

---

## **AIMMS Language Reference - Sparse Execution Engine**

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  $\mathcal{A}\mathcal{M}\mathcal{S}\text{-}\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  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.

**Part IV**

---

**Sparse Execution**

## Chapter 12

### The AIMMS Sparse Execution Engine

In this chapter, we provide you with a look under the hood of the AIMMS sparse execution engine. It's not only interesting to know what AIMMS can do, but also, to some extent, how it is done. An understanding of the inner workings of the AIMMS execution engine may also give you a framework to understand why some formulations of AIMMS statements are more efficient than others, leading to more efficient applications. Increasing the efficiency of your application will help making it a success.

*Learning about  
sparse execution*

The AIMMS execution system borrows and extends two simple but powerful concepts from sparse matrix technology. These concepts are:

*Sparse matrix  
technology*

- only store the nonzero values, and
- do not compute  $0+0$  and  $0*x$  ( $x$  any number), because these computations always result in  $0.0$  and these results are not stored.

There are, of course, much more aspects to sparse matrix technology. These aspects, however, are not relevant to the discussion in this chapter.

The AIMMS extensions to these borrowed concepts are that

*AIMMS  
extensions*

- only non-default values are stored, whereby the default is a selectable value, and
- many operations like OR and AND have similar behavior as + and \*, respectively.

Note, however, that other operators, such as the / and = operators, will have to consider zeros:

- the computation  $0.0 / 0.0$  results in UNDF, and
- the computation  $0.0 = 0.0$  results in  $1.0$

The result of these computations are not equal to  $0.0$  and need to be stored; 'sparse execution' is not applicable for these operators.

## 12.1 Storage and basic operations of the execution engine

In this section we present, in a step-by-step manner, the operations that, combined, build up the AIMMS sparse execution engine. The data storage method upon which these operations work is called an *ordered view*.

The AIMMS execution engine stores the data according to the concept of an ordered view. An ordered view is an ordered sparse collection of the non-default elements of an identifier. The order is the lexicographical order of the indices of that identifier. Because of this order

*Ordered view*

- the non-default elements of the identifier can be visited in a lexicographic order one at a time, and
- a particular tuple can be found efficiently using values for the indices.

The running example, used in this section and presented below, contains the two parameters  $A(i, j)$  and  $B(i, j)$ , where  $i$  and  $j$  are indices in a set  $S$  containing the elements  $\{a1..a5\}$ . The defaults of these parameters are 0.0, and they contain following data:

*Running example*

```

A(i,j) := data table
  a1 a2 a3 a4 a5
! ---
a1      2      5
a2 2      3  2
a3
a4 4
a5
;

B(i,j) := data table
  a1 a2 a3 a4 a5
! ---
a1      3      2
a2
a3 5      1  2
a4 4
a5
;

```

The ordered views of A and B are presented in the composite tables below:

```

Composite table:
  i j A
! ---
a1 a2 2
a1 a5 5
a2 a1 2
a2 a3 3
a2 a4 2
a4 a1 4 ;

Composite table:
  i j B
! ---
a1 a2 3
a1 a5 2
a3 a1 5
a3 a3 1
a3 a4 2
a4 a1 4 ;

```

There is nothing really new here; an ordered view corresponds to an relational table in database terminology, with a (database) index on the primary keys  $i$  and  $j$ . A characteristic of both representations is that they can be easily searched given explicit values for  $i$  and  $j$ .

*Like an index in relational databases*

In the following sections we will classify the algebraic operations in AIMMS according to their behavior in the AIMMS sparse execution engine, and discuss the effects of combining multiple operations or changing the natural index order.

*Basic operations*

---

### 12.1.1 The + operator: union behavior

The first statement in the running example is the simple addition of the matching elements into a parameter  $C(i, j)$ :

*First statement*

$$C(i, j) := A(i, j) + B(i, j);$$

As illustrated in Figure 12.1, this statement can be executed in a sparse manner by merging the ordered views of A and B and adding the values along the way.

