Analyzing Your Data

The key to creating an efficient MDDB is thorough analysis of both your data and its users. Armed with this analysis, you can determine

- the best order for the classification variables in the NWAY cube
- whether or not you should create subcubes
- which classification variables should be included in the subcubes.

While it is possible to create an MDDB that consists of the NWAY cube only, this approach would require that the NWAY cube fulfill all requests made by OLAP clients. While this approach saves on storage space, you can improve query performance if you create subcubes based on anticipated client requests. In order to create an MDDB that requires the least storage space while providing users with the best response time, consider the following issues:

- the warehouse data and how it will be used in the OLAP application
- the business problem and user expectations
- the need for an iterative approach.

By choosing and ordering classification variables for the NWAY cube and subcubes based on the needs of users, you will create an MDDB that meets the functional priorities dictated by your organization's business requirements.
Creating useful and efficient MDDBs is best accomplished as an iterative process that fits into the overall data warehouse and business intelligence strategies of your organization. You can analyze the usage patterns to determine whether your MDDB is defined correctly and make adjustments if necessary. For example, if a defined subcube is rarely or never accessed, it can be safely removed from the MDDB creation. Alternately, if there are subcubes that are not defined but are frequently requested, they can be added to the MDDB creation. (The MDDB must be rebuilt after adding or removing subcubes to the definition because subcubes cannot be added to or removed from existing MDDBs.)

Using a Spiral Diagram as a Starting Point

One way you can create an initial MDDB that will meet most users’ requirements and performance expectations is to analyze your data by using a spiral diagram. By placing the classification variables on axes representing the hierarchies of the MDDB, you can develop an acceptable data model for the MDDB.

Imagine that you have a base table that contains retail sales information. The classification variables in this table can be grouped into four dimensions, or groups of variables with similar characteristics:

- product (SECTOR, GRP, FAMILY, ARTICLE)
- time (YEAR, MONTH, DAY)
- geography (REGION, SHOP)
- supplier (GRP_SUPP, SUPPLIER).

First, create an axis for each dimension. Then, place the classification variables on the appropriate axes (working from the outside to the center) in ascending order of cardinality (number of unique values), as shown in the following diagram. Each variable becomes a dimensional level for that dimension, with the outermost variable at the top dimensional level.
The placement of the axes in relation to each other can be significant. You might want to try several arrangements to find one that works best for you. The following are possible arrangements:

- arrange axes in descending order of likelihood of use
- arrange axes in descending order of cardinality at the top dimensional level
- arrange axes in descending order of number of levels.

Now you can draw a spiral on the diagram that will indicate a general ordering scheme for the variables. Starting on the outside with the classification variable with the lowest cardinality that is also the one most likely to be of interest to users, draw a line from this variable to an adjacent variable on the diagram. Continue in this manner, spiraling in toward the center. As in this example, you might have to deviate slightly from this pattern when multiple variables near the center have high cardinality. Here, the spiral is drawn so that FAMILY is after SUPPLIER instead of DAY.
You can then produce an ordered list of classification variables based on the diagram. For this example, the order would be:

YEAR
SECTOR
REGION
GRP_SUPP
MONTH
GRP
SHOP
SUPPLIER
FAMILY
DAY
ARTICLE

From this list, we can develop an intelligent initial choice of crossings. Start with the whole list and successively drop the last item in a “stair-step” fashion to form a new crossing, as follows:

YEAR SECTOR REGION GRP_SUPP MONTH GRP SHOP SUPPLIER FAMILY DAY ARTICLE
YEAR SECTOR REGION GRP_SUPP MONTH GRP SHOP SUPPLIER FAMILY
YEAR SECTOR REGION GRP_SUPP MONTH GRP SHOP
YEAR SECTOR REGION GRP_SUPP MONTH
YEAR SECTOR REGION
YEAR SECTOR
From this list of crossings, you can define the NWAY cube and subcubes of the MDDB accordingly.

**Ensuring Optimal Ordering of Classification Variables**

In Version 7 and later releases, the order in which you specify classification variables for both the NWAY cube and any subcubes affects not only performance during creation of the MDDB but also reporting performance when the MDDB is used. The procedure for ensuring optimal ordering of the classification variables requires two steps:

1. Assess cardinality of each variable.
2. Sort the base table.

The goal in assessing the cardinality (number of unique values) of the variables is to determine their order based on ascending cardinality. If you specify the classification variables in this order when you create the MDDB, you might not achieve the lowest possible MDDB size, but the MDDB should be of an acceptable size, and this will help reduce disk I/O time dramatically. (See “Advanced Optimal Ordering Techniques” on page 15 for information on how to more accurately determine optimal classification variable ordering.)

The following example illustrates this analysis. Assume that your base table contains six classification variables with unique values for each variable.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of Unique Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFICE</td>
<td>50</td>
</tr>
<tr>
<td>SITE</td>
<td>35,000</td>
</tr>
<tr>
<td>PAGE</td>
<td>52,000</td>
</tr>
<tr>
<td>YEAR</td>
<td>10</td>
</tr>
<tr>
<td>MONTH</td>
<td>12</td>
</tr>
<tr>
<td>DAY</td>
<td>31</td>
</tr>
</tbody>
</table>

Before creating an MDDB based on this base table, you would sort the table based on the following order:

```
YEAR MONTH DAY OFFICE SITE PAGE
```

That is, sort the base table by each classification variable, starting with the variable with the lowest number of unique values and ending with the variable with the largest number of unique values. Then, use this same order when you specify classification variables for the NWAY cube and subcubes.

