

---

## **AIMMS Language Reference - Read and Write Statement**

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 24

### The READ and WRITE Statements

In order to help you separate the model description and its input and output data, AIMMS offers the READ and WRITE statements for dynamic data transfer between your modeling application and external data sources such as

*Dynamic data transfer*

- ASCII data files, and
- database tables in external ODBC- or OLE DB-compliant databases.

This chapter first introduces the READ and WRITE statements in the form of an extended example. Subsequently, their semantics are presented in full detail including issues such as filtering, domain checking, and slicing.

*This chapter*

---

#### 24.1 A basic example

The aim of this section is to give you an overview of the READ and WRITE statements through a short illustrative example. It shows how to read data from and write data to ASCII files and database tables. It is based on the familiar transport problem with the following input data:

*Getting started*

- the set *Cities*,
- the compound set *Routes* from *Cities* to *Cities*,
- the parameters *Supply(i)* and *Demand(i)* for each city *i*, and
- the parameters *Distance(i, j)* and *TransportCost(i, j)* for each route between two cities *i* and *j*.

For the sake of simplicity, it is assumed that there is only a single output, the actual *Transport(i, j)* along each route.

The input data can be conveniently given in the form of tables. One for the identifiers defined over a single city like *Supply* and *Demand*, and the other for the identifiers defined over a tuple of cities like *Distance* and *TransportCost*. These tables can be provided in the form of ASCII files as in Table 24.1 (format explained in Section 26.3). Alternatively, the data can be obtained from particular tables in a database. This example assumes the following database tables exist:

*Format of input data*

- *CityData* for the one-dimensional parameters, and

- RouteData for the two-dimensional parameters.

COMPOSITE TABLE				COMPOSITE TABLE			
Cities	Supply	Demand		i	j	Distance	TransportCost
! -----	-----	-----		! -----	-----	-----	-----
Amsterdam	50			Amsterdam	Rotterdam	85	1.00
Rotterdam	100			Amsterdam	Antwerp	170	2.50
Antwerp	75	25		Amsterdam	Berlin	660	10.00
Berlin		125		Amsterdam	Paris	510	8.25
Paris		75		Rotterdam	Antwerp	100	1.20
;				Rotterdam	Berlin	700	10.00
				Rotterdam	Paris	440	7.50
				Antwerp	Berlin	725	11.00
				Antwerp	Paris	340	5.00
				Berlin	Paris	1050	17.50
				;			

Table 24.1: Example data set for the transport model

### 24.1.1 Simple data transfer

The simplest use of the READ statement is to initialize data from a fixed name ASCII data file, or a database table. To read all the data from each source, the following groups of statements will suffice

*Simple data initialization*

```
read from file "transport.inp" ;

read from table CityData;
read from table RouteData;
```

Such statements are typically found in the body of the predefined procedure MainInitialization.

When a data source also contains data for identifiers that are of no interest to your particular application (but may be to others), AIMMS allows you to restrict the data transfer to a specific selection of identifiers in that data source. For instance, the following READ statement will only read the identifiers Distance and TransportCost, not changing the current contents of the AIMMS identifiers Supply and Demand.

*Reading identifier selections*

```
read Distance, TransportCost from file "transport.inp" ;
```

Similar identifier selections are possible when reading from a database table.

After your model has computed the optimal transport, you may want to write the solution `Transport(i,j)` to an ASCII output file for future reference. You can do this by calling the `WRITE` statement, which has equivalent syntax to the `READ` statement. The transfer of `Transport(i,j)` to the file `transport.out` is accomplished by the following `WRITE` statement.

*Writing the solution*

```
write Transport to file "transport.out" ;
```

If you omit an identifier selection, AIMMS will write all model data to the file. When writing to a database table, AIMMS can of course only transfer data for those identifiers that are known in the table that you are writing to.

File data transfer is not restricted to files with a fixed name. To choose the name of the data file either during execution or from within the end-user interface, you have several options:

*File name need not be explicit*

- replace the filename string in the `READ` and `WRITE` statements with a string-valued parameter holding the filename, or
- use a `FILE` identifier (for ASCII files only).

---

### 24.1.2 Set initialization and domain checking

When you are reading the initial data of the transport model from an external data source several situations can occur:

*Domain restrictions*

- you just want to initialize the set `Cities` from the data source,
- the set `Cities` has already been initialized, and you want to retrieve the parametric data for existing cities only, or
- the set `Cities` has already been initialized, but you want to extend it on the basis of the data read from the external data source.