*Merging rows*

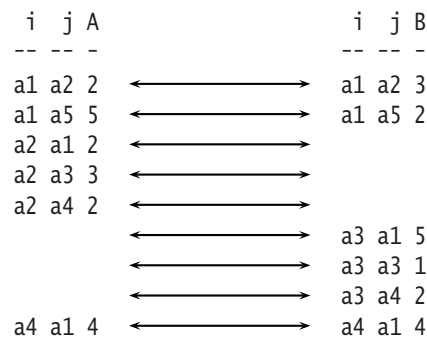


Figure 12.1: Sparse execution of the + operator

In this figure each arrow represents a computed result. The behavior of the + operator is called *sparse union* behavior: the union of the rows of A and B is taken to form the rows of C and it is sparse because we do not need to consider tuples  $(i, j)$  for which  $A(i, j)$  and  $B(i, j)$  are both 0.0.

*Union behavior*

Other operators, such as OR, XOR, <, > and <> have a similar behavior. They can also be implemented using the union of the rows and performing the appropriate operation along the way.

*Similar operators*

---

### 12.1.2 The \* operator: intersection behavior

The second statement in the running example is the simple multiplication of the matching elements into a parameter  $D(i, j)$ .

*Second statement*

$$D(i, j) := A(i, j) * B(i, j);$$

This statement can be executed in a sparse manner by intersecting the ordered views of A and B and multiplying the values along the way. Intersection is sufficient because only for tuples  $(i, j)$  for which both  $A(i, j)$  and  $B(i, j)$  are nonzero, a nonzero is computed. This is illustrated in the Figure 12.2

*Matching rows*

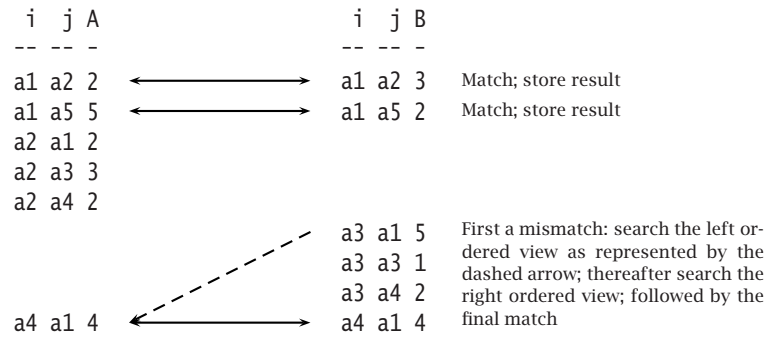


Figure 12.2: Sparse execution of the \* operator

Note that the ordered views of both A and B are searchable and, thus, finding the matching elements can be implemented efficiently. We call this behavior *sparse intersection* behavior. Because only matching rows need to be considered, sparse intersection operators are much more efficient than sparse union operators.

*Intersection behavior*

Other operators, such as the AND and \$ operators exhibit a similar behavior. They can also be implemented using the intersection of the rows and performing the appropriate operation along the way.

*Similar operators*

---

### 12.1.3 The = operator: dense behavior

The third statement in the running example checks whether the corresponding values are equal.

*Third statement*

$$E(i, j) := (A(i, j) = B(i, j));$$

This statement is admittedly a bit artificial. However, such conditions are frequently part of larger expressions and must be considered as well. The key observation is that the comparison  $0.0 = 0.0$  evaluates to true. In AIMMS the value of true is represented by the numerical value 1.0. Therefore, the result of  $E(i, j)$  is:

*Comparing values*

```
E(i,j) := data table
      a1 a2 a3 a4 a5
!  ---
a1  1   1   1   1
a2   1   1   1   1
a3   1   1   1   1
a4  1  1  1  1  1
a5  1  1  1  1  1 ;
```

Because the comparison between two zeros also results in a nonzero, all possible combinations of  $(i, j)$  have to be considered. Therefore, this operation exhibits a *dense* behavior, i.e. the operation cannot be performed in a sparse manner. Dense operators have the worst possible efficiency.

*Dense behavior*

Other operators, such as  $/$ ,  $**$ ,  $<=$  and  $=>$  have a similar behavior. They also need to be implemented by considering all possibilities and evaluating along the way.

*Similar operators*

Increasing the number of indices or increasing the size of the sets will make the number of rows to be considered for such operations grow rapidly. High-dimensional dense operations are a possible cause of performance glitches in your application.

