

---

## **AIMMS Language Reference - Communicating with Databases**

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit [www.aimms.com](http://www.aimms.com) or order your hard-copy at [www.lulu.com/aimms](http://www.lulu.com/aimms).

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

Email: [info@aimms.com](mailto:info@aimms.com)  
WWW: [www.aimms.com](http://www.aimms.com)

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation.  $\text{T}_{\text{E}}\text{X}$ ,  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ , and  $\text{A}_{\text{M}}\text{S}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$  are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

**Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.**

**In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.**

This documentation was typeset by Paragon Decision Technology B.V. using  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  and the LUCIDA font family.

## Chapter 25

# Communicating With Databases

One of the most important capabilities of the READ and WRITE statements in AIMMS is its ability to transfer data with ODBC- and OLE DB-compliant databases. Although there are similarities between the basic concepts of data storage in databases and those in AIMMS, they are sufficiently different to justify a separate chapter in this manual.

*Communicating with databases*

This chapter deals with the intricacies of data transfer from and to databases. It first discusses the link between data in AIMMS and a table in a database. Then it explains the database-specific requirements regarding the READ and WRITE statements. Next comes a discussion on how to access stored procedures, followed by a description how to send SQL statements directly to a particular data source.

*This chapter*

---

### 25.1 The DATABASE TABLE declaration

You can make a database table known to AIMMS by means of a DATABASE TABLE declaration in your application. Inside this declaration you can specify the ODBC or OLE DB data source name of the database and the name of the database table from which you want to read, or to which you want to write. The list of attributes of a DATABASE TABLE is given in Table 25.1.

*Database tables*

Attribute	Value-type	See also page
INDEX DOMAIN	<i>index-domain</i>	42
DATA SOURCE	<i>string-expression</i>	
TABLE NAME	<i>string-expression</i>	
OWNER	<i>string-expression</i>	
PROPERTY	ReadOnly	
MAPPING	<i>mapping-list</i>	
TEXT	<i>string</i>	19
COMMENT	<i>comment string</i>	19
CONVENTION	<i>convention</i>	462

Table 25.1: DATABASE TABLE attributes

The mandatory `DATA SOURCE` attribute specifies the ODBC or OLE DB data source name of the database you want to link with. Its value must be a string or a string parameter. If you are unsure about the data source name by which a particular database is known, AIMMS will help you. While completing the declaration form of a database table, AIMMS will automatically let you choose from the available data sources on your system using the `DATA SOURCE` wizard. AIMMS supports the following data source types:

*The `DATA SOURCE` attribute*

- ODBC file data sources (.dsn extension),
- OLE DB connections (.udl extension), and
- ODBC user and system data sources (no extension).

In addition, you can specify the name of an AIMMS string parameter, holding the name of any of the above data source types. If the data source you are looking for is not available in this list, you can set up a link to that database from within the wizard.

With the `TABLE NAME` attribute you must specify the name of the table or view within the data source to which the `DATABASE TABLE` is mapped. Once you have provided the `DATA SOURCE` attribute, the `TABLE NAME` wizard will let you select any table or view available in the specified data source.

*The `TABLE NAME` attribute*

The following declaration illustrates the simplest possible `DATABASE TABLE` declaration.

*Example*

```

DATABASE TABLE:
  identifier : RouteData
  data source : "Topological Data"
  table name : "Route Definition" ;

```

It will connect to an ODBC user or system data source called “Topological Data”, and in that data source search for a table named “Route Definition”.

The `OWNER` attribute is for advanced use only. By default, when connecting to a database server, you will have access to all tables and stored procedures which are visible to you. In case a table name appears more than once, but is owned by different users, by default a connection is made to the table instance owned by yourself. By specifying the `OWNER` attribute you can gain access to the table instance owned by the indicated user.

*The `OWNER` attribute*

With the `PROPERTY` attribute of a `DATABASE TABLE` you can specify whether the declared table is `ReadOnly`. Specifying a database table as `ReadOnly` will prevent you from inadvertently modifying its content. If you do not provide this property, the database table will default to read-write permissions unless the server does not allow write access.

