Database Structures for Mathematical Programming Models

Robert Fourer
Department of Industrial Engineering and Management Sciences,
Northwestern University, Evanston, IL 60208-3119, U.S.A.

Abstract. In both the design and use of large-scale mathematical programming systems, a substantial portion of the effort has no direct relation to the variables and constraints, but is instead concerned with the description, manipulation and display of data. Established principles of database design do not apply directly to mathematical programming, however, because there are significant differences of organization and content between the data for an optimization model and the data for a conventional database application such as payroll or order entry.

This paper derives fundamental principles of database construction for large-scale mathematical programming, by use of a steel mill planning model as an example. Alternative formulations for the model—which incorporate aspects of production and network linear programming—are presented at the outset, and are shown to correspond to relational and hierarchical database schemes that have contrasting strengths and weaknesses. A particular implementation of the steel optimization package is then presented as an illustration. A concluding section puts this work into perspective, by surveying and categorizing a variety of approaches for providing data management features in mathematical programming applications. The views of data offered by this paper's approach are seen to differ substantially from the views offered by traditional mathematical programming systems, and certain "intermediate" strategies for integration of database and mathematical programming software are identified as having particular promise for future work.

Published version: *Decision Support Systems* **20** (1997) 317–344.

This work has been supported in part by grants from the American Iron and Steel Institute, and by grant DDM-8908818 from the National Science Foundation.

1. Introduction

The work recounted here grew out of a project to design an optimization package for steel mill planning. Because the project was supported by the American Iron and Steel Institute, it was to be based on a generic model—one that any particular steel company could specialize to its own operations, simply by supplying its own data. Users of the model would be concerned mainly with entering and maintaining their data, and with reporting the optimal production levels.

In light of the data's central role, it was decided to implement the optimization package in the context of a database management system. The user would thus enter the description of a steel mill as a collection of materials and facilities records of a prescribed structure. An associated linear program would then be automatically generated and solved, and the optimal values would be automatically left in appropriate fields. Finally, the results would be displayed as desired, by use of the database system's varied reporting options.

As the implementation of the steel optimization package has progressed, it has become clear that many of the underlying principles are of much broader applicability. Certainly, the generic linear programming model can describe other productive enterprises that transform flows of raw materials into diverse finished products, through a series of processing steps. There are equally important general principles at work, however, in the way that the database structure is related to the formulation of the linear program.

The goal of this paper is to suggest some of the principles of database construction for linear programming (and for large-scale optimization in general) by use of the steel optimization model as an example. Two likely formulations of the linear program are introduced initially, and are shown to correspond to relational and hierarchical database schemes that have contrasting strengths and weaknesses. An implementation of the steel optimization package is then described, and its presentation of the data is contrasted to the data views typically provided by specialized optimization software. In conclusion, a variety of promising generalizations and alternatives are discussed.

The next part of this introduction briefly describes traditional connections between mathematical programming and database management systems, and cites some notable previous studies and implementations. The remainder of the paper is then outlined in greater detail.

1.1 Background

Large-scale linear programming applications optimize over hundreds or thousands of variables, subject to comparably large numbers of constraints. The maintenance of these models has long been recognized to involve a substantial task of data management. Indeed, in both the design and use of a mathematical programming system (or MPS), a substantial portion of the effort has no direct relation to the objective and constraints, but is instead concerned with the description, manipulation and display of data. This holds true both for traditional matrix generation systems [2, 38] and for more modern systems based on algebraic modeling languages

[8, 18] or matrix block schematics [10, 47].

The data required by mathematical programming models is not typified by the payroll and order entry data that appear as examples in textbooks on database management [14, 45]. As described for example by Palmer [37], an MPS's data tables tend to be smaller but more complex in structure, and tend to contain more numerical (as opposed to symbolic) information. Designers have responded to these features by creating data representations that are specific to MPS software. Most popular is a scheme of tables whose rows and columns are specially labeled [46], and whose entries may be maintained independently or may be generated automatically from a larger corporate database [30].

The characteristics of an MPS's data are not so special, however, as to preclude the use of standard database structures for purposes of mathematical programming. The similarities between MPS data tables and relational database tables have been remarked upon by several authors, including Welch [46], Choobineh [11] and Müller-Merbach [34]. MPS implementations using the so-called network database model were investigated by Bonczek, Holsapple and Whinston [7] and by Stohr and Tanniru [44]; more recently, an implementation of an MPS that employs aspects of the hierarchical and relational database models has been described by Baker [1].

Other researchers (to be cited later) have proposed ways in which standard database concepts or software can be integrated with—or interfaced to—various mathematical programming systems. In some cases the database aspect takes the lead, while the ability to optimize is regarded as an added feature. In other designs the lead is given to the mathematical programming aspect, with the database being just one feature of an integrated MPS. A third approach employs both a general-purpose database management system and a mathematical programming system, which pass information between them.

1.2 Outline

Section 2 presents two formulations of our linear programming example, which differ in how they describe the indexing of model components. Sections 3 and 4 then describe relational and hierarchical database structures, respectively, that correspond to the organization of data in the two formulations. Each structure has certain advantages and disadvantages, as explained in Section 5, with respect to ease of use, data storage, and data retrieval.

The relationships between the linear programming formulations (in Section 2) and the database structures (in Sections 3 and 4) are not merely a fortuitous consequence of the application at hand. One can readily discern general rules that connect familiar algebraic characterizations of data to familiar database concepts such as records, fields, and keys. Sections 3 and 4 also codify these rules and comment on their application.

The implementation designed for the steel optimization project is described in Section 6. Particular attention is given in this section to practical issues of speed and convenience, and to the appearance and mechanics of the user interface.

As a complement to Section 6's very specific focus, the concluding section com-

pares a variety of strategies for integrating database and linear programming software. Sections 7.1 and 7.2 survey ideas for the addition of linear programming features to database management systems, and for the incorporation of database features into linear programming systems, respectively; the latter also observes how traditional MPS's differ fundamentally from typical database management systems in their ways of organizing data and presenting it to the user. Section 7.3 describes several promising options for intermediate levels of integration between database and optimization software. Different integration strategies are seen to correspond to different kinds of coordination between a modeling system's algebraic description of data and a database management system's data scheme.

2. Formulations

We adopt the following general formulation that covers all linear programs (LPs) to be considered in this paper:

$$\begin{array}{ll} \text{Maximize} & \sum_{j=1}^{n} c_j x_j \\ \text{Subject to} & l_i^{\text{row}} \le \sum_{j=1}^{n} a_{ij} x_j \le u_i^{\text{row}}, \quad i = 1, \ldots, m \\ & l_j^{\text{col}} \le x_j \le u_j^{\text{col}}, \qquad\qquad\quad j = 1, \ldots, n \end{array}$$

It will be of conceptual and practical value to investigate database structures that are capable of representing any LP of this form, by storing all l_i^{row}, u_i^{row}, c_j, l_j^{col} and u_j^{col}, along with all a_{ij} that are nonzero.

