AIMMS Language Reference - Execution Efficiency Cookbook

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

Chapter 13

Execution Efficiency Cookbook

This chapter

Typically, when you start running your model with realistic, large-scale data sets, execution performance becomes an important issue. In this chapter, we discuss various techniques that you can use to improve the execution efficiency of your model.

Dividing the time spent

The running time of AIMMS applications can be divided in the time spent by AIMMS itself and the time spent by the solution algorithms (i.e. solvers) used by AIMMS.

Time spent by solvers

The time used by the solvers mostly depends, apart from the quality of the solver, on the specific formulation of the mathematical program to be solved. Finding a formulation that can be efficiently solved is often a challenging task and is beyond the scope of this chapter. For a detailed discussion, you are referred to the extensive literature that exists on this subject.

Time spent by AIMMS

AIMMS itself typically spends most of its time on the execution of assignment statements and the generation of constraints. This time depends on several factors. A few of these factors are:

- the size of the sets and the data set size used in your model,
- the efficiency of the AIMMS execution engine, and
- the language constructs used to formulate the execution statements and constraints.

Understanding AIMMS *execution*

At AIMMS we are committed to continuously improving the efficiency of the AIMMS execution engine and the AIMMS matrix generator. The efficiency of your application, however, does not only depend on the efficiency of AIMMS, but also on the specific formulation of your model and the language constructs that you have used. A global understanding of the AIMMS execution engine, as presented in Chapter 12, may provide a good background on which to start reconsidering particular formulations that lead to bottlenecks in execution performance in your application.

Analysis tools

In addition, AIMMS provides you with two tools for analyzing execution bottlenecks, namely the **Identifier Cardinalities** and **Profiler** tools. The use of both tools is described in Chapter 8 of the AIMMS User's Guide.

Analyzing cardinalities

The **Identifier Cardinalities** tool can help you to discover identifiers with a large number of elements. Such identifiers, when used in statements and constraints, may lead to efficiency bottlenecks throughout your model. Whenever you are able to reduce the number of elements associated with such identifiers, by leaving out irrelevant elements, the execution efficiency of your model will improve at several places. Naturally, such reductions are not possible when all the elements are relevant to the computation of the solution. In Section 13.1, we discuss two frequently observed and effective approaches to reducing the number of elements in both one-dimensional sets and multidimensional identifiers.

Analyzing statements

With the AIMMS **Profiler** tool you can identify the individual statements and constraints on which the AIMMS execution engine spends most of its time. Even if the inefficiencies are not the result of superfluous identifier cardinalities, it may still be possible to review and rewrite such statements and constraints in order to improve the execution efficiency of your application. In Section 13.2 we discuss potential bottlenecks and alternative formulations for particular statements and constraints.

Simple precautions

Before you begin tuning your application, you may want to set aside a copy of the application and inputs with known results. You can then set up a script that executes each of these tests using the AIMMS command line option `--run-only` (see also Chapter 18 of AIMMS The User's guide). In addition, you may wish to regularly commit your sources to a version control system in order to track the changes you make over time.

13.1 Reducing the number of elements

Application phases

In general, one can divide an application in three phases:

1. reading input data, often referred to as reading and preprocessing,
2. processing data, often referred to as the core model, and
3. writing output results, often referred to as reporting.

Interactive applications add the on/off switching of various application features, the setting of tuning parameters, the consideration of various scenarios, the output to screen, and so on. This does not change the basic concept, however. It only means that the inputs come from various sources and the outputs go to various destinations. An important observation is that, usually, most of the computation time is spent in the core model as this involves:

- the execution of assignments,

- the evaluation of definitions,
- the generation of constraints, and
- the execution of one or more SOLVE statements.

Reducing the core model

Obviously, the fewer data we have in the core model, the sooner we're finished. Often, a considerable percentage of the data read in during the data input phase is irrelevant to the final result. We could, therefore, consider spending more time in the data input phase and try to remove such irrelevant data with the primary objective of reducing the amount of data used in the core model. Experience shows that this effort is usually, but not always, worthwhile.

Reducing the number of elements

In this section, two complementary methods of reducing the model size are considered, namely reducing the number of elements in

- one-dimensional sets, and
- multidimensional identifiers.

13.1.1 Size reduction of one-dimensional sets

Two approaches

If, after the data input phase, a one-dimensional set contains a large number of elements that are irrelevant to the core model, there are two possible approaches to removing them from computations in the core model. These are:

- adding a condition to all identifiers indexed over that set, or
- introducing a subset of active elements, and using an index to that active subset.

These two approaches are illustrated below.

Tanks example

As a running example, consider a collection of tanks. Let us introduce a few identifiers related to tanks:

```
Set Periods {
    Index        :  t;
}
Set Tanks {
    Index        :  Tnks;
}
Set BrokenTanks {
    SubsetOf     :  Tanks;
}
Parameter StrategicReserve {
    IndexDomain  :  Tnks;
}



Parameter SizeOfTank {
    IndexDomain  :  Tnks;
```

```
}
Parameter TankIsRelevant {
    IndexDomain  :  Tnks;
    Range        :  binary;
    Definition   :  {
        1 $ [ ( not Tnks in BrokenTanks                       ) AND
              ( SizeOfTank( Tnks ) > StrategicReserve( Tnks ) )     ]
    }
}
Variable TankLevel {
    IndexDomain  :  (t,Tnks) | TanksIsRelevant( Tnks );
}
Constraint TankLimit {
    IndexDomain  :  (t,Tnks) | TanksIsRelevant( Tnks );
    Definition   :  TankLevel( t,Tnks ) <= SizeOfTank( Tnks );
}
```

The example above illustrates the first approach, in which the restriction on the tanks is embodied by the parameter TankIsRelevant.