*The `PROPERTY` attribute*

By default, AIMMS tries to map the column names used in a database table onto the AIMMS identifiers of the same name. Such an implicit mapping is, of course, not always possible. When you link to an existing database that was not specifically designed for your AIMMS application, it is very likely that the column names do not correspond to the names of your AIMMS identifiers. Therefore, the MAPPING attribute lets you override this default. The database columns explicitly mapped through the MAPPING attribute are added to the set of implicit mappings constructed by AIMMS. The column names from the database table used in a mapping list must be quoted.

*The MAPPING attribute*

The following declarations demonstrate the use of mappings in a DATABASE TABLE declaration. This example assumes the set and parameter declarations of Section 24.1 plus the existence of the relation Routes given by

*Example*

```
SET:
  identifier : Routes
  subset of  : (Cities, Cities) ;
```

The following mapped database declaration will take care of the necessary column to identifier mapping.

```
DATABASE TABLE:
  identifier : RouteData
  data source : "Topological Data"
  table name : "Route Definition"
  mapping :
    "from"      --> i,                ! name substitution
    "to"        --> j,
    "dist"      --> Distance(i,j),
    "fcost"     --> TransportCost(i,j,'fixed'), ! slicing
    "vcost"     --> TransportCost(i,j,'variable'),
    ("from","to") --> Routes ;        ! mapping to relation
```

The first three lines of the MAPPING attribute provide a simple name translation from a column in the database table to an AIMMS identifier. You can only use this type of mapping if the structural form of the database table (i.e. the primary key) coincides with the domain of the AIMMS identifier.

*Name substitution*

If the number of attributes in the primary key of a database table is lower than the dimension of the intended AIMMS identifier, you can also map a column name to a *slice* of an AIMMS identifier of the proper dimension, as shown in the fcost and vcost mapping. You can do this by replacing one or more of the indices in the identifier's index space with a reference to a fixed element.

*Mapping columns to slices*

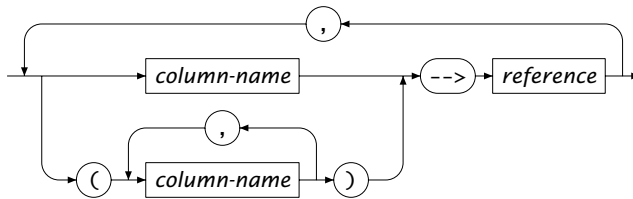
As shown in the last line of the MAPPING attribute, you can let the complete primary key in a database table correspond with a simple set, or with a relation (see Section 3.2.3) in AIMMS. This correspondence is specified by mapping the tuple of primary attributes of the table onto the AIMMS set itself, or onto an index into this set. The primary attributes in the tuple are mapped in a one-to-one fashion onto the indices in the relation.

*Mapping  
primary key to  
relation*

The syntax of the MAPPING attribute is given by the following diagram.

*Syntax*

*mapping-list :*



With the CONVENTION attribute you can indicate to AIMMS that the external data is stored with the units provided in the specified convention. If the unit specified in the convention differs from the unit that AIMMS uses to store its data internally, the data is scaled at the time of transfer. For the declaration of CONVENTIONS you are referred to Section 30.8.

*The CONVENTION  
attribute*

In addition, you can use CONVENTIONS to convert calendar data from the calendar slot format used within your model to the format expected by the database and vice versa. The use of CONVENTIONS for this purpose is discussed in full detail in Section 31.10. For non-calendar related date-time values you can use the predefined identifier `OBDCDateTimeFormat` to accomplish this (see Section 25.8).

*Date  
conversions*

## 25.2 Indexed database tables

While the MAPPING attribute allows you to map data columns in a database table onto a slice of a higher-dimensional AIMMS identifier, a different type of slicing is required when the primary key of a database table contains *exogenous* columns that are of no interest to your application. Consider, for instance, the following situations.

