10 Database Access

The structure of indexed data in AMPL has much in common with the structure of the relational tables widely used in database applications. The AMPL `table` declaration lets you take advantage of this similarity to define explicit connections between sets, parameters, variables, and expressions in AMPL, and relational database tables maintained by other software. The `read table` and `write table` commands subsequently use these connections to import data values into AMPL and to export data and solution values from AMPL.

The relational tables read and written by AMPL reside in files whose names and locations you specify as part of the `table` declaration. To work with these files, AMPL relies on *table handlers*, which are add-ons that can be loaded as needed. Handlers may be provided by the vendors of solvers or database software. AMPL has built-in handlers for two simple relational table formats useful for experimentation, and the AMPL web site provides a handler that works with the widely available ODBC interface.

This chapter begins by showing how AMPL entities can be put into correspondence with the columns of relational tables, and how the same correspondences can be described and implemented by use of AMPL's `table` declaration. Subsequent sections present basic features for reading and writing external relational tables, additional rules for handling complications that arise when reading and writing the same table, and mechanisms for writing a series of tables or columns and for reading spreadsheet data. The final section briefly describes some standard and built-in handlers.

10.1 General principles of data correspondence

Consider the following declarations from `diet.mod` in Chapter 2, defining the set `FOOD` and three parameters indexed over it:

```
set FOOD;
param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];
```

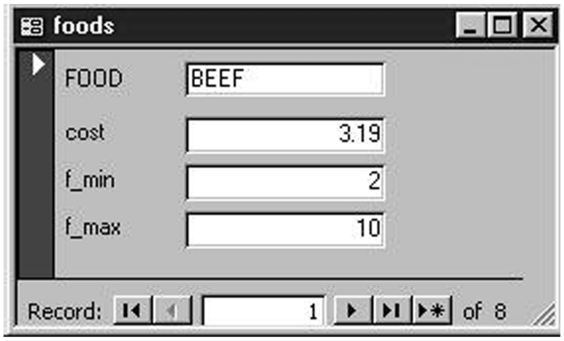
A *relational table* giving values for these components has four *columns*:

```
FOOD    cost    f_min   f_max
BEEF    3.19    2       10
CHK     2.59    2       10
FISH    2.29    2       10
HAM     2.89    2       10
MCH     1.89    2       10
MTL     1.99    2       10
SPG     1.99    2       10
TUR     2.49    2       10
```

The column headed `FOOD` lists the members of the AMPL set also named `FOOD`. This is the table's *key* column; entries in a key column must be unique, like a set's members, so that each key value identifies exactly one row. The column headed `cost` gives the values of the like-named parameter indexed over set `FOOD`; here the value of `cost["BEEF"]` is specified as 3.19, `cost["CHK"]` as 2.59, and so forth. The remaining two columns give values for the other two parameters indexed over `FOOD`.

The table has eight *rows* of data, one for each set member. Thus each row contains all of the table's data corresponding to one member — one food, in this example.

In the context of database software, the table rows are often viewed as data *records,* and the columns as *fields* within each record. Thus a data entry form has one entry field for each column. A form for the diet example (from Microsoft Access) might look like Figure 10-1. Data records, one for each table row, can be entered or viewed one at a time by using the controls at the bottom of the form.



FOOD	BEEF
cost	3.19
f_min	2
f_max	10

Figure 10-1: Access data entry form.

Parameters are not the only entities indexed over the set `FOOD` in this example. There are also the variables:

```
var Buy {j in FOOD} >= f_min[j], <= f_max[j];
```

and assorted result expressions that may be displayed:

```
ampl: model diet.mod;
ampl: data diet2a.dat;

ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032

ampl: display Buy, Buy.rc, {j in FOOD} Buy[j]/f_max[j];
:        Buy         Buy.rc    Buy[j]/f_max[j]    :=
BEEF    5.36061    8.88178e-16     0.536061
CHK     2          1.18884         0.2
FISH    2          1.14441         0.2
HAM    10         -0.302651        1
MCH    10         -0.551151        1
MTL    10         -1.3289          1
SPG     9.30605    0               0.930605
TUR     2          2.73162         0.2
;
```

All of these can be included in the relational table for values indexed over `FOOD`:

FOOD	cost	f_min	f_max	Buy	BuyRC	BuyFrac
BEEF	3.19	2	10	5.36061	8.88178e-16	0.536061
CHK	2.59	2	10	2	1.18884	0.2
FISH	2.29	2	10	2	1.14441	0.2
HAM	2.89	2	10	10	-0.302651	1
MCH	1.89	2	10	10	-0.551151	1
MTL	1.99	2	10	10	-1.3289	1
SPG	1.99	2	10	9.30605	0	0.930605
TUR	2.49	2	10	2	2.73162	0.2

Where the first four columns would typically be read into AMPL from a database, the last three are results that would be written back from AMPL to the database. We have invented the column headings `BuyRC` and `BuyFrac`, because the AMPL expressions for the quantities in those columns are typically not valid column headings in database management systems. The `table` declaration provides for input/output and naming distinctions such as these, as subsequent sections will show.

Other entities of `diet.mod` are indexed over the set `NUTR` of nutrients: parameters `n_min` and `n_max`, dual prices and other values associated with constraint `Diet`, and expressions involving these. Since nutrients are entirely distinct from foods, however, the values indexed over nutrients go into a separate relational table from the one for foods. It might look like this:

```
NUTR  n_min   n_max   NutrDual
A       700   20000    0
B1      700   20000    0
B2      700   20000    0.404585
C       700   20000    0
CAL   16000   24000    0
NA        0   50000   -0.00306905
```

As this example suggests, any model having more than one indexing set will require more than one relational table to hold its data and results. Databases that consist of multiple tables are a standard feature of relational data management, to be found in all but the simplest ‘‘flat file’’ database packages.

Entities indexed over the same higher-dimensional set have a similar correspondence to a relational table, but with one key column for each dimension. In the case of Chapter 4’s `steelT.mod`, for example, the following parameters and variables are indexed over the same two-dimensional set of product-time pairs:

```
set PROD;      # products
param T > 0;   # number of weeks

param market {PROD,1..T} >= 0;
param revenue {PROD,1..T} >= 0;
var Make {PROD,1..T} >= 0;
var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t];
```

A corresponding relational table thus has two key columns, one containing members of `PROD` and the other members of `1..T`, and then a column of values for each parameter and variable. Here’s an example, corresponding to the data in `steelT.dat`:

```
PROD   TIME  market  revenue   Make   Sell
bands   1     6000      25     5990   6000
bands   2     6000      26     6000   6000
bands   3     4000      27     1400   1400
bands   4     6500      27     2000   2000
coils   1     4000      30     1407    307
coils   2     2500      35     1400   2500
coils   3     3500      37     3500   3500
coils   4     4200      39     4200   4200
```

Each ordered pair of items in the two key columns is unique in this table, just as these pairs are unique in the set `{PROD,1..T}`. The `market` column of the table implies, for example, that `market["bands",1]` is 6000 and that `market["coils",3]` is 3500. From the first row, we can also see that `revenue["bands",1]` is 25, `Make["bands",1]` is 5990, and `Sell["bands",1]` is 6000. Again various names from the AMPL model are used as column headings, except for `TIME`, which must be invented to stand for the expression `1..T`. As in the previous example, the column headings can be any identifiers acceptable to the database software, and the `table` declaration will take care of the correspondences to AMPL names (as explained below).

AMPL entities that have sufficiently similar indexing generally fit into the same relational table. We could extend the `steelT.mod` table, for instance, by adding a column for values of

```
var Inv {PROD,0..T} >= 0;
```

The table would then have the following layout:

PROD	TIME	market	revenue	Make	Sell	Inv
bands	0	.	.	.	.	10
bands	1	6000	25	5990	6000	0
bands	2	6000	26	6000	6000	0
bands	3	4000	27	1400	1400	0
bands	4	6500	27	2000	2000	0
coils	0	.	.	.	.	0
coils	1	4000	30	1407	307	1100
coils	2	2500	35	1400	2500	0
coils	3	3500	37	3500	3500	0
coils	4	4200	39	4200	4200	0

We use ``.'' here to mark table entries that correspond to values not defined by the model and data. There is no `market["bands",0]` in the data for this model, for example, although there does exist a value for `Inv["bands",0]` in the results. Database packages vary in their handling of ``missing'' entries of this sort.

Parameters and variables may also be indexed over a set of pairs that is read as data rather than being constructed from one-dimensional sets. For instance, in the example of `transp3.mod` from Chapter 3, we have:

```
set LINKS within {ORIG,DEST};
param cost {LINKS} >= 0;   # shipment costs per unit
var Trans {LINKS} >= 0;    # actual units to be shipped
```

A corresponding relational table has two key columns corresponding to the two components of the indexing set `LINKS`, plus a column each for the parameter and variable that are indexed over `LINKS`:

ORIG	DEST	cost	Trans
GARY	DET	14	0
GARY	LAF	8	600
GARY	LAN	11	0
GARY	STL	16	800
CLEV	DET	9	1200
CLEV	FRA	27	0
CLEV	LAF	17	400
CLEV	LAN	12	600
CLEV	STL	26	0
CLEV	WIN	9	400
PITT	FRA	24	900
PITT	FRE	99	1100
PITT	STL	28	900
PITT	WIN	13	0

The structure here is the same as in the previous example. There is a row in the table only for each origin-destination pair that is actually in the set `LINKS`, however, rather than for every possible origin-destination pair.