This form is too general, however, for our prospective users to work with it directly. They need a much more specific formulation, as described below, that speaks of familiar materials and facilities, and of their limits, yields, costs and capacities.

We begin by introducing our specific model informally, using a few steelmaking concepts for examples. Then we present two formal descriptions, which differ mainly in how they define and use index sets. Sections 3 and 4 will show how these two descriptions naturally give rise to two different database structures.

2.1 An informal description

We consider a generic continuous-flow production process; raw materials enter, various transformations to intermediate goods are performed, and finished products leave. Profit is the total revenue from sales of the finished products, less the costs of acquiring the raw materials and making the transformations. Our problem is to run the plant at the most profitable levels of activity, in one future planning period. This kind of model is appropriate for, among other things, a quarterly or annual model of a steel mill's production. An early application at an American company is described in a 1958 paper by Fabian [16]; subsequent examples include a World Bank study of optimization models for the steel sector of the Mexican economy, described by Kendrick, Meeraus and Alatorre [29], and a strategic planning model for a German steel company by Bielefeld, Walter and Wartmann [4].

Specification of our model begins with a list of *materials,* and the following data for each:

- The cost per unit of material bought, and the minimum and maximum quantities that can be bought.
- The revenue per unit of material sold, and the minimum and maximum quantities that can be sold.

Normally, raw materials (such as coal, ore, or limestone) can only be bought, while finished products (tempered coils, pipes) are only sold. Intermediates (pig iron, slabs, unfinished coils) can often be neither bought nor sold, but there are exceptions depending on market conditions. We can set a product's maximum bought or sold to zero to indicate that no buying or selling of the product is possible. To "load" the facility for a specified level of production, the minimum sales amounts may be set equal to the maximum for every finished product.

For each material, the model may also optionally specify a list of *conversions* to other materials. Each conversion has a given yield and cost per unit (of the material being converted). Conversions often serve as a bookkeeping device. For example, tempered coils might be converted into five different products, each representing coils destined for a different market, and each with its own revenue per ton and sales limits. Some of the coils of secondary grade could also be downgraded to scrap by means of a conversion.

The major transformations of materials at a steel mill cannot be described as simple conversions. We must rather define a collection of *facilities* at which transformations occur. Each facility houses one or more productive *activities,* which use and produce materials in certain proportions. Specifically, the following information is provided for each activity at a facility:

- The amount of each input required by a unit of activity.
- The amount of each output produced by a unit of activity.
- The (variable) cost per unit of activity.
- Upper and lower limits on the number of units of activity.
- The number of units of activity that can be accommodated by one unit of the facility's overall capacity.

Naturally, there are upper and lower limits on the overall capacity of each facility. There are similar limits on a facility's total use of each input and total production of each output. Where the lower and upper limits are not critical, the lower may be set to 0, and the upper to a large, effectively infinite, value.

A common example of a facility is a rolling mill at which several finished products are made. The rolling of each product must be modeled as a separate activity, since it produces a separate output. The units of each activity are in tons (of product), while the units of the facility's capacity are in hours. Thus the model specifies each activity's capacity use in tons per hour. In effect, the activities are competing for the limited capacity of the facility; the optimization automatically allocates capacity to best account for factors such as yield, cost of production, and potential for revenue from sales.

A different example occurs at a basic oxygen furnace, where all activities produce liquid steel. Here activities differ in the amounts of the various inputs that they use to produce a ton of steel; each represents, in effect, a different recipe for steel production. Both the units of each activity and the units of the furnace's capacity are measured in tons of steel produced, so the capacity use figure is just 1. (The specification of discrete recipes can be viewed as approximating the solution of a more general blending problem at the furnace.)

We can characterize this model as a kind of hybrid between production and network flow, as depicted in Figure 2–1. The circles in the figure represent material balances, and the rectangles represent facilities. Each circle may have an inbound arc for purchases, or an outbound arc for sales. An arc from a circle to a rectangle denotes input of a material to a facility, and an arc from a rectangle to a circle denotes output of a material from a facility. An arc from a circle to another circle shows a conversion of one material to another.

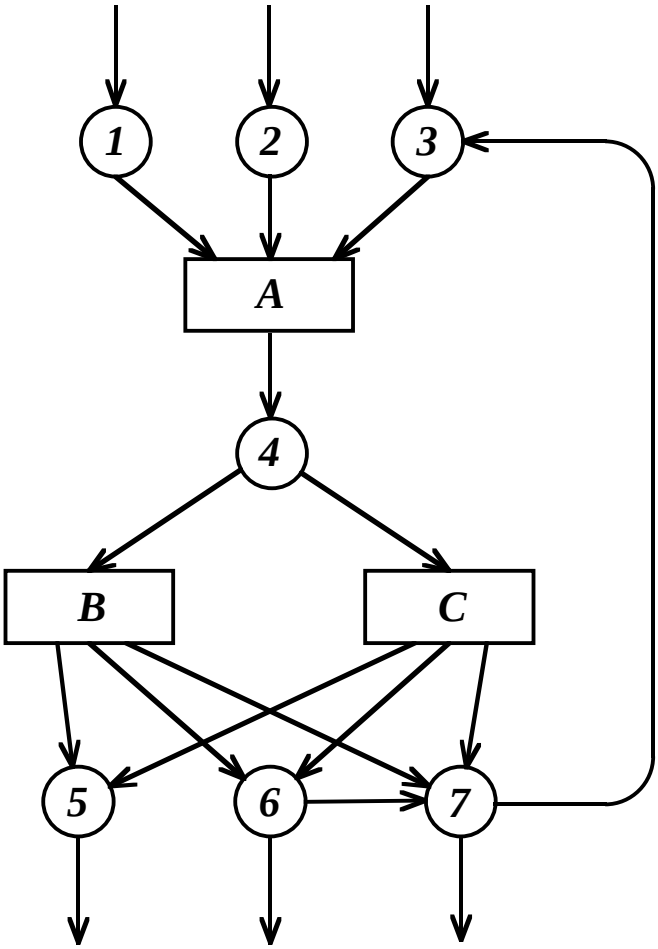


Figure 2–1. *Structure of a small instance of the multi-facility production problem.* Raw materials (1, 2 and 3) are transformed by facility A to an intermediate (4), which is then transformed by facilities B and C to finished product of primary and secondary grades (5 and 6) as well as scrap (7) that can be sold or recycled. Secondary product may also be scrapped if there is insufficient market for it; this option, modeled as a conversion, is represented by the arrow from (6) to (7).

It is convenient to imagine that the circles toward the top of the diagram correspond to raw materials, while those at the bottom correspond to finished products. Thus there is a general flow of materials from top to bottom. Exceptions to this general flow may occur, however, as a result of by-products that can be re-used in production. The most common example is steel scrap, which occurs as an output of every rolling and finishing facility, and which (if not sold) may be recycled as an input to steelmaking furnaces. Natural gas from coke ovens and blast furnaces can also be recycled, in a more complex scheme that produces heat or electricity.