Introducing active subsets

To illustrate the second approach, we change the above model section by introducing the *active subset* ActiveTanks and modifying the declaration of the variable TankLevel and the constraint TankLimit as presented below.

```
Set ActiveTanks {
    SubsetOf     :  Tanks;
    Index        :  tnk, tnk2;
    Definition   :  { Tnks | TankIsRelevant(Tnks) };
}
Variable TankLevel {
    IndexDomain  :  (t,tnk);
}
Constraint TankLimit {
    IndexDomain  :  (t,tnk);
    Definition   :  TankLevel( t,tnk ) <= SizeOfTank( tnk );
}
```

The core model still consists of the variable TankLevel and the constraint TankLimit but their index domain has been changed. These identifiers are now declared over active tanks only. Because of this change in the index domain, the parameter TankIsRelevant is no longer needed in their index domain condition.

Speedup by active subsets

One may argue that nothing is gained because the selection through TankIsRelevant is now replaced by the index tnk of the active subset ActiveTanks. However, the AIMMS execution engine has been tuned to select relevant elements of parameters and variables through indices in subsets. The selection via a condition such as TankIsRelevant(Tnks) will force AIMMS to retrieve the values for:

- the parameter or variable at hand,
- the parameter TanksIsRelevant, and then
- combine these values using the 'such that' operator |.

Both approaches produce identical results and limit the core model execution to relevant elements only. The first approach using the TankIsRelevant condition takes more execution time than the second approach using an index in the active subset ActiveTanks because this latter approach selects the relevant elements more directly.

Multiple active subsets

Intuitively you might expect the improvement to be minor because probably only a few tanks, if any, are removed from the collection of all tanks. However, for other indices of the model the gain may be significant. More significant gains may be observed, for example, when

- you study a few periods from a large model calendar,
- you study a few scenarios from a large database of scenarios,
- you study a rather limited region,
- there are only a few crudes available from a large collection of available crudes, or
- there are only a few products ordered from a large catalog.

A large dimensional identifier, indexed over multiple active subsets, will have the effect.

Starting with a core model

What if your model does not limit the number of elements in one-dimensional sets at all? Following the active subset approach, as illustrated above, you will have to modify the core model wherever you use the root set or an index in the root set. In such a situation, you can also implement "active subsets" by introducing a superset of the root set, and letting the original root set take on the role of an active subset.

Example

We continue the running example by presenting a core model version of it.

```
Set Periods {
    Index        :  t;
}
Set Tanks {
    Index        :  tnk;
}
```

```
Parameter SizeOfTank {
    IndexDomain  :  tnk;
}
Variable TankLevel {
    IndexDomain  :  (t,tnk);
}
Constraint TankLimit {
    IndexDomain  :  (t,tnk);
    Definition   :  TankLevel( t,tnk ) <= SizeOfTank( tnk );
}
```

In implementing the active subset approach, we introduce a new superset AllTanks and redefine the original set Tanks as an active subset of the superset AllTanks as follows.

```
Set AllTanks {
    Index        :  Tnks;
}
Set BrokenTanks {
    SubsetOf     :  AllTanks;
}
Parameter StrategicReserve {
    IndexDomain  :  Tnks;
}
Parameter TankIsRelevant {
    IndexDomain  :  Tnks;
    Range        :  binary;
    Definition   :  {
        1 $ [ ( not Tnks in BrokenTanks                      ) AND
              ( SizeOfTank( Tnks ) > StrategicReserve( Tnks ) )     ]
    }
}
Set Tanks {
    SubsetOf     :  AllTanks;
    Index        :  tnk;
    Definition   :  { Tnks | TankIsRelevant(Tnks) };
}
Parameter SizeOfTank {
    IndexDomain  :  Tnks;
    Comment      :  Now Wrt AllTanks instead of Tanks;
}
```

Note that the variable and constraint declarations in the core model above have not been altered, but their size has been reduced by the size reduction in the set Tanks.

13.1.2 Size reduction of multidimensional identifiers

Limiting multi-dimensional identifiers

Having illustrated limiting the number of elements in one-dimensional sets, we want to consider limiting the number of elements in multidimensional parameters, variables, and constraints. The AIMMS language facilitates this through the IndexDomain attribute.

Index domain conditions

Domain conditions can be specified in the IndexDomain attribute of multidimensional parameters, variables, and constraints. Whenever such an identifier is assigned, generated, or referenced in an expression, AIMMS will automatically add the domain condition so keeping your assignments and constraints more concise and efficient.

Continued example

We illustrate this by extending the above example as follows.

```
Variable Flow {
    IndexDomain  : (t,tnk,tnk2);
}
Constraint TankLevelBalance {
    IndexDomain  : (t,tnk) | t <> first(Periods);
    Definition   : {
        TankLevel(t-1,tnk)              ! Level of previous period
                   - Sum( tnk2, Flow(t,tnk,tnk2) ) ! Flow out of the tank
                   + Sum( tnk2, Flow(t,tnk2,tnk) ) ! Flow in to the tank
                   = TankLevel(t,tnk)              ! Current level
    }
    Comment      : {
        Level at end of previous period
            minus outflow
            plus  inflow is
            level at end of current period
    }
}
```