The following statements impose domain restrictions on the `READ` statement.

*The READ statements*

```
read Cities
  from file "transport.inp" ;

read Supply, Demand
  from file "transport.inp"
  filtering i ;

read Supply, Demand
  from file "transport.inp" ;
```

The first READ statement is a straightforward initialization of the set *Cities*. By default, AIMMS reads in replace mode, which implies that any previous contents of the set *Cities* is overwritten. *Initializing sets*

The second READ statement assumes that the set *Cities* has already been initialized. From all entries of the identifiers *Supply* and *Demand* it will only read those which correspond to existing elements in the set *Cities*, and skip over the data from the remaining entries. *Domain checking*

The third READ statement differs from the second in that the clause 'FILTERING i' has been omitted. As a result, AIMMS will not reject data that does not correspond to an existing label in the set *Cities*, but will read all available *Supply* and *Demand* data, and extend the set *Cities* accordingly. *Extending domain sets*

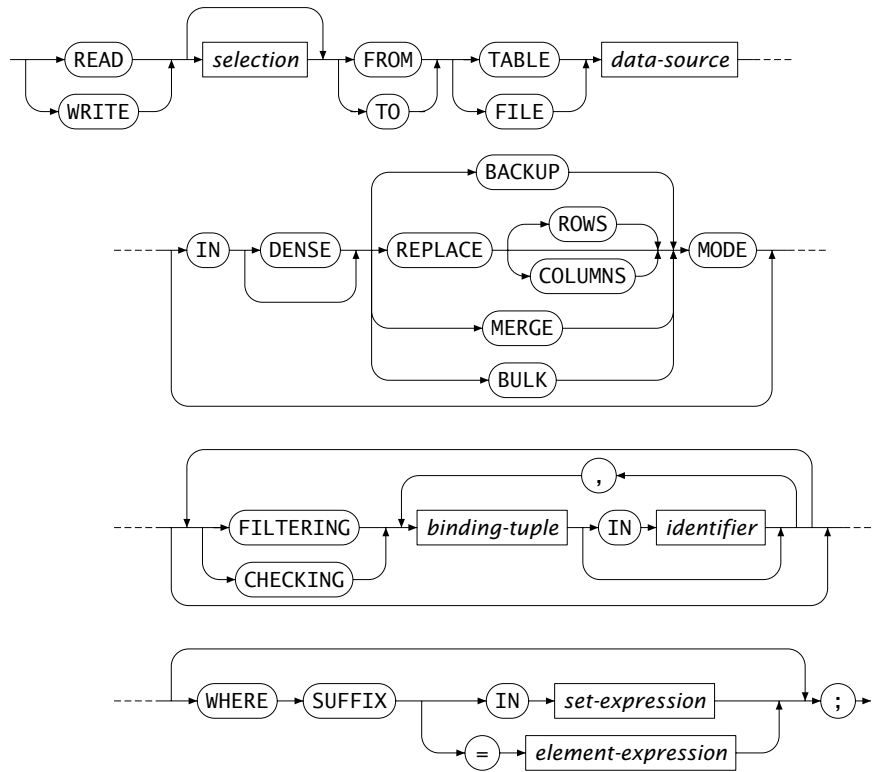
---

## 24.2 Syntax of the READ and WRITE statements

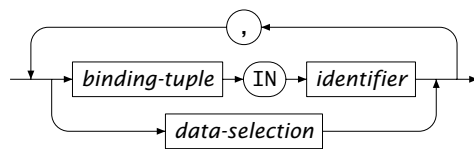
In READ and WRITE statement you can specify the data source type, what data will be transferred, and in what mode. The syntax of the statements reflect these aspects. *READ and WRITE statements*

*Syntax*

*read-write-statement* :



*selection* :



The data source of a READ or WRITE statement in AIMMS can be either

*Data sources*

- a FILE represented by either
  - a FILE identifier,
  - a string constant, or
  - a scalar string reference,
- a TABLE represented by either
  - a DATABASE TABLE identifier,
  - an element parameter with a range that is a subset of the pre-declared set `AllDatabaseTables`

Strings for file data sources refer either to an absolute path or to a relative path. All relative paths are taken relative to the project directory.

Assuming that `UserSelectedFile` is a FILE identifier, and `UserFilename` a string parameter, then the following statements illustrate the use of strings and FILE identifiers.