*Exogenous  
columns in  
primary key*

- You are linking to a database table that contains data for a huge set of cities, but your model only deals with a single city that is not explicitly part of the model formulation. For your application the city column is exogenous.
- A table in a database contains several versions of a particular data set, where the version number is represented by an additional version column in the table. For your application the version column is exogenous.

In your AIMMS application you can deal with these situations by partitioning a single table inside the database into a set of *virtual* lesser-dimensional tables indexed by the exogenous column(s). You can do this by declaring the database table to have an INDEX DOMAIN corresponding to the sets that map onto the exogenous columns. In subsequent READ and WRITE statements you can then refer to a particular instance of a virtual table through a reference to the database table with an explicit set element or an element parameter.

*Indexed  
DATABASE TABLES*

The following example assumes that the table "Route Definition" contains several versions of the data, each identified by the value of an additional column version. In the AIMMS model, this column is associated with a set TableVersions given by the following declaration.

*Example*

```
SET:
  identifier : TableVersions
  index      : v
  parameter  : LatestVersion ;
```

The following declaration will provide a number of virtual tables indexed by v.

```
DATABASE TABLE:
  identifier : RouteData
  index domain : v
  data source : "Topological Data"
  table name : "Route Definition"
  mapping    : "version" --> v,
              "from"    --> i,
              "to"      --> j,
              "dist"    --> Distance(i,j),
              "cost"    --> TransportCost(i,j) ;
```

Note that the index v in the index domain is mapped onto the column version in the table.

In order to obtain the set of TableVersions you can follow one of two strategies:

*Data transfer  
with indexed  
tables*

- you can obtain the set of the available versions from the table "Route Definition" itself by declaring another DATABASE TABLE in AIMMS

```
DATABASE TABLE:
  identifier : VersionTable
  data source : "Topological Data"
  table name : "Route Definition"
  mapping    :
    "version" --> TableVersions ;
```

- or, you can obtain the versions from a separate table in a relational database declared similarly as above.

A typical sequence of actions for data transfer with indexed tables could then be the following.

- Read the set of all possible versions from VersionTable:

```
read TableVersions from table VersionTable ;
```

- Obtain the value of LatestVersion from within the language or the graphical user interface.
- Read the data accordingly:

```
read Distance, TransportCost from RouteData(LatestVersion) ;
```

---

### 25.3 Database table restrictions

The AIMMS READ and WRITE statements are intended to directly transfer data to and from a *single* ASCII or case file, or a *single* table in a database. This is the simplest form of communication with a database. If you need more advanced control over the connection with a particular database, you can access stored procedures within the database using AIMMS. Such procedures can be implemented by the database designer to accomplish advanced tasks that go beyond the ordinary. The use of stored procedures is discussed in Section 25.5.

*Data transfer to single tables*

When you are connecting to a table in a database through a READ or WRITE statement, you do not have to make a connection to the server explicitly. The database table declaration and the ODBC/OLE DB configuration files on your system provide sufficient information to allow AIMMS to make the connection automatically whenever needed. If you need to log on to the database, you will be prompted with a log on screen. On some systems it is possible to store log on information in the ODBC/OLE DB data source file.

*Automatic connection*

There is a fundamental difference in the storage of data in AIMMS and the storage of data in a database table. Whereas AIMMS stores its data separately per identifier, a database table stores the data of several indexed identifiers in records all indexed by the same single index tuple. This difference implies that AIMMS has to impose some additional restrictions on data transfer with database tables that are not needed when reading from or writing to either AIMMS case files or ASCII files.

*Different data representation*

In order to be able to define the semantics of the READ and WRITE statements to database tables in an unambiguous way, AIMMS makes a number of (reasonable) assumptions about the database tables in an external database. It is, however, not always possible for AIMMS to verify these assumptions, and unexpected effects may occur when they do not hold. The following assumptions about database tables are made.