10.2 Examples of table-handling statements

To transfer information between an AMPL model and a relational table, we begin with a `table` declaration that establishes the correspondence between them. Certain details of this declaration depend on the software being used to create and maintain the table. In the case of the four-column table of diet data defined above, some of the possibilities are as follows:

- For a Microsoft Access table in a database file `diet.mdb`:

```
table Foods IN "ODBC" "diet.mdb":
   FOOD <- [FOOD], cost, f_min, f_max;
```

- For a Microsoft Excel range from a workbook file `diet.xls`:

```
table Foods IN "ODBC" "diet.xls":
   FOOD <- [FOOD], cost, f_min, f_max;
```

- For an ASCII text table in file `Foods.tab`:

```
table Foods IN:
   FOOD <- [FOOD], cost, f_min, f_max;
```

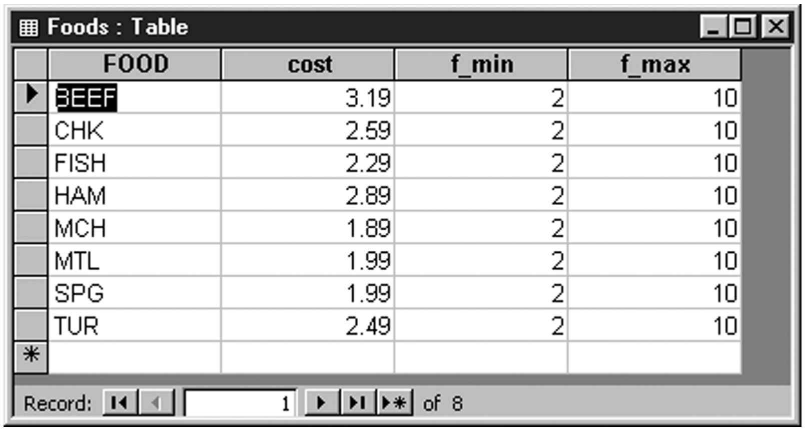
Each `table` declaration has two parts. Before the colon, the declaration provides general information. First comes the table name — `Foods` in the examples above — which will be the name by which the table is known within AMPL. The keyword `IN` states that the default for all non-key table columns will be read-only; AMPL will read values *in* from these columns and will not write out to them.

Details for locating the table in an external database file are provided by the character strings such as `"ODBC"` and `"diet.mdb"`, with the AMPL table name (`Foods`) providing a default where needed:

- For Microsoft Access, the table is to be read from database file `diet.mdb` using AMPL's ODBC handler. The table's name within the database file is taken to be `Foods` by default.

- For Microsoft Excel, the table is to be read from spreadsheet file `diet.xls` using AMPL's ODBC handler. The spreadsheet range containing the table is taken to be `Foods` by default.

- Where no details are given, the table is read by default from the ASCII text file `Foods.tab` using AMPL's built-in text table handler.


Foods : Table

FOOD	cost	f_min	f_max
BEEF	3.19	2	10
CHK	2.59	2	10
FISH	2.29	2	10
HAM	2.89	2	10
MCH	1.89	2	10
MTL	1.99	2	10
SPG	1.99	2	10
TUR	2.49	2	10

Figure 10-2: Access relational table.

In general, the format of the character strings in the `table` declaration depends upon the table handler being used. The strings required by the handlers used in our examples are described briefly in Section 10.7, and in detail in online documentation for specific table handlers.

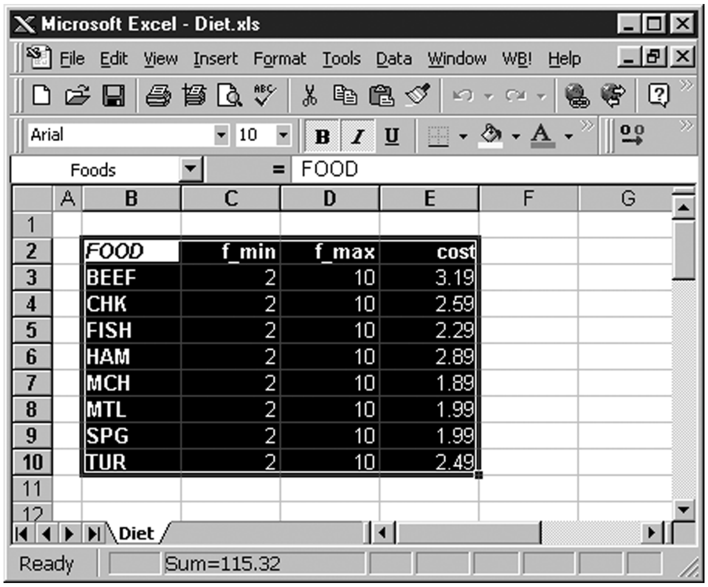
After the colon, the `table` declaration gives the details of the correspondence between AMPL entities and relational table columns. The four comma-separated entries correspond to four columns in the table, starting with the key column distinguished by surrounding brackets `[...]`. In this example, the names of the table columns (`FOOD`, `cost`, `f_min`, `f_max`) are the same as the names of the corresponding AMPL components. The expression `FOOD <- [FOOD]` indicates that the entries in the key column `FOOD` are to be copied into AMPL to define the members of the set `FOOD`.

The `table` declaration only defines a correspondence. To read values from columns of a relational table into AMPL sets and parameters, it is necessary to give an explicit

```
    read table
```

command.

Thus, if the data values were in an Access relational table like Figure 10-2, the `table` declaration for Access could be used together with the `read table` command to read the members of `FOOD` and values of `cost`, `f_min` and `f_max` into the corresponding AMPL set and parameters:


Microsoft Excel - Diet.xls
File Edit View Insert Format Tools Data Window WB! Help
Arial
10
Foods
= FOOD
Diet
Ready
Sum=115.32

FOOD	f_min	f_max	cost
BEEF	2	10	3.19
CHK	2	10	2.59
FISH	2	10	2.29
HAM	2	10	2.89
MCH	2	10	1.89
MTL	2	10	1.99
SPG	2	10	1.99
TUR	2	10	2.49

Figure 10-3: Excel worksheet range.

```
ampl: model diet.mod;
ampl: table Foods IN "ODBC" "diet.mdb":
ampl?    FOOD <- [FOOD], cost, f_min, f_max;
ampl: read table Foods;
ampl: display cost, f_min, f_max;
:       cost f_min f_max    :=
BEEF    3.19    2    10
CHK     2.59    2    10
FISH    2.29    2    10
HAM     2.89    2    10
MCH     1.89    2    10
MTL     1.99    2    10
SPG     1.99    2    10
TUR     2.49    2    10
;
```

(The `display` command confirms that the database values were read as intended.) If the data values were instead in an Excel worksheet range like Figure 10-3, the values would be read in the same way, but using the `table` declaration for Excel:

```
ampl: model diet.mod;
ampl: table Foods IN "ODBC" "diet.xls":
ampl?    FOOD <- [FOOD], cost, f_min, f_max;
ampl: read table Foods;
```

And if the values were in a file `Foods.tab` containing a text table like this:

```
ampl.tab 1 3
FOOD     cost    f_min    f_max
BEEF     3.19    2        10
CHK      2.59    2        10
FISH     2.29    2        10
HAM      2.89    2        10
MCH      1.89    2        10
MTL      1.99    2        10
SPG      1.99    2        10
TUR      2.49    2        10
```

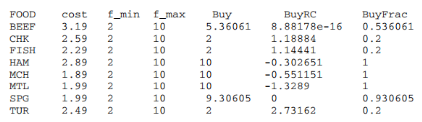
the declaration for a text table would be used:

```
ampl: model diet.mod;
ampl: table Foods IN: FOOD <- [FOOD], cost, f_min, f_max;
ampl: read table Foods;
```

Because the AMPL table name `Foods` is the same in all three of these examples, the `read table` command is the same for all three: `read table Foods`. In general, the `read table` command only specifies the AMPL name of the table to be read. All information about what is to be read, and how it is to be handled, is taken from the named table's definition in the `table` declaration.

To create the second (7-column) relational table example of the previous section, we could use a pair of table declarations:

```
table ImportFoods IN "ODBC" "diet.mdb" "Foods":
   FOOD <- [FOOD], cost, f_min, f_max;
table ExportFoods OUT "ODBC" "diet.mdb" "Foods":
   FOOD <- [FOOD], Buy, Buy.rc ~ BuyRC,
   {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac;
```



FOOD	cost	f_min	f_max	Buy	BuyRC	BuyFrac
BEEF	3.19	2	10	5.36061	8.88178e-16	0.536061
CHK	2.59	2	10	2	1.18884	0.2
FISH	2.29	2	10	2	1.14441	0.2
HAM	2.89	2	10	10	-0.302651	1
MCH	1.89	2	10	10	-0.551151	1
MTL	1.99	2	10	10	-1.3289	1
SPG	1.99	2	10	9.30605	0	0.930605
TUR	2.49	2	10	2	2.73162	0.2

or a single table declaration combining the input and output information:

```
table Foods "ODBC" "diet.mdb": [FOOD] IN, cost IN,
   f_min IN, f_max IN, Buy OUT, Buy.rc ~ BuyRC OUT,
   {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac OUT;
```

These examples show how the AMPL table name (such as `ExportFoods`) may be different from the name of the corresponding table within the external file (as indicated by the subsequent string `"Foods"`). A number of other useful options are also seen here: `IN` and `OUT` are associated with individual columns of the table, rather than with the whole table; `[FOOD] IN` is used as an abbreviation for `FOOD <- [FOOD]`; columns of the table are associated with the values of variables `Buy` and expressions `Buy.rc` and `Buy[j]/f_max[j]`; `Buy.rc ~ BuyRC` and `{j in FOOD} Buy[j]/f_max[j] ~ BuyFrac` associate an AMPL expression (to the left of the `~` operator) with a database column heading (to the right).