*Examples*

```
read from file "C:\Data\Transport\initial.dat" ;
read from file "data\initial.dat" ;
read from file UserFileName ;
read from file UserSelectedFile ;
```

The *selection* in a READ or WRITE statement determines which data you want to transfer from or to an ASCII file, or database table. A selection is a list of references to sets, parameters, variables and constraints. During a WRITE statement, AIMMS accepts certain restrictions on each reference to restrict the amount of data written (as explained below). Note, however, that AIMMS does not accept all types of restrictions which are syntactically allowed by the syntax diagram of the READ and WRITE statements.

*Specifying a selection*

If you do not specify a selection during a READ statement, AIMMS will transfer the data of all identifiers stored in the table or file that can be mapped onto identifiers in your model. If you do not specify a selection for a WRITE statement to an ASCII file, all identifiers declared in your model will be written. When writing to a database table, AIMMS will write data for all columns in the table as long as they can be mapped onto AIMMS identifiers.

*Default selection*

You can apply the following filtering qualifiers on READ and WRITE statements to restrict the data selection:

*Filtering the selection*

- the FILTERING or CHECKING clauses restrict the domain of all transferred data in both the READ and WRITE statements, and
- an arbitrary logical condition can be imposed on each individual parameter and variable in a WRITE statement.

You can use both the FILTERING and CHECKING clause to restrict the tuples for which data is transferred between a data source and AIMMS. During a WRITE statement there is no difference in semantics, and you can use both clauses interchangeably. During a READ statement, however, the FILTERING clause will skip over all data outside of the filtering domain, whereas the CHECKING clause will issue a runtime error when the data source contains data outside of the filtering domain. This is useful feature for catching typing errors in ASCII data files.

*FILTERING versus CHECKING*

The following examples illustrate filtering and the use of logical conditions imposed on index domains.

*Examples*

```
read Distance(i,j) from table RouteTable
  filtering i in SourceCities, (i,j) in Routes;
```

```
write Transport( (i,j) | Sum(k, Transport(i,k)) > MinimumTransport )
to table RouteTable ;
```

If you need more advanced filtering on the records in a database table, you can use the database to perform this for you. You can

- define *views* to create temporary tables when the filtering is based on a non-parameterized condition, or
- use *stored procedures* with arguments to create temporary tables when the filtering is based on a parameterized condition.

*Advanced filtering on records*

The resulting tables can then be read using a simple form of the READ statement.

AIMMS allows you to transfer data from and to a file or a database table in either *merge* mode or *replace* mode. If you have not selected a mode in either a READ or WRITE statement, AIMMS will transfer the data in replace mode by default. When you are writing data to an ASCII data file, AIMMS also supports a *backup* mode. In the special case that you are writing to an Oracle or SQL Server database table, the *bulk* mode can be used to speed up the writing.

*Merge, replace, backup or bulk mode*

When AIMMS reads data in merge mode, it will overwrite existing elements for all read identifiers, and add new elements as necessary. It is important to remember that in this mode, if there is no data read for some of the existing elements, they keep their current value.

*Reading in merge mode*

When AIMMS writes data in merge mode, the semantics is dependent on the type of the data source.

*Writing in merge mode*

- If the data source is an ASCII file, AIMMS will *append* the newly written data to the end of the file.
- If the data source is a database table, AIMMS will merge the new values into the existing values, creating new records as necessary.

When AIMMS reads data in replace mode, it will empty the existing data of all identifiers in the identifier selection, and then read in the new data.

*Reading in replace mode*

When AIMMS writes data in replace mode, the semantics is again dependent on the type of the data source.

*Writing in replace mode*

- If the data source is an ASCII file, AIMMS will *overwrite the entire contents* of the file with the newly written data. Thus, if the file also contained data for identifiers that are not part of the current identifier selection, their data is lost by the WRITE statement.

- If the data source is a database table, AIMMS will either empty all columns in the table that are mapped onto identifiers in the identifier selection (default, REPLACE COLUMNS mode), or will remove all records in the table not written by this write statement (REPLACE ROWS mode). The REPLACE COLUMNS and REPLACE ROWS modes are discussed in more detail in Section 25.3).

When you are transferring data to an ASCII file, AIMMS supports writing in backup mode in addition to the merge and replace modes. The backup mode lets you write out files which can serve as an ASCII backup to a (binary) AIMMS case file. When writing in backup mode, AIMMS

*Writing in  
backup mode*

- skips all identifiers on the identifier list which possess a nonempty definition (and, consequently, cannot be read in from a datafile),
- skips all identifiers for which the property NoSave has been set, and
- writes the contents of all remaining identifiers in such an order that, upon reading the data from the file, all domain sets are read before any identifiers defined over such domain sets.