2.2 First formulation

Our first algebraic description of the above problem is characterized by the use of ordered "tuples" of indices to describe productive possibilities. It is presented in full in Appendix A; we comment here on some of its more important conventions and features.

The model is built upon two fundamental sets of objects: a set $\mathcal{M}$ of materials, and a set $\mathcal{F}$ of facilities. Certain data values, such as the purchase limits for each material and the capacity of each facility, are indexed directly over these sets. However, much of the data is indexed over sets of ordered pairs and triples, whose components are taken from $\mathcal{M}$ and $\mathcal{F}$:

$\mathcal{M}^{\text{conv}}$	$\subseteq \mathcal{M} \times \mathcal{M}$	conversions
$\mathcal{F}^{\text{in}}$	$\subseteq \mathcal{F} \times \mathcal{M}$	facility inputs
$\mathcal{F}^{\text{out}}$	$\subseteq \mathcal{F} \times \mathcal{M}$	facility outputs
$\mathcal{F}^{\text{act}}$	$\subseteq \mathcal{F} \times$ **?**	facility activities
$\mathcal{A}^{\text{in}}$	$\subseteq \mathcal{F} \times \mathcal{M} \times$ **?**	activity inputs
$\mathcal{A}^{\text{out}}$	$\subseteq \mathcal{F} \times \mathcal{M} \times$ **?**	activity outputs

The construction and use of these sets are readily understood from the descriptions in Appendix A. However, the latter three have special characteristics that require further comment.

Consider the set $\mathcal{F}^{\text{act}}$. As defined in Appendix A, $(i,k) \in \mathcal{F}^{\text{act}}$ means that k is an activity available at facility i. Thus we can reasonably regard $\mathcal{F}^{\text{act}}$ as a subset of $\mathcal{F} \times \mathcal{A}$, where $\mathcal{A}$ is a previously unmentioned set of activities. Indeed, all of the three occurrences of **?** above would properly be references to $\mathcal{A}$.

We do not refer explicitly to the set $\mathcal{A}$ in our formulation, however, because activities do not play a role in the model apart from their association with facilities. There are no model components indexed directly over activities. Even if activities at different facilities were to have the same name, they would not necessarily be the same activity, and the linear program would be unaffected if the names of any of them were changed. These distinctions will eventually be seen to carry over to the database structure.

In the case of $\mathcal{A}^{\text{in}}$ (and analogously for $\mathcal{A}^{\text{out}}$), the model defines the domain of the triples more restrictively than the summary above:

$$\mathcal{A}^{\text{in}} \subseteq -(i,j,k) : (i,j) \in \mathcal{F}^{\text{in}} \text{ and } (i,k) \in \mathcal{F}^{\text{act}}{}''.$$

The set on the right is just $\mathcal{F}^{\text{in}} \bowtie \mathcal{F}^{\text{act}}$, the natural join of $\mathcal{F}^{\text{in}}$ and $\mathcal{F}^{\text{act}}$. We restrict $\mathcal{A}^{\text{in}}$ to this set because, for material j to be an input to activity k at facility i, the material must be an allowable input at the facility, or $(i,j) \in \mathcal{F}^{\text{in}}$, and the activity must exist at the facility, or $(i,k) \in \mathcal{F}^{\text{act}}$.

The only data values indexed over $\mathcal{A}^{\text{in}}$ are the technological coefficients, a^{in}_{ijk}. We could thus dispense with $\mathcal{A}^{\text{in}}$ entirely by indexing a^{in}_{ijk} directly over $\mathcal{F}^{\text{in}} \bowtie \mathcal{F}^{\text{act}}$, and by letting a^{in}_{ijk} be zero if activity k does not actually use input j. However, we would then lose a certain consistency in the data definition. In our current formulation,

collections of data values are indexed either over a single fundamental set ($\mathcal{M}$ or $\mathcal{F}$) or over a subset of a cartesian product of fundamental sets (such as $\mathcal{F} \times \mathcal{M}$). This convention, which extends to the variables and constraints as well, is not always followed in linear programming formulations, but we will see that it is valuable to the design of an associated relational database structure.

2.3 Second formulation

Our second algebraic description is characterized by a different way of representing the index sets, in which the ordered pairs and triples are replaced by indexed collections of subsets. As seen in Appendix B, where the full formulation is presented, this change has immediate implications for how the model is expressed, even though the model represents the same class of linear programs as before.

The six collections of subsets are written as follows in Appendix B:

$\mathcal{M}_j^{\text{conv}}$	$\subseteq \mathcal{M}$	conversions from material j
$\mathcal{F}_i^{\text{in}}$	$\subseteq \mathcal{M}$	inputs at facility i
$\mathcal{F}_i^{\text{out}}$	$\subseteq \mathcal{M}$	outputs from facility i
$\mathcal{F}_i^{\text{act}}$	$\subseteq$ **?**	activities at facility i
$\mathcal{A}_{ik}^{\text{in}}$	$\subseteq \mathcal{F}_i^{\text{in}}$	inputs to activity k at facility i
$\mathcal{A}_{ik}^{\text{out}}$	$\subseteq \mathcal{F}_i^{\text{out}}$	outputs from activity k at facility i

This change does not affect the model's names for the data values; both formulations have u_{ij}^{in}, a_{ijk}^{out}, and so forth. The difference is in how the models describe the indexing of this data. For example, the first formulation specifies a value u_{ij}^{in} for each $(i,j) \in \mathcal{F}^{\text{in}}$, while the second specifies a u_{ij}^{in} for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{in}}$. Sections 3 and 4 will show how these different ways of describing the data can correspond to different ways of organizing it in a database.

The **?** above refers to the same phantom "set of activities" as in the first formulation. There is no question as to the domains of $\mathcal{A}_{ik}^{\text{in}}$ and $\mathcal{A}_{ik}^{\text{out}}$, however. For example, the inputs available to activity k at facility i are a subset of the inputs available to all activities at facility i. Hence $\mathcal{A}_{ik}^{\text{in}} \subseteq \mathcal{F}_i^{\text{in}} \subseteq \mathcal{M}$; each $\mathcal{A}_{ik}^{\text{in}}$ is actually a sub-subset. The values a_{ijk}^{in} are correspondingly defined for all $i \in \mathcal{F}$, $k \in \mathcal{F}_i^{\text{act}}$, and $j \in \mathcal{A}_{ik}^{\text{in}}$.

3. Relational Structures

A subset of a cartesian product of sets is a *relation* in a mathematical sense. Thus it is not surprising that data indexed over pairs or triples from cartesian products, as in our first formulation, has a natural representation in relational database structures.

This section develops and comments upon likely relational schemes for our models. To establish principles and terminology, we first focus on the simple general linear programming model that was introduced at the beginning of Section 2. We observe how the nonzero coefficients of the model can be viewed as being indexed over ordered pairs, and propose a collection of relational tables appropriate to the structure of the data. We finally generalize from this example to a series of explicit rules for deriving relational database schemes from linear programming models, and explore the application of these rules to the considerably more complicated case of the multi-facility production model.