这里外部表的列名是ImportFoods这样，而内部希望是Foods，就用这种方式对应

根据例子猜测，当需要加入运算列的时候，使用out，否则使用in即可，此处后三列均为写入列

To write meaningful results back to the Access database, we need to read all of the diet model's data, then solve, and then give a `write table` command. Here's how it all might look using separate `table` declarations to read and write the Access table `Foods`:

```
ampl: model diet.mod;
ampl: table ImportFoods IN "ODBC" "diet.mdb" "Foods":
ampl?    FOOD <- [FOOD], cost, f_min, f_max;
ampl: table Nutrs IN "ODBC" "diet.mdb": NUTR <- [NUTR],
ampl?    n_min, n_max;
ampl: table Amts IN "ODBC" "diet.mdb": [NUTR, FOOD], amt;
ampl: read table ImportFoods;
ampl: read table Nutrs;
ampl: read table Amts;
ampl: solve;
ampl: table ExportFoods OUT "ODBC" "diet.mdb" "Foods":
ampl?    FOOD <- [FOOD],
ampl?    Buy, Buy.rc ~ BuyRC,
ampl?    {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac;
ampl: write table ExportFoods;
```

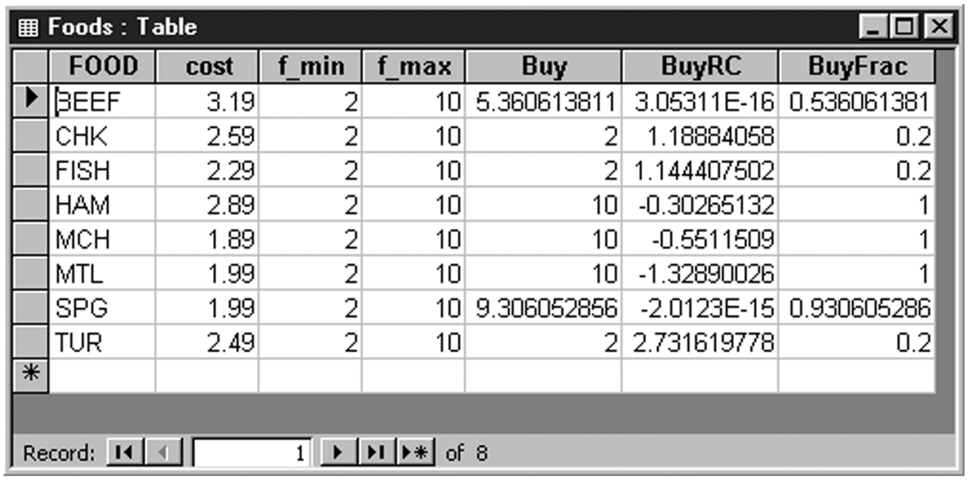
and here is an alternative using a single declaration to both read and write `Foods`:

```
ampl: model diet.mod;
ampl: table Foods "ODBC" "diet.mdb":
ampl?    [FOOD] IN, cost IN, f_min IN, f_max IN,
ampl?    Buy OUT, Buy.rc ~ BuyRC OUT,
ampl?    {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac OUT;
ampl: table Nutrs IN "ODBC" "diet.mdb":
ampl?    NUTR <- [NUTR], n_min, n_max;
ampl: table Amts IN "ODBC" "diet.mdb": [NUTR, FOOD], amt;
ampl: read table Foods;
ampl: read table Nutrs;
ampl: read table Amts;
ampl: solve;
ampl: write table Foods;
```

Either way, the Access table `Foods` would end up having three additional columns, as seen in Figure 10-4.

The same operations are handled similarly for other types of database files. In general, the actions of a `write table` command are determined by the previously declared AMPL table named in the command, and by the status of the external file associated with the AMPL table through its `table` declaration. Depending on the circumstances, the `write table` command may create a new external file or table, overwrite an existing table, overwrite certain columns within an existing table, or append columns to an existing table.

The `table` declaration is the same for multidimensional AMPL entities, except that there must be more than one key column specified between brackets `[` and `]`. For the steel production example discussed previously, the correspondence to a relational table could be set up like this:



Foods : Table

FOOD	cost	f_min	f_max	Buy	BuyRC	BuyFrac
BEEF	3.19	2	10	5.360613811	3.05311E-16	0.536061381
CHK	2.59	2	10	2	1.18884058	0.2
FISH	2.29	2	10	2	1.144407502	0.2
HAM	2.89	2	10	10	-0.30265132	1
MCH	1.89	2	10	10	-0.5511509	1
MTL	1.99	2	10	10	-1.32890026	1
SPG	1.99	2	10	9.306052856	-2.0123E-15	0.930605286
TUR	2.49	2	10	2	2.731619778	0.2

Record: 1 of 8

Figure 10-4: Access relational table with output columns.

```
table SteelProd "ODBC" "steel.mdb":
   [PROD, TIME], market IN, revenue IN,
   Make OUT, Sell OUT, Inv OUT;
```

Here the key columns `PROD` and `TIME` are not specified as `IN`. This is because the parameters to be read in, `market` and `revenue`, are indexed in the AMPL model over the set `{PROD, 1..T}`, whose membership would be specified by use of other, simpler tables. The `read table SteelProd` command merely uses the `PROD` and `TIME` entries of each database row to determine the pair of indices (subscripts) that are to be associated with the `market` and `revenue` entries in the row.

Our transportation example also involves a relational table for two-dimensional entities, and the associated `table` declaration is similar:

```
table TransLinks "ODBC" "trans.xls" "Links":
   LINKS <- [ORIG, DEST], cost IN, Trans OUT;
```

The difference here is that `LINKS`, the AMPL set of pairs over which `cost` and `Trans` are indexed, is part of the data rather than being determined from simpler sets or parameters. Thus we write `LINKS <- [ORIG, DEST]`, to request that pairs from the key columns be read into `LINKS` at the same time that the corresponding values are read into `cost`. This distinction is discussed further in the next section.

As you can see from even our simple examples so far, `table` statements tend to be cumbersome to type interactively. Instead they are usually placed in AMPL programs, or scripts, which are executed as described in Chapter 13. The `read table` and `write table` statements may be included in the scripts as well. You can define a table and then immediately read or write it, as seen in some of our examples, but a script is often more readable if the complex `table` statements are segregated from the statements that read and write the tables.

The rest of this chapter will concentrate on `table` statements. Complete sample scripts and Access or Excel files for the diet, production, and transportation examples can be obtained from the AMPL web site.

10.3 Reading data from relational tables

To use an external relational table for reading only, you should employ a `table` declaration that specifies a read/write status of `IN`. Thus it should have the general form

```
table table-name IN string-list_opt :
    key-spec, data-spec, data-spec, ... ;
```

where the optional *string-list* is specific to the database type and access method being used. (In the interest of brevity, most subsequent examples do not show a *string-list*.) The *key-spec* names the key columns, and the *data-spec* gives the data columns. Data values are subsequently read from the table into AMPL entities by the command

```
read table table-name ;
```

which determines the values to be read by referring to the `table` declaration that defined *table-name*.

Reading parameters only

To assign values from data columns to like-named AMPL parameters, it suffices to give a bracketed list of key columns and then a list of data columns. The simplest case, where there is only one key column, is exemplified by

```
table Foods IN: [FOOD], cost, f_min, f_max;
```

This indicates that the relational table has four columns, comprising a key column `FOOD` and data columns `cost`, `f_min` and `f_max`. The data columns are associated with parameters `cost`, `f_min` and `f_max` in the current AMPL model. Since there is only one key column, all of these parameters must be indexed over one-dimensional sets.

When the command

```
read table Foods
```

is executed, the relational table is read one row at a time. A row's entry in the key column is interpreted as a subscript to each of the parameters, and these subscripted parameters are assigned the row's entries from the associated data columns. For example, if the relational table is

FOOD	cost	f_min	f_max
BEEF	3.19	2	10
CHK	2.59	2	10
FISH	2.29	2	10
HAM	2.89	2	10
MCH	1.89	2	10
MTL	1.99	2	10
SPG	1.99	2	10
TUR	2.49	2	10

processing the first row assigns the values 3.19 to parameter `cost['BEEF']`, 2 to `f_min['BEEF']`, and 10 to `f_max['BEEF']`; processing the second row assigns 2.59 to `cost['CHK']`, 2 to `f_min['CHK']`, and 10 to `f_max['CHK']`; and so forth through the six remaining rows.

At the time that the `read table` command is executed, AMPL makes no assumptions about how the parameters are declared; they need not be indexed over a set named `FOOD`, and indeed the members of their indexing sets may not yet even be known. Only later, when AMPL first uses each parameter in some computation, does it check the entries read from key column `FOOD` to be sure that each is a valid subscript for that parameter.

The situation is analogous for multidimensional parameters. The name of each data column must also be the name of an AMPL parameter, and the dimension of the parameter's indexing set must equal the number of key columns. For example, when two key columns are listed within the brackets:

```
table SteelProd IN: [PROD, TIME], market, revenue;
```

the listed data columns, `market` and `revenue`, must correspond to AMPL parameters `market` and `revenue` that are indexed over two-dimensional sets.