**Advanced Optimal Ordering Techniques**

Although the method described in the previous section for determining the acceptable classification variable order is suitable for the majority of cases, it will not
necessarily result in optimal ordering. This section provides information about how to achieve better MDDB performance through a more in-depth analysis of classification variable ordering called the derived ordering technique.

In order to understand how the best ordering can be obtained, it is important to understand the MDDB internal structures. When an MDDB is created, it is divided into two separate pieces: the data and the navigational index, which describes how to access the data. The size of the data is fixed in all cases, but depending on the structure of the data, the index can vary in size by 50 percent or more. Therefore, optimal results can be obtained if you work to minimize the index size wherever possible. Note that variation in index size is determined by the relationships between the classification variables. The analysis variables have no impact on the size of the index.

**Ordering Example**

The best way to understand the nature of the relationship between the classification variables is to look at an example. Imagine that you have three classification variables, YEAR, QUARTER, and MONTH, and that the data spans a ten-year period as shown in the following table.

<table>
<thead>
<tr>
<th>YEAR</th>
<th>QUARTER</th>
<th>MONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>Q1</td>
<td>January</td>
</tr>
<tr>
<td>1990</td>
<td>Q1</td>
<td>February</td>
</tr>
<tr>
<td>1990</td>
<td>Q1</td>
<td>March</td>
</tr>
<tr>
<td>1990</td>
<td>Q2</td>
<td>April</td>
</tr>
<tr>
<td>1990</td>
<td>Q2</td>
<td>May</td>
</tr>
<tr>
<td>1990</td>
<td>Q2</td>
<td>June</td>
</tr>
<tr>
<td>1990</td>
<td>Q3</td>
<td>July</td>
</tr>
<tr>
<td>1990</td>
<td>Q3</td>
<td>August</td>
</tr>
<tr>
<td>1990</td>
<td>Q3</td>
<td>September</td>
</tr>
<tr>
<td>1990</td>
<td>Q4</td>
<td>October</td>
</tr>
<tr>
<td>1990</td>
<td>Q4</td>
<td>November</td>
</tr>
<tr>
<td>1990</td>
<td>Q4</td>
<td>December</td>
</tr>
<tr>
<td>1991</td>
<td>Q1</td>
<td>January</td>
</tr>
<tr>
<td>1991</td>
<td>Q1</td>
<td>February</td>
</tr>
<tr>
<td>1991</td>
<td>Q1</td>
<td>March</td>
</tr>
<tr>
<td>1991</td>
<td>Q2</td>
<td>April</td>
</tr>
<tr>
<td>1991</td>
<td>Q2</td>
<td>May</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For any given MONTH value, there are 10 corresponding YEAR values and one corresponding QUARTER value. For any given YEAR value, there are 12 corresponding...
MONTH values and 4 corresponding QUARTER values. For any given QUARTER value, there are 10 corresponding YEAR values and 3 corresponding MONTH values. Put another way, you can view the relationships between the classification variables as follows:

YEAR members have a 1 to 4 relationship with QUARTER members.
YEAR members have a 1 to 12 relationship with MONTH members.
QUARTER members have a 1 to 3 relationship with MONTH members.
QUARTER members have a 1 to 10 relationship with YEAR members.
MONTH members have a 1 to 10 relationship with YEAR members.
MONTH members have a 1 to 1 relationship with QUARTER members.

The next section illustrates how these relationships affect the index size for an MDDB.

**Calculating Index Size**

The navigational index of an MDDB consists of nodes that represent combinations of classification variable values. The number of nodes in an index depends on which variables are at the different levels of the index. The index size, that is, the number of nodes, can be calculated with a simple formula. For an index containing three variables, you can use either of the following equivalent formulas:

\[
\text{Index nodes} = x + xy + xyz
\]

or

\[
\text{Index nodes} = x(1 + y(1 + z))
\]

where

- \(x\) is the number of members in the top-level variable
- \(y\) is the number of members of the second-level variable that relate to each member of the top-level variable
- \(z\) is the number of members of the third-level variable that relate to each member of the second-level variable.

Using the above formula on each of the six possible scenarios, or possible orderings, you get these results in the following table. The goal is to find the scenario with the smallest number of index nodes: the smaller the number of nodes, the better the performance.

<table>
<thead>
<tr>
<th>Table 3.3 Calculating Index Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scenario</strong></td>
</tr>
<tr>
<td>MONTH YEAR QUARTER</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The optimal classification variable ordering for the MDDB is QUARTER MONTH YEAR, with 136 index nodes. If you refer to the relationships between all the classification variables, you can see that the YEAR variable should be last in the order. This is because all members of the YEAR variable will always cross with all the members of all the other classification variables. Therefore, no matter where the YEAR variable exists in the ordering, it will always expand out fully for every member of the next-higher level of the index.

Note: Remember also that once you establish the optimal classifier ordering, you should also sort the base table into that order to speed up the MDDB building process.