3.1 An ordered-pair view of the general model

We can think of the shape of the general linear program as being determined by two sets:

$\mathcal{I} = -1, \ldots, m''$ is the set of constraints (or coefficient rows)
$\mathcal{J} = -1, \ldots, n''$ is the set of variables (or coefficient columns)

Two "right-hand side" values are associated with each constraint:

l_i^{row} = the lower limit on constraint i, for each $i \in \mathcal{I}$
u_i^{row} = the upper limit on constraint i, for each $i \in \mathcal{I}$

Each variable has a "profit" and two bounds, as well as an activity level:

c_j = the profit (if positive) or cost (if negative) per unit of variable j, for each $j \in \mathcal{J}$
x_j = the activity level of variable j, for each $j \in \mathcal{J}$
l_j^{col} = the lower limit on variable j, for each $j \in \mathcal{J}$
u_j^{col} = the upper limit on variable j, for each $j \in \mathcal{J}$

Finally, the nonzero coefficients are indexed over a set of constraint-variable pairs:

$\mathcal{C} \subseteq \mathcal{I} \times \mathcal{J}$ is the set of coefficient nonzeroes:
$(i, j) \in \mathcal{C}$ means that variable j is used in constraint i
a_{ij} = the coefficient of variable j in constraint i, for each $(i, j) \in \mathcal{C}$

This approach to indexing the coefficients makes sense for most large linear programs, including those that derive from our production model, because each variable figures in just a few of the many constraints.

The set $\mathcal{C}$ of pairs has a *dimension* of 2, whereas the sets $\mathcal{I}$ and $\mathcal{J}$ of simple members have dimension 1. All of a set's members have a number of components equal to the dimension, and no two members are the same; these conventions, while

usually taken for granted in a mathematical context, must be made explicit to ensure that the analogous database is well defined.

With minor modifications, our general formulation of the linear program can accommodate the above view of the data:

$$\begin{array}{lll} \text{Maximize} & \sum_{j\in\mathcal{J}} c_j x_j & \\ \text{Subject to} & l_i^{\text{row}} \le \sum_{(i,j)\in\mathcal{C}} a_{ij}x_j \le u_i^{\text{row}}, & \text{for all } i \in \mathcal{I} \\ & l_j^{\text{col}} \le x_j \le u_j^{\text{col}}, & \text{for all } j \in \mathcal{J} \end{array}$$

The summation denoted by $\sum_{(i,j)\in\mathcal{C}} a_{ij}x_j$ is taken for each fixed $i \in \mathcal{I}$; it is interpreted as the sum over all j such that (i,j), for the given i, is a member of $\mathcal{C}$. This is the same convention that is used in Appendix A for the production model.

3.2 A relational database for the general model

A relational database is a collection of *files.* For our linear program, the simplest file is the one that identifies the constraints and their associated limits. This file contains a *record* for each constraint $i \in \mathcal{I}$. Every record has three *fields:*

row_name, a unique identifier for the constraint;
row_min, the constraint's lower limit;
row_max, the constraint's upper limit.

The *row_name* field is the database analogue of the subscript i; it is designated the *key* for this file, which must be different for each constraint record. The values stored in row_min and row_max naturally correspond to l_i^{row} and u_i^{row} in the algebraic formulation.

The structure of the constraints file would be represented in conventional notation as the relation CONSTRAINTS (*row_name*, row_min, row_max). The file's content would be depicted by exhibiting its records and fields as the *rows* and *columns,* respectively, of a relational *table:*

row_name	row_min	row_max
1	0.0e+00	1.0e+30
2	3.0e+01	3.0e+01
3	4.5e+02	4.8e+02
4	-1.0e+30	0.0e+00
5	-1.0e+30	1.0e+30

We adopt the terminology of record and field in this paper, however, to avoid confusion with the customary use of row and column in linear programming, as synonyms for constraint and variable. We also find it convenient to present the structure of the constraint file graphically, by means of the following concise diagram:

CONSTRAINTS
row_name
row_min
row_max

The TITLE of the file is at top, followed by the field names, with the key in *italics*. This kind of diagram will be seen to extend conveniently to depict a variety of relational and hierarchical structures.

The variables file for the general model is similarly presented as:

VARIABLES
col_name
col_profit
col_min
col_optimal
col_max

Here *col_name* is the variable's unique identifier, and the key for the file. The following four fields store the values called c_j, l_j^{col}, x_j, and u_j^{col} in the mathematical formulation. Strictly speaking, the field col_optimal is superfluous, since it holds the values of variables x_j rather than any data required to describe the linear program. However, for practical purposes it is highly desirable to be able to store the solution values in the same files as the data. For the same reason, we could have included a field in the constraints file for the dual values.

It remains to define a file to hold the nonzero coefficients. Its structure can be diagrammed as follows:

COEFFICIENTS
coeff_row
coeff_col
coeff_value

Each record's coeff_value field holds the nonzero coefficient of the variable specified by the *coeff_col* field, in the constraint specified by the *coeff_row* field. The pair (*coeff_row*, *coeff_col*) is the key, since every coefficient corresponds to a different combination of constraint and variable.

Two essential requirements remain unstated in the above diagram of the coefficients file: that each *coeff_row* entry must match a key (*row_name*) entry in the CONSTRAINTS file, and that each *coeff_col* entry must match a key (*col_name*) entry in the VARIABLES file. In the terminology of relational databases, *coeff_row* and *coeff_col* are *foreign keys* of the COEFFICIENTS relation. Following Date (1990), we use an arrow in our diagram to identify foreign key references:

COEFFICIENTS
coeff_row → CONSTRAINTS
coeff_col → VARIABLES
coeff_value

Each → represents a *many-to-one* relationship. For example, since there may be many nonzero coefficients in one constraint, there may be many COEFFICIENTS records that have the same constraint identifier in their *coeff_row* field. But each

constraint identifier may appear only once in the *row_name* field of the CONSTRAINTS file. (Because the many-to-one relationship is not, strictly speaking, with the CONSTRAINTS file but rather with the *row_name* field in that file, we could be more precise by saying something like coeff_row → CONSTRAINTS.*row_name*; indeed, such an expression might be forced on us if certain relations admitted more than one key. Since in this paper the relationship is always with the unique key field of a file, however, we omit the field designation to keep the notation concise.)

3.3 Principles of relational structure

In even the simple example above, we can see many principles for deriving a relational database structure from a linear programming formulation. Most fundamentally, there is a connection between sets (such as $\mathcal{I}$, $\mathcal{J}$ or $\mathcal{C}$) and files:

> ◇ **Rule R1** *(files)*: For each set $\mathcal{S}$ in the model, there is a corresponding file in the database.