When `read table SteelProd` is executed, each row's entries in the key columns are interpreted as a pair of subscripts to each of the parameters. Thus if the relational table has contents

PROD	TIME	market	revenue
bands	1	6000	25
bands	2	6000	26
bands	3	4000	27
bands	4	6500	27
coils	1	4000	30
coils	2	2500	35
coils	3	3500	37
coils	4	4200	39

processing the first row will assign 6000 to `market['bands',1]` and 25 to `revenue['bands',1]`; processing the second row will assign 6000 to `market['bands',2]` and 26 to `revenue['bands',2]`; and so forth through all eight rows. The pairs of subscripts given by the key column entries must be valid for `market` and `revenue` when the values of these parameters are first needed by AMPL, but the parameters need not be declared over sets named `PROD` and `TIME`. (In fact, in the model from which this example is taken, the parameters are indexed by `{PROD, 1..T}` where `T` is a previously defined parameter.)

Since a relational table has only one collection of key columns, AMPL applies the same subscripting to each of the parameters named by the data columns. These parameters are thus usually indexed over the same AMPL set. Parameters indexed over similar sets may also be accommodated in one database table, however, by leaving blank any entries in rows corresponding to invalid subscripts. The way in which a blank entry is indicated is specific to the database software being used.

Values of unindexed (scalar) parameters may be supplied by a relational table that has one row and no key columns, so that each data column contains precisely one value. The corresponding `table` declaration has an empty *key-spec*, `[]`. For example, to read a value for the parameter `T` that gives the number of periods in `steelT.mod`, the `table` declaration is

```
table SteelPeriods IN: [], T;
```

and the corresponding relational table has one column, also named `T`, whose one entry is a positive integer.

Reading a set and parameters

It is often convenient to read the members of a set from a table's key column or columns, at the same time that parameters indexed over that set are read from the data columns. To indicate that a set should be read from a table, the *key-spec* in the `table` declaration is written in the form

set-name `<-` `[`*key-col-spec*`,` *key-col-spec*`,` ...`]`

The `<-` symbol is intended as an arrow pointing in the direction that the information is moved, from the key columns to the AMPL set.

The simplest case involves reading a one-dimensional set and the parameters indexed over it, as in this example for `diet.mod`:

```
table Foods IN: FOOD <- [FoodName], cost, f_min, f_max;
```

When the command `read table Foods` is executed, all entries in the key column `FoodName` of the relational table are read into AMPL as members of the set `FOOD`, and the entries in the data columns `cost`, `f_min` and `f_max` are read into the like-named AMPL parameters as previously described. If the key column is named `FOOD` like the AMPL set, the appropriate `table` declaration becomes

```
table Foods IN: FOOD <- [FOOD], cost, f_min, f_max;
```

In this special case only, the *key-spec* can also be written in the abbreviated form `[FOOD] IN`.

An analogous syntax is employed for reading a multidimensional set along with parameters indexed over it. In the case of `transp3.mod`, for instance, the `table` declaration could be:

```
table TransLinks IN: LINKS <- [ORIG, DEST], cost;
```

When `read table TransLinks` is executed, each row of the table provides a pair of entries from key columns `ORIG` and `DEST`. All such pairs are read into AMPL as members of the two-dimensional set `LINKS`. Finally, the entries in column `cost` are read into parameter `cost` in the usual way.

As in our previous multidimensional example, the names in brackets need not correspond to sets in the AMPL model. The bracketed names serve only to identify the key columns. The name to the left of the arrow is the only one that must name a previously declared AMPL set; moreover, this set must have been declared to have the same dimension, or arity, as the number of key columns.

It makes sense to read the set `LINKS` from a relational table, because `LINKS` is specifically declared in the model in a way that leaves the corresponding data to be read separately:

```
set ORIG;
set DEST;
set LINKS within {ORIG,DEST};
param cost {LINKS} >= 0;
```

By contrast, in the similar model `transp2.mod`, `LINKS` is defined in terms of two one-dimensional sets:

```
set ORIG;
set DEST;
set LINKS = {ORIG,DEST};
param cost {LINKS} >= 0;
```

and in `transp.mod`, no named two-dimensional set is defined at all:

```
set ORIG;
set DEST;
param cost {ORIG,DEST} >= 0;
```

In these latter cases, a `table` declaration would still be needed for reading parameter `cost`, but it would not specify the reading of any associated set:

```
table TransLinks IN: [ORIG, DEST], cost;
```

Separate relational tables would instead be used to provide members for the one-dimensional sets `ORIG` and `DEST` and values for the parameters indexed over them.

When a `table` declaration specifies an AMPL set to be assigned members, its list of *data-spec*s may be empty. In that case only the key columns are read, and the only action of `read table` is to assign the key column values as members of the specified AMPL set. For instance, with the statement

```
table TransLinks IN: LINKS <- [ORIG, DEST];
```

a subsequent `read table` statement would cause just the values for the set `LINKS` to be read, from the two key columns in the corresponding database table.

Establishing correspondences

An AMPL model's set and parameter declarations do not necessarily correspond in all respects to the organization of tables in relevant databases. Where the difference is substantial, it may be necessary to use the database's query language (often SQL) to derive temporary tables that have the structure required by the model; an example is given in the discussion of the ODBC handler later in this chapter. A number of common, simple differences can be handled directly, however, through features of the `table` declaration.

Differences in naming are perhaps the most common. A `table` declaration can associate a data column with a differently named AMPL parameter by use of a *data-spec* of the form *param-name* ˜ *data-col-name*. Thus, for example, if table `Foods` were instead defined by

```
table Foods IN:
   [FOOD], cost, f_min ˜ lowerlim, f_max ˜ upperlim;
```

the AMPL parameters `f_min` and `f_max` would be read from data columns `lowerlim` and `upperlim` in the relational table. (Parameter `cost` would be read from column `cost` as before.)

A similarly generalized form, *index* ˜ *key-col-name*, can be used to associate a kind of dummy index with a key column. This index may then be used in a subscript to the optional *param-name* in one or more *data-spec*s. Such an arrangement is useful in a number of situations where the key column entries do not exactly correspond to the subscripts of the parameters that are to receive table values. Here are three common cases.

Where a numbering of some kind in the relational table is systematically different from the corresponding numbering in the AMPL model, a simple expression involving a key column *index* can translate from the one numbering scheme to the other. For example, if time periods were counted from 0 in the relational table data rather than from 1 as in the model, an adjustment could be made in the `table` declaration as follows:

```
table SteelProd IN: [p ˜ PROD, t ˜ TIME],
   market[p,t+1] ˜ market, revenue[p,t+1] ˜ revenue;
```

In the second case, where AMPL parameters have subscripts from the same sets but in different orders, key column *index*es must be employed to provide a correct index order. If `market` is indexed over `{PROD, 1..T}` but `revenue` is indexed over `{1..T, PROD}`, for example, a `table` declaration to read values for these two parameters should be written as follows:

```
table SteelProd IN: [p ˜ PROD, t ˜ TIME],
   market, revenue[t,p] ˜ revenue;
```

Finally, where the values for an AMPL parameter are divided among several database columns, key column *index*es can be employed to describe the values to be found in each column. For instance, if the `revenue` values are given in one column for `"bands"` and in another column for `"coils"`, the corresponding `table` declaration could be written like this:

```
table SteelProd IN: [t ~ TIME],
   revenue["bands",t] ~ revbands,
   revenue["coils",t] ~ revcoils;
```

It is tempting to try to shorten declarations of these kinds by dropping the ~ *data-col-name*, to produce, say,

```
table SteelProd IN:
   [p ~ PROD, t ~ TIME], market, revenue[t,p];   # ERROR
```

This will usually be rejected as an error, however, because `revenue[t,p]` is not a valid name for a relational table column in most database software. Instead it is necessary to write

```
table SteelProd IN:
   [p ~ PROD, t ~ TIME], market, revenue[t,p] ~ revenue;
```

to indicate that the AMPL parameters `revenue[t,p]` receive values from the column `revenue` of the table.

More generally, a ~ synonym will have to be used in any situation where the AMPL expression for the recipient of a column's data is not itself a valid name for a database column. The rules for valid column names tend to be the same as the rules for valid component names in AMPL models, but they can vary in details depending on the database software that is being used to create and maintain the tables.

Reading other values

In a `table` declaration used for input, an *assignable* AMPL expression may appear anywhere that a parameter name would be allowed. An expression is assignable if it can be assigned a value, such as by placing it on the left side of `:=` in a `let` command.

Variable names are assignable expressions. Thus a `table` declaration can specify columns of data to be read into variables, for purposes of evaluating a previously stored solution or providing a good initial solution for a solver.

Constraint names are also assignable expressions. Values ''read into a constraint'' are interpreted as initial dual values for some solvers, such as MINOS.

Any variable or constraint name qualified by an assignable suffix is also an assignable expression. Assignable suffixes include the predefined `.sstatus` and `.relax` as well as any user-defined suffixes. For example, if the diet problem were changed to have integer variables, the following `table` declaration could help to provide useful information for the CPLEX solver (see Section 14.3):

```
table Foods IN: FOOD IN,
   cost, f_min, f_max, Buy, Buy.priority ~ prior;
```

An execution of `read table Foods` would supply members for set `FOOD` and values for parameters `cost`, `f_min` and `f_max` in the usual way, and would also assign initial values and branching priorities to the `Buy` variables.

10.4 Writing data to relational tables

To use an external relational table for writing only, you should employ a `table` declaration that specifies its read/write status to be `OUT`. The general form of such a declaration is

```
table table-name OUT string-list_opt :
    key-spec, data-spec, data-spec, ... ;
```

where the optional *string-list* is specific to the database type and access method being used. (Again, most subsequent examples do not include a *string-list*.) AMPL expression values are subsequently written to the table by the command

```
write table table-name ;
```

which uses the `table` declaration that defined *table-name* to determine the information to be written.