*Beware!*

---

#### 12.1.4 Behavior of combined operations

The fourth statement is a variation of the third statement:

*Fourth statement*

```
EP(i,j) := ( A(i,j) = B(i,j) ) $ A(i,j);
```

Though the operation  $=$  remains dense, the entire right hand side of the assignment statement is limited to only those tuples  $(i, j)$  for which  $A(i, j)$  is nonzero. This is called a domain condition on the expression. The net effect is that the condition on the expression speeds up the efficiency behavior from dense to sparse behavior. The result of this fourth assignment is:

*Speeding up*

```
EP(i,j) := data table
      a1 a2 a3 a4 a5
!  ---
```

```

a1
a2
a3
a4 1
a5          ;

```

When your model contains a statement that performs badly because of a dense operation, using a domain condition can remedy the problem. Often, it is possible to formulate a domain condition that does not alter the result of the computation, but which does allow AIMMS to execute the statement in a sparse manner.

*Preventing  
dense behavior*

---

### 12.1.5 Summation

The fifth statement, as detailed below, is a preamble to the sixth statement and illustrates a language construct where sparse evaluation is straightforward. This fifth statement is a simple aggregation of the parameter  $A(i, j)$  into a parameter  $AI(i)$ :

*Fifth statement*

```
AI(i) := Sum( j, A(i, j) );
```

This operation is illustrated in Figure 12.3.

```

i  j  A
-- -- -
a1 a2 2 } 7
a1 a5 5 }
a2 a1 2 } 7
a2 a3 3 }
a2 a4 2 }
a4 a1 4 } 4

```

Figure 12.3: Sparse execution of the Sum operator

Each brace represents a group of values corresponding to a particular value of  $i$ . As the elements in a group are adjacent in this ordered view, the result of  $AI$  can be computed by a single pass over the ordered view of  $A$ . The order of the running indices in the statement is  $[i, j]$ . The first running index  $i$  is already part of the left hand side of the assignment, and  $j$  is added to this list as part of the sum.

*Running indices  
and identifier  
indices match*

Because the order of the running indices matches the order of the indices in the identifier  $A(i, j)$ , the result of the sum can be computed in a single pass over the ordered view of  $A(i, j)$ .

*Single pass is sufficient*

---

### 12.1.6 Reordered views

The sixth statement is a small variation of the fifth statement above. This sixth statement is an aggregation of the parameter  $A$  into a parameter  $AJ(j)$ :

*Sixth statement*

```
AJ(j) := Sum( i, A(i,j) );
```

The elements that belong to the same group  $j$  are not adjacent in the ordered view of  $A$ , as the order of the indices in this statement is  $[j, i]$  which does not match the order of the indices in  $A(i, j)$ .

*Non-matching index order*

In order to get the elements in the same group adjacent again, AIMMS maintains other views of the parameter  $A$  called *reordered views*. A reordered view of an ordered view is a lexicographic order of the elements such that the order of the indices in the identifier matches the order of the running indices. A reordered view and the grouping according to this view are illustrated in Figure 12.4.

*Reordered views*

```

j  i  A
-- -- -
a1 a2 2 } 6
a1 a4 4 }
a2 a1 2 } 2
a3 a2 3 } 3
a4 a2 2 } 2
a5 a1 5 } 4
```

Figure 12.4: Sparse execution of the reordered Sum operator

Again, each brace represents a group of values corresponding to a particular value of  $j$ . As the elements in a group are adjacent in this reordered view, the result of  $AJ$  can be computed by a single pass over this reordered view of  $A$ . AIMMS generates and maintains reordered views on a need-be basis. They do take up memory, however.

*Single pass is sufficient*

---

## 12.2 Modifying the sparsity

Now that we've glanced at the execution engine's inner workings, you may be wondering about the following questions.

*Questions*

- Does sparse execution influence the results of your model?
- Does AIMMS have sparse versions of operators that are dense by nature?

Sparse execution never influences the results of your model. AIMMS only applies sparse intersection or sparse union when it is applicable. It does not in any way influence the results of your model compared to simply considering all combinations of the running indices, but only the efficiency with which these results are obtained.