To avoid having to say "the file corresponding to $\mathcal{S}$" again and again, we henceforth refer to such a file as the "$\mathcal{S}$-file".

A set's dimension, and its use in indexing other model components (such as l_i^{row}, x_j or a_{ij}), determine the fields of the corresponding file:

> ◇ **Rule R2** *(key fields)*: The $\mathcal{S}$-file has a number of key fields equal to the dimension of $\mathcal{S}$.
>
> ◇ **Rule R3** *(data fields)*: The $\mathcal{S}$-file has an additional data field for each model entity indexed over $\mathcal{S}$.

Because a set's members are distinct and all have the same dimension, **R2** always specifies a key that is valid in relational database terms.

The above rules determine the basic structure of relational files. We also make a connection between the membership of a set and the content of a file:

> ◇ **Rule R4** *(records)*: The $\mathcal{S}$-file has a record corresponding to each member of $\mathcal{S}$.

This means that, for example, our VARIABLES file contains a record for each variable in the model. For the variable whose identifier is j, the fields of the associated record contain j together with the numerical values c_j, l_j^{col}, x_j, and u_j^{col}. Whereas in the algebraic formulation these quantities were indexed independently over $\mathcal{J}$—the phrase "for each $j \in \mathcal{J}$" appears four separate times—in the database scheme it is most natural to group all similarly indexed entities together.

A final rule specifies how containment restrictions (such as $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{J}$) are represented in the database:

> ◇ **Rule R5** *(many-to-one relationships)*: For each containment restriction of the form $(i_1, \ldots, i_D) \in \mathcal{S} \Rightarrow i_d \in \mathcal{T}$, the dth key in the $\mathcal{S}$-file has a many-to-one relationship to the $\mathcal{T}$-file.

In the case of the expression $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{J}$, there are two containment restrictions: $(i,j) \in \mathcal{C} \Rightarrow i \in \mathcal{I}$ and $(i,j) \in \mathcal{C} \Rightarrow j \in \mathcal{J}$. Thus many-to-one relationships are defined in the database from a key of the $\mathcal{C}$-file to the $\mathcal{I}$-file (*coeff_row* $\rightarrow$ CONSTRAINTS), and from a key of the $\mathcal{C}$-file to the $\mathcal{J}$-file (*coeff_col* $\rightarrow$ VARIABLES).

3.4 A relational database for the production model

We now examine how the above rules can give a database scheme for the multi-facility production model set forth in Appendix A. In the process we point out a situation in which a generalization to **R5** might be desirable.

The production model explicitly defines two 1-dimensional sets. First is the set $\mathcal{M}$ of materials, over which are indexed l_j^{buy}, x_j^{buy}, u_j^{buy}, c_j^{buy} and l_j^{sell}, x_j^{sell}, u_j^{sell}, c_j^{sell}. The resulting file is:

MATERIALS
mat_name
buy_min
buy_opt
buy_max
buy_cost
sell_min
sell_opt
sell_max
sell_cost

Second is the set $\mathcal{F}$ of facilities, which indexes just l_i^{cap} and u_i^{cap}. It gives rise to the following file:

FACILITIES
fac_name
cap_min
cap_max

All of the others are subsets of pairs or triples.

The 2-dimensional sets $\mathcal{F}^{\text{in}} \subseteq \mathcal{F} \times \mathcal{M}$ and $\mathcal{F}^{\text{out}} \subseteq \mathcal{F} \times \mathcal{M}$ are directly analogous $\mathcal{C}$ above. The resulting files are:

FACILITY_INPUTS
in_fac $\rightarrow$ FACILITIES
in_mat $\rightarrow$ MATERIALS
in_min
in_opt
in_max

FACILITY_OUTPUTS
out_fac $\rightarrow$ FACILITIES
out_mat $\rightarrow$ MATERIALS
out_min
out_opt
out_max

The key pair (*in_fac*, *in_mat*) uniquely defines a facility input, and the remaining three fields correspond to what are called l_{ij}^{in}, x_{ij}^{in}, and u_{ij}^{in} in Appendix A. The

situation is entirely analogous for facility outputs.

The model's other two 2-dimensional sets give rise to files that are only a bit different. The set $\mathcal{M}^{\text{conv}} \subseteq \mathcal{M} \times \mathcal{M}$ yields two key fields that have many-to-one relationships to the same materials field:

MATERIAL_CONVERSIONS
from_mat → MATERIALS
to_mat → MATERIALS
conv_yield
conv_cost
conv_opt

The set $\mathcal{F}^{\text{act}}$ has only one many-to-one relationship:

ACTIVITIES
act_fac → FACILITIES
act_name
act_min
act_opt
act_max
act_cost
act_cap_rate

The lack of a relationship for *act_name* is a direct consequence of the absence of an explicit set $\mathcal{A}$ of single activities, which was discussed in Section 2. Notice that the formulation in Appendix A does not specify a containment restriction of the form $\mathcal{F}^{\text{act}} \subseteq \mathcal{F} \times \mathcal{A}$, which (from rule **R5**) would imply two many-to-one relationships as previously explained. Instead the formulation states only $\mathcal{F}^{\text{act}} \subseteq -(i,k) : i \in \mathcal{F}''$, which embodies only the one restriction that $(i,k) \in \mathcal{F}^{\text{act}} \Rightarrow i \in \mathcal{F}$.

It remains to consider the two 3-dimensional sets. Each member of $\mathcal{A}^{\text{in}}$ is a triple (i,j,k), where i is a facility, j is a material, and k is an activity; and each member indexes one data value, a^{in}_{ijk}. Thus, by rules **R2** and **R3**, the associated file has three key fields and a data field, as follows:

ACTIVITY_INPUTS
act_in_fac → FACILITIES
act_in_mat → MATERIALS
act_in
act_in_rate

The two many-to-one relationships in this scheme would be implied from rule **R5** by $\mathcal{A}^{\text{in}} \subseteq -(i,j,k) : i \in \mathcal{F}$ and $j \in \mathcal{M}''$. As discussed in Section 2, however, the model actually incorporates the stronger condition that $\mathcal{A}^{\text{in}} \subseteq -(i,j,k) : (i,j) \in \mathcal{F}^{\text{in}}$ and $(i,k) \in \mathcal{F}^{\text{act}\,\prime\prime}$, or equivalently $(i,j,k) \in \mathcal{A}^{\text{in}} \Rightarrow (i,j) \in \mathcal{F}^{\text{in}}$ and $(i,j,k) \in \mathcal{A}^{\text{in}} \Rightarrow (i,k) \in \mathcal{F}^{\text{act}}$. Because the condition to the right of $\Rightarrow$ involves a 2-dimensional rather than a 1-dimensional set, these restrictions go beyond the situations covered

by **R5**; they might be indicated in our relational database notation by

$$(act\_in\_fac, act\_in\_mat) \rightarrow \texttt{FACILITY\_INPUTS},$$
$$(act\_in\_fac, act\_in) \rightarrow \texttt{ACTIVITIES}.$$

Restrictions of these kinds can be enforced by relational database systems, but not so readily as the simpler many-to-one relationships that we have specified in our rules. The situation for $\mathcal{A}^{\text{out}}$ is exactly analogous; one need only substitute "out" for all occurrences of "in" above.