*Assumptions about database tables*

- Every database table is in second normal form, i.e. every non-primary column in the table is functionally dependent on the primary key.

- Every primary column in a database table is mapped onto an index in an AIMMS domain set.
- Every non-primary column in a database table is mapped onto a (slice of an) AIMMS identifier, such that the specific index domain of this identifier precisely matches the primary key of the database table according to the existing index mapping.

AIMMS will not allow all identifier selections to be read from or written to database tables. An identifier selection is allowed when the following conditions hold for its components.

*Assumptions about identifier selections*

- All parameter and variable references must have the same domain after slicing. The resulting domain must correspond to the primary key of the database table.
- During a WRITE statement in REPLACE mode you can only write a (simple or compound) set mapped onto the primary key of a database table as long as there are no non-primary columns, or when the selection comprises all the columns of the table.
- AIMMS allows each domain set associated with a primary column in a table of any dimension to be read from that table.

The above rules can be summarized by stating that the database table can be transformed into an AIMMS composite table for the indexed identifiers in it.

*Simply stated*

Identifier selections in READ and WRITE statements form a one-to-one correspondence with a sparse set of records in the database table. During a READ statement the sparsity pattern is determined by all the records in the database table. During a WRITE statement the sparsity pattern is determined by all indexed identifiers in the selection. Records will be written for only those tuples for which at least one of the indexed identifiers or tuple references has a non-default value. Thus, the transferred data resulting from a WRITE statement is equivalent to the single composite table in AIMMS for all indexed identifiers in the selection.

*Selections are sparse*

Writing data to a database in either merge or replace mode may lead to the creation of new records in a database table. New records will be created when AIMMS writes a tuple for a key for which no record is available. If the table has non-primary attributes for which no data is written, AIMMS will leave these attributes empty when it creates new rows.

*Creation of records*

AIMMS supports two replace modes for writing: REPLACE COLUMNS mode (default) and REPLACE ROWS mode.

*Removal of records: REPLACE COLUMNS/ROWS mode*

- In REPLACE or REPLACE COLUMNS mode, AIMMS will only remove data from columns mapped onto identifiers in your model. Rows will only be re-

moved from the database table if *all* columns in the table are mapped onto identifiers in your model.

- In REPLACE ROWS mode, AIMMS will remove all rows whose primary key does not correspond to an index tuple being written during the WRITE statement. Columns that are not mapped onto identifiers in your model, either are assigned their default value specified in the database, or a NULL value otherwise. As a consequence, you should make sure that all non-nullable columns in the table are mapped onto identifiers in your model (or have a default value in the database) during REPLACE ROWS mode.

AIMMS will only remove records if the selection you are writing comprises all the columns in the database table, including the set mapped onto the primary key. In this way, AIMMS ensures that no data is lost in the table inadvertently.

Using the DATABASE TABLE interface it is only possible to filter records using simple domain conditions formulated in a FILTERING clause. For huge database tables it may be desirable to use more advanced filtering techniques designed to restrict the number of records to be transferred. This can be done inside the database application itself in the following two ways.

*Filtering on records*

- Create a *view* in the database that does the filtering for you, and then use the standard READ statement. This is the most straightforward approach, and is sufficient if the filter does not depend on AIMMS data.
- Create a *stored procedure* in the database that can be activated through a DATABASE PROCEDURE in AIMMS. This allows you to filter records dependent on the value of some AIMMS identifiers that are used as arguments of the stored procedure (see Section 25.5).

Although AIMMS allows mapping primary keys to relations in your model, data of identifiers declared over a compound set, or a compound set itself cannot be read from or written to a database table directly. If you want to READ or WRITE compound sets or data declared over compound sets, you have to use intermediate relations and/or parameters with an expanded domain to accomplish this.

*No support for compound sets*

---

## 25.4 Data removal

The AIMMS database interface offers limited capabilities to manage the tables in a database. Such management is typically done through the use of stored procedures within a database. AIMMS, however, offers you the possibility to remove data from a database table by means of the EMPTY or TRUNCATE statement.

