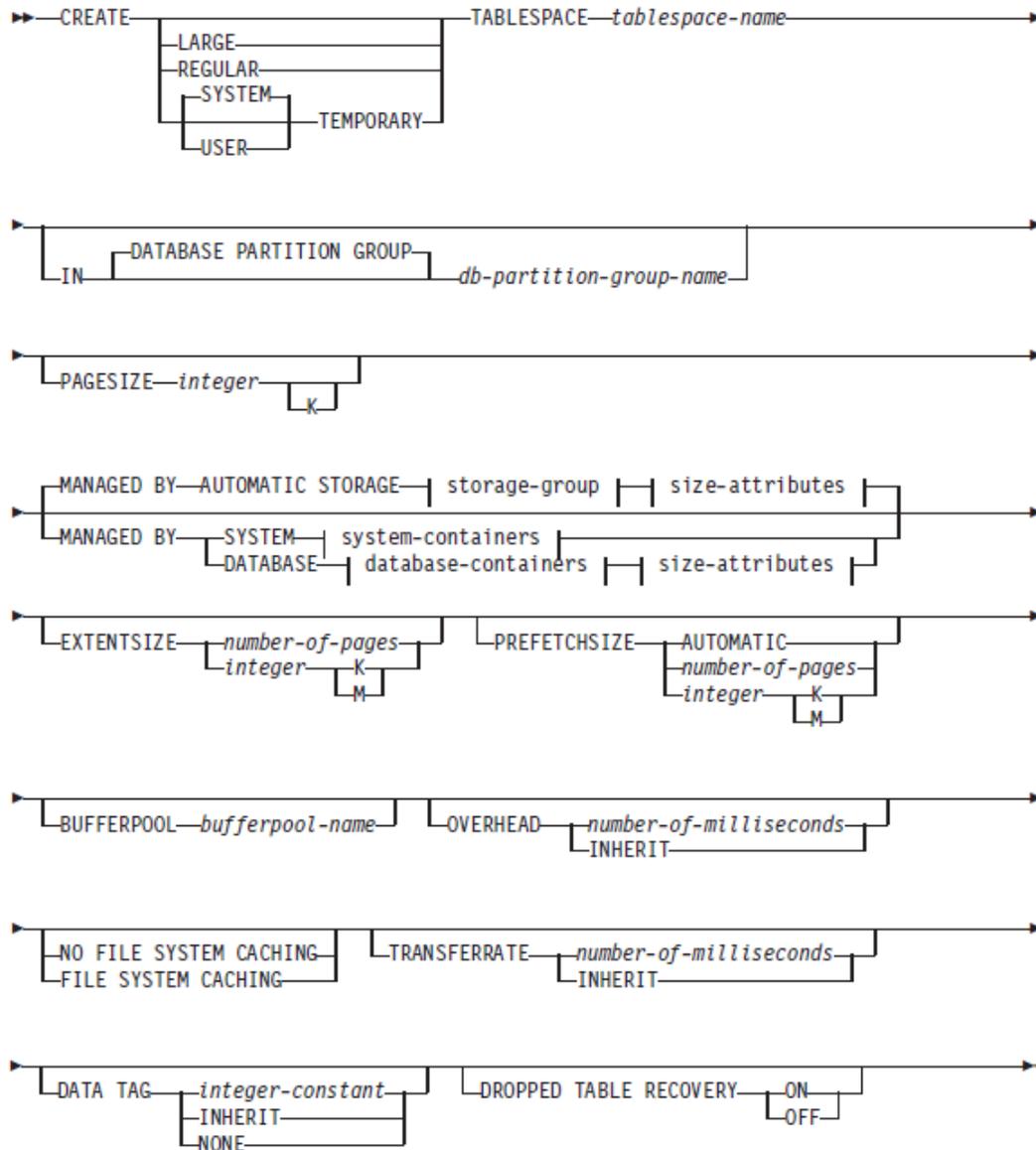
CREATE GLOBAL TEMPORARY TABLE

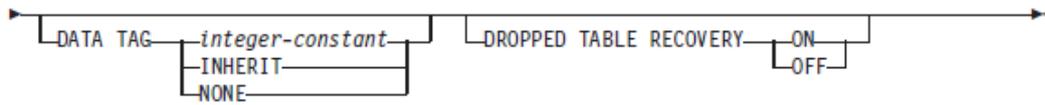
This new table type of DB2V10.5 is not supported by MEGA.