The completed database scheme, consisting of eight relational tables, can be used to support a broad variety of queries. We defer further discussion, however, to the comparison in Section 5 with the alternative hierarchical structure.

4. Hierarchical Structures

We now seek to, in effect, repeat the previous section's development, but working from the formulation of Appendix B rather than Appendix A. The result is a hierarchical data structure that will be seen, in the sequel, to have notable advantages as well as disadvantages.

Use of the term "hierarchical" in this context is suggested by Date [14, Section 26.5]; the same kind of structure is also often called "nested relational" or "non first normal form." Whereas the fields of a relational database record must contain single data items such as names or numbers, some fields in a hierarchical database file may themselves be structured as files. These subfiles (or nested relations) nicely capture the intent of the indexed subsets, such as $\mathcal{M}_j^{\text{conv}}$ and $\mathcal{A}_{ik}^{\text{in}}$, that appear in Appendix B's formulation.

The organization of this section parallels that of the previous one. We first rewrite the simple linear programming model (from the beginning of Section 2) so that the nonzero coefficients are specified by use of an indexed collection of subsets, and then show how the coefficient values are naturally represented within a hierarchical database structure. Finally we generalize to a series of explicit rules for deriving hierarchical database schemes from linear programming models, and apply these rules to the case of the multi-facility production model.

4.1 An indexed-subset view of the general model

Just as in the previous section, we imagine the shape of the general linear program as being determined a set $\mathcal{I}$ of constraints and a set $\mathcal{J}$ of variables, which index l_i^{row}, u_i^{row} and c_j, x_j, l_j^{col}, u_j^{col} respectively. The difference lies in the description of the coefficients.

Suppose that we define, for each j, a subset of $\mathcal{I}$ to specify which constraints use x_j:

$\mathcal{C}_j \subseteq \mathcal{I}$ is a subset of column nonzeroes:
$i \in \mathcal{C}_j$ means that variable j is used in constraint i

Then the nonzero coefficient values are described as follows:

a_{ij} = the coefficient of variable j in constraint i, for each $j \in \mathcal{J}$ and $i \in \mathcal{C}_j$

These are the same coefficients as in the previous section; all that has changed is the way that their indexing is presented.

With insignificant modifications, our general formulation of the linear program can accommodate this view as well:

$$\begin{array}{lll} \text{Maximize} & \sum_{j\in\mathcal{J}} c_j x_j & \\ \text{Subject to} & l_i^{\text{row}} \le \sum_{j\in\mathcal{J}:i\in\mathcal{C}_j} a_{ij}x_j \le u_i^{\text{row}}, & \text{for all } i \in \mathcal{I} \\ & l_j^{\text{col}} \le x_j \le u_j^{\text{col}}, & \text{for all } j \in \mathcal{J} \end{array}$$

The sum over "$j \in \mathcal{J} : i \in \mathcal{C}_j$" does seem rather awkward, however. Since there is one of these sums for each constraint, it might be preferable to define a subset for

each $i \in \mathcal{I}$ (rather than for each $j \in \mathcal{J}$):

$\mathcal{C}_i \subseteq \mathcal{J}$ is a subset of row nonzeroes:
$j \in \mathcal{C}_i$ means that variable j is used in constraint i
a_{ij} = the coefficient of variable j in constraint i, for each $i \in \mathcal{I}$ and $j \in \mathcal{C}_i$

Then the formulation simplifies to the following:

$$\begin{array}{lll} \text{Maximize} & \sum_{j \in \mathcal{J}} c_j x_j & \\ \text{Subject to} & l_i^{\text{row}} \leq \sum_{j \in \mathcal{C}_i} a_{ij} x_j \leq u_i^{\text{row}}, & \text{for all } i \in \mathcal{I} \\ & l_j^{\text{col}} \leq x_j \leq u_j^{\text{col}}, & \text{for all } j \in \mathcal{J} \end{array}$$

Of course, there may be other concerns than the convenience of the algebraic formulation. We will later observe, for example, that the subsets $\mathcal{C}_j$ are preferred for generating the "column-wise" data structures required by linear programming algorithms.

4.2 A hierarchical database for the general model

With respect to the unindexed 1-dimensional sets, a hierarchically structured database looks the same as a relational one. Thus we begin with the following two files from Section 3:

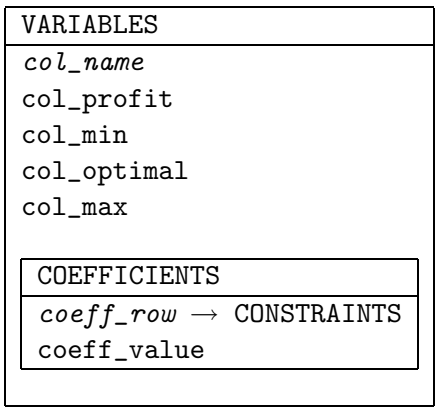
CONSTRAINTS
row_name
row_min
row_max

VARIABLES
col_name
col_profit
col_min
col_optimal
col_max

Consider now one of the sets $\mathcal{C}_j$, for a particular j. $\mathcal{C}_j$ is itself a 1-dimensional set, whose members are restricted to lie in $\mathcal{I}$; and for each $i \in \mathcal{C}_j$, there is a coefficient value. Thus we can imagine that $\mathcal{C}_j$ gives rise to a relational file as follows:

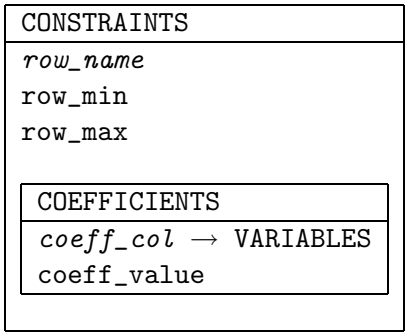
COEFFICIENTS
coeff_row → CONSTRAINTS
coeff_value

There is one of these files for each variable, just as there are *col_name*, col_profit, col_min, col_optimal, and col_max fields for each variable. Thus, in the hierarchical database scheme, a COEFFICIENTS file is incorporated as a *subfile* of the VARIABLES file. We depict this situation as follows:



The user of this structure can regard `COEFFICIENTS` as just another field of the `VARIABLES` file. Rather than containing a single value like other fields, however, this field is itself an entire relational subfile of values. This subfile has its own collection of records, each containing a *coeff_row* field and a `coeff_value` field.