*Data removal*

The EMPTY statement can remove data from a database table in two manners.

*Empty database columns*

- When you use a database table identifier in the identifier selection in an EMPTY statement, AIMMS will remove all data from that table.
- When you use a database table identifier behind the IN clause in an EMPTY statement, AIMMS will empty all columns in the corresponding database table which are mapped onto the AIMMS identifiers in the identifier selection of that EMPTY statement.

For more details on the EMPTY statement, refer to Sections 23.3.

The examples in this paragraph illustrate various uses of the EMPTY statement applied to database tables.

*Examples*

- The following statement removes all data from the table CityTable.

```
empty CityTable ;
```

- The following statement removes the data from the table CityTable that maps onto the AIMMS identifier Demand.

```
empty Demand in CityTable ;
```

- The following statement removes the data associated with the version OldVersion from the indexed table RouteTable(v). The data associated with other versions will remain intact.

```
empty RouteTable(OldVersion) ;
```

- The following statement removes the data from the table RouteTable(v) for all versions v in the set Versions.

```
empty RouteTable;
```

- The following statement removes the data from the table RouteTable(v) for all versions sv in the subset SelectedVersions of the set Versions.

```
empty RouteTable(sv);
```

- The following statement removes the data in the column mapped onto the AIMMS identifier Transport and associated with the version LatestVersion from the indexed table RouteTable(v).

```
empty Transport in RouteTable(LatestVersion) ;
```

In the special case that you are connecting to an Oracle or SQL Server database table, you can use the TRUNCATE statement to empty the database in a very fast way. The drawbacks of the TRUNCATE statement is that it can only be applied to a data table from an Oracle or SQL Server data source and that a rollback is not possible after the table has been emptied. In case the underlying data source does not support truncating, depending on the setting of the option `Warning_truncate_table` a warning is issued and AIMMS will use the slower EMPTY statement to empty the table.

*Truncate  
database tables*

The following statement removes all data from the table `CityTable` at once.

*Example*

```
truncate table CityTable ;
```

---

## 25.5 Executing stored procedures and SQL queries

When transferring data from or to a database table, you may need more sophisticated control over the data link than offered by the standard DATABASE TABLE interface. AIMMS offers you this additional control by letting you have access to stored procedures as well as letting you execute SQL statements directly. The following two paragraphs provide some examples where such control may be useful.

*Sophisticated  
control*

Your application may require its data in a somewhat different form than is directly available in the database. In this case you may have to perform some pre-processing of the data in the database. Similarly, you may want to perform post-processing in the database after writing data to it. In such circumstances you may call a stored procedure to perform these tasks for you.

*Useful for data  
processing*

In some cases, the required data for your application may need to be the result of a parameterized query of the database, i.e. a database table whose contents is dependent on one or more parameters which are only known during runtime. Such dynamic tables are usually obtained as the *result set* of a stored procedure or of a parameterized query. In this case AIMMS will allow you to use a stored procedure call or a dynamically composed SQL query inside the READ statement as if it were a database table. Please note that it's currently not possible to read a result set from an Oracle stored procedure, since Oracle uses a non-standard mechanism for that (involving so-called *ref cursors*).

*Useful for  
dynamic access*

Every stored procedure or SQL query that you want to call from within AIMMS must be declared as a DATABASE PROCEDURE within your application. The attributes of a DATABASE PROCEDURE are listed in Table 25.2.

*The DATABASE  
PROCEDURE  
declaration*

Attribute	Value type	See also page
DATA SOURCE	<i>string</i>	375
ARGUMENTS	<i>argument-list</i>	144
STORED PROCEDURE	<i>string-expression</i>	
SQL QUERY	<i>string-expression</i>	
OWNER	<i>string-expression</i>	375
PROPERTY MAPPING	UseResultSet <i>mapping-list</i>	376
COMMENT	<i>comment string</i>	19
CONVENTION	<i>convention</i>	377, 462

Table 25.2: DATABASE PROCEDURE attributes

A DATABASE PROCEDURE in AIMMS can represent either a (dynamically created) SQL query or a call to a stored procedure. AIMMS makes the distinction on the basis of the STORED PROCEDURE and SQL QUERY attributes. If the STORED PROCEDURE attribute is nonempty, AIMMS assumes that the DATABASE PROCEDURE represents a stored procedure and expects the SQL QUERY attribute to be empty, and vice versa.