**Extending the Example**

The previous example can be extended by adding two classification variables: PRODUCT (with 8 members: surfboards, swimsuits, water skis, jet skis, snowboards, snow skis, parkas, and dog sleds) and COUNTRY (with 4 members: U.S., Canada, Germany, and Italy).

The relationships between the classification variables are as follows:

<table>
<thead>
<tr>
<th>Table 3.4 Classification Variable Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Classification Variable</td>
</tr>
<tr>
<td>YEAR</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Notice now that the relationships are not as straightforward as before. For instance, each product will likely only be purchased six months out of a year, but those six months could fall over three quarters. Furthermore, each member in the COUNTRY variable crosses with all other members of all other classification variables. The formula for calculating index size that we used in the first example will not work for this example. When working with five classification variables, you use this formula:

\[ \text{Index nodes} = x(1 + y (1 + z (1 + a (1 + b)))) \]

where \(a\) and \(b\) follow the pattern, established in the previous example, for the fourth and fifth variables.

The task now is to use the above information about the relationships between the classification variables to produce the ordering for \(x\), \(y\), \(z\), \(a\), and \(b\) that will yield the smallest number of index nodes. By examining the relationships between the classification variables, you can deduce the optimal classifier ordering to be

QUARTER MONTH PRODUCT COUNTRY YEAR

The corresponding formula is

\[ 4 (1 + 3 (1 + 4 (1 + 4 (1 + 10)))) = 2176 \text{ index nodes} \]

By placing PRODUCT after MONTH in the order, the PRODUCT members per QUARTER member are reduced by half. Referring to the original example, you can also deduce that QUARTER should precede MONTH. Because COUNTRY crosses with all members from all other classification variables, it should follow all other classification variables that have member relations fewer than the number of COUNTRY members. Because the YEAR variable has a 1 to 10 relationship when following any other classification variable, COUNTRY is placed after YEAR in the order.
If you compare this derived ordering to ordering based solely on ascending cardinality, you can see the derived ordering to be far superior. Ordering based on ascending cardinality would yield

```
QUARTER COUNTRY PRODUCT YEAR MONTH
```

which produces an index of size

\[
4 (1 + 4 (1 + 8 (1 + 10 (1 + 12)))) = 16788 \text{ index nodes}
\]

While this ordering would probably produce acceptable performance, it is not optimal. When it is possible to do so, you should use the derived ordering technique described here.

Unfortunately, it is unlikely that the data you work with will have classification variable relationships that are as simple and clearly defined as the ones used in these examples. Ordering classification variables becomes much more difficult when there is a variable whose members form unpredictable relationships with members from other classification variables. If you find that your data contains this type of variable, you can either create a tool that measures the relationships between classification variables through their members, or if sub-optimal ordering is acceptable, you can order the variables based on ascending cardinality. In many cases, variables of this type are large, so it might be safe to assume that such a variable should be placed last in the ordering. One way to verify this assumption is to determine if the variable in question has \(x\) or more members, where \(x\) is the number of members of the next largest classification variable. If this is the case, the variable should be last in the ordering.

**Additional Considerations**

Additional discrepancies occur when classification variable relationships are not equivalent for all of a classification variable's members. For example, if we added the members wreath, Christmas tree, Santa Claus, Rudolph, and Frosty the Snowman to our PRODUCT variable, we would have to increase the MONTH/PRODUCT relationship to nine although summer months would only have four PRODUCT member crossings, and five of the six winter months would only have four PRODUCT member crossings. Unfortunately, the arithmetic required to consider these sub-relationships is so complex that it is far more practical to estimate the effect that the sub-relationships have.

Data sparsity is another factor that can create discrepancies in classifier ordering calculations. Data sparsity is difficult to determine in data sets, and even if there were a method for determining data sparsity within individual sub-trees, the storage and management of such statistics would not be practical. Your best option is to estimate, based on experience, the effect data sparsity will have under different ordering scenarios.

Finally, keep in mind that, with a large number of classification variables, the calculation required to analyze the classification variable member relationship may be quite time consuming. The number of relationships will be \(n (n - 1)\), where \(n\) is the number of classifiers. It is important to recognize that the performance benefits will justify the time spent performing this analysis. If you create an MDDB so that it has the smallest possible footprint, I/O time is reduced each time that the MDDB is loaded.

While there are no ordering techniques that will work in all cases, you can use the techniques discussed here to make an educated approximation of the optimal order. The method of analyzing classification variable member relationships should prove effective in the majority of cases. Even if it does not yield optimal ordering, it will probably allow you to apply an ordering that will result in acceptable performance and disk space usage.
MDDB Memory Optimization

The main factor affecting MDDB performance is the selection and ordering of classification variables in the NWAY cube and subcubes (see “Analyzing Your Data” on page 11 for more information), but you can also optimize performance by specifying how much memory the MDDB should be able to use during creation and usage. This capability allows you to limit the resources that an individual MDDB request will take. You specify the memory limitations using the VMEMSIZE= and PKTSIZE= options in the MDDB procedure for creation of and reporting from MDDBs. See “Building an MDDB with the MDDB Procedure” on page 24 for details on MDDB procedure syntax.

The PKTSIZE parameter determines the size of an internal heap that is used for disk I/O in servicing MDDB requests. If users view multiple cubes at the same time (for instance, by using expanders in the SAS/EIS software Multidimensional Database viewer), the PKTSIZE parameter must be reduced because four heaps of the size determined by the PKTSIZE setting will be created for every cube opened. If the PKTSIZE parameter is set too low, the amount of disk space used by the cubes’ temporary files expands exponentially.