Indexes

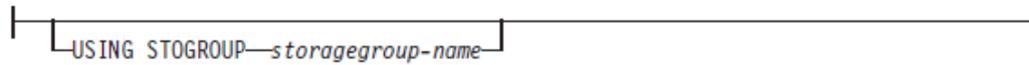


Tablespaces

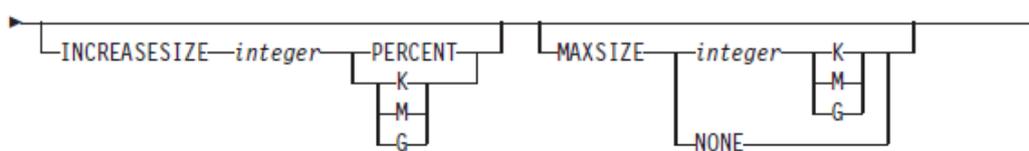
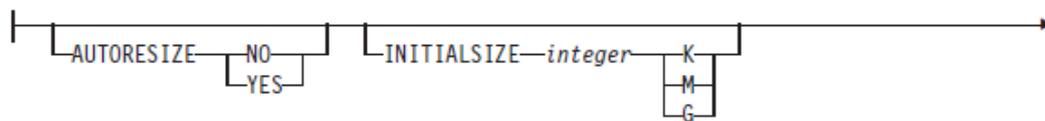




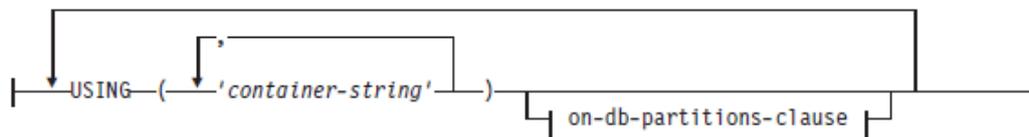
storage-group:



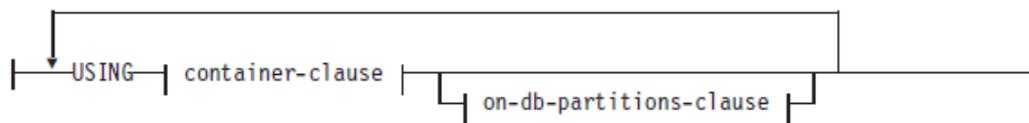
size-attributes:



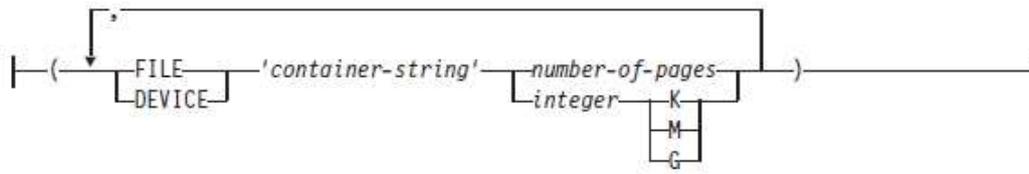
system-containers:



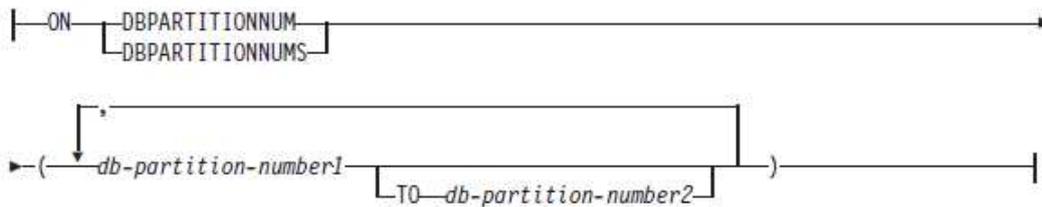
database-containers:



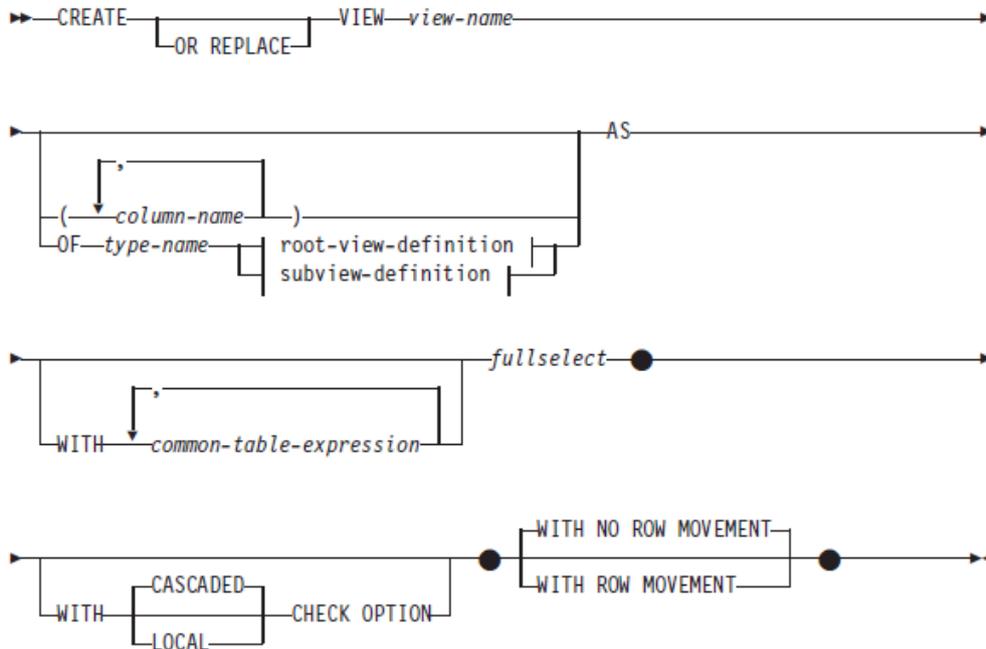
container-clause:



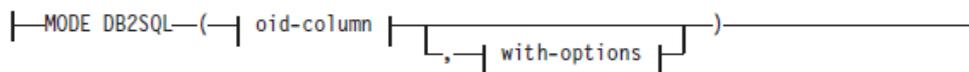
on-db-partitions-clause:



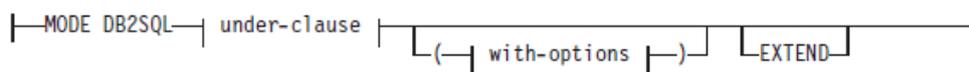
Views



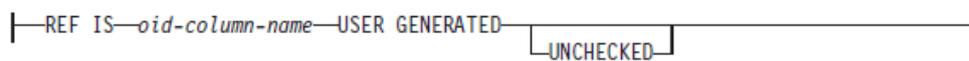
root-view-definition:



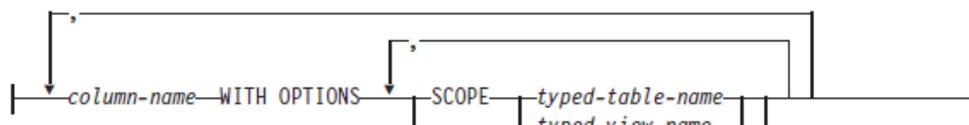
subview-definition:



old-column:



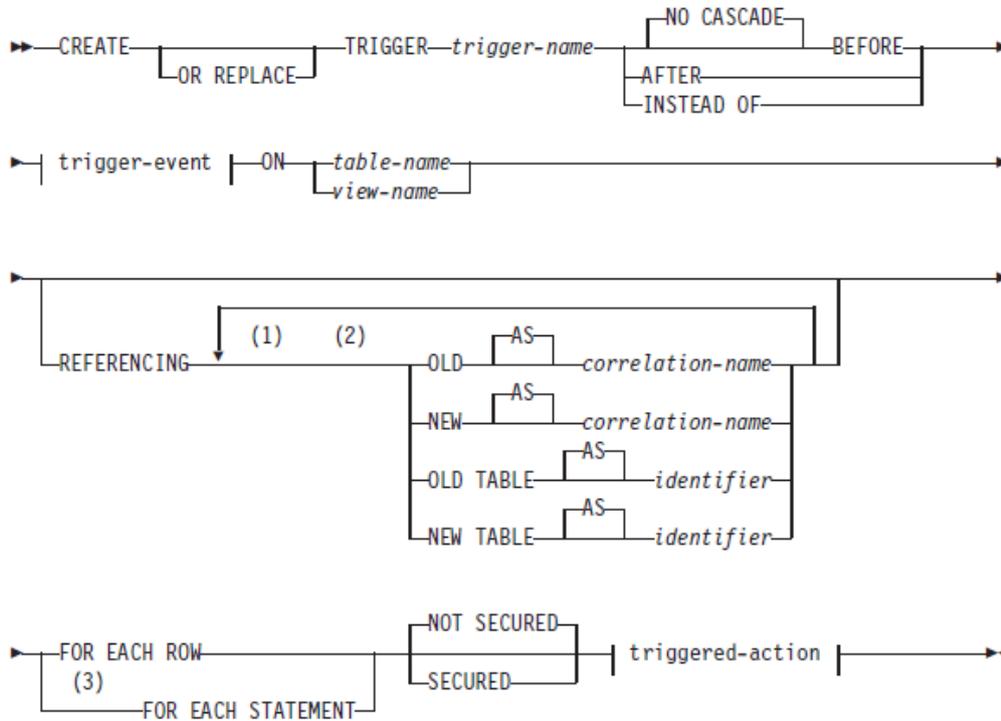
with-options:



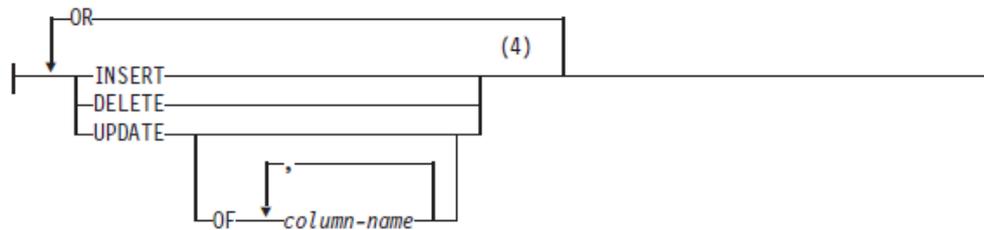
Example:

```
CREATE VIEW MA_PROJ
AS SELECT PROJNO, PROJNAME, RESPEMP
FROM PROJECT WHERE SUBSTR(PROJNO, 1, 2) = ' MA'
```