*Sparse execution is correct*

AIMMS does support sparse versions of some dense operators, but the sparse versions will in general lead to different results. By adding \$ characters to dense operators you modify these operators into sparse ones. That is why we call the \$ characters added to these operators *sparsity modifiers*.

*Sparsity modifiers*

Sparsity modifiers may be added to the left-hand side of a dense operator, to the right-hand side, or to both. It causes the operator only to return a nonzero result if the associated operand(s) are nonzero. Such a change to an operator may, however, change its results in a way you may or may not want.

*Left and right operand*

Let us now consider a few examples when such a modification is applicable. The first example of using a sparsity modifier is in the efficient guarding against division by zero errors. Without the use of sparsity modifiers, we can accomplish this as follows.

*A first example: the /\$ operator*

```
! Leave A(i,j) zero when C(i,j)+D(i,j) is zero in order to
! avoid division by zero errors.
! This is accomplished by repeating the denominator in the condition.
A(i,j) := ( B(i,j) / (C(i,j)+D(i,j)) ) $ (C(i,j)+D(i,j)) ;
```

In the example, we only divide by  $C(i)+D(i)$  if this sum is nonzero. Note that this subexpression is actually computed twice. AIMMS provides a notational convenience in the form of \$ sparsity modifiers as follows.

```
! Leave A(i,j) zero when C(i,j)+D(i,j) is zero in order to
! avoid division by zero errors.
! This is accomplished by using the /$ division operator
! which sparsely skips 0.0's.
A(i,j) := B(i,j) /$ (C(i,j)+D(i,j)) ;
```

The /\$ operator is defined as the / operator except when the right hand side is 0.0. In that case the \$ sparsity modifier defines it as 0.0. An added advantage is that the sub-expression  $C(i)+D(i)$  is only computed once.

The second example is in the merging of new results into a set of existing results. Without the use of a sparsity modifier you can accomplish this as follows.

*The merge operator := \$*

```
! Only overwrite elements of E(i,j) when the result
! F(i,j) + G(i,j) is nonzero.
! This is accomplished by repeating the RHS of the
! assignment as a domain condition.
E((i,j) | F(i,j)+G(i,j)) := F(i,j)+G(i,j) ;
```

Using the \$ sparsity modifier this can be obtained equivalently as follows.

```
! Only overwrite elements of E(i,j) when the result
! F(i,j) + G(i,j) is nonzero.
! This is accomplished by using the $ sparsity
! modifier on the assignment operator:
E(i,j) := $ F(i,j)+G(i,j) ;
```

Table 12.1 summarizes the operators to which the \$ sparsity modifier can be applied, and whether it can be applied to the left-hand side operand, to the right-hand side operand, or to both.

*Where allowed?*

Operator	Sparsity modifier allowed	
	\$ left	\$ right
^	yes	yes
*	-	-
/	-	yes
+, -	-	-
=, <>, <, >, <=, >, >=	yes	yes
:=	yes	yes
+=, -=	yes	-
*=, /=, ^=	yes	yes
\$, ONLYIF, AND, OR, XOR	-	-

Table 12.1: Sparsity modifiers of binary operators

In addition to modifying the behavior of binary operators, the \$ sparsity modifier can also be applied to iterative operators. The interpretation in this case is that the iterative operator in the presence of a \$ modifier will only be applied to tuples for which the expression yields a nonzero value.

*Modifying iterative operators*

The third and last example of the \$ sparsity modifier is on the Min operator. Suppose you want to find the smallest nonzero distance between a particular node and other nodes. This can be modeled as follows:

*Example: the Min\$ operator*

```
! Find the smallest nonzero distance:
MinimalDistance(i) := Min(j | Distance(i,j), Distance(i,j));
```

The 'nonzero' restriction is taken care of by repeating the argument of the Min operator in its domain condition. By using the \$ sparsity modifier we can abbreviate the above as follows:

```
! Find the smallest nonzero distance:
MinimalDistance(i) := Min$(j, Distance(i,j));
```

Table 12.2 summarizes the iterative operators to which the \$ sparsity modifier can be applied.