For Version 7 SAS/MDDB Server software, the PKTSIZE value should be reset for all MDDBs by using the methods described below. Resetting the PKTSIZE parameter necessitates resetting the VMEMSIZE parameter as well, as described in the following section.

Setting Parameters for MDDB Creation

When creating an MDDB, calculate the PKTSIZE parameter using the following formula:

\[
PKTSIZE = \frac{3 \times VMEMSIZE}{2} \times 1024
\]

where

- PKTSIZE is the packet size expressed in kilobytes (KB)
- VMEMSIZE is the VMEM parameter expressed in megabytes (MB)
- 1024 converts MB to KB.

PKTSIZE is expressed in kilobytes because the PKTSIZE option in the MDDB procedure expects a value in kilobytes. The maximum recommended PKTSIZE is 96000 KB, and the minimum recommended PKTSIZE is 6000 KB.

When PKTSIZE is calculated, VMEMSIZE must be recalculated as well. If VMEMSIZE is not recalculated, there is a chance that the packet size will consume all the system memory, causing poor performance or failure. Use the following formula to recalculate VMEMSIZE:

\[
VMEMSIZE_{\text{new}} = \frac{3 \times VMEMSIZE_{\text{old}}}{4} + \frac{PKTSIZE}{1024}
\]

where

- PKTSIZE is the packet size calculated previously, expressed in kilobytes
- VMEMSIZE_{\text{old}} is the previous VMEMSIZE parameter, expressed in megabytes
VMEMSIZE\textsubscript{new} is the new VMEMSIZE parameter, expressed in megabytes. 1024 converts KB to MB.

The VMEMSIZE parameter explicitly sets the total memory that can be used by the MDDB. The default setting, which is also the recommended minimum setting, is 16 MB. The recommended maximum setting depends on the amount of RAM in the system where the MDDB resides. Use the following table to determine the maximum VMEMSIZE setting for your system.

<table>
<thead>
<tr>
<th>System RAM</th>
<th>Recommended Maximum VMEMSIZE Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 MB-96 MB</td>
<td>System RAM minus 16 MB</td>
</tr>
<tr>
<td>97 MB-256 MB</td>
<td>System RAM minus 32 MB</td>
</tr>
<tr>
<td>257 MB-512 MB</td>
<td>System RAM minus 128 MB</td>
</tr>
<tr>
<td>513 MB-768 MB</td>
<td>System RAM minus 128 MB</td>
</tr>
</tbody>
</table>

These values provide a general guideline for setting the VMEMSIZE parameter. Determining the optimal VMEMSIZE setting for the system could require that you consider platform-specific factors and the needs of other processes that will run concurrently.

As an example, suppose that your current VMEMSIZE parameter is 16 MB. Using the previous equations, you calculate the PKTSIZE parameter to be

\[
PKTSIZE = \frac{3 \times 16MB}{2} \times 1024KB/MB = 24576KB
\]

and the new VMEMSIZE parameter to be

\[
VMEMSIZE_{new} = \frac{3 \times 16MB}{4} + \frac{24576KB}{1024KB/MB} = 36MB
\]

### Setting Parameters for MDDB Reporting

When reading and, therefore, calculating totals in an MDDB, calculate the PKTSIZE parameter using the following equation:

\[
PKTSIZE = \frac{3 \times VMEMSIZE}{8} \times 1024
\]

Calculate a new VMEMSIZE by using the same equation as in the previous section.

### Using the MAX Packet Size

If the packet size you calculate, either for creating or reading an MDDB, exceeds the recommended maximum packet size (96000 KB), use the recommended maximum.
Stored and Derived Statistics

When you create an MDDB, you have to decide which statistics should be stored with the MDDB. The statistics that can be stored with an MDDB are a minimum sufficient set that can be used to calculate other statistics when the MDDB is accessed. You can specify up to eight statistics for each analysis variable to be stored with the MDDB:

- $N$
- $SUM$
- $SUMWGT$
- $UWSUM$
- $NMISS$
- $USS$
- $MIN$
- $MAX$

Depending on which of the eight statistics are stored, up to 13 additional statistics can be calculated by SAS/MDDB Server software at run time, allowing for up to 21 available statistics. The following table indicates which statistics can be derived and which stored statistics are required:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Required Stored Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMISS</td>
<td>NMISS</td>
</tr>
<tr>
<td>SUMWGT</td>
<td>SUMWGT (available only if a WEIGHT is specified)</td>
</tr>
<tr>
<td>SUM</td>
<td>SUM (However, if a WEIGHT is specified, then SUM is the weighted sum.)</td>
</tr>
<tr>
<td>AVG</td>
<td>N, SUM (However, if a WEIGHT is specified, then SUMWGT and SUM are required instead.)</td>
</tr>
<tr>
<td>UWSUM</td>
<td>UWSUM</td>
</tr>
<tr>
<td>MIN</td>
<td>MIN</td>
</tr>
<tr>
<td>MAX</td>
<td>MAX</td>
</tr>
<tr>
<td>USS</td>
<td>USS</td>
</tr>
<tr>
<td>RANGE</td>
<td>MIN, MAX</td>
</tr>
<tr>
<td>PCTN</td>
<td>N</td>
</tr>
<tr>
<td>PCTSUM</td>
<td>SUM</td>
</tr>
<tr>
<td>CSS</td>
<td>N, SUM, USS (However, if a WEIGHT is specified, then SUMWGT, SUM, and USS are required instead.)</td>
</tr>
<tr>
<td>VAR, STD, STDERR, CV, T, PRT, LCLM, UCLM</td>
<td>N, SUM, USS (However, if a WEIGHT is specified, then SUMWGT is also required.)</td>
</tr>
</tbody>
</table>