Note that, using this formulation, AIMMS generates matrix columns for every possible pair of tanks, whereas in practice only a small selection can have an actual flow. If this selection of possible connections between tanks is represented by a relation TankConnections, the constraint TankLevelBalance could be written more efficiently as:

```
Set TankConnections {
    SubsetOf     : (AllTanks, AllTanks);
}
Variable Flow {
    IndexDomain  : (t,tnk,tnk2);
}
Constraint TankLevelBalance {
    IndexDomain  : (t,tnk) | t <> first(Periods);
    Definition   : {
        TankLevel(t-1,tnk)
            - Sum( tnk2 | (tnk,tnk2) in TankConnections, Flow(t,tnk,tnk2) )
            + Sum( tnk2 | (tnk2,tnk) in TankConnections, Flow(t,tnk2,tnk) )
            = TankLevel(t,tnk)
    }
}
```

Note the repetition of the condition in the above formulation. This is because the condition is actually a restriction on the Flow variable, and should therefor be a part of its declaration. This leads to a much more concise formulation, as presented below.

```
Variable Flow {
    IndexDomain  : (t,tnk,tnk2) | (tnk,tnk2) in TankConnections;
}
Constraint TankLevelBalance {
    IndexDomain  : (t,tnk) | t <> first(Periods);
    Definition   : {
        TankLevel(t-1,tnk)
            - Sum( tnk2, Flow(t,tnk,tnk2) )
            + Sum( tnk2, Flow(t,tnk2,tnk) )
            = TankLevel(t,tnk)
    }
}
```

Using binary parameters

A frequently observed alternative to using relations is the use of binary parameters. The above example could then be written as follows:

```
Parameter TankIsConnected {
    IndexDomain  : (tnk,tnk2);
    Range        : {0, 1};
}
Variable Flow {
    IndexDomain  : (t,tnk,tnk2) | TankIsConnected(tnk,tnk2);
}
```

The outflow term of `TankLevelBalance` will then be generated as if it were written:

```
    Sum( tnk2, Flow(t,tnk,tnk2) $ TankIsConnected(tnk,tnk2) )
```

The notation using binary parameters is equivalent to that with relations. Which option you use is only a matter of taste and style.

Why use index domain conditions?

We would encourage you to employ index domain conditions, as using them has the following advantages:

1. Index domain conditions speed up the execution because:
   - They exclude irrelevant elements in assignments to parameters with an index domain condition,
   - Having index domain conditions on variables effectively makes the referencing of such variables sparse, as only relevant columns are generated, and
   - Index domain conditions on a constraint avoid the generation of irrelevant rows of that constraint.
2. Index domain conditions permits concise formulations. As illustrated above, you do not need to include the domain condition of the `Flow` variables while constructing the `TankLevelBalance` constraint. Moreover, you do not need to worry that you mightforget such a condition at a particular place in the model.

3. Whenever you determine a more restrictive condition on an identifier A, you only need to change your model at one place, namely in the index domain condition of that identifier A. You don't need to go through the entire model changing every reference to the identifier A.

Tight conditions

To make index domain conditions as effective as possible, they should remove all, or almost all, irrelevant combinations. Constructing such "tight" index domain conditions, can be far from straightforward. However, the time spent on constructing tight index domain conditions often pays off with a significant reduction in the total execution time of your model.

13.2 Analyzing and tuning statements

Analyzing and tuning statements

As illustrated in the previous section, carefully reviewing the number of elements in active subsets and the index domain conditions may lead to significant reductions in execution time. Additional reductions can be obtained by analyzing and rewriting specific time-consuming statements and constraints. In this section we will discuss a procedure which you can follow to identify and resolve potential inefficiencies in your model.

Suggested approach

You can use the AIMMS profiler to identify computational bottlenecks in your model. If you have found a particular bottleneck, you may want to use the checklist below to quickly find relevant information for the problem at hand. For each question that you answer with a yes you may want to follow the suggested option.

- Is the bottleneck a repeated expression where the combined execution of all instances takes up a lot of time? If so, you can either
  - manually replace the expression by a new parameter containing the repeated expression as a definition. Do not forget to check the NoSave property if you do not want that newly defined parameter to be stored in cases.
  - or let AIMMS do it for you, by setting the option `subst low dim expr class` to an appropriate value for your application. See also the help associated with that option.

  For a worked example, see also Subsection 13.2.4
- Is the bottleneck due to debugging/obsolete code? If so, delete it, move it to the Comment attribute, or enclose the time-consuming debugging code in something like an `IF ( DebugMode ) THEN` and ENDIF pair.
- Are you using dense operators such /, =, ^, or dense functions such as Log, Exp, Cos in which a zero argument has a non-zero result? An overview of the efficiency of such functions and operators can be found in Section 12.3. Could you add index domain conditions to make the execution of the time-consuming expressions more sparse, without changing the final result?

- Is the bottleneck part of a FOR statement? If so, is that FOR statement really necessary? For a detailed discussion about the need for and alternatives to FOR statements, see Section 13.2.1.
- Is the bottleneck the condition of the FOR statement that takes up most of the time? This is shown in the profiler by a large net time for the FOR statement. Section 13.2.2 discusses why the conditions of FOR statements may absorb a lot of computational time and discusses alternatives.
- Does the body of a FOR, WHILE, or REPEAT statement contain a SOLVE statement, and is AIMMS spending a lot of time regenerating constraints (as shown in the profiling times of the constraints)? If so, consider modifying the generated mathematical programs directly using the GMP library as discussed in Chapter 16.
- Does your model contain a defined parameter over an index, say t, and do you use this parameter inside a FOR loop that runs over that same index t? Inefficient use of this construct is indicated by the AIMMS profiler through a high hitcount for that defined parameter. See Section 13.2.3 for an example and an alternative formulation.
- Is the bottleneck an expression with several running indices? Contains this expression non-trivial sub-expressions with fewer running indices? If the answer is yes, consult Section 13.2.4 for a detailed analysis of two examples.
- Does the expression involve a parameter or a variable that is bound with a non-zero default? Section 13.2.5 discusses the possible adverse timing effects of using non-zero defaults in expressions, and how to overcome these.
- Would you expect a time-consuming assignment to take less time given the sparsity of the identifiers involved? This may be one of those rare occasions in which the specific order of running indices has an effect on the execution speed. Although tackling this type of bottleneck may be very challenging, Section 13.2.6 hopefully offers sufficient clues through an illustrative example.
- Are you using ordered sets? Reordering the elements in a set can slow execution significantly as detailed in Section 13.2.7.