*SQL query or stored procedure*

With the STORED PROCEDURE attribute you can specify the name of the stored procedure within the ODBC/OLE DB data source that you want to be called. The STORED PROCEDURE wizard will let you select any stored procedure name available within the specified ODBC/OLE DB data source. If the stored procedure that you want to call is not owned by yourself, or if there are name conflicts, you should specify the owner with the OWNER attribute.

*The STORED PROCEDURE attribute*

You can use the SQL QUERY attribute to specify the SQL query that you want to be executed when the DATABASE PROCEDURE is called. The value of this attribute can be any string expression, allowing you to generate a dynamic SQL query using the arguments of the DATABASE PROCEDURE.

*The SQL QUERY attribute*

With the ARGUMENTS attribute you can indicate the list of *scalar* arguments of the database procedure. The specified arguments must have a matching declaration in a declaration section local to the DATABASE PROCEDURE. If the DATABASE PROCEDURE represents a stored procedure, the argument list is interpreted as the argument list of the stored procedure. When you use the STORED PROCEDURE wizard, AIMMS will automatically enter the argument list, including their AIMMS prototype, for you. For a DATABASE PROCEDURE representing an SQL query, you can use the arguments in composing the SQL query string.

*The ARGUMENTS attribute*

For SQL queries all arguments must be Input arguments, as the query cannot modify them. For stored procedures, the STORED PROCEDURE wizard will by default set the input-output type of each argument equal to its SQL input-output type. However, if you want to discard the result of any output argument, you can change its type to Input.

*Input-output type*

With the PROPERTY attribute of a DATABASE PROCEDURE you can indicate the intended use of the procedure.

*The PROPERTY attribute*

- When you do not specify the property UseResultSet, AIMMS lets you call the DATABASE PROCEDURE as if it were an AIMMS procedure.
- When you do specify the property UseResultSet, AIMMS lets you use the DATABASE PROCEDURE as a parameterized table in the READ statement. In that case, you can also provide a MAPPING attribute to specify the mapping from column names in the result set onto the corresponding AIMMS identifiers.

The following declarations will make two stored procedures contained in the data source "Topological Data" available in your AIMMS application. The local declarations of all arguments are omitted for the sake of brevity. They are all assumed to be Input arguments.

*Stored procedure examples*

```

DATABASE PROCEDURE:
  identifier      : StoreSingleTransport
  data source     : "Topological Data"
  stored procedure : "SP_STORE_SINGLE_TRANSPORT"
  arguments      : (from, to, transport) ;
DATABASE PROCEDURE:
  identifier      : SelectTransportNetwork
  data source     : "Topological Data"
  stored procedure : "SP_DISTANCE"
  arguments      : MaxDistance
  property       : UseResultSet
  mapping        :
    "from"       --> i,
    "to"         --> j,
    "dist"       --> Distance(i,j),
    ("from","to") --> Routes ;

```

The procedure StoreSingleTransport can be used like any other AIMMS procedure, as in the following statement.

```

StoreSingleTransport( 'Amsterdam', 'Rotterdam',
                    Transport('Amsterdam', 'Rotterdam') );

```

The second procedure SelectTransportNetwork can be used in a READ statement as if it were a database table, as illustrated below.

```

read from table SelectTransportNetwork( UserSelectedDistance );

```

The following example illustrates the declaration of a DATABASE PROCEDURE representing a direct SQL query. Its aim is to delete those records in the specified table for which the column `VersionCol` equals the specified version. Both arguments must be declared as local Input string parameters.