When a database is structured in this way, all of the information about a variable, including a list of its nonzero coefficients, can be retrieved from a single record of the `VARIABLES` file. On the other hand, there is no obvious way to retrieve all nonzero coefficients in a single constraint. For such a retrieval to be equally easy, a different subfile, analogous to the set $\mathcal{C}_i$, would have to be added instead to the `CONSTRAINTS` file:



Just as the sets $\mathcal{C}_i$ and $\mathcal{C}_j$ tend to be convenient for different purposes, the subfiles of the `CONSTRAINTS` and `VARIABLES` files are useful for viewing the database in different ways. This characteristic represents both an advantage and a disadvantage for the hierarchical structure, but we defer further discussion to the comparison with the relational structure in Section 5.

4.3 Principles of hierarchical structure

In the hierarchical as in the relational case, our simple example suggests a number of principles for deriving a database structure from a linear programming formulation. Individual sets (such as $\mathcal{I}$ or $\mathcal{J}$) not indexed over other sets in the model have the same connection to files as before:

◇ **Rule H1a** *(files)*: For each unindexed set $\mathcal{S}$ in the model, there is a corresponding file in the database.

A companion rule makes the analogous connection between indexed collections of sets (such as $\mathcal{C}_j$ or $\mathcal{C}_i$) and subfiles:

◇ **Rule H1b** *(subfiles)*: For each collection of sets $\mathcal{T}_s$ indexed over $s \in \mathcal{S}$, the $\mathcal{S}$-file has a subfile corresponding to the collection $\mathcal{T}_s$.

This subfile rule has a straightforward extension, as we will see in the discussion of the production model below, to the case of "sub-subfiles" $\mathcal{U}_{st}$ indexed over $t \in \mathcal{T}_s$. (As in Section 3, we use the term "$\mathcal{S}$-file" to stand for "the file corresponding to $\mathcal{S}$"; and we use the term "$\mathcal{T}_s$-subfile" analogously.)

Every file or subfile has a key field, and a data field for each entity indexed over the underlying set:

◇ **Rule H2** *(key fields)*: Each file or subfile has one key field.

◇ **Rule H3** *(data fields)*: The $\mathcal{S}$-file (or $\mathcal{T}_s$-subfile) has an additional data field for each model entity indexed over $\mathcal{S}$ (or $\mathcal{T}_s$).

Considering our example, how do we know whether the entity a_{ij}, $j \in \mathcal{J}$, $i \in \mathcal{C}_j$ is indexed over $\mathcal{J}$ or over $\mathcal{C}_j$? We can conclude that it is indexed over $\mathcal{C}_j$, because there is exactly one piece of data a_{ij} for each $i \in \mathcal{C}_j$; it cannot be indexed over $\mathcal{J}$, on the other hand, because there are many numbers a_{ij} for each $j \in \mathcal{J}$.

The connection between the membership of a set and the content of a file is also much the same as before:

◇ **Rule H4** *(records)*: The $\mathcal{S}$-file (or $\mathcal{T}_s$-subfile) has a record corresponding to each member of $\mathcal{S}$ (or $\mathcal{T}_s$).

Thus our `VARIABLES` file contains a record for each variable in the model, just as in the relational case. For the variable whose identifier is j, however, the associated record comprises not only the fields that contain j together with numerical values c_j, l_j^{col}, x_j, and u_j^{col}, but also a `COEFFICIENTS` subfile for variable j. For each nonzero coefficient of variable j there is a record in this subfile whose fields contain i and a_{ij}.

It remains to prescribe the handling of containment restrictions in a hierarchical database:

◇ **Rule H5** *(many-to-one relationships)*: For each containment restriction of the form $j \in \mathcal{T}_s \Rightarrow j \in \mathcal{R}$, the key record in the $\mathcal{T}_s$-subfile has a many-to-one relationship to the $\mathcal{R}$-file.

In the case of the expression $\mathcal{C}_j \subseteq \mathcal{I}$ from our example, the containment restriction is $i \in \mathcal{C}_j \Rightarrow i \in \mathcal{I}$. Thus there is a many-to-one relationship from the key of the $\mathcal{C}_j$-subfile to the $\mathcal{I}$-file, represented by *`coeff_row`* $\rightarrow$ `CONSTRAINTS` in our database diagram.

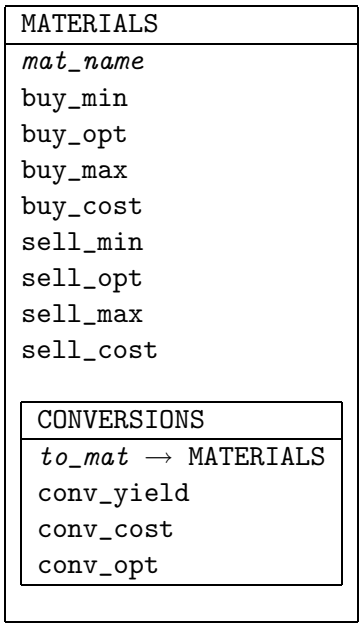
We have made no mention of multi-dimensional sets in this development, to emphasize that relationships such as $j \in \mathcal{J}$, $i \in \mathcal{C}_j \subseteq \mathcal{I}$, using indexed collections of

1-dimensional sets, can take the place of expressions such as $(i, j) \in \mathcal{C} \subseteq \mathcal{I} \times \mathcal{J}$ using a 2-dimensional set. The rules **R1**–**R5** and **H1**–**H5** are in fact compatible, however, and may be used together to provide a richer variety of indexing arrangements. For example, these rules can accommodate a collection of sets $\mathcal{T}_{qr}$ indexed over $(q, r) \in \mathcal{S} \subseteq \mathcal{Q} \times \mathcal{R}$, or a collection of 2-dimensional sets $\mathcal{T}_s \subseteq \mathcal{Q} \times \mathcal{R}$ indexed over $s \in \mathcal{S}$.

4.4 A hierarchical database for the production model

We conclude by showing how the above rules determine a hierarchical data scheme for Appendix B's version of the multi-facility production model.

The set $\mathcal{M}$ gives rise, by rule **H1a**, to a file that is like our previous example of the `VARIABLES` file. The members $j \in \mathcal{M}$ index not only the values l_j^{buy}, x_j^{buy}, u_j^{buy}, c_j^{buy} and l_j^{sell}, x_j^{sell}, u_j^{sell}, c_j^{sell}, but also the subsets $\mathcal{M}_j^{\text{conv}}$. Thus rules **H1b**, **H2** and **H3** specify that the corresponding `MATERIALS` file must have a key field and eight data fields, plus a subfile. Since for each $j' \in \mathcal{M}_j^{\text{conv}}$ there are further values $a_{jj'}^{\text{conv}}$, $c_{jj'}^{\text{conv}}$ and $x_{jj'}^{\text{conv}}$, the subfile must by the same rules have a key field and three data fields. We thus arrive at a scheme that can be diagrammed as follows:



The record for a particular material contains, in addition to the purchase and sales data, a subrecord for each thing that the particular material can be converted to. The notation *to_mat* → `MATERIALS` indicates that the target of each conversion must itself be a material in the database; it corresponds by rule **M5** to the containment restriction $j' \in \mathcal{M}_j^{\text{conv}} \Rightarrow j' \in \mathcal{M}$, which is implied by the statement $\mathcal{M}_j^{\text{conv}} \subseteq \mathcal{M}$ in the model's algebraic formulation.