A `table` declaration for writing specifies an external file and possibly a relational table within that file, either explicitly in the *string-list* or implicitly by default rules. Normally the named external file or table is created if it does not exist, or is overwritten otherwise. To specify that instead certain columns are to be replaced or are to be added to a table, the `table` declaration must incorporate one or more *data-spec*s that have read/write status `IN` or `INOUT`, as discussed in Section 10.5. A specific table handler may also have its own more detailed rules for determining when files and tables are modified or overwritten, as explained in its documentation.

The *key-spec*s and *data-spec*s of `table` declarations for writing external tables superficially resemble those for reading. The range of AMPL expressions allowed when writing is much broader, however, including essentially all set-valued and numeric-valued expressions. Moreover, whereas the table rows to be read are those of some existing table, the rows to be written must be determined from AMPL expressions in some part of a `table` declaration. Specifically, rows to be written can be inferred either from the *data-spec*s, using the same conventions as in `display` commands, or from the *key-spec*. Each of these alternatives employs a characteristic `table` syntax as described below.

Writing rows inferred from the data specifications

If the *key-spec* is simply a bracketed list of the names of key columns,

```
[key-col-name, key-col-name, ...]
```

the `table` declaration works much like the `display` command. It determines the external table rows to be written by taking the union of the indexing sets stated or implied in the *data-spec*s. The format of the *data-spec* list is the same as in `display`, except that all of the items listed must have the same dimension.

In the simplest case, the *data-spec*s are the names of model components indexed over the same set:

```
table Foods OUT: [FoodName], f_min, Buy, f_max;
```

When `write table Foods` is executed, it creates a key column `FoodName` and data columns `f_min`, `Buy`, and `f_max`. Since the AMPL components corresponding to the data columns are all indexed over the AMPL set `FOOD`, one row is created for each member of `FOOD`. In a representative row, a member of `FOOD` is written to the key column `FoodName`, and the values of `f_min`, `Buy`, and `f_max` subscripted by that member are written to the like-named data columns. For the data used in the diet example, the resulting relational table would be:

```
FoodName  f_min    Buy      f_max
BEEF        2      5.36061    10
CHK         2      2          10
FISH        2      2          10
HAM         2     10          10
MCH         2     10          10
MTL         2     10          10
SPG         2      9.30605    10
TUR         2      2          10
```

Tables corresponding to higher-dimensional sets are handled analogously, with the number of bracketed key-column names listed in the *key-spec* being equal to the dimension of the items in the *data-spec*. Thus a table containing the results from `steelT.mod` could be defined as

```
table SteelProd OUT: [PROD, TIME], Make, Sell, Inv;
```

Because `Make` and `Sell` are indexed over `{PROD,1..T}`, while `Inv` is indexed over `{PROD,0..T}`, a subsequent `write table SteelProd` command would produce a table with one row for each member of the union of these sets:

```
PROD   TIME   Make   Sell   Inv
bands    0     .      .      10
bands    1    5990   6000     0
bands    2    6000   6000     0
bands    3    1400   1400     0
bands    4    2000   2000     0
coils    0     .      .       0
coils    1    1407    307  1100
coils    2    1400   2500     0
coils    3    3500   3500     0
coils    4    4200   4200     0
```

Two rows are empty in the columns for `Make` and `Sell`, because `("bands",0)` and `("coils",0)` are not members of the index sets of `Make` and `Sell`. We use a ‘‘`.`’’ here to indicate the empty table entries, but the actual appearance and handling of empty entries will vary depending on the database software being used.

If this form is applied to writing suffixed variable or constraint names, such as the dual and slack values related to the constraint `Diet`:

```
table Nutrs OUT: [Nutrient],
   Diet.lslack, Diet.ldual, Diet.uslack, Diet.udual;  # ERROR
```

a subsequent `write table Nutrs` command is likely to be rejected, because names with a ‘‘dot’’ in the middle are not allowed as column names by most database software:

```
ampl: write table Nutrs;
Error executing "write table" command:
   Error writing table Nutrs with table handler ampl.odbc:
   Column 2’s name "Diet.lslack" contains non-alphanumeric
      character ’.’.
```

This situation requires that each AMPL expression be followed by the operator ˜ and a corresponding valid column name for use in the relational table:

```
table Nutrs OUT: [Nutrient],
   Diet.lslack ˜ lb_slack, Diet.ldual ˜ lb_dual,
   Diet.uslack ˜ ub_slack, Diet.udual ˜ ub_dual;
```

This says that the values represented by `Diet.lslack` should be placed in a column named `lb_slack`, the values represented by `Diet.ldual` should be placed in a column named `lb_dual`, and so forth. With the table defined in this way, a `write table Nutrs` command produces the intended relational table:

```
Nutrient    lb_slack    lb_dual    ub_slack    ub_dual
   A         1256.29    0          18043.7      0
   B1         336.257   0          18963.7      0
   B2           0       0.404585   19300        0
   C          982.515   0          18317.5      0
   CAL       3794.62    0           4205.38     0
   NA       50000       0              0       -0.00306905
```

The ˜ can also be used with unsuffixed names, if it is desired to assign the dabatase column a name different from the corresponding AMPL entity.

More general expressions for the values in data columns require the use of dummy indices, in the same way that they are used in the data-list of a `display` command. Since indexed AMPL expressions are rarely valid column names for a database, they should generally be followed by ˜ *data-col-name* to provide a valid name for the corresponding relational table column that is to be written. To write a column `servings` containing the number of servings of each food to be bought and a column `percent` giving the amount bought as a percentage of the maximum allowed, for example, the `table` declaration could be given as either

```
table Purchases OUT: [FoodName],
   Buy ˜ servings, {j in FOOD} 100*Buy[j]/f_max[j] ˜ percent;
```

or

```
table Purchases OUT: [FoodName],
   {j in FOOD} (Buy[j] ˜ servings,
                100*Buy[j]/f_max[j] ˜ percent);
```

Either way, since both *data-spec*s give expressions indexed over the AMPL set `FOOD`, the resulting table has one row for each member of that set:

```
FoodName   servings   percent
BEEF        5.36061    53.6061
CHK         2          20
FISH        2          20
HAM        10         100
MCH        10         100
MTL        10         100
SPG         9.30605    93.0605
TUR         2          20
```

The expression in a *data-spec* may also use operators like sum that define their own dummy indices. Thus a table of total production and sales by period for steelT.mod could be specified by

```
table SteelTotal OUT:  [TIME],
   {t in 1..T} (sum {p in PROD} Make[p,t] ~ Made,
                sum {p in PROD} Sell[p,t] ~ Sold);
```

As a two-dimensional example, a table of the amounts sold and the fractions of demand met could be specified by

```
table SteelSales OUT: [PROD, TIME], Sell,
   {p in PROD, t in 1..T} Sell[p,t]/market[p,t] ~ FracDemand;
```

The resulting external table would have key columns PROD and TIME, and data columns Sell and FracDemand.

Writing rows inferred from a key specification

An alternative form of table declaration specifies that one table row is to be written for each member of an explicitly specified AMPL set. For the declaration to work in this way, the *key-spec* must be written as

set-spec -> [*key-col-spec*, *key-col-spec*, ...]

In contrast to the arrow <- that points from a key-column list to an AMPL set, indicating values to be read into the set, this form uses an arrow -> that points from an AMPL set to a key column list, indicating information to be written from the set into the key columns.

An explicit expression for the row index set is given by the *set-spec*, which can be the name of an AMPL set, or any AMPL set-expression enclosed in braces { }. The *key-col-spec*s give the names of the corresponding key columns in the database. Dummy indices, if needed, can appear either with the *set-spec* or the *key-col-spec*s, as we will show.

The simplest case of this form involves writing database columns for model components indexed over the same one-dimensional set, as in this example for diet.mod:

```
table Foods OUT: FOOD -> [FoodName], f_min, Buy, f_max;
```

When write table Foods is executed, a table row is created for each member of the AMPL set FOOD. In that row, the set member is written to the key column FoodName, and the values of f_min, Buy, and f_max subscripted by the set member are written to the like-named data columns. (For the data used in our diet example, the resulting table would be the same as for the `FoodName` table given previously in this section.) If the key column has the same name, `FOOD`, as the AMPL set, the appropriate `table` declaration becomes

```
table Foods OUT: FOOD -> [FOOD], f_min, Buy, f_max;
```

In this special case only, the *key-spec* can also be written in the abbreviated form `[FOOD] OUT`.

The use of `~` with AMPL names and suffixed names is governed by the considerations previously described, so that the example of diet slack and dual values would be written

```
table Nutrs OUT: NUTR -> [Nutrient],
   Diet.lslack ~ lb_slack, Diet.ldual ~ lb_dual,
   Diet.uslack ~ ub_slack, Diet.udual ~ ub_dual;
```

and `write table Nutrs` would give the same table as previously shown.

More general expressions for the values in data columns require the use of dummy indices. Since the rows to be written are determined from the *key-spec*, however, the dummies are also defined there (rather than in the *data-spec*s as in the alternative form above). To specify a column containing the amount of a food bought as a percentage of the maximum allowed, for example, it is necessary to write `100*Buy[j]/f_max[j]`, which in turn requires that dummy index `j` be defined. The definition may appear either in a *set-spec* of the form `{` *index-list* `in` *set-expr* `}`:

```
table Purchases OUT: {j in FOOD} -> [FoodName],
   Buy[j] ~ servings, 100*Buy[j]/f_max[j] ~ percent;
```

or in a *key-col-spec* of the form *index* ~ *key-col-name*:

```
table Purchases OUT: FOOD -> [j ~ FoodName],
   Buy[j] ~ servings, 100*Buy[j]/f_max[j] ~ percent;
```

These two forms are equivalent. Either way, as each row is written, the index `j` takes the key column value, which is used in interpreting the expressions that give the values for the data columns. For our example, the resulting table, having key column `FoodName` and data columns `servings` and `percent`, is the same as previously shown. Similarly, the previous example of the table `SteelTotal` could be written as either

```
table SteelTotal OUT: {t in 1..T} -> [TIME],
   sum {p in PROD} Make[p,t] ~ Made,
   sum {p in PROD} Sell[p,t] ~ Sold;
```

or

```
table SteelTotal OUT: {1..T} -> [t ~ TIME],
   sum {p in PROD} Make[p,t] ~ Made,
   sum {p in PROD} Sell[p,t] ~ Sold;
```

The result will have a key column `TIME` containing the integers 1 through `T`, and data columns `Made` and `Sold` containing the values of the two summations. (Notice that since `1..T` is a set-expression, rather than the name of a set, it must be included in braces to be used as a *set-spec*.)