See “VAR Statement” on page 27 for details on specifying a WEIGHT and using the SUMWGT statistic.
Building an MDDB

There are four ways that you can build an MDDB. You can use
- the MDDB procedure
- SAS/EIS software
- the SAS/MDDB Server classes
- SAS/Warehouse Administrator software.

This section provides instructions on how to build an MDDB by using each of these methods. You can choose one method over another based on the SAS software products that you use and with which you are most familiar.

Note: Regardless of the method that you choose, the information that you specify when creating an MDDB is similar to that specified when using the SUMMARY procedure. The NWAY cube correlates to the NWAY data produced by the SUMMARY procedure, and subcube data correlates to _TYPE_ records produced by the SUMMARY procedure. If you are already familiar with the SUMMARY procedure, keeping these similarities in mind could help you understand how to create MDDBs.

Once an MDDB has been created, you can copy or transport it to any platform that supports the MDDB object type. For information on copying or transporting MDDBs, refer to “Transporting MDDBs Across Operating Environments” on page 39.

Building an MDDB with the MDDB Procedure

This section provides the syntax for the MDDB procedure and explains how to use the procedure to create an MDDB. An example PROC MDDB statement is also provided.

PROC MDDB <option(s)>;
   CLASS var1 var2 ... /<order-options>;
   HIERARCHY class-var1 class-var2 ... /<NAME=name | "name" DISPLAY=YES | NO>
   VAR var1 var2 ... /<stat-options>;
PROC MDDB Statement

You can use the following options in the PROC MDDB statement:

DATA= dsname
   Use the DATA= option to specify the name of a SAS table that is to be used as the source for the MDDB. If you do not specify a table name, _LAST_ is used.

OUT= libref.outmddb
   Use the OUT= option to specify the name of the MDDB that you are creating. The OUT= option is required.

IN= libref.inmddb
   The IN= option can be used during an incremental update of an MDDB, in which case the name of the MDDB specified in the IN= option is the existing MDDB. The
DATA= option is used to specify the name of the table that contains the incremental data that will be added to the data in the input MDDB and written out to the MDDB specified in the OUT= option.

The IN= option can also be used to convert an MDDB from a previous format (prior to Version 7) into the new format. In this case, the IN= option refers to the MDDB that is in the previous version format, and the OUT= option specifies the name of the new MDDB that will be created. The CONVERT option must be specified.

LABEL= description

Use the LABEL= option to specify a description to be stored with the MDDB. The character description string can be up to 256 characters long. Enclose the description in quotes if it contains embedded blanks. This parameter is optional.

PW="password"

You can use the PW= option to specify a password that is to be associated with the MDDB. The password must be no more than eight characters and is not case-sensitive. Any passwords that are specified in the MDDB name will override the password specified as an option in the PROC MDDB statement. This parameter is optional.

You can specify read, write, and alter passwords using the same syntax as for data sets. For example, each of the following is valid:

```
out=libname.memname (pw = "password"
  read = "read_password"
  write = "write_password"
  alter = "alter_password")
```

You can also specify the PWREQ= option to control whether a password requestor window appears when a required password is either missing or incorrect. By default, the password requestor window does not appear when creating an MDDB but does appear when reading an MDDB. To use the PWREQ= option, specify

```
pwreq = ‘NO’ | ‘YES’
```

For example, you can specify the following code to ensure that a password requestor window appears when the required password is missing or incorrect:

```
libname.memname (pw = "password"
  pwreq = ‘YES’)
```

Use the following as guidelines when you specify passwords:

- You can specify any combination of pw, read, write, and alter passwords.
- Spaces and quotes (single or double) are optional. The password is not case-sensitive.
- If you specify only pw="password", that password is used for the read, write, and alter passwords.
- The read, write, and alter passwords override the pw password if the pw password is also specified.

VMEMSIZE=msize

The VMEMSIZE= option indicates how much RAM (in megabytes) can be used during processing. This parameter is optional.

PKTSIZE=psize

The PKTSIZE= option is used to specify the size (in kilobytes) of a block that is read from a file. This is especially important during remote file reads when a
smaller block size will decrease the overall network traffic. The block size should never go below 8 KB. This parameter is optional.