13.2.1 Consider the use of FOR statements

Why avoid the FOR statement?

The AIMMS execution system is designed for efficient bulk execution of assignment statements, plus set and parameter definitions and constraints. A consequence of this design choice is that computation time is spent, just before the execution of such an executable object, analyzing and initializing that object. This is usually worthwhile except when only one element is computed at a time. Consider the following two fragments of AIMMS code that have the same final result. The first fragment uses a FOR statement:

```
for ( (i,j) | B(i,j) ) do ! Only when B(i,j) exists we want to
```

```
    A(i,j) := B(i,j);     ! overwrite A(i,j) with it.
endfor ;
```

The second fragment avoids the FOR statement:

```
A((i,j) | B(i,j)) := B(i,j); ! Overwrite A(i,j) only when B(i,j) exists
```

In the first fragment, the initialization and analysis is performed for every iteration of the FOR loop. In the second fragment the initialization and analysis is performed only once. Using the $ sparsity modifier on the assignment operator := (see also Section 12.2), the statement can be formulated even more compactly and efficiently as:

```
A(i,j) :=$ B(i,j); ! Merge B(i,j) in A(i,j)
```

In the above example, the FOR statement is used only to restrict the domain of execution of a single assignment. While using the FOR statement in this manner may seem normal to programmers, the execution engine of AIMMS can deal with conditions on assignment statements much more efficiently. As such, the use of the FOR statement is superfluous and time consuming.

When not *to remove the FOR statement*

Now that the FOR statement has been made to look inefficient, you are probably wondering why has it been introduced in the AIMMS language in the first place? Well, simply because sometimes it is needed. And it is only inefficient if used unnecessarily. So when is the FOR statement applicable? Two typical examples are:

- generating a text report file, and
- in algorithmic code inside the core model.

We will discuss these examples in the next two paragraphs.

Generating text reports

The AIMMS DISPLAY statement is a high level command that outputs an identifier in tabular, list, or composite list format with a limited amount of control. In addition, the output of the DISPLAY statement can always be read back by AIMMS, and, to enable that requirement, the name of the identifier is always included in the output. Thus, the AIMMS DISPLAY statement usually fails to meet the specific formatting requirements of your application domain, and you end up needing control over the position of the output on an element-by-element basis. This requires the use of FOR statements. However, depending on the purpose of your text report file, there might be very good alternatives available:

- When this reporting is for printing purposes only, you may want to consider the AIMMS print pages as explained in AIMMS The User's Guide Chapter 14. These print pages look far better than text reports.

- When the report file is for communication with other programs, you may want to consider whether communication using relational databases (see Chapter 27), or through XML (see Chapter 30) form better alternatives. For communication with EXCEL or OpenOffice Calc, a library of dedicated functions is built in AIMMS (see Chapter 29).

Algorithmic code inside the core model

A FOR statement is needed whenever your model contains two statements where:

- the computation of the last statement depends on the computation of the first statement, and
- the computation of the first statement depends on the results of the last statement obtained during a previous iteration.

Iterating unneccesarily

FOR statements may be especially inefficient, if the condition of a FOR statement allows elements for which none of the statements inside the FOR loop modify the data in your model or generate output. This is illustrated in the following example.

Transposing a distance matrix

Consider a distance matrix, D(i,j), with only a few entries per row in its lower left half containing the distances to near neighbors. You also want it to contain the reverse distances. One, inefficient, but valid, way to formulate that in AIMMS is as follows:

```
for ( (i,j) | i > j ) do ! The condition 'i > j' ensures we only
    D(i,j) := D(j,i) ;   ! write to the upper right of D.
endfor ;
```

Why inefficient?

There are two reasons why the above is inefficient:

- Although there is a condition on the FOR loop, this condition permits many combinations of (i,j) that do not invoke execution as D(i,j) was sparse to begin with. A tempting improvement would be to add D(j,i) to the condition on the FOR loop. However, this will lead to other problems, however, as will be explained in the next section.
- As explained in Section 12.1.6, AIMMS maintains reordered views. For each non-zero value computed and assigned to the identifier D(i,j), AIMMS will need to adapt the reordered view for D(j,i), and re-initialize searching in that reordered view.

Suggested modification

In the example at hand we can move the condition on the FOR loop to the assignment itself and simply remove the FOR statement altogether (but not its contents). The example then reads:

```
D((i,j) | i > j) := D(j,i) ; ! The condition 'i > j' ensures we only
                             ! write to the upper right of D.
```

Using application domain knowledge

We can improve the assignment further by noting that we are actually merging the transposed lower half in the identifier itself, and that there is no conflict in the elements. This can be achieved by a $ sparsity modifier on the assignment operator. The $ sparsity modifier and the opportunity it offers are introduced in Section 12.2. The example can then be written as:

```
D(i,j) :=$ D(j,i); ! Merge the transpose of the lower half in the identifier itself.
```

13.2.2 Ordered sets and the condition of a FOR statement

Modifying the FOR condition

The condition placed on a FOR statement is like any other expression evaluated one element at a time. However, during that evaluation, the identifiers referenced in the condition may have been modified by the statements inside the FOR loop. In general, this is not a problem, except when the range of the running index of the FOR statement is an *ordered* set. In that situation, the evaluation of the condition itself becomes time consuming as the tuples satisfying the condition have to be repeatedly computed and sorted, as illustrated below.