Tables corresponding to higher-dimensional sets are handled analogously, with the number of *key-col-spec*s listed in brackets being equal to the dimension of the *set-spec*. Thus a table containing the results from `steelT.mod` could be defined as

```
table SteelProd OUT:
   {PROD, 1..T} -> [PROD, TIME], Make, Sell, Inv;
```

and a subsequent `write table SteelProd` would produce a table of the form

```
PROD   TIME   Make   Sell   Inv
bands    1    5990   6000     0
bands    2    6000   6000     0
bands    3    1400   1400     0
bands    4    2000   2000     0
coils    1    1407    307  1100
coils    2    1400   2500     0
coils    3    3500   3500     0
coils    4    4200   4200     0
```

This result is not quite the same as the table produced by the previous `SteelProd` example, because the rows to be written here correspond explicitly to the members of the set `{PROD, 1..T}`, rather than being inferred from the indexing sets of `Make`, `Sell`, and `Inv`. In particular, the values of `Inv["bands",0]` and `Inv["coils",0]` do not appear in this table.

The options for dummy indices in higher dimensions are the same as in one dimension. Thus our example `SteelSales` could be written either using dummy indices defined in the *set-spec*:

```
table SteelSales OUT:
   {p in PROD, t in 1..T} -> [PROD, TIME],
   Sell[p,t] ~ sold, Sell[p,t]/market[p,t] ~ met;
```

or with dummy indices added to the *key-col-spec*s:

```
table SteelSales OUT:
   {PROD,1..T} -> [p ~ PROD, t ~ TIME],
   Sell[p,t] ~ sold, Sell[p,t]/market[p,t] ~ met;
```

If dummy indices happen to appear in both the *set-spec* and the *key-col-spec*s, ones in the *key-col-spec*s take precedence.

10.5 Reading and writing the same table

To read data from a relational table and then write results to the same table, you can use a pair of `table` declarations that reference the same file and table names. You may also be able to combine these declarations into one that specifies some columns to be read and others to be written. This section gives examples and instructions for both of these possibilities.

Reading and writing using two `table` declarations

A single external table can be read by one `table` declaration and later written by another. The two `table` declarations follow the rules for reading and writing given above.

In this situation, however, one usually wants `write table` to add or rewrite selected columns, rather than overwriting the entire table. This preference can be communicated to the AMPL table handler by including input as well as output columns in the `table` declaration that is to be used for writing. Columns intended for input to AMPL can be distinguished from those intended for output to the external table by specifying a read/write status column by column (rather than for the table as a whole).

As an example, an external table for `diet.mod` might consist of columns `cost`, `f_min` and `f_max` containing input for the model, and a column `Buy` containing the results. If this is maintained as a Microsoft Access table named `Diet` within a file `diet.mdb`, the `table` declaration for reading data into AMPL could be

```
table FoodInput IN "ODBC" "diet1.mdb" "Diet":
   FOOD <- [FoodName], cost, f_min, f_max;
```

The corresponding declaration for writing the results would have a different AMPL *table-name* but would refer to the same Access table and file:

```
table FoodOutput "ODBC" "diet1.mdb" "Diet":
   [FoodName], cost IN, f_min IN, Buy OUT, f_max IN;
```

When `read table FoodInput` is executed, only the three columns listed in the `table FoodInput` declaration are read; if there is an existing column named `Buy`, it is ignored. Later, when the problem has been solved and `write table FoodOutput` is executed, only the one column that has read/write status `OUT` in the `table FoodOutput` declaration is written to the Access table, while the table's other columns are left unmodified.

Although details may vary with the database software used, the general convention is that overwriting of an entire existing table or file is intended only when *all* data columns in the `table` declaration have read/write status `OUT`. Selective rewriting or addition of columns is intended otherwise. Thus if our AMPL table for output had been declared

```
table FoodOutput "ODBC" "diet1.mdb" "Diet":
                 [FoodName], Buy OUT;
```

then all of the data columns in Access table `Diet` would have been deleted by `write table FoodOutput`, but the alternative

```
table FoodOutput "ODBC" "diet1.mdb" "Diet":
                 [FoodName], Buy;
```

would have only overwritten the column `Buy`, as in the example we originally gave, since there is a data column (namely `Buy` itself) that does not have read/write status `OUT`. (The default, when no status is given, is `INOUT`.)

Reading and writing using the same `table` declaration

In many cases, all of the information for both reading and writing an external table can be specified in the same `table` declaration. The *key-spec* may use the arrow `<-` to read contents of the key columns into an AMPL set, `->` to write members of an AMPL set into the key columns, or `<->` to do both. A *data-spec* may specify read/write status `IN` for a column that will only be read into AMPL, `OUT` for a column that will only be written out from AMPL, or `INOUT` for a column that will be both read and written.

A `read table` *table-name* command reads only the key or data columns that are specified in the declaration of *table-name* as being `IN` or `INOUT`. A `write table` *table-name* command analogously writes to only the columns that are specified as `OUT` or `INOUT`.

As an example, the declarations defining `FoodInput` and `FoodOutput` above could be replaced by

```
table Foods "ODBC" "diet1.mdb" "Diet":
   FOOD <- [FoodName], cost IN, f_min IN, Buy OUT, f_max IN;
```

A `read table Foods` would then read only from key column `FoodName` and data columns `cost`, `f_min` and `f_max`. A later `write table Foods` would write only to the column `Buy`.

10.6 Indexed collections of tables and columns

In some circumstances, it is convenient to declare an indexed collection of tables, or to define an indexed collection of data columns within a table. This section explains how indexing of these kinds can be specified within the `table` declaration.

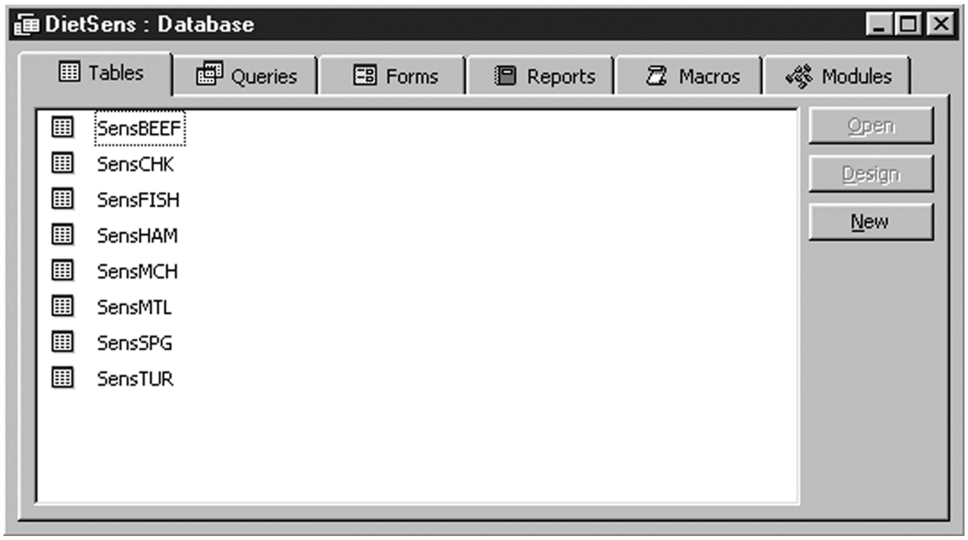
To illustrate indexed collections of tables, we present a script (Chapter 13) that automatically solves a series of scenarios stored separately. To illustrate indexed collections of columns, we show how a two-dimensional spreadsheet table can be read.

All of our examples of these features make use of AMPL's character-string expressions to generate names for series of files, tables, or columns. For more on string expressions, see Sections 13.7 and A.4.2.

Indexed collections of tables

AMPL table declarations can be indexed in much the same way as sets, parameters, and other model components. An optional `{`*indexing-expr*`}` follows the *table-name*:

`table` *table-name* `{`*indexing-expr*`}`$_{opt}$ *string-list*$_{opt}$ `:` ...



SensBEEF
SensCHK
SensFISH
SensHAM
SensMCH
SensMTL
SensSPG
SensTUR

Figure 10-5: Access database with tables of sensitivity analysis.

