AIMMS Language Reference - The AIMMS 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.

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

Email: info@aimms.com
WWW: www.aimms.com

Part IV

Sparse Execution

Chapter 12

The AIMMS Sparse Execution Engine

Learning about sparse execution

In this chapter, we look under the hood of the AIMMS sparse execution engine. It is 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 for understanding why some formulations of AIMMS statements are more efficient than others, leading to more efficient applications. Increasing the efficiency of your application will help make it a success.

Sparse matrix technology

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

- only store the non-zero 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 consequently not stored.

AIMMS *extensions*

The AIMMS extensions to these borrowed concepts are that:

- only non-default values are stored, where the default is a selectable value, and
- many operations such as `OR` and `AND` have similar behaviors 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 results of these computations are not equal to 0.0 and need to be stored; and therefore 'sparse execution' is not applicable to 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, when combined, build up the AIMMS sparse execution engine. The data storage method with which these operations work is called an *ordered view*.

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:

- 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.

Running example

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 default values of these parameters are 0.0, and they contain the following data:

```
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 ;
```

Like an index in relational databases

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.

Basic operations

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.

12.1.1 The + operator: union behavior

First statement

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

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

Merging rows

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 as one progresses.

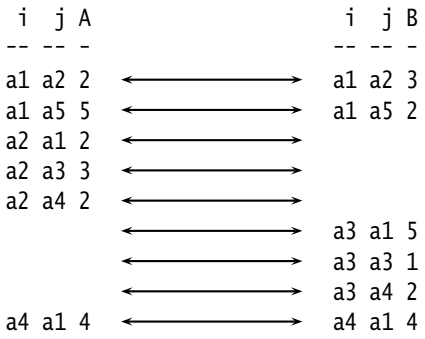

Figure 12.1: Sparse execution of the + operator

Union behavior

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

Similar operators

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

12.1.2 The * operator: intersection behavior

Second statement

The second statement in the running example is the simple multiplication of the matching elements resulting in parameter D(i,j):

```
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 corresponding values. Intersection is sufficient because only for those tuples (i,j) for which both A(i,j) and B(i,j) are non-zero, will a non-zero be computed. This is illustrated in the Figure 12.2

Matching rows

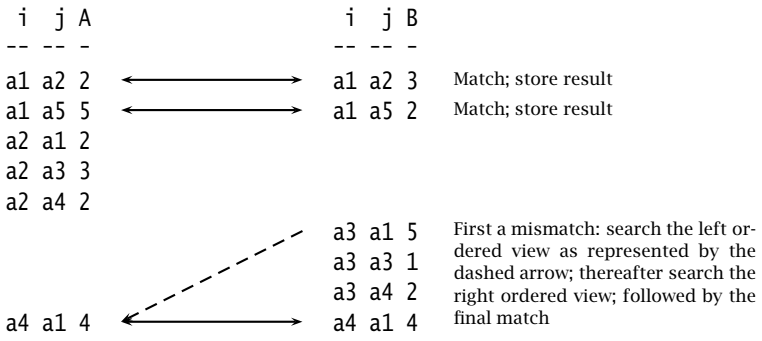

Figure 12.2: Sparse execution of the * operator

Note that the ordered views of both A and B are searchable and, thus, finding the matching elements can be efficiently implemented. 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 similar behavior. They can also be implemented using the intersection of the rows and performing the appropriate operation.

Similar operators

12.1.3 The = operator: dense behavior

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

Third statement

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

This statement is admittedly somewhat artificial. However, such conditions are frequently part of larger expressions and must be considered. The key observation is that the comparison 0.0 = 0.0 evaluates to true. In AIMMS the value '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
  a2     1        1
  a3     1        1
  a4  1  1  1  1  1
  a5  1  1  1  1  1   ;
```

Dense behavior

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

Similar operators

Other operators, such as /, **, <= and => demonstrate similar behavior. They also need to be implemented by considering all the possibilities and evaluating as one progresses.

Beware!

Increasing the number of indices, or increasing the size of the sets will make the number of rows to be considered in such operations grow rapidly. Large-dimensional dense operations are a potential cause of performance glitches in an application.

12.1.4 Behavior of combined operations

Fourth statement

The fourth statement is a variation of the third statement:

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

Speeding up

Although 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 non-zero. This is known as a domain condition on the expression. The net effect on the expression is that this condition speeds up efficient behavior by moving from dense to sparse behavior. The result of this fourth assignment is:

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

Preventing dense behavior

If your model contains a statement that performs badly due to 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.