Continued example

Let us again consider the example of the previous section with the parameter D now added to the FOR loop condition, and the set S ordered lexicographically. As an efficient formulation has already been presented in the previous section, it looks somewhat artificial, but similar structures may appear in real-life models.

```
Set S {
    Index      : i,j;
    OrderBy    : i ! lexicographic ordering.;
    Body       : {
        for ( (i,j) | ( i > j ) AND D(j,i) ) do ! Only execute the statements in the
            D(i,j) := D(j,i) ;                  ! loop when this is essential.
        endfor
    }
}
```

What does AIMMS *do in this example?*

First note that the FOR statement respects the ordering of the set S. Because of this ordering, AIMMS will first evaluate the entire collection of tuples satisfying the condition (i > j) AND D(j,i), and subsequently order this collection according to the ordering of the set S. Next, the body of the FOR statement is executed for every tuple in the ordered tuple collection. However, when an identifier, such as D in this example, is modified inside the body of the FOR loop AIMMS will need to recompute the ordered tuple collection, and continue where it left off. This not only sounds time consuming, it is.

FOR as a bottleneck

If the following three conditions are met, the condition of a FOR statement becomes time consuming:

- the indices of a FOR statement have a specified element order,
- the condition of the FOR statement is changed by the statements inside the loop, and
- the product of the cardinality of the sets associated with the running indices of the FOR statement is very large.

if these three conditions are met, AIMMS will issue a warning when the number of re-evaluations reaches a certain threshold.

Improving efficiency

There are several ways to improve the efficiency of inefficient FOR statements. To understand this, it is necessary to explain a little more about the execution strategies available to AIMMS when evaluating FOR statements, as each strategy has its own merits and drawbacks. Therefore, consider the FOR statement:

```
for ( (i,j,k) | Expression(i,j,k) ) do
   ! statements ...
endfor;
```

where i, j and k are indices of some sets, each with a specified ordering, and Expression(i,j,k) is some expression over the indices i, j and k.

The sparse strategy

The first strategy, called the *sparse* strategy, fully evaluates Expression(i,j,k), and stores the result in temporary storage before executing the FOR statement. Subsequently, for each tuple (i,j,k) for which a non-zero value is stored, the statements within the FOR loop are executed. If an identifier is modified during the execution of these statements, then the condition Expression(i,j,k) has to be fully re-evaluated.

The dense strategy

The second strategy, called the *dense* strategy, evaluates Expression(i,j,k) for all possible combinations of indices (i,j,k). As soon as a non-zero result is found the statements are executed. Re-evaluation is avoided, but at the price of considering every (i,j,k) combination.

The unordered strategy

The third strategy, called the *unordered* strategy, uses the normal sparse execution engine of AIMMS but ignores the specified order of the indices. This may, however, give different results, especially when the FOR loop contains one or more DISPLAY/PUT statements or uses lag and lead operators in conjunction with one or more of the ordered indices.

Selecting a strategy

By prefixing the FOR statement with one of the keywords SPARSE, ORDERED, or UNORDERED (as explained in Section 8.3.4), you can force AIMMS to adopt a particular strategy. If you do not explicitly specify a strategy, AIMMS uses the sparse strategy by default, and only issues a warning if an identifier referenced inside the FOR loop is modified and the second evaluation of Expression(i,j,k) gives a non-empty result.

Improving efficiency

Given the above, you have the following options for improving the efficiency of the FOR statement.

- Rewrite the FOR statement such that the condition does not change during each iteration.
- Prefix the FOR statement with the keyword UNORDERED such that the unordered strategy will be set. You can safely choose this strategy if the element order is not relevant for the FOR statement. In all other cases, the semantics of the FOR statement will be changed.
- Prefix the FOR statement with the keyword ORDERED such that the dense strategy is selected. You can safely choose this strategy if the condition on the running indices evaluates to true for a significant number of all possible combinations of the tuples (i,j,k).
- Prefix the FOR statement with the keyword SPARSE to adopt the sparse strategy. However, all warnings will be suppressed relating to the condition on the running indices needing to be evaluated multiple times. You can choose this strategy if the condition needs to be re-evaluated in only a few iterations.

13.2.3 Combining definitions and FOR loops

Dependency is symbolic

As explained in Section 7.1, the dependency structure between set and parameter definitions is based only on symbol references. AIMMS' evaluation scheme recomputes a defined parameter *in its entirety* even if only a single element in its inputs has changed. This negatively affects performance when such a defined parameter is used inside a FOR loop and its input is changed inside that same FOR loop.

A simulation example

A typical example occurs when using definitions in simulations over time. In simulations, computations are often performed period by period, referring back to data from previous period(s). The relation used to computate the stock of a particular product p in period t can easily be expressed by the following definition and then used inside the body of a procedure.

```
Parameter ProductStock {
    IndexDomain  :  (p,t);
    Definition   :  ProductStock(p,t-1) + Production(p,t) - Sales(p,t);
}
Procedure ComputeProduction {
```

```
    Body         : {
        for ( t ) do
            ! Compute Production(p,t) partly based on the stock for period (t-1)
            Production(p,t) := Max( ProductionCapacity(p),
                                    MaxStock(p) - ProductStock(p,t-1) + Sales(p,t) );
        endfor ;
    }
}
```

During every iteration, the production in period t is computed on the basis of the stock in the previous period and the maximum production capacity. However, because of the dependency of ProductStock with respect to Production, AIMMS will re-evaluate the definition of ProductStock in its entirety for *each* period before executing the assignment for the next period. Although the FOR loop is not really necessary here, it is used for illustrative purposes.