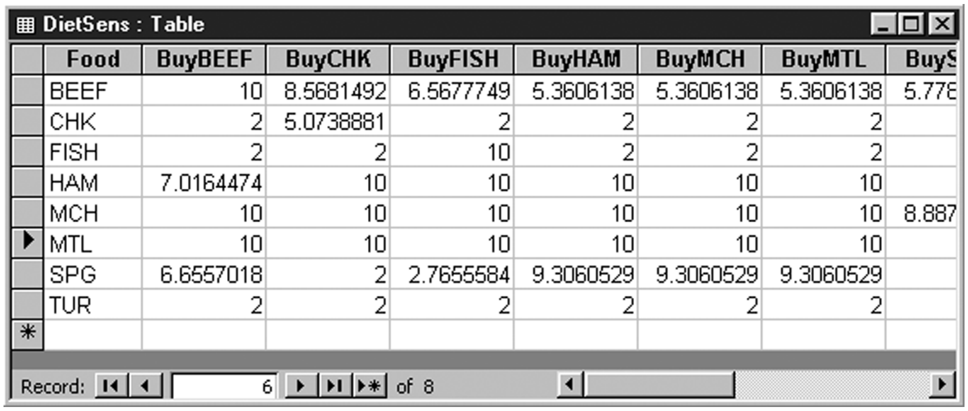
One table is defined for each member of the set specified by the *indexing-expr*. Individual tables in this collection are denoted in the usual way, by appending a bracketed subscript or subscripts to the *table-name*.

As an example, the following declaration defines a collection of AMPL tables indexed over the set of foods in `diet.mod`, each table corresponding to a different database table in the Access file `DietSens.mdb`:

```
table DietSens {j in FOOD}
   OUT "ODBC" "DietSens.mdb" ("Sens" & j):
      [Food], f_min, Buy, f_max;
```

Following the rules for the standard ODBC table handler, the Access table names are given by the third item in the *string-list*, the string expression `("Sens" & j)`. Thus the AMPL table `DietSens["BEEF"]` is associated with the Access table `SensBEEF`, the AMPL table `DietSens["CHK"]` is associated with the Access table `SensCHK`, and so forth. The following AMPL script uses these tables to record the optimal diet when there is a two-for-the-price-of-one sale on each of the foods:

```
for {j in FOOD} {
   let cost[j] := cost[j] / 2;
   solve;
   write table DietSens[j];
   let cost[j] := cost[j] * 2;
}
```


DietSens : Table

Food	BuyBEEF	BuyCHK	BuyFISH	BuyHAM	BuyMCH	BuyMTL	BuyS
BEEF	10	8.5681492	6.5677749	5.3606138	5.3606138	5.3606138	5.778
CHK	2	5.0738881	2	2	2	2	
FISH	2	2	10	2	2	2	
HAM	7.0164474	10	10	10	10	10	
MCH	10	10	10	10	10	10	8.887
MTL	10	10	10	10	10	10	
SPG	6.6557018	2	2.7655584	9.3060529	9.3060529	9.3060529	
TUR	2	2	2	2	2	2	

Record: 6 of 8

Figure 10-6: Alternate Access table for sensitivity analysis.

For the data in diet2a.dat, the set FOOD has eight members, so eight tables are written in the Access database, as seen in Figure 10-5. If instead the table declaration were to give a string expression for the second string in the *string-list*, which specifies the Access filename:

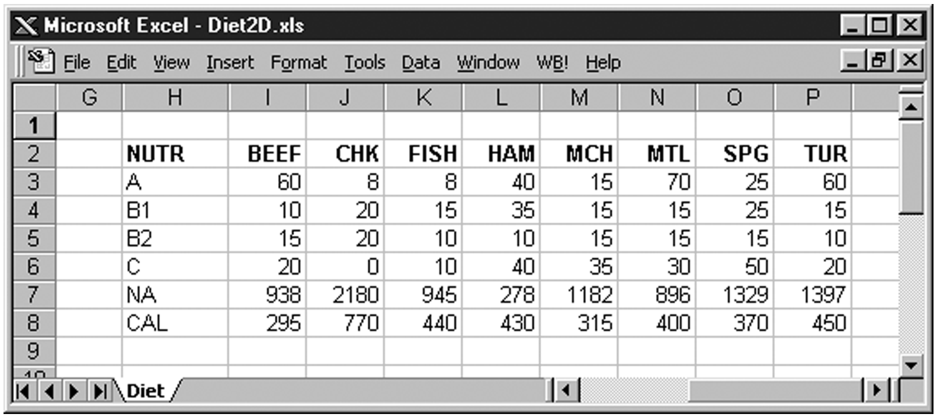
```
table DietSens {j in FOOD}
   OUT "ODBC" ("DietSens" & j & ".mdb"):
      [Food], f_min, Buy, f_max;
```

then AMPL would write eight different Access database files, named DietSensBEEF.mdb, DietSensCHK.mdb, and so forth, each containing a single table named (by default) DietSens. (These files must have been created before the write table commands are executed.)

A string expression can be used in a similar way to make every member of an indexed collection of AMPL tables correspond to the same Access table, but with a different *data-col-name* for the optimal amounts:

```
table DietSens {j in FOOD} "ODBC" "DietSens.mdb":
   [Food], Buy ~ ("Buy" & j);
```

Then running the script shown above will result in the Access table of Figure 10-6. The AMPL tables in this case were deliberately left with the default read/write status, INOUT. Had the read/write status been specified as OUT, then each write table would have overwritten the columns created by the previous one.


Microsoft Excel - Diet2D.xls
File Edit View Insert Format Tools Data Window WB! Help
Diet

	G	H	I	J	K	L	M	N	O	P
1										
2		NUTR	BEEF	CHK	FISH	HAM	MCH	MTL	SPG	TUR
3		A	60	8	8	40	15	70	25	60
4		B1	10	20	15	35	15	15	25	15
5		B2	15	20	10	10	15	15	15	10
6		C	20	0	10	40	35	30	50	20
7		NA	938	2180	945	278	1182	896	1329	1397
8		CAL	295	770	440	430	315	400	370	450
9										

Figure 10-7: Two-dimensional AMPL table in Excel.

Indexed collections of data columns

Because there is a natural correspondence between data columns of a relational table and indexed collections of entities in an AMPL model, each *data-spec* in a `table` declaration normally refers to a different AMPL parameter, variable, or expression. Occasionally the values for one AMPL entity are split among multiple data columns, however. Such a case can be handled by defining a collection of data columns, one for each member of a specified indexing set.

The most common use of this feature is to read or write two-dimensional tables. For example, the data for the parameter

```
param amt {NUTR,FOOD} >= 0;
```

from `diet.mod` might be represented in an Excel spreadsheet as a table with nutrients labeling the rows and foods the columns (Figure 10-7). To read this table using AMPL's external database features, we must regard it as having one key column, under the heading `NUTR`, and data columns headed by the names of individual foods. Thus we require a `table` declaration whose *key-spec* is one-dimensional and whose *data-spec*s are indexed over the AMPL set `FOOD`:

```
table dietAmts IN "ODBC" "Diet2D.xls":
   [i ~ NUTR], {j in FOOD} <amt[i,j] ~ (j)>;
```

The *key-spec* `[i ~ NUTR]` associates the first table column with the set `NUTR`. The *data-spec* `{j in FOOD}` <...> causes AMPL to generate an individual *data-spec* for each member of set `FOOD`. Specifically, for each `j in FOOD`, AMPL generates the *data-spec* `amt[i,j] ~ (j)`, where `(j)` is the AMPL string expression for the heading of the external table column for food `j`, and `amt[i,j]` denotes the parameter to which the values in that column are to be written. (According to the convention used here and in other AMPL declarations and commands, the parentheses around `(j)` cause it to be interpreted as an expression for a string; without the parentheses it would denote a column name consisting of the single character `j`.)

A similar approach works for writing two-dimensional tables to spreadsheets. As an example, after `steelT.mod` is solved, the results could be written to a spreadsheet using the following `table` declaration:

```
table Results1 OUT "ODBC" "steel1out.xls":
   {p in PROD} -> [Product],
      Inv[p,0] ~ Inv0,
      {t in 1..T} < Make[p,t] ~ ('Make' & t),
                    Sell[p,t] ~ ('Sell' & t),
                    Inv[p,t]  ~ ('Inv' & t) >;
```

or, equivalently, using `display`-style indexing:

```
table Results2 OUT "ODBC" "steel2out.xls":
   [Product],
      {p in PROD} Inv[p,0] ~ Inv0,
      {t in 1..T} < {p in PROD} (Make[p,t] ~ ('Make' & t),
                                 Sell[p,t] ~ ('Sell' & t),
                                 Inv[p,t] ~  ('Inv' & t) ) >;
```

The key column labels the rows with product names. The data columns include one for the initial inventories, and then three representing production, sales, and inventories, respectively, for each period, as in Figure 10-8. Conceptually, there is a symmetry between the row and column indexing of a two-dimensional table. But because the tables in these examples are being treated as relational tables, the `table` declaration must treat the row indexing and the column indexing in different ways. As a result, the expressions describing row indexing are substantially different from those describing column indexing.

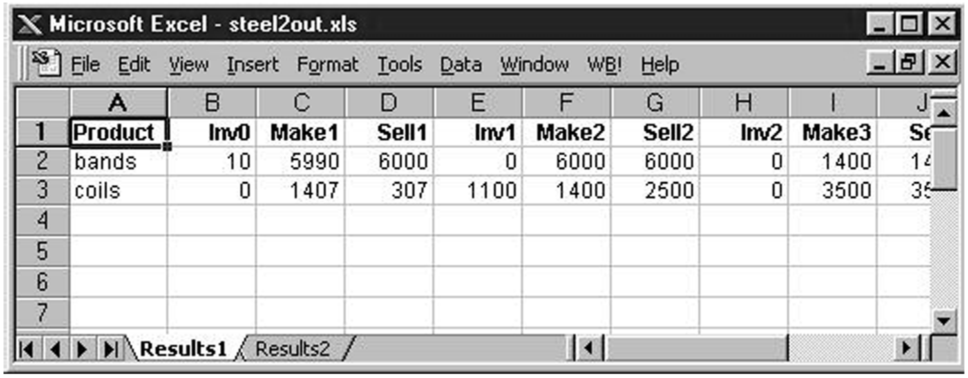
As these examples suggest, the general form for specifying an indexed collection of table columns is

{*indexing-expr*} < *data-spec*, *data-spec*, *data-spec*, ... >

where each *data-spec* has any of the forms previously given. For each member of the set specified by the *indexing-expr*, AMPL generates one copy of each *data-spec* within the angle brackets <...>. The *indexing-expr* also defines one or more dummy indices that run over the index set; these indices are used in expressions within the *data-spec*s, and also appear in string expressions that give the names of columns in the external database.