*Where allowed?*

Iterative operator	Sparsity modifier allowed
	\$ added
Sort, NBest	yes
Intersection, First, Last, Nth	yes
ArgMin, ArgMax	-
Sum, Union	yes
Prod	-
Min, Max	yes
Statistical operators (see also page 83)	yes
ForAll	-
Other logical operators (see also page 92)	-

Table 12.2: Sparsity modifiers of iterative operators

Conclusively, we can say that the \$ sparsity modifier is a notational convenience which you may or may not like. In the end it is up to you to decide whether or not to use it. You do this by weighing its advantage and disadvantages. Our point of view on this is discussed briefly below.

*Usage of sparsity modifiers*

Using sparsity modifiers has the following advantages.

*Advantages*

- It allows for a more compact notation. In the examples above, the domain condition is replaced by a strategically placed \$ sparsity modifier thereby reducing the overall expression. There are many models out there with multiple line subexpressions and then the reduction is surely not insignificant.
- It is more efficient. The zeros in your model are usually abundantly present and form a special case. So special you want them ignored as the corresponding entries should also not turn up in the results. In addition, you want them to be ignored as quickly as possible: don't spend any computation time on them.

Like any new notation it takes time to get used to it. This holds both for you as a modeler but also for the people you want to communicate your model to. In order to alleviate this disadvantage you may want to add a few brief comments on the modified operators you use such as “:= \$ operator used here to merge the result in to the existing data”.

*Disadvantages*

---

### 12.3 Overview of operator efficiency

In this section you find an overview of the efficiency of all unary, binary and iterative operators in AIMMS.

*Operator efficiency*

The unary operators and functions presented in Table 12.3 are divided into two groups: sparse and dense.

*Unary operators and functions*

- *sparse* When the argument is 0.0 the result is 0.0. The result of only those tuples need to be computed for which the argument has a nonzero value.
- *dense* When the argument is 0.0 the result is not equal to 0.0. The result of all possible tuples need to be computed.

The binary operators presented in Table 12.4 can be divided into three groups:

*Binary operators*

- *intersection sparse*: When either of the arguments is 0.0 the result is 0.0. The result of only those tuples need to be computed when both arguments are not equal to 0.0. This corresponds to taking the intersection of the set of tuples for which the arguments are defined.
- *union sparse*: When both arguments are 0.0 the result is 0.0. The result of only those tuples need to be computed when at least one of the arguments is not equal to 0.0. This corresponds to taking the union of the set of tuples for which the arguments are defined.

sparse		dense	
-	Sinh	NOT	Cos
Sin	Tanh	Cos	Cosh
Tan	ArcSin	Exp	ArcCos
Round	ArcTan	Log	ArcCosh
Floor	ArcSinh	Log10	Factorial
Ceil	ArcTanh		
Trunc	Sqr		
Sqrt			

Table 12.3: Sparseness of unary operators and functions

- *dense*: When both arguments are 0.0 the result is not equal to 0.0. In that case the expression needs to be evaluated for all possible combinations of values of the indices, unless these combinations are limited by a sparse operator elsewhere in the same expression. This corresponds to taking the Cartesian product of the ranges of the indices.

intersection	union	dense
*	+	^
\$	-	/
ONLYIF	<>	=
AND	<	<=
	>	>=
	OR	Permutation
	XOR	Combination

Table 12.4: Sparseness of binary operators

The iterative operators presented in Table 12.5 are divided into three groups as follows:

*Iterative operators*

- *sparse* The value 0.0 of the argument does not influence the result and can be safely ignored. The iterative operator only considers existing entries of its argument.
- *almost sparse* The second 0.0 of the argument does not influence the result. Execution starts dense which means the iterative operator considers all possible tuples. But after the first 0.0 has been encountered, execution continues sparse.
- *dense* The value 0.0 of the argument influences the result. The iterative operator considers all possible combinations.

<b>sparse</b>	<b>almost sparse</b>	<b>dense</b>	
Sum	Max	Mean	SampleDeviation
Prod	Min	GeometricMean	PopulationDeviation
Exists	ArgMax	HarmonicMean	Skewness
Forall	ArgMin	RootMeanSquare	Kurtosis
Count		Median	RankCorrelation

Table 12.5: Sparseness of iterative operators