Backup mode is not supported during a READ statement, or when writing to a database.

With the introduction of AIMMS 3.4 the WRITE statement supports bulk writing to database tables. This *bulk* mode has the same effect as the *replace rows* mode. The big advantage of writing in bulk mode, is the high speed that can be obtained (up to 30 times as fast as the other write modes in AIMMS). The drawbacks are that this mode can only be applied to Oracle and SQL Server data sources and that a rollback is not possible after executing the write statement.

*Writing in bulk  
mode*

If a write command is executed in *bulk* mode, and it appears that the underlying data source does not support bulk writing, depending on the setting of the option `Warning_illegal_bulk_mode_write` a warning is issued and AIMMS will try to find an alternative mode to write the data.

*Illegal bulk  
mode writing*

Data in AIMMS is stored for non-default values only, and, by default, AIMMS only writes these non-default values to a database. In order to write the default values as well to the database table at hand, you can add the *dense* keyword before each of the WRITE modes discussed above. This will cause AIMMS to write all possible values, including the defaults, for all tuple combinations considered in the WRITE statement. Care should be taken that writing in *dense* mode does not lead to an excessive amount of records being stored in the database.

*Writing data in  
a dense mode*

Whenever elements in a domain set have been removed by a READ statement in replace mode, AIMMS will *not* cleanup all identifiers defined over that domain. Instead, it will leave it up to you to use the CLEANUP statement to remove the inactive data that may have been created.

*Replacing sets*

For every READ and WRITE statement you can indicate whether or not you want domain filtering to take place during the data transfer. If you want domain filtering to be active, you must indicate the list of indices, or domain conditions to be filtered in either a FILTERING or CHECKING clause. In case of ambiguity which index position in a parameter you want to have filtered you must specify indices in the set or parameter reference.

*Domain filtering*

The following READ statements are not accepted because both Routes and Distance are defined over Cities  $\times$  Cities, and it is unclear to which position the filtered index *i* refers.

*Example*

```
read Routes    from table RouteTable filtering i ;
read Distance from table RouteTable filtering i ;
```

This ambiguity can be resolved by explicitly adding the relevant indices as follows.

```
read (i,j) in Routes from table RouteTable filtering i ;
read Distance(i,j)  from table RouteTable filtering i ;
```

When you have activated domain filtering on an index or index tuple, AIMMS will limit the transfer of data dependent on further index restrictions.

*Semantics of domain filtering*

- During a READ statement only the data elements for which the value of the given index (tuple) lies within the specified set are transferred. If no further index restriction has been specified, transfer will take place for all elements of the corresponding domain set.
- During a WRITE statement only those data elements are transferred for which the index (tuple) is contained in the AIMMS set given in the (optional) IN clause. If no set has been specified, and the data source is a database table, the transfer is restricted to only those tuples that are already present in the table. When the data source is an ASCII file the latter type of domain filtering is not meaningful and therefore ignored by AIMMS.

In the following two READ statements the data transfer for elements associated with *i* and (*i,j*), respectively, is further restricted through the use of the sets SourceCities and Routes.

*READ example*

```
read Distance(i,j) from table RouteTable filtering i in SourceCities ;
read Distance(i,j) from table RouteTable filtering (i,j) in Routes ;
```

In the following two WRITE statements, the values of the variable Transport(i, j) are written to the database table RouteTable for those tuples that lie in the AIMMS set SelectedRoutes, or for which records in the table RouteTable are already present, respectively.

*WRITE example*

```
write Transport(i,j) to table RouteTable filtering (i,j) in SelectedRoutes ;  
write Transport(i,j) to table RouteTable filtering (i,j) ;
```

The FILTERING clause in the latter WRITE statement would have been ignored by AIMMS when the data source was an ASCII data file.

Using the WHERE clause of the WRITE statement you can instruct AIMMS, for all identifiers in the identifier selection, to write the data of either a specified suffix or a set of suffices to file, rather than their level values. The WHERE clause can only be specified during a WRITE statement to a FILE, and the corresponding set or element expression must refer to a subset of, or element in, the predefined set AllSuffixNames.

*Writing selected  
suffices using  
the WHERE clause*

The following WRITE statement will write the values of the .Violation suffix of to the file ViolationsReport.txt for all variables in the project.

*Example*

```
write AllVariables to file "ViolationsReport.txt" where suffix = 'Violation';
```