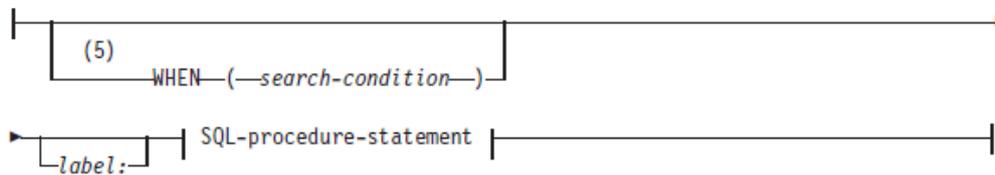
Triggers

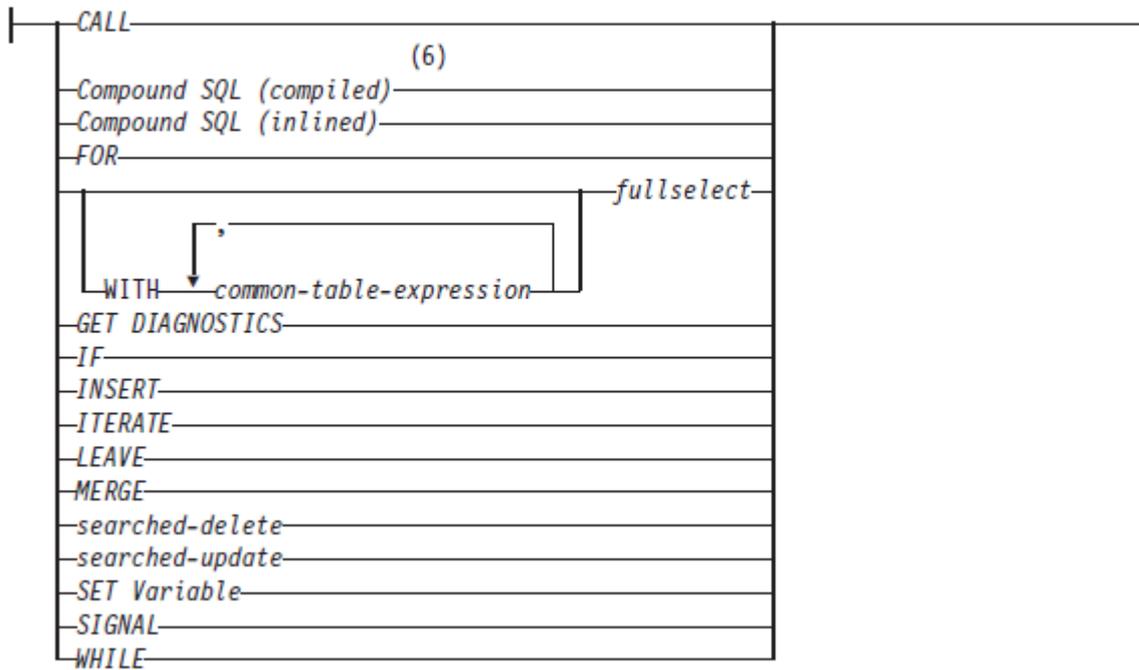


trigger-event:



triggered-action:



SQL-procedure-statement:

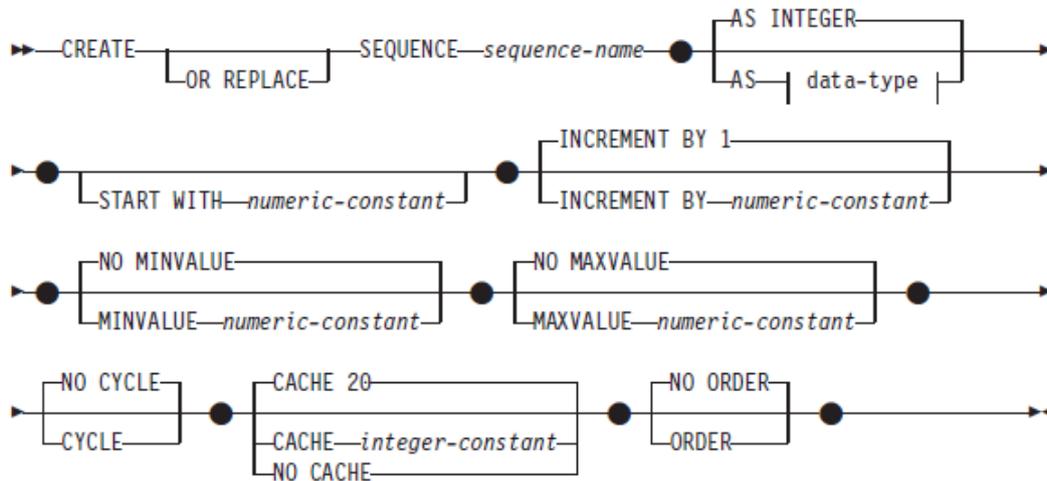
Example:

```

CREATE TRIGGER REORDER
AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
REFERENCING NEW TABLE AS NTABLE
FOR EACH STATEMENT
BEGIN ATOMIC
SELECT ISSUE_SHIP_REQUEST (MAX_STOCKED - ON_HAND, PARTNO)
FROM NTABLE
WHERE (ON_HAND < 0.10 * MAX_STOCKED) ;END