**CONVERT**

The CONVERT option enables you to convert MDDBs that were created with SAS/MDBD Server Release 6.12 or Release 6.09E to the new format. The following example illustrates how to use the CONVERT option to convert an MDDB created in Release 6.12 to the Version 7 format:

```plaintext
proc mddb
  in=v6lib.mddb
  out=v7lib.newmddb
  convert;

**CLASS Statement**

**CLASS** var1 var2 ... / <order-options>;

Use the CLASS statement to specify variables from the base table that are to be used as the classification variables in the MDDB.

You can specify one or more CLASS statements. However, a given variable can only appear once in all CLASS statements. The class variable can be either numeric or character. If you do not specify a sort order, ASCENDING is used.

You can use the following options in the CLASS statement:

- **order-options**

  Use the `order-options` in the CLASS statement to specify the sort order for the classification variables. You can specify any of the following order options:
  - ASCENDING
  - DESCENDING
  - ASCFORMATTED
  - DESCFORMATTED
  - DSORDER.

  You can also specify a different sort order for each CLASS variable. To do this, use a separate CLASS statement for each variable to be sorted.

**HIERARCHY Statement**

**HIERARCHY** class-var1 class-var2 ... / <NAME= name | “name” DISPLAY=YES | NO>;

You can define one or more subcubes to be stored in your MDDB by using a HIERARCHY statement. If you do not specify a hierarchy, only the NWAY cube hierarchy is stored in the MDDB. You can specify multiple CLASS variables; however, you can specify a CLASS variable only once in each HIERARCHY statement.

You can use the following options in the HIERARCHY statement:

- **NAME= name | “name”**

  Use the `NAME=` option to specify a name for your hierarchy. If the name contains a space or blank, it must be enclosed in quotes (see example, below).

  If you do not specify a name for your hierarchy, the default name HIER n is used, where n is a number (beginning with 1).

- **DISPLAY= YES | NO**

  This option will only have an effect at the time when someone chooses to register this MDDB in a SAS/EIS repository. At that time, a value of YES will be
interpreted to mean that the specific hierarchy should be registered as a drill hierarchy. The default value of NO indicates that this hierarchy should not be specifically registered.

The following examples illustrate how to specify a subcube using the HIERARCHY statement:

```
hierarchy country region division /name=geo display=YES;
hierarchy country region division /name="geographic hierarchy";
```

**VAR Statement**

```
VAR var1 var2 ... /<stat-options>;
```

The VAR statement enables you to specify variables from the base table to be used as the analysis variables in the MDDB. You can specify one or more VAR statements. However, a given variable can only appear once in all VAR statements. The variables must be numeric. If you do not specify a statistic, SUM is used.

You can use the following options in the VAR statement:

```
stat-options
```

Use the stat-options in the VAR statement to specify the statistics to be stored for each analysis variable. Separate each statistic with a space. You can specify any of the following statistics options:

- MAX
- MIN
- N
- NMISS
- SUM
- SUMWGT
- USS
- UWSUM
- WEIGHT= numeric variable to use to weight the analysis variable

If you specify WEIGHT=, its value must be the name of a numeric variable in the data set. If you also specify SUMWGT, the weighted sum will be stored in the MDDB. If you specify only WEIGHT=, the weight will be used in calculating the SUM statistic, but the weighted sum will not be stored, and the other statistics that would be calculated based on the weighted sum will not be calculated (that is, they will have missing values).

If you specify SUMWGT but do not specify WEIGHT=, then the request to store SUMWGT will be ignored.

**Example 1: Building an MDDB Using the MDDB Procedure**

This example shows you how to use the MDDB procedure to build an MDDB from the source table SASHELP.PRDSALE. The SASHELP.PRDSALE table contains the classification columns COUNTRY, REGION, DIVISION, PRODTYPE, PRODUCT, QUARTER, YEAR, and MONTH. The analysis variables are ACTUAL and PREDICT. Based on logical assumptions about how users would want to drill down through the data, you can write a PROC MDDB statement to create the correct MDDB with multiple subcubes that will meet anticipated user requests.
The resulting MDDB is called SASUSER.MDDB. The NWAY cube contains each of the classification variables. One analysis variable, PREDICT, has the statistic SUM; the other analysis variable, ACTUAL, has the statistics N, NMISS, SUM, USS, MIN, and MAX. The HIERARCHY statements create subcubes that optimize drill-down performance. No matter where a user is in any of the drill hierarchies, there is a subcube with related aggregations.

Building an MDDB with SAS/EIS Software

This section provides instructions on how to use SAS/EIS software to build an MDDB. You supply information about the MDDB in a series of SAS/EIS windows. When you build an MDDB using SAS/EIS software, the MDDB will be registered automatically in the SAS/EIS metabase facility.

To build an MDDB with SAS/EIS software, you must first register the detail data in a SAS/EIS repository. The types of SAS/EIS reports that are produced from the MDDB will help you determine how to register the detail data. Your registration should contain columns defined with the CATEGORY and ANALYSIS attributes and can contain the HIERARCH table attribute. See the SAS/EIS software online Help for more details on the data requirements for specific reports.

Once you have determined the data requirements, register the detail data in a SAS/EIS repository. Then follow the steps below to build the MDDB.

1. Invoke SAS/EIS software, and double-click Build EIS in the EIS Main Menu.

2. In the Build EIS window, specify a path and an application database, if you have not previously done so. Select Add.

3. In the Add window, select Data Access from the Object Databases list box. Then select Multidimensional database from the Objects list box and select Build. The Multidimensional Database window appears, where you enter all the information needed to create an MDDB.

4. In the Multidimensional Database window, type a name and description in the Name and Description fields. Then select the right arrow beside the MDDB field to open the MDDB window.

5. In the MDDB window, specify information on where to save and register the MDDB that you are creating. You must register the MDDB in a repository. Use the down arrow beside the Repository field to specify a repository. You can also add password protection to the MDDB in this window. Select OK to return to the Multidimensional Database window.

6. Select the right arrow beside the Table field to open the Select Table window, where you specify the registered table to be used as input for the MDDB. Select the detail data that you registered in the repository. Select OK to return to the Multidimensional Database window.
7 Select the right arrow beside the **Dimensions** field to open the Column Selection window, where you select the dimension and analysis columns. Select **OK** to return to the Multidimensional Database window.

8 Select **Create** to build the MDDB. You will receive a message indicating that the MDDB has been successfully built. You can now specify your MDDB (instead of a table) in the objects that use MDDBs as input. If you do not select **Create**, the MDDB is not created or registered until the EIS application runs.

Note: Re-executing the MDDB application, by editing the MDDB and selecting **Create** or **Test** from the Build EIS window, or by using the RUNEIS APPL=as-app-name command, will cause the MDDB to be re-created, overwriting any previous changes. △

---

**Building an MDDB with the SAS/MDDB Server Classes**

SAS/MDDB Server includes two classes that you can use to create MDDBs:

- MDDB Class
- MDDB_C Class.

Note: Two additional classes, MDDB_H and MDDB_M, are provided to help you work with MDDBs. The MDDB_H class reads an existing MDDB and returns header information. The MDDB_M class reads and returns data from the MDDB. For details on these classes, see the SAS/MDDB Server online Help. △

This section summarizes the functionality of the two classes that allow you to create MDDBs. For complete documentation of the classes, refer to the SAS/MDDB Server online Help.

**MDDB Class**

The MDDB class reads and summarizes a SAS table and stores the minimum sufficient set of summarized data in an MDDB library member. The methods specific to the MDDB class are:

- `_handleError`
  - handles errors that might occur during MDDB processing.

- `_convert`
  - converts MDDBs from pre-indexing changes to the new format (with indexing changes).

- `_summary`
  - summarizes a table and creates the MDDB.

- `_updateMddb`
  - updates an MDDB with the latest information specified in the table and in the original MDDB.

**MDDB_C Class**

The MDDB_C class enables you to create an MDDB from any data source. You can use this class to create an MDDB that is
not dependent on a particular data source (see “Creating an MDDB: Cell-by-Cell” in the SAS/MDDB Server online Help under “MDDB_C Class”)

- dependent on a particular data source—that is, an MDDB built from a data set created by PROC SUMMARY (see “Creating an MDDB: Using a SUMMARY Data Set” in the SAS/MDDB Server online Help under “MDDB_C Class”).

The methods specific to the MDDB_C class are

- **_addNode**
  - adds a node to the currently open cube.

- **_closeCube**
  - closes the current open cube.

- **_closeMmdb**
  - closes the MDDB and stores it on disk.

- **_defineClass**
  - defines a classifier that is specified in _OPEN_MDBB_.

- **_fillFromSummaryDS**
  - fills an MDDB with data from a summary table.

- **_handleError**
  - handles errors that might occur during MDDB processing.

- **_isMmdbComplete**
  - returns a value that indicates whether the minimum amount of data has been entered such that a _CLOSE_MDBB_ method can be called.

- **_isMmdbOpen**
  - returns a value that indicates whether an MDDB is open.

- **_openCube**
  - opens a cube specified in _OPEN_MDBB_ and adds nodes to it.

- **_openMmdb**
  - opens an MDDB and sets up basic header information.

**Example 2: Building an MDDB Using the MDDB Class**

The SCL code in this section shows how you could build the same MDDB as in “Building an MDDB with the MDDB Procedure” on page 24 using the MDDB class.