*SQL query example*

```

DATABASE PROCEDURE:
  identifier : DeleteTableVersion
  data source : "Topological Data"
  arguments  : (DeleteTable, DeleteVersion)
  SQL query  :
    FormatString( "DELETE FROM %s WHERE VersionCol = '%s'",
                  DeleteTable, DeleteVersion );

```

In addition to executing SQL queries through DATABASE PROCEDURES, AIMMS also allows you to execute SQL statements directly within a data source. The interface for this mechanism is simple, and forms a convenient alternative for a DATABASE PROCEDURE when you want to execute a single SQL statement only once.

*Executing SQL statements directly*

You can send SQL statements to a data source by calling the procedure `DirectSQL` with the following prototype:

*The procedure DirectSQL*

■ `DirectSQL(data-source, SQL-string)`

Both arguments of the procedure should be string expressions. Note that in case the SQL statement also produces a result set, then this set is ignored by AIMMS.

The following call to `DirectSQL` drops a table called "Temporary\_Table" from the data source "Topological Data".

*Example*

```

DirectSQL( "Topological Data",
           "DROP TABLE Temporary_Table" );

```

The procedure `DirectSQL` does not offer direct capabilities for parameterizing the SQL string with AIMMS data. Instead, you can use the function `FormatString` to construct symbolic SQL statements with terms based on AIMMS identifiers.

*Use FormatString*

---

## 25.6 Database transactions

By default, AIMMS places a transaction around any *single* WRITE statement to a database table. In this way, AIMMS makes sure that the complete WRITE statement can be rolled back in the event of a database error during the execution of that WRITE statement. You can increase the amount of transactional control over READ and WRITE statements through the procedures

*Transactions*

- StartTransaction(*isolation-level*)
- CommitTransaction
- RollbackTransaction

With the procedure StartTransaction you can manually initiate a database transaction. As a consequence, you can commit or roll back the changes in the database caused by all WRITE statements executed within the context of the transaction simultaneously. You can specify the exact semantics of the transaction through its only (optional) argument *isolation-level*, which must be an element from the predefined set AllIsolationLevels. You cannot call StartTransaction recursively, i.e. you must call CommitTransaction or RollbackTransaction prior to the next call to StartTransaction. The procedure returns a value of 1 if the transaction was started successfully, or 0 otherwise.

*The procedure StartTransaction*

Besides the ability to commit roll back *all* the changes made to the database during the transaction, AIMMS supports the following isolation levels for transactions:

*The set AllIsolationLevels*

- ReadUncommitted: a transaction operating at this level can see uncommitted changes made by other transactions,
- ReadCommitted (default): a transaction operating at this level cannot see changes made by other transactions until those transactions are committed,
- RepeatableRead: a transaction operating at this level is guaranteed not to see any changes made by other transactions in values it has already read during the transaction, and
- Serializable: a transaction operating at this level guarantees that all concurrent transactions interact only in ways that produce the same effect as if each transaction were entirely executed one after the other.

Note that not all database servers may support all of these isolation levels, and may cause the call to StartTransaction to fail.

Through the procedure CommitTransaction you can commit all the changes that you have made to the database since the previous call to StartTransaction. The function returns a value of 1 if the changes are committed successfully, or 0 otherwise.

*The procedure CommitTransaction*

With the procedure RollbackTransaction you can roll back (i.e. undo) all the changes that you have made to the database since the previous call to StartTransaction. The function returns a value of 1 if the changes are rolled back successfully, or 0 otherwise.

*The procedure RollbackTransaction*

---

## 25.7 Testing the presence of data sources and tables

When you want to run an AIMMS-based application on the computer of an end-user, you may want to make sure that the data sources and database tables required to run the application successfully are present, prior to actually initiating any data transfer. Normally, trying to execute a READ and WRITE statements on a nonexisting data source or database table causes AIMMS to generate run-time errors, which might be confusing to your end-users. By first verifying the presence of the required data sources and database tables, you are able to generate error messages which are more meaningful to your end-users.