```

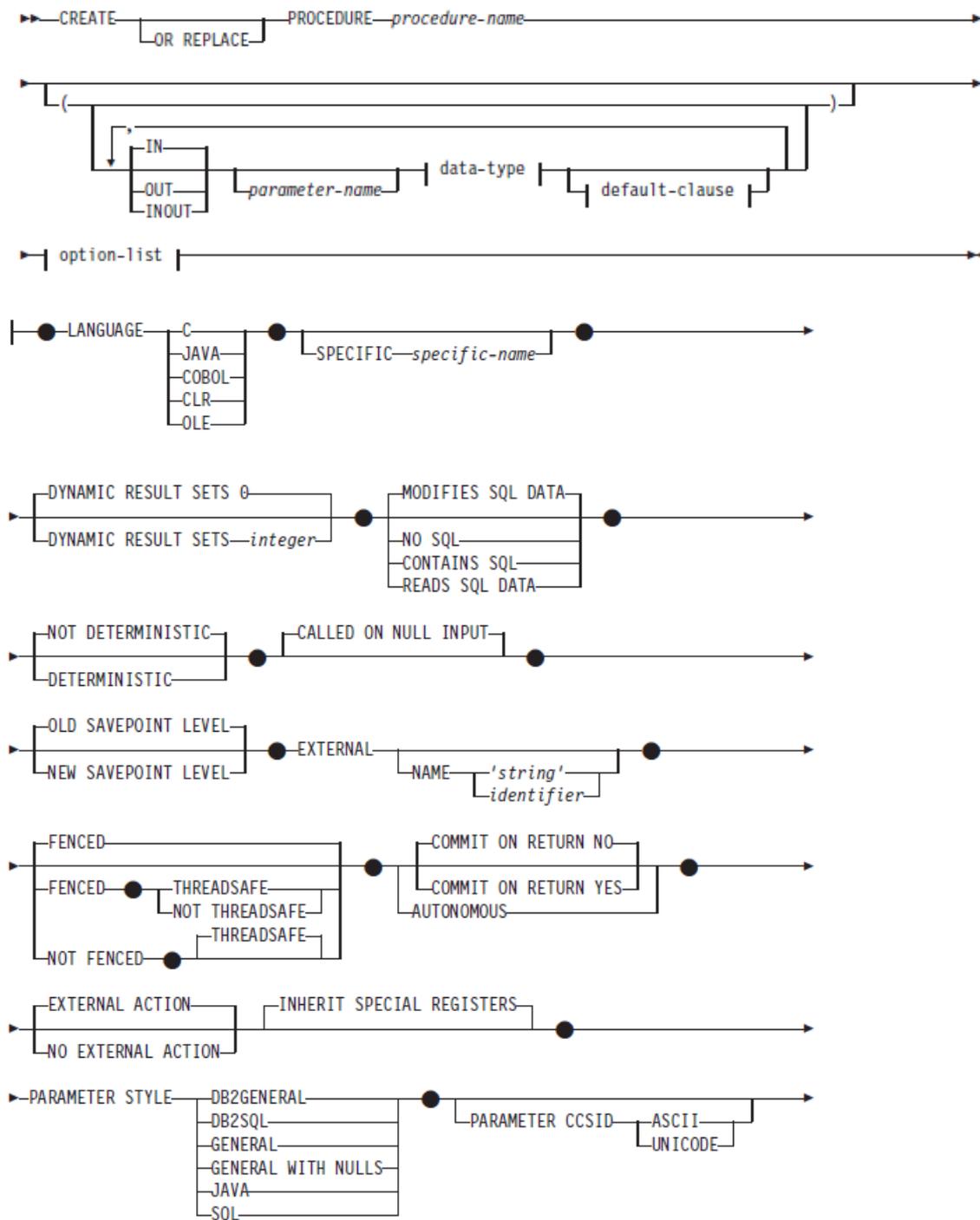
Sequences



Example:

```
CREATE SEQUENCE ORG_SEQ
START WITH 1
INCREMENT BY 1
NO MAXVALUE
NO CYCLE
  CACHE 24
```

Procedures



Example:

```

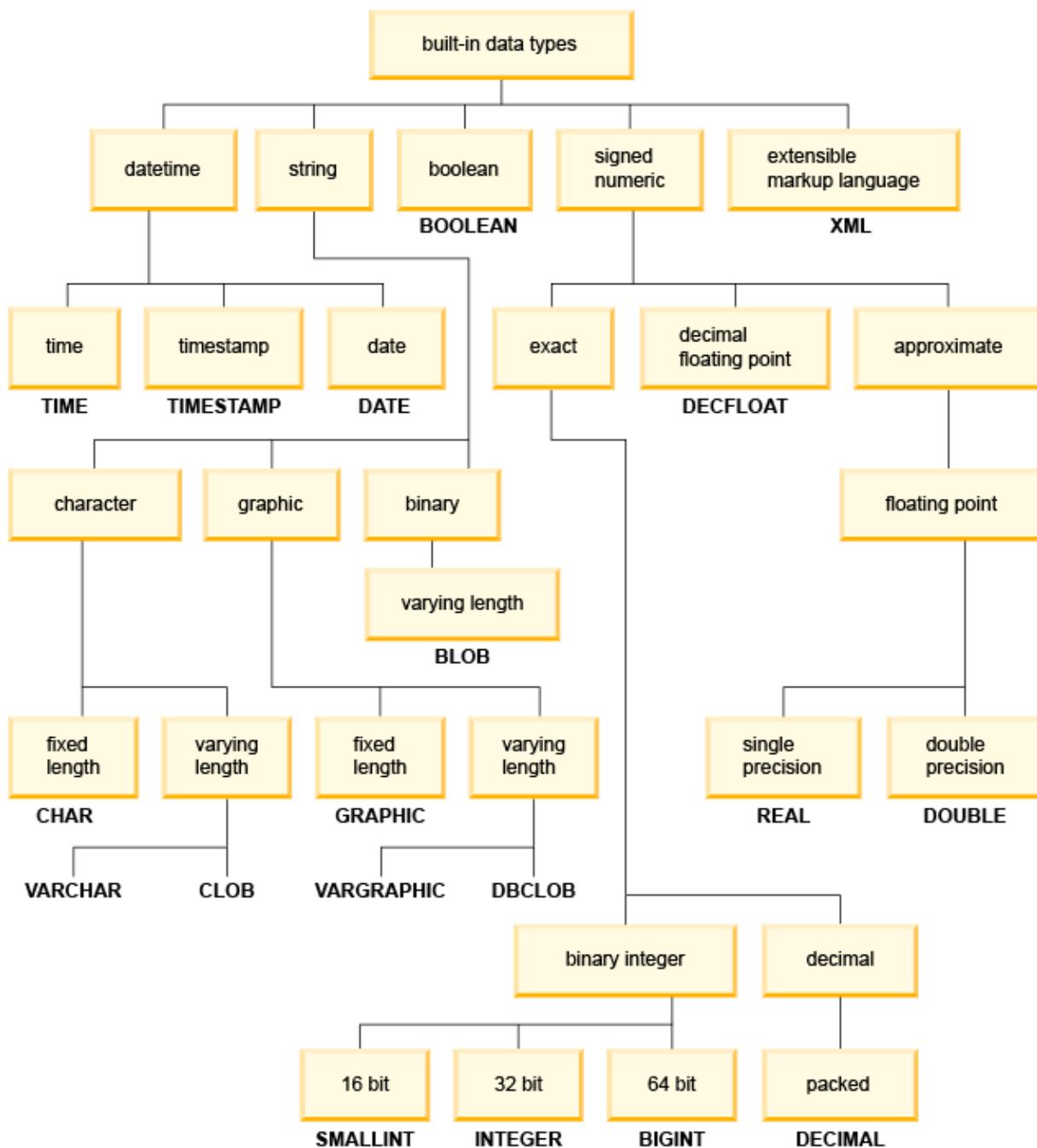
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,
OUT COST DECIMAL(7,2),
OUT QUANTITY INTEGER)
EXTERNAL NAME ' parts.onhand'
LANGUAGE JAVA PARAMETER STYLE JAVA
    
```

Creating Datatype Packages

DB2V10.5 Datatypes

A complete description of datatypes is accessible from the following link:

http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.sql.ref.doc/doc/r0008483.html?cp=SSEPGG_10.5.0%2F2-12-2-3&lang=fr