Improved formulation

In this example, execution times can be reduced by moving the definition of ProductStock to an explicit assignment in the FOR loop.

```
Parameter ProductStock {
    IndexDomain  :  (p,t);
    ! Definition attribute is empty.
}
Procedure ComputeProduction {
    Body         : {
        for ( t ) do
            ! Compute Production(p,t) partly based on the stock for period (t-1)
            Production(p,t) := Max( ProductionCapacity(p),
                                    MaxStock(p) - ProductStock(p,t-1) + Sales(p,t) );

            ! Then compute stocks for current period t
            ProductStock(p,t) := ProductStock(p,t-1) + Production(p,t) - Sales(p,t);
        endfor ;
    }
}
```

In this formulation, only one slice of the ProductStock parameter is computed per period. A drawback of this formulation is that it will have to be restated at various places in your model if the inputs of the definition are assigned at several places in your model.

Use of macros

As an alternative, you might consider the use of a Macro (see also Section 6.4) to localize the defining expression of ProductStock at a single node in the model tree. The disadvantage of macros is that they cannot be used in DISPLAY statements, or saved to cases.

When to avoid definitions

As illustrated above, it is best to avoid definitions, if, within a FOR loop, you only need a slice of that definition to modify the inputs for another slice of that same definition. AIMMS is not designed to recognize this situation and will repeatly evaluate the entire definition. The AIMMS profiler will expose such definitions by their high hitcount.

13.2.4 Identifying lower-dimensional subexpressions

Lower-dimensional subexpressions

Repeatedly performing the same computation is obviously a waste of time. In this section, we will discuss a special, but not uncommon, instance of such behavior, namely lower-dimensional sub-expressions. A lengthy expression, that runs over several indices, can have distinct subexpressions that depend on fewer indices. Let us illustrate this with two examples, the first being an artificial example to explain the principle, and the second a larger example that has actually been encountered in practice and permits the discussion of related issues.

Artificial example

Consider the following artificial example:

```
F(i,k) := G(i,k) * Sum[j | A(i,j) = B(i,j), H(j)] ;
```

For every value of i, the sub-expression Sum[j | A(i,j) = B(i,j), H(j)] results in the same value for each k. Currently, the AIMMS execution engine will repeatedly compute this value. It is more efficient to rewrite the example as follows.

```
FP(i) := Sum[j | A(i,j) = B(i,j), H(j)] ;
F(i,k) := G(i,k) * FP(i) ;
```

The principle of introducing an identifier for a specific sub-expression often leads to dramatic performance improvements, as illustrated in the following real-life example.

A complicated assignment

Consider the following 4-dimensional assignment involving region-terminal-terminal-region transports. Here, sr and dr (source region and destination region) are indices in a set of Regions with m elements and st and dt (source terminal and destination terminal) are indices in a set of Terminals with n elements.

```
Transport( (sr,st,dt,dr) |          TRDistance(sr,st) <= MaxTRDistance(st) AND
                                    TRDistance(dr,dt) <= MaxTRDistance(dt) AND
   sr <> dr AND MinTransDistance <= RRDistance(sr,dr) <= MaxTransDistance  AND
   st <> dt AND MinTransDistance <= TTDistance(st,dt) <= MaxTransDistance
         ) := Demand(sr,dr);
```

The domain condition states that region-terminal-terminal-region transport should only be assigned if the various distances between regions and/or terminals satisfy the given bounds.

Efficiency analysis

The <= operator is dense and be evaluated for all possible values of the indices. The subexpression TRDistance(sr,st) <= MaxTRDistance(st), for example, will be evaluated for every possible value of dr and dt, even though it only depends on sr and st. In other words, we're computing the same thing over and over again.

Effect of the AND operator

There are multiple AND operators in this example. The AND operator is sparse, and oten, sparse operators make execution quick. However, they fail to do just that in this particular example. Bear with us. Although the AND operator is a sparse binary operator, its effectiveness depends on how effectively the intersection is taken. What are we taking the intersection of? If we consider a particular argument of the AND operators: TRDistance(sr,st) <= MaxTRDistance(st), as the operator <= is dense and this argument will be computed for all tuples {(sr,st,dt,dr)} even though the results will be mostly 0.0's. The domain of evaluation for this argument is thus the full Cartesian product of four sets. The evaluation domain of the other arguments of the AND operators will be the same. So, in this example, we are repeatedly taking the intersection of a Cartesian product with itself, resulting in that same Cartesian product. Thus, the AND operator will be evaluated for all tuples in {(sr,st,dt,dr)} even though this operator is sparse.

Example reformulation

In the formulation below, we've named the following sub-expressions.

```
ConnectableRegionalTerminal( (sr,st) | TRDistance(sr,st) <= MaxTRDistance(st) ) := 1;

ConnectableRegions( (sr,dr) | sr <> dr AND
        MinTransDistance <= RRDistance(sr,dr) <= MaxTransDistance )  := 1;

ConnectableTerminals( (st,dt) | st <> dt AND
        MinTransDistance <= TTDistance(st,dt) <= MaxTransDistance )  := 1;
```

In each of these three assignments, each condition depends fully on the running indices and thus its evaluation is not unnecessarily repeated. By substituting the three newly introduced identifiers in the condition the original assignment becomes:

```
Transport( (sr,st,dt,dr) |
           ConnectableRegionalTerminal(sr,st)     AND
           ConnectableRegionalTerminal(dr,dt)     AND
           ConnectableRegions(sr,dr)              AND
           ConnectableTerminals(st,dt) )
         := Demand(sr,dr);
```

The newly created identifiers are all sparse, and the sparse operator AND can effectively use this created sparsity in its arguments.