10.7 Standard and built-in table handlers

To work with external database files, AMPL relies on table handlers. These are add-ons, usually in the form of shared or dynamic link libraries, that can be loaded as needed.


Microsoft Excel - steel2out.xls
File Edit View Insert Format Tools Data Window WB! Help
Results1
Results2

	A	B	C	D	E	F	G	H	I	J
1	Product	Inv0	Make1	Sell1	Inv1	Make2	Sell2	Inv2	Make3	Se
2	bands	10	5990	6000	0	6000	6000	0	1400	14
3	coils	0	1407	307	1100	1400	2500	0	3500	35
4										
5										
6										
7										

Figure 10-8: Another two-dimensional Excel table.

AMPL is distributed with a ‘‘standard’’ table handler that runs under Microsoft Windows and communicates via the Open Database Connectivity (ODBC) application programming interface; it recognizes relational tables in the formats used by Access, Excel, and any other application for which an ODBC driver exists on your computer. Additional handlers may be supplied by vendors of AMPL or of database software.

In addition to any supplied handlers, minimal ASCII and binary relational table file handlers are built into AMPL for testing. Vendors may include other built-in handlers. If you are not sure which handlers are currently seen by your copy of AMPL, the features described in A.13 can get you a list of active handlers and brief instructions for using them.

As the introductory examples of this chapter have shown, AMPL communicates with handlers through the *string-list* in the `table` declaration. The form and interpretation of the *string-list* are specific to each handler. The remainder of this section describes the *string-list*s that are recognized by AMPL’s standard ODBC handler. Following a general introduction, specific instructions are provided for the two applications, Access and Excel, that are used in many of the examples in preceding sections. A final subsection describes the string-lists recognized by the built-in binary and ASCII table handlers.

Using the standard ODBC table handler

In the context of a declaration that begins `table` *table-name* ..., the general form of the *string-list* for the standard ODBC table handler is

`"ODBC"` `"`*connection-spec*`"` `"`*external-table-spec*`"`$_{opt}$ `"verbose"`$_{opt}$

The first string tells AMPL that data transfers using this table declaration should employ the standard ODBC handler. Subsequent strings then provide directions to that handler.

The second string identifies the external database file that is to be read or written upon execution of `read table` *table-name* or `write table` *table-name* commands. There are several possibilities, depending on the form of the *connection-spec* and the configuration of ODBC on your computer.

If the *connection-spec* is a filename of the form *name*`.`*ext*, where *ext* is a 3-letter extension associated with an installed ODBC driver, then the named file is the database file. This form can be seen in a number of our examples, where filenames of the forms *name*`.mdb` and *name*`.xls` refer to Access and Excel files, respectively.

Other forms of *connection-spec* are more specific to ODBC, and are explained in online documentation. Information about your computer's configuration of ODBC drivers, data source names, file data sources, and related entities can be examined and changed through the Windows ODBC control panel.

The third string normally gives the name of the relational table, within the specified file, that is to be read or written upon execution of `read table` or `write table` commands. If the third string is omitted, the name of the relational table is taken to be the same as the *table-name* of the containing `table` declaration. For writing, if the indicated table does not exist, it is created; if the table exists but all of the `table` declaration's *data-spec*s have read/write status `OUT`, then it is overwritten. Otherwise, writing causes the existing table to be modified; each column written either overwrites an existing column of the same name, or becomes a new column appended to the table.

Alternatively, if the third string has the special form

`"SQL=`*sql-query*`"`

the table declaration applies to the relational table that is (temporarily) created by a statement in the Structured Query Language, commonly abbreviated SQL. Specifically, a relational table is first constructed by executing the SQL statement given by *sql-query*, with respect to the database file given by the second string in the `table` declaration's *string-list*. Then the usual interpretations of the `table` declaration are applied to the constructed table. All columns specified in the declaration should have read/write status `IN`, since it would make no sense to write to a temporary table. Normally the *sql-query* is a `SELECT` statement, which is SQL's primary device for operating on tables to create new ones.

As an example, if you wanted to read as data for `diet.mod` only those foods having a cost of $2.49 or less, you could use an SQL query to extract the relevant records from the `Foods` table of your database:

```
table cheapFoods IN "ODBC" "diet.mdb"
   "SQL=SELECT * FROM Foods WHERE cost <= 2.49":
   FOOD <- [FOOD], cost, f_min, f_max;
```

Then to read the relevant data for parameter `amt`, which is indexed over nutrients and foods, you would want to read only those records that pertained to a food having a cost of

$2.49 or less. Here is one way that an SQL query could be used to extract the desired records:

```
option selectAmts "SELECT NUTR, Amts.FOOD, amt "
   "FROM Amts, Foods "
      "WHERE Amts.FOOD = Foods.FOOD and cost <= 2.49";

table cheapAmts IN "ODBC" "diet.mdb" ("SQL=" & $selectAmts):
   [NUTR, FOOD], amt;
```

Here we have used an AMPL option to store the string containing the SQL query. Then the table declaration's third string can be given by the relatively short string expression `"SQL=" & $selectAmts`.

The string `verbose` after the first three strings requests diagnostic messages — such as the DSN= string that ODBC reports using — whenever the containing table declaration is used by a `read table` or `write table` command.

Using the standard ODBC table handler with Access and Excel

To set up a relational table correspondence for reading or writing Microsoft Access files, specify the *ext* in the second string of the *string-list* as `mdb`:

`"ODBC" "`*filename*`.mdb" "`*external-table-spec*`"` $_{opt}$

The file named by the second string must exist, though for writing it may be a database that does not yet contain any tables.

To set up a relational table correspondence for reading or writing Microsoft Excel spreadsheets, specify the *ext* in the second string of the *string-list* as `xls`:

`"ODBC" "`*filename*`.xls" "`*external-table-spec*`"` $_{opt}$

In this case, the second string identifies the external Excel workbook file that is to be read or written. For writing, the file specified by the second string is created if it does not exist already.

The *external-table-spec* specified by the third string identifies a spreadsheet range, within the specified file, that is to be read or written; if this string is absent, it is taken to be the *table-name* given at the start of the `table` declaration. For reading, the specified range must exist in the Excel file. For writing, if the range does not exist, it is created, at the upper left of a new worksheet having the same name. If the range exists but all of the table declaration's *data-spec*s have read/write status `OUT`, it is overwritten. Otherwise, writing causes the existing range to be modified. Each column written either overwrites an existing column of the same name, or becomes a new column appended to the table; each row written either overwrites entries in an existing row having the same key column entries, or becomes a new row appended to the table.

When writing causes an existing range to be extended, rows or columns are added at the bottom or right of the range, respectively. The cells of added rows or columns must be empty; otherwise, the attempt to write the table fails and the `write table` command elicits an error message. After a table is successfully written, the corresponding range is created or adjusted to contain exactly the cells of that table.

Built-in table handlers for text and binary files

For debugging and demonstration purposes, AMPL has built-in handlers for two very simple relational table formats. These formats store one table per file and convey equivalent information. One produces ASCII files that can be examined in any text editor, while the other creates binary files that are much faster to read and write.

For these handlers, the `table` declaration's *string-list* contains at most one string, identifying the external file that contains the relational table. If the string has the form

```
"filename.tab"
```

the file is taken to be an ASCII text file; if it has the form

```
"filename.bit"
```

it is taken to be a binary file. If no *string-list* is given, a text file *table-name*`.tab` is assumed.

For reading, the indicated file must exist. For writing, if the file does not exist, it is created. If the file exists but all of the `table` declaration's *data-spec*s have read/write status `OUT`, it is overwritten. Otherwise, writing causes the existing file to be modified; each column written either replaces an existing column of the same name, or becomes a new column added to the table.

The format for the text files can be examined by writing one and viewing the results in a text editor. For example, the following AMPL session,

```
ampl: model diet.mod;
ampl: data diet2a.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032
ampl: table ResultList OUT "DietOpt.tab":
ampl?    [FOOD], Buy, Buy.rc, {j in FOOD} Buy[j]/f_max[j];
ampl: write table ResultList;
```

produces a file `DietOpt.tab` with the following content:

```
ampl.tab 1 3
FOOD Buy Buy.rc 'Buy[j]/f_max[j]'
BEEF 5.360613810741701 8.881784197001252e-16 0.5360613810741701
CHK 2 1.1888405797101402 0.2
FISH 2 1.1444075021312856 0.2
HAM 10 -0.30265132139812223 1
MCH 10 -0.5511508951406658 1
MTL 10 -1.3289002557544745 1
SPG 9.306052855924973 -8.881784197001252e-16 0.9306052855924973
TUR 1.9999999999999998 2.7316197783461176 0.19999999999999998
```

In the first line, `ampl.tab` identifies this as an AMPL relational table text file, and is followed by the numbers of key and non-key columns, respectively. The second line gives the names of the table's columns, which may be any strings. (Use of the ~ operator to specify valid column-names is not necessary in this case.) Each subsequent line gives the values in one table row; numbers are written in full precision, with no special formatting or alignment.