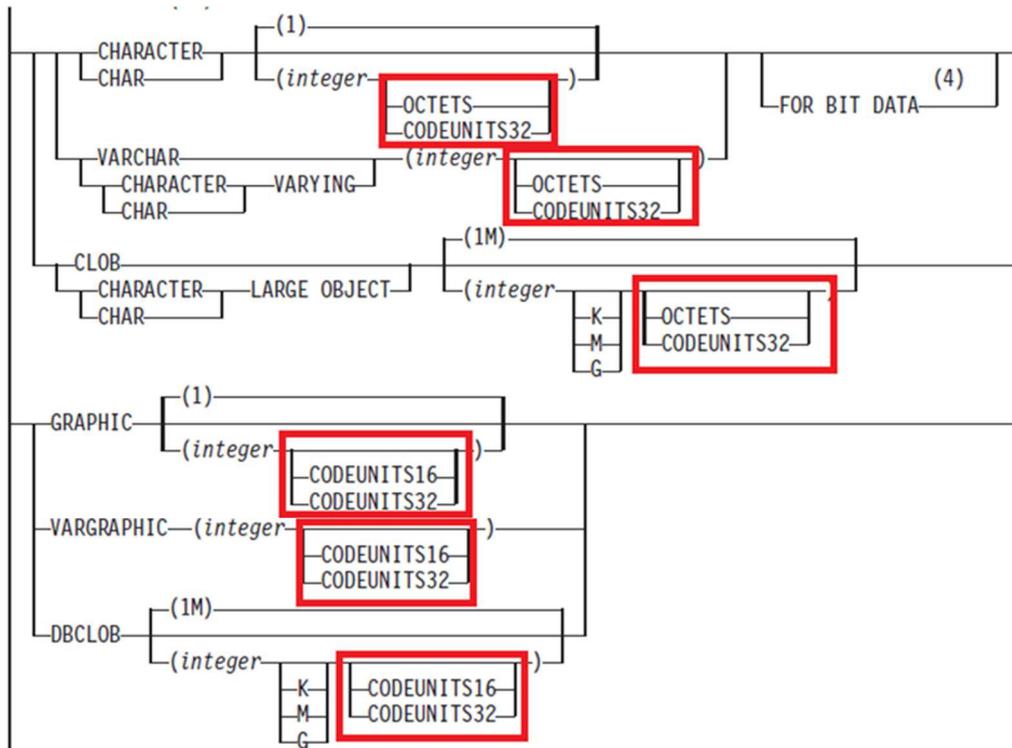
NCHAR, NVARCHAR, NCLOB

The NCHAR, NVARCHAR, NCLOB datatypes are not supported by MEGA.

OCTETS, CODEUNITS32, CODEUNITS16

OCTETS, CODEUNITS32 and CODEUNITS16 are units of length for the character string datatype.

They are not supported by MEGA.



LONG VARCHAR, LONG VARGHAPHIC

The LONG VARCHAR and LONG VARGRAPHIC datatypes are deprecated in version 10.5 but remain accepted in DB2 10.5 syntax.

Physical Parameters

Colors used:

Gray: parameters where reverse engineering is non-directional.

Red: parameters not processed.

Green: parameters reverse engineered.

Tables

- ✚ Clause type : table_check
 - CHECK NAME (String)
 - ENFORCED(Enumeration: ENFORCED,NOT ENFORCED,NOT ENFORCED TRUSTED, NOT ENFORCED NOT TRUSTED)
- ✚ Clause type : table_block
 - DATA CAPTURE (Enumeration: CHANGES,NONE)
 - NOT LOGGED INITIALLY(Boolean)
 - WITH RESTRICT ON DROP
 - COMPRESSION MODE (Enumeration: COMPRESSION NO,COMPRESSION YES ADAPTIVE,COMPRESSION YES STATIC)
 - VALUE COMPRESSION (Boolean)
 - CCSID (UNICODE,ASCII)
 - SECURITY POLICY(String)
 - OPTIONS (VarChar)

 - AS ROW (Enumeration : BEGIN,END)
- ✚ Clause type : organize_clause
 - ORGANIZE BY (Enumeration :ROW,COLUMN,DIMENSIONS,KEY SEQUENCE,INSERT TIME)
 - ROW USING (Boolean)
 - ALLOW OVERFLOW(Boolean)
 - PCTFREE (Integer)
 - Not processed:
 -
- ✚ Clause type : organize_sequence_key_clause
 - COLUMN (Link)
 - STARTING_CONSTANT
 - ENDING_CONSTANT
- ✚ Clause type : distribution_clause
 - DISTRIBUTE BY (Enumeration :HASH,REPLICATION)
 - DISTRIBUTION COLUMNS(Link)
- ✚ Clause type : partitioning_clause
 - RANGE(Boolean)
- ✚ Clause type : partitioning_range_clause
 - PARTITION NAME(String)

- **EVERY(String)**
- Used Data Group(Link)
- ✚ Clause type: boundary_clause
 - BOUNDARY INCLUSIVE(INCLUSIVE,EXCLUSIVE)
 - BOUNDARY TYPE(STARTING,ENDING)
 - BOUNDARY VALUES(String)
- ✚ Clause type: tablespace_option
 - **CYCLE(Boolean)**
 - Used Data Group (link)
 - Used Data Group 2 (link)
 - Used Data Group 3 (link)
- ✚ Clause type : partition_expression
 - COLUMN(Link)
 - NULLS ORDER (Enumeration : FIRST, LAST)
- ✚ Clause type : table_period_defintion
 - PERIOD MODE (Enumeration : BUSINESS_TIME, SYSTEM_TIME)
 - BEGIN COLUMN(Link)
 - END COLUMN(Link)

Columns

- ✚ Clause type : column_block
 - COMPACT (Boolean)
 - COMPRESS (Boolean)
 - INLINE LENGTH (Boolean)
 - LOGGED (String)
 - IMPLICITLY HIDDEN (Enumeration : IMPLICITLY HIDDEN, NOT HIDDEN)
- ✚ Clause type : generated_column_spec
 - GENERATED (Enumeration : GENERATED ALWAYS, GENERATED DEFAULT)
 - GENERATION TYPE (Enumeration : AS GENERATED EXPRESSION, AS ROW CHANGE TIMESTAMP, AS ROW TRANSACTION TIMESTAMP, AS ROW TRANSACTION STARTD ID)
 - START WITH (String)
 - INCREMENT (String)
 - MAX VALUE (OPENED ENUMERATION : NO MAXVALUE)
 - MIN VALUE (OPENED ENUMERATION : NO MINVALUE)

- CACHE (OPENED ENUMERATION : NO CACHE)
- CYCLE (Boolean)
- ORDER (Boolean)
- GENERATE EXPRESSION (VarChar)