Effect of reformulation

A modified version of the above example was sent to us by a customer. While the original formulation took several minutes to execute for a given large dataset, the reformulation only took a few seconds.

A bit of general advice

Perhaps a modeling style which avoids the need for substitutions is best. The easy way is to let AIMMS identify the places in which such substitutions can be made by switching the options in the option category Aimms - Tuning - Substitute Lower Dimension Expressions to appropriate settings. The disadvantage of this easy method is that some opportunities are missed as AIMMS cannot guarantee the equivalence of the formulations, and some replacements are missed. For instance, in the above example, AIMMS will create an identifier for both TRDistance(sr,st) <= MaxTRDistance(st) and TRDistance(dr,dt) <= MaxTRDistance(dt), even though only one suffices. You can avoid substitutions by keeping your expressions brief relating only a few identifiers at a time. This will also help to keep your model readable.

13.2.5 Parameters with non-zero defaults

Sparse execution expects 0.0's

Sparse execution is based on the effect of the number 0.0 on addition and multiplication. When other numbers are used as a default, all possible elements of these parameters need to be considered rather than only the stored ones. The advice is not to use such parameters in intensive computations. In the example below, the summation operator will need to consider every possible element of P rather than only its non-zeros.

```
Parameter P {
    IndexDomain  :  (i,j);
    Default      :  1;
    Body         : {
        CountP := Sum( (i,j), P(i,j) )
    }
}
```

Appropriate use of default

Identifiers with a non-zero default, may be convenient, however, in the interface of your application as the GUI of AIMMS can display non-default elements only.

The NonDefault function

For parameters with a non-zero default, you still may want to execute only for its non-default values. For this purpose, the function NonDefault has been introduced. This function allows one to limit execution to the data actually stored in such a parameter. Consider the following example where the non defaults of P are summed:

```
CountP := Sum( (i,j)| NonDefault(P(i,j)), P(i,j) );
```

In the above example the summation is limited to only those entries in P(i,j) that are stored. If you would rather use familiar algebraic notation, instead of the dedicated function NonDefault, you can change the above example to:

```
CountP := Sum( (i,j) | P(i,j) <> 1, P(i,j) );
```

This statement also sums only the non-default values of P. AIMMS recognizes this special use of the <> operator as actually using the NonDefault function; the summation operator will only consider the tuples (i,j) that are actually stored for P.

Suffices of variables

Note that the suffices .Lower and .Upper of variables are like parameters with a non-zero default. For example in a free variable the default of the .lower suffix is -inf and the default of the suffix .upper is inf.

13.2.6 Index ordering

Index ordering

In rare cases, the particular order of indices in a statement may have an adverse effect on its performance. The efficiency aspects of index ordering, when they occur, are inarguably the most difficult to understand and recognize. Again, this inefficiency is best explained using an example.

An artificial example

Consider the following assignment statement:

```
FS(i,k) := Sum( j, A(i,j) * B(j,k) );
```

If A(i,j) and B(j,k) are binary parameters, where

- for any given i, the parameter A(i,j) maps to a single j, and,
- for any given j, the parameter B(j,k) maps to a single k,

one would intuitively expect that the assignment could be executed rather efficiently. When actually executing the statement, it may therefore come as an unpleasant surprise that it takes a seemingly unexplainable amount of time.

An analysis

In the qualitative analysis above, implicitly the index order i selects j, and j selects a few k's, or, in AIMMS terminology, a running index order [i,j,k]. The actual running index order of AIMMS is, however, first the indices [i,k] from the assignment operator, followed by the index [j] from the summation operator: [i,k,j]. The effect of the actual index order is that, for a given value of index i, the relevant values of index k cannot be restricted using the parameter chain A(i,j)-B(j,k) without the aid of an intermediate running index j. Consequently, AIMMS has to try every combination of (i,k).

Reformulation

Given the above analysis, the preferred index ordering [i,j,k] can be accomplished by introducing an intermediate identifier FSH(i,j,k), and replacing the original assignment by the following statements.

```
FSH(i,j,k) := A(i,j) * B(j,k);
FS(i,k) := Sum( j, FSH(i,j,k) );
```

With a real-life example, where the range of the indices i, j and k contained over 10000 elements, the observed improvement was more than a factor 50.

Not for +

A similar improvement could be obtained for the following example:

```
FSP(i,k) := Sum( j, A(i,j) + B(j,k) );
```

Here a value is computed for each (i,k) of FSP, because, for every i, there is a non-zero A(i,j), and for every k, there is a non-zero B(j,k).

13.2.7 Set element ordering

Data entry order

By default, all elements in a root set are numbered internally by AIMMS in a consecutive manner according to their *data entry order*, i.e. the order in which the elements have been added to the set. Such additions can be either explicit or implicit, and may take plac, for example when the model text contains references to explicit elements in the root set, or by reading the set from files, databases, or cases.

Multi-dimensional storage

The storage of multidimensional data defined over a root set is always based on this internal and consecutive numbering of root set elements. More explicitly, all tuple-value pairs associated with a multidimensional identifier are stored according to a strict right-to-left ordering based on the respective root set numberings.