*Fail safe access*

You can test the presence of data sources and database tables on a host computer through the functions

*Syntax*

- `TestDataSource(data-source)`
- `TestDatabaseTable(data-source, table-name)`

Both *data-source* and *table-name* are string arguments.

With the procedure `TestDataSource` you can check whether the ODBC/OLE DB data source named *data-source* is present on the host computer on which your AIMMS application is being run. The procedure returns 1 if the data source is present, or 0 otherwise.

*The procedure  
TestDataSource*

The function `TestDatabaseTable` lets you check whether a given table named *table-name* exists in the data source named *data-source*. The procedure returns 1 if the database table is present in the given data source, or 0 otherwise. However, the procedure `TestDatabaseTable` will not let you check whether the table contains the columns which you expect it to contain. If you try to access columns in the database table which are not present during either a READ or WRITE statement, AIMMS will still generate a run-time error to this effect.

*The procedure  
TestDatabase-  
Table*

---

## 25.8 Dealing with date-time values

Special care is required when you want to read data from or write data to a database which represents a date, a time, or a time stamp in the database table. The ODBC/OLE DB technology uses a fixed string format for each of these data types. Most likely, this format will not coincide with the format that you use to store dates and times in your modeling application.

*Mapping  
date-time values*

When a column in a database table containing date-time values maps onto a CALENDAR in your AIMMS model, AIMMS will automatically convert the date-time values to the associated time slot format of the calendar, and store the corresponding values for the appropriate time slots.

*Mapped onto calendars*

By default, AIMMS assumes that the date-time values mapped onto a particular CALENDAR are stored in the database according to the same time zone (ignoring daylight saving time) as specified in the TIMESLOT FORMAT attribute of that calendar (see also Section 31.7.4). In the absence of such a time zone specification, AIMMS will assume the local time zone (without daylight saving time). You can override the time zone through the TIMESLOT FORMAT attribute of a CONVENTION. The use of CONVENTIONS with respect to CALENDARS is discussed in full detail in Section 31.10.

*Time zone translation*

If a date-time column in a database table does not map onto a CALENDAR in your model, you can still convert the ODBC/OLE DB date-time format into the date or time representation of your preference, using the predefined string parameter ODBCDateTimeFormat defined over the set of AllIdentifiers. With it, you can specify, on a per identifier basis, the particular format that AIMMS should use to store dates and/or times using the formats discussed in Section 31.7. AIMMS will never perform a time zone conversion for non-calendar data, and will ignore ODBCDateTimeFormat when it contains a date-time format specification for a CALENDAR.

*The ODBCDateTimeFormat parameter*

If you do not specify a date-time format for a particular identifier, and the column does not map onto a CALENDAR, AIMMS will assume the fixed ODBC/OLE DB format. These formats are:

*Unmapped columns*

- YYYY-MM-DD hh:mm:ss.ttttt for date-time columns,
- YYYY-MM-DD for date columns, and
- hh:mm:ss for time columns.

When you are unsure about the specific type of a date/time/date-time column in the database table during a WRITE action, you can always store the AIMMS data in date-time format, as AIMMS can convert these to both the date and time format. During a READ action, AIMMS will always translate into the type for the column type.

A stock ordering model contains the following identifiers:

*Example*

- the set Products with index p, containing all products kept in stock,
- an ordinary set OrderDates with index d, containing all ordering dates, and
- a string parameter ArrivalTime(p,d) containing the arrival time of the goods in the warehouse.

The order dates should be of the format '980320', whilst the arrival times should be formatted as '12:30 PM' or '9:01 AM'. Using the time specifiers of Section 31.7, you can accomplish this through the following assignments to the predefined parameter `ODBCDateTimeFormat`:

```
ODBCDateTimeFormat( 'OrderDates' ) := "%y%m%d";  
ODBCDateTimeFormat( 'ArrivalTime' ) := "%h:%M %p";
```