Indexes

✚ Clause type : index_block

- BUSINESS_TIME WITHOUT OVERLAPS (Boolean)
- PARTITIONED (Enumeration : PARTITIONED ,NOT PARTITIONED)
- Used Data Group (Link)
- EXTEND USING (String)
- VALUE LIST (VarChar)
- PCTFREE (Percent)
- LEVEL2 PCTFREE (Percent)
- MINPCTUSED (String)
- REVERSE SCANS (ALLOW,DISALLOW)
- PAGE SPLIT (HIGHT,LOW,SYMMETRIC)
- COLLECT(STATISTICS,DETAILED STATISTICS,SAMPLED DETAILED STATISTICS,UNSAMPLED DETAILED STATISTICS)
- COMPRESS(YES,NO)
- NULL KEYS (INCLUDE NULL KEYS, EXCLUDE NULL KEYS)
- INCLUDE(Link)

Tablespaces

✚ Clause type : tablespace_block

- STORE(LARGE,LOB,LONG,REGULAR,SYSTEM TEMPORARY,USER TEMPORARY)
- NODEGROUP(Boolean)
- NODEGROUP NAME(String)
- PAGESIZE(String)
- MANAGED(MANAGE BY AUTOMATIC STORAGE,MANAGED BY DATABASE,MANAGED BY SYSTEM)
- EXTENTSIZE(String)
- PREFETCHSIZE(String)
- BUFFERPOOL(String)
- OVERHEAD(String)
- TRANSFERRATE(String)

- DROPPED TABLE RECOVERY(ON,OFF)
- AUTORESIZE(Boolean)
- INCREASESIZE(String)
- INITIALSIZE(String)
- MAXSIZE(String)
- FILE SYSTEM CACHING (FILE SYSTEM CACHING
- NO FILE SYSTEM CACHING (NO FILE SYSTEM CACHING,NO NO FILE SYSTEM CACHING)
- DATA TAG (String)
- ✚ Clause type : containers
 - Container-type(FILE,DEVICE)
 - Container-strings(VarChar)
 - Container –size (String)
 - container-numbers(String)

Unmanaged physical parameters

✚ AS

Specifies that the columns of the table are based on the attributes of the structured type identified by type-name1. If type-name1 is specified without a schema name, the type name is resolved by searching the schemas on the SQLpath (defined by the FUNCPATH preprocessing option for static SQL and by the CURRENT PATH register for dynamic SQL). The type name must be the name of an existing user-defined type (SQLSTATE 42704) and it must be an instantiable structured type (SQLSTATE 428DP) with at least one attribute (SQLSTATE 42997).

✚ LIKE

Specifies that the columns of the table have exactly the same name and description as the columns of the identified table (table-name1), view (view-name) or nickname (nickname). The name specified after LIKE must identify a table, view or nickname that exists in the catalog, or a declared temporary table. A typed table or typed view cannot be specified (SQLSTATE428EC).

✚ TYPE_NAME

Creates a typed table, which takes its structure from the specified composite type (name optionally schema-qualified). A typed table is tied to its type; for example the table will be dropped if the type is dropped (with DROP TYPE ... CASCADE). When a typed table is created, then the data types of the columns are determined by the underlying composite type and are not specified by the CREATE TABLE command. But the CREATE TABLE command can add defaults and constraints to the table and can specify storage parameters.

UNIQUE Constraint

Creating a UNIQUE constraint automatically creates a unique index of the same name. It is therefore recommended not to generate unique indexes.

In the generation options of a database, select one of the two options “Create Index” and “Create Index[Unique]”.

CREATE INDEX:	<input type="checkbox"/> no	<input checked="" type="checkbox"/> yes
CREATE PROCEDURE:	<input type="checkbox"/> no	<input checked="" type="checkbox"/> yes
CREATE INDEX [PK]:	<input type="checkbox"/> no	<input checked="" type="checkbox"/> yes
CREATE INDEX[UNIQUE]:	<input type="checkbox"/> no	<input checked="" type="checkbox"/> yes

The same rule applies on PK constraints.

Organize_sequence_key-clause Clause

The “organize_sequence_key-clause” clause is reverse engineered into a check constraint.

For example:

Became:

A Unique index is also created on columns used in “organize_sequence_keyclause” clauses.

In DB2V10.5 reverse engineering of this clause is not supported.

‘EVERY \ DB2’ Properties

The “EVERY” property has no physical value in DB2 system tables. If this property is declared on a partition, it means that the partition is composed of several partitions.

For example:

```
CREATE TABLE PARTITION_range_4  
  
(  
ID INTEGER,  
CONTENTS CLOB  
)  
PARTITION BY RANGE (ID) (STARTING 1 ENDING 100, STARTING 101 ENDING 400 EVERY 100)
```

Is equal to:

```
CREATE TABLE PARTITION_range_4  
  
(  
ID INTEGER,  
CONTENTS CLOB  
)  
PARTITION BY RANGE (ID) (STARTING 1 ENDING 100, STARTING 101 ENDING 200, STARTING 201 ENDING 300, STARTING  
301 ENDING 400)
```