12.1.5 Summation

Fifth statement

The fifth statement, as detailed below, is a step towards 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)` in a parameter `AI(i)`:

```
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 ⎫
a2 a3 3 ⎬ 7
a2 a4 2 ⎭
a4 a1 4 } 4
```

Figure 12.3: Sparse execution of the `Sum` operator

Running indices and identifier indices match

Each pairing 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 in 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.

Single pass is sufficient

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

12.1.6 Reordered views

Sixth statement

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

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

Non-matching index order

This time, 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).

Reordered views

In order to regain adjacency of the elements in the same group, AIMMS maintains other views of the parameter A known as *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.

```
 j  i A
-- -- -
a1 a2 2  }
a1 a4 4  } 6
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

Single pass is sufficient

Again, each pairing 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 results of AJ can be computed by a single pass over this reordered view of A. AIMMS generates and maintains reordered views on an as needs basis. They do, however, take up memory.

12.2 Modifying the sparsity

Questions

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

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

Sparse execution is correct

Sparse execution never changes 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 the possible combinations of the running indices, but only the efficiency with which these results are obtained.

Sparsity modifiers

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

Left and right operands

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 non-zero result if the associated operand(s) are non-zero. Such a change to an operator may, however, change its results in a way you may, or may not, want.

A first example: the /$ operator

Let us now consider a few examples where 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.

```
! 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 non-zero. 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 merge operator :=$

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

```
! Only overwrite elements of E(i,j) when the result
! F(i,j) + G(i,j) is non-zero.
! 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 equivalently obtained as follows.

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

Where allowed?

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.

Operator	Sparsity modifier allowed	
	$ left	$ right
^	yes	yes
*	no	no
/	no	yes
+, -	no	no
=, <>, <, <=, >, >=	yes	yes
:=	yes	yes
+=, -=	yes	no
*=, /=, ^=	yes	yes
$, ONLYIF AND, OR, XOR	no	no

Table 12.1: Sparsity modifiers of binary operators

Modifying iterative operators

In addition to modifying the behavior of binary operators, the $ sparsity modifier can also be applied to iterative operators. The effect 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 non-zero value.

Example: the Min$ operator

The third and final example of the $ sparsity modifier provided here is on the Min operator. Suppose you want to find the smallest non-zero distance between a particular node and other nodes. This can be modeled as follows:

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

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

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

Where allowed?

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

Iterative operator	Sparsity modifier allowed
	$ added
Sort, NBest	yes
Intersection,	yes
First, Last, Nth	no
ArgMin, ArgMax	yes
Sum, Union	no
Prod	yes
Min, Max	yes
Statistical operators (see also page 86)	yes
ForAll	no
Other logical operators (see also page 95)	no

Table 12.2: Sparsity modifiers of iterative operators

Usage of sparsity modifiers

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

Advantages

Using sparsity modifiers has the following advantages.

- It enables a more compact notation. In the examples above, the domain condition is replaced by a strategically placed $ sparsity modifier thereby reducing the overall expression. Many models have with multiple line subexpressions and with these the reduction is not insignificant.
- It is more efficient. There are usually abundant zeros in a model. You want them ignored so that the corresponding entries do not appear in the results. In addition, you want them to be ignored as quickly as possible: so as not to waste any computation time on them.

Disadvantages

As with any new notation it takes time to get used to it. This holds both for you as a modeler and 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 into the existing data`".

12.3 Overview of operator efficiency

Operator efficiency

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

Unary operators and functions

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

- *sparse*: Here, when the argument is 0.0, the result is 0.0. The result needs to be computed only for those tuples for which the argument has a non-zero value.
- *dense*: Here, when the argument is 0.0, the result is not equal to 0.0. The results of all possible tuples need to be computed.

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: Sparsen and dense unary operators and functions

Binary operators

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

- *intersection sparse*: Here, when either of the arguments is 0.0, the result is 0.0. The result of only those tuples need to be computed where 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*: Here, when both arguments are 0.0, the result is 0.0. The result of only those tuples need to be computed where 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.

- *dense*: Here, when both arguments are 0.0, the result is not equal to 0.0. In this 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

Iterative operators

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

- *sparse* A value 0.0 of an argument does not influence the result and can safely be ignored. The iterative operator only considers existing entries of its argument.
- *almost sparse* A second 0.0 in the argument does not influence the result. The execution starts in a dense meaning that the iterative operator considers all possible tuples. However, after a first 0.0 has been encountered, execution continues in a sparse manner.
- *dense* A value 0.0 in the argument influences the result. The iterative operator considers all possible combinations.

sparse	almost sparse	dense	
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