Indexed execution

By default, all indexed execution taking place in AIMMS, either through implied loops induced by indexed assignments or through explicit FOR loops, employs the same strict right-to-left ordering of root set elements. Thus, there is a perfect match between the execution order and the order in which identifiers referenced in such loops are stored internally. As a consequence, it is very easy for AIMMS to synchronize the tuple for which execution is currently due with an ordered route through all the non-zero tuples in the identifiers involved in the statement. This principle is the basis of the sparse execution engine underlying AIMMS.

Execution over ordered sets

Inefficiency is introduced if the elements in a set over which a loop takes place have been ordered differently from the data entry order, either because of an ordering principle specified in the OrderBy attribute of the set declaration or through an explicit Sort operation. Consequently, there will no lomger be a direct match between the execution order of the loop and the storage order of the non-zero identifier values. Depending on the precise type of statement, this may result in no, slight or serious increase in the execution time of the statement, as AIMMS may have to perform randomly-placed lookups for particular tuples. These random lookups are much more time consuming than running over the data only once in an ordered fashion.

Effect on FOR loops

In particular, you should avoid using FOR statements in which the running index is an index in a set with a nondefault ordering whenever possible. If not, AIMMS is forced to execute such FOR statements using the imposed nondefault ordering and, as a result, *all* identifier lookups within the FOR loop are random. In such a situation, you should carefully consider whether ordered execution is really essential. If not, it is advisable to leave the original set unordered, and create an ordered *subset* (containing all the elements of the original set) for use when the nondefault element ordering is required.

Effect on assignments

In most situations, the efficiency of indexed assignments is not affected by the use of indices in sets with a nondefault ordering. AIMMS has only to rely on the nondefault ordering if an assignment contains special order-dependent constructs such as lag and lead operators. In all other cases, AIMMS can use the default data entry order.

Complete reordering

If a nondefault ordering of some sets in your model causes a serious increase in execution times, you may want to apply the CLEANDEPENDENTS statement (see also Section 25.3) to those roots sets that are the cause of the increase of execution times. The CLEANDEPENDENTS statement will fully renumber the elements in the root set according to their current ordering, and rebuild all data defined over it according to this new numbering.

Use sparingly

As all identifiers defined over the root set have to be completely rebuilt, CLEANDEPENDENTS is an inherently expensive operation. You should, therefore, only use it when really necessary.

13.2.8 Using AIMMS' advanced diagnostics

Using diagnostic warnings

In order to help you create correct and efficient applications, AIMMS is regularly extended with diagnostics that incorporate the recognition of new types of problematic situations. These diagnostics may help you detect model formulations that lead to sub-optimal performance and/or ambiguous results.

These diagnostics can be controled through various options in the **Warning** category.

Apply diagnostics regularly

As the list of diagnostic options is regularly extended, and some of the formulation problems depend on the model data and, thus, can only be detected at runtime, you are advised to apply the diagnostics provided by AIMMS on a regular basis during your application tests. Section 8.4.4 describes a way in which you can switch on all the diagnostic options by just changing the value of the two options `strict_warning_default` and `common_warning_default`.

Diagnostic options

Below we provide a list of performance-related diagnostics that may help you tune the performance of your model:

- **`Warning_repeated_iterative_evaluation:`** If the arguments of an iterative operator do not depend on some of the indices, the iterative operator is repeatedly evaluated with the same result. Consider the assignment `a(i) := sum(j,b(j));` in which the sum does not depend on the index `i` and so the same value is computed for every value of `i`. See also Subsection 13.2.4.
- **`Warning_unused_index:`** If an index is not used inside the argument(s) or index domain condition of an iterative iterator, this leads to inefficient execution. In the assignment `a(i) := sum((j,k),b(i,j));`, the index `k` is not used in the summation. Further, when an index in the index domain of a constraint is not used inside the definition of that constraint this is likely to lead to the generation of duplicate rows.
- **`Warning_duplicate_row:`** At the end of generating a mathematical program it is verified that there are no duplicate rows inside that mathematical program. This might be caused by two constraints having the same definition. Besides consuming more memory, duplicate rows cause the problem to become degenerate and may cause the problem to become more difficult to solve. This warning is not supported for mathematical programs of type MCP or MPCC because, for these types the row col mapping is also relevant and duplicate rows cannot be simply eliminated.
- **`Warning_duplicate_column:`** At the end of generating a mathematical progrram it is verified that there are no duplicate columns inside that mathematical program. Besides consuming more memory, duplicate columns result in the generated mathematical program having non-unique solutions.
- **`Warning_trivial_row:`** Generating and eliminating trivial rows such as 0 <= 1 takes time.

The help for the option category `AIMMS - Progress, errors & warnings - warnings` provides more information on these options.

13.3 Summary

The recipe boils down to three steps

This chapter consists of a recipe for fine tuning an existing AIMMS application such that AIMMS more efficiently executes the definitions and statements and efficiently generates the constraints. The recipe consists of the following three steps:

- First, construct active subsets by removing all elements for which the variables are fixed in advance. These active subsets should then be used throughout your core model. This reduces the work each time, even in the evaluation of the index domain conditions to be constructed next.
- Second, construct index domain conditions for the parameters, variables, and constraints of the core model. This will make several dense expressions seemingly execute more sparse, because only a limited number of elements are evaluated. Especially with variables and constraints this avoids the generation of columns for fixed variables and empty constraints. Thus the number of bottlenecks in your application is further reduced.
- Finally, use the AIMMS profiler to pinpoint those assignment statements, FOR loops, and constraints that still absorb a considerable amount of computation time, and analyze and possible modify them. A checklist that can be used for this analysis has been presented in Section 13.2.