```sas
/*-- load the MDDB class to create the MDDB entry from data set--*/
/*-- using the CLASS instead of the PROC --*/
dcl object dataid=_new_ sasshelp.mpdb.mmdb();
init:
    /*-- create classification variables list --*/
    classlist=makelist();
    rc=setnitemc(classlist, 'ASCENDING', 'PRODUCT');
    rc=setnitemc(classlist, 'ASCENDING', 'PRODTYPE');
    rc=setnitemc(classlist, 'ASCENDING', 'YEAR');
```
rc=setnitemc(classlist, 'ASCENDING', 'QUARTER');
rc=setnitemc(classlist, 'ASCENDING', 'MONTH');
rc=setnitemc(classlist, 'ASCENDING', 'COUNTRY');
rc=setnitemc(classlist, 'ASCENDING', 'REGION');
rc=setnitemc(classlist, 'ASCENDING', 'DIVISION');

//-- create hierarchies/subcubes--*/

hlist=makelist();
h2list=makelist();
rc=insertc(h2list, 'COUNTRY', -1);
rc=insertc(h2list, 'REGION', -1);
rc=insertc(h2list, 'DIVISION', -1);
rc=setniteml(hlist, h2list, 'GEOGRAPHIC HIERARCHY');

h2list=makelist();
rc=insertc(h2list, 'PRODUCT', -1);
rc=insertc(h2list, 'YEAR', -1);
rc=setniteml(hlist, h2list, 'PRODUCT TIME HIERARCHY');

h2list=makelist();
rc=insertc(h2list, 'YEAR', -1);
rc=insertc(h2list, 'QUARTER', -1);
rc=insertc(h2list, 'MONTH', -1);
rc=insertl(hlist, h2list, -1);

//-- setup analysis and applicable stats--*/

alist=makelist();
a2list=makelist();
rc=insertc(a2list, 'SUM',-1);
a3list=makelist();
rc=insertc(a3list, 'N',-1);
rc=insertc(a3list, 'NMISS',-1);
rc=insertc(a3list, 'SUM',-1);
rc=insertc(a3list, 'USS',-1);
rc=insertc(a3list, 'MIN',-1);
rc=insertc(a3list, 'MAX',-1);
rc=setniteml(alist, a2list, 'PREDICT');
rc=insertl(alist, a3list, -1, 'ACTUAL');

put 'Creating mddb: sasuser.mddb';

dataid._summary('SASHELP.PRDSALE', 'SASUSER.MDDB', classlist, hlist, alist, "MDDB from SASHELP.PRDSALE");

rc=rc;
return;
Building an MDDB with SAS/Warehouse Administrator Software

SAS/Warehouse Administrator software provides a graphical user interface that enables you to specify the classification variables, analysis variables, and summary levels (hierarchies) for an MDDB. Additionally, you specify a location for storing the MDDB and other information specific to features of SAS/Warehouse Administrator software. For details about how to create an MDDB using SAS/Warehouse Administrator software, refer to the SAS/Warehouse Administrator User's Guide.

Updating an MDDB

It is not unusual to have gigabytes of source data that need to be summarized. Summarizing this volume of data can take hours or even days, depending on the type of CPU and other factors surrounding the data. Therefore, it is important to be able to update an MDDB with new data without having to completely rebuild the MDDB. SAS/MDDB Server software allows for incremental updates.

When you update an existing MDDB, remember that you cannot change the basic structure of the MDDB. You cannot add a classification variable, analysis variable, or statistic that was not in the original MDDB definition. In addition, you cannot add or remove subcubes from the MDDB. If you want to perform any of these tasks, you will need to rebuild the entire MDDB from the original detail data.

This section discusses how to update an existing MDDB using the MDDB procedure, SAS/EIS software, and the MDDB class. Regardless of the method you use, an MDDB will be created containing the new data. During this process, the old MDDB must still exist, so make sure there is enough disk space before the update is attempted. When the process is complete, you must copy the new MDDB into a permanent library (if it is not already in one), delete the old MDDB, and then rename the new MDDB so that it has the same name as the old MDDB. If you do not give the new MDDB the name of the old MDDB and you are using the MDDB in SAS/EIS applications, the MDDB will not be recognized by your SAS/EIS metabase registrations.

Note: No explicit action is required to update an MDDB using SAS/Warehouse Administrator. When an MDDB is defined as part of the data flow of your data warehouse, it will be automatically updated by SAS/Warehouse Administrator software when the base table for the MDDB is updated.

Updating an MDDB Using the MDDB Procedure

To update an existing MDDB using the MDDB procedure, you specify the existing MDDB using the IN= option, specify the table that contains the update data using the DATA= option, and specify the name of the updated MDDB using the OUT= option. Note that the values of the IN= and OUT= options must be different. You can also use the LABEL= option to specify a label for the updated MDDB. When you update an existing MDDB, the CLASS, VAR, and HIERARCHY statements are ignored.

"Example 1: Building an MDDB Using the MDDB Procedure" on page 27 shows how to create an MDDB called SASUSER.MDDB. To update that MDDB, you will use a PROC MDDB statement to create a new MDDB that contains the update data. Data for the new MDDB will come from the existing MDDB and from an updated base table.

In the following statement, the DATA= option specifies the name of the updated base table, the IN= option specifies the existing MDDB, and the OUT= option specifies the name of the new MDDB. Note that the LABEL= option is also used.
Building and Updating MDDBs

Updating an MDDB Using the MDDB Class

The _updateMDDB method of the MDDB class updates an MDDB with the latest information specified in the table and in the original MDDB. For complete documentation of the _updateMDDB method, refer to the SAS/MDDB Server online Help.

Updating an MDDB Using SAS/EIS Software

SAS/EIS software provides a graphical user interface that enables you to incrementally update an existing MDDB. The update data must exist in a SAS table that contains the variables in the MDDB. The format, informat, length, and type of the MDDB variables must also match those on the table. Any SAS table variables that are not in the MDDB are ignored during the update.

The Incremental MDDB Updates window in SAS/EIS software enables you to specify:

- a SAS name that is unique for the application type in the current application database
- an optional description that appears in the application windows and in application selection lists
- the name of the MDDB to be updated
- the repository in which the MDDB registration is stored
- whether the MDDB will have a read-protect password
- the SAS table used to update the MDDB.

For detailed instructions on how to update an MDDB using SAS/EIS software, refer to the SAS/EIS software online Help.

Updating an MDDB Using the MDDB Class

After the procedure runs, you would use the DATASETS procedure to delete the old MDDB and rename the new MDDB so that it has the same name as the old MDDB:

```
proc mddb data=sasuser.sales in=sasuser.mddb
   out=sasuser.mddbnew label='Updated sales MDDB: 15 Nov 1998';
run;
```

```
proc datasets library=sasuser;
   delete mddb /mt=mddb;
run;
   change mddbnew=mddb;
run;
quit;
```