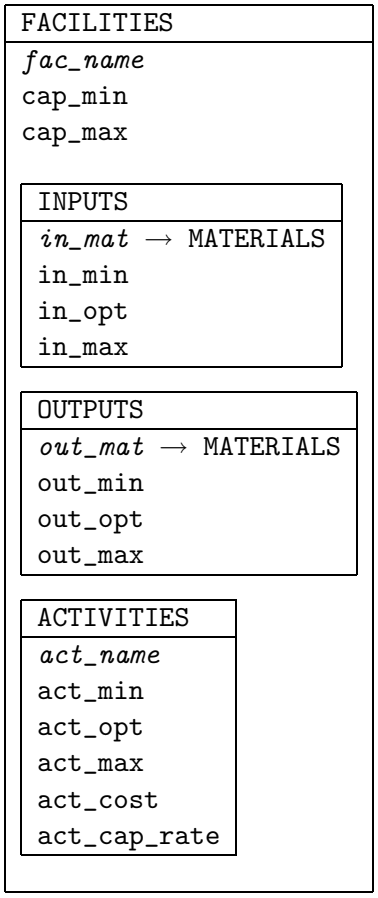
Some studies of hierarchical database structures [41] have found it convenient to assume that every subfile contains at least one record. In a typical multi-facility planning application, however, only a few of the materials are subject to conversions;

that is to say, $\mathcal{M}_j^{\text{conv}}$ is the empty set for many of the materials j. For this situation to be properly represented in the database, it is necessary that the corresponding records in the `MATERIALS` file be allowed to have "empty" `CONVERSIONS` subfiles.

We can apply analogous reasoning to start building a hierarchical scheme for facilities. The members $i \in \mathcal{F}$ index the values l_i^{cap}, u_i^{cap} and the subsets $\mathcal{F}_i^{\text{in}}$, $\mathcal{F}_i^{\text{out}}$, $\mathcal{F}_i^{\text{act}}$; members of the subsets also index certain values, as follows:

$$
\begin{aligned}
&j \in \mathcal{F}_i^{\text{in}}: \quad l_{ij}^{\text{in}},\ x_{ij}^{\text{in}},\ u_{ij}^{\text{in}} \\
&j \in \mathcal{F}_i^{\text{out}}: \ l_{ij}^{\text{out}},\ x_{ij}^{\text{out}},\ u_{ij}^{\text{out}} \\
&k \in \mathcal{F}_i^{\text{act}}: \ l_{ik}^{\text{act}},\ x_{ik}^{\text{act}},\ u_{ik}^{\text{act}},\ c_{ik}^{\text{act}},\ r_{ik}^{\text{act}}
\end{aligned}
$$

It follows that there should be a key field, two data fields, and three subfiles. Each subfile should have its own key field, and three to five data fields. The result looks like this:

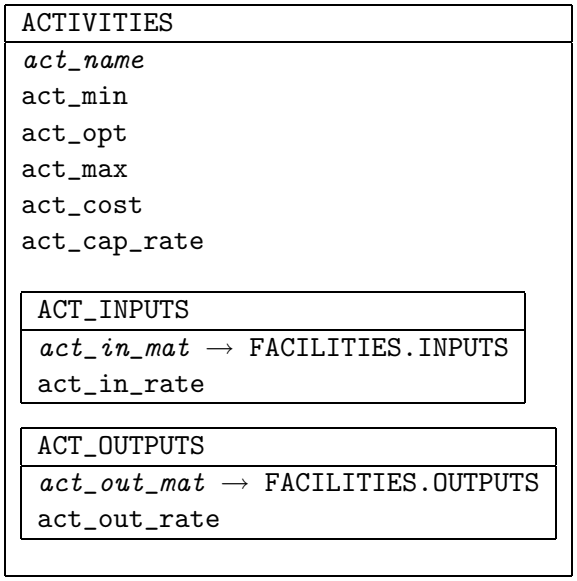


Here each facilities record contains three collections of subrecords, rather than just one as in the case of the materials.

Still missing from this structure, however, is the detailed activities data. For each k in the indexed set $\mathcal{F}_i^{\text{act}}$, the model specifies not only the values l_{ik}^{act}, x_{ik}^{act}, and so forth, but also the sets $\mathcal{A}_{ik}^{\text{in}}$ and $\mathcal{A}_{ik}^{\text{out}}$, which are in turn also used to index data values:

$j \in \mathcal{A}_{ik}^{\text{in}}$: a_{ijk}^{in}
$j \in \mathcal{A}_{ik}^{\text{out}}$: a_{ijk}^{out}

Since we have here two sets ($\mathcal{A}_{ik}^{\text{in}}$, $\mathcal{A}_{ik}^{\text{out}}$) indexed over a set ($\mathcal{F}_i^{\text{act}}$) that is itself indexed over a set ($\mathcal{F}$), we are led to extend Rule **H1b** to let sub-subfiles appear within subfiles. Specifically, two sub-subfiles that correspond to $\mathcal{A}_{ik}^{\text{in}}$ and $\mathcal{A}_{ik}^{\text{out}}$ must appear within the subfile `ACTIVITIES` that corresponds to $\mathcal{F}_i^{\text{act}}$. Calling the sub-subfiles `ACT_INPUTS` and `ACT_OUTPUTS`, the subfile for activities is diagrammed as follows:



The foreign key specification *act_in_mat* → `FACILITIES.INPUTS` represents the natural generalization of rule **H5** to the model's containment restriction $\mathcal{A}_{ik}^{\text{in}} \subseteq \mathcal{F}_i^{\text{in}}$. It specifies a many-to-one relationship from the key field of the sub-subfile corresponding to $\mathcal{A}_{ik}^{\text{in}}$ (that is, *act_in_mat*), to the subfile corresponding to $\mathcal{F}_i^{\text{in}}$ (which we denote `FACILITIES.INPUTS` because it is the `INPUTS` subfile of the `FACILITIES` file). The comparable relationship in the relational context is not so easy to represent, as we have already noted in Section 3.

The full facilities file is represented by substituting the above `ACTIVITIES` subfile diagram for the one that appears within the previously described `FACILITIES` diagram. A complete hierarchical database structure for the production problem thus consists of just two files, but with extensive internal structures of subfiles and sub-subfiles. The diagrams for both the hierarchical and relational cases are collected in Appendix C, where the reader may more easily compare them. We proceed next to consider their relative advantages and disadvantages.

5. Comparisons

The implementation to be described in Section 6 employs a database management system that supports both hierarchical and relational schemes. Thus, notwithstanding the current popularity of the relational model, for our application the choice of scheme has been more than an academic matter.

Before proceeding to the implementational details, we consider here some general contrasts between the relational and hierarchical approaches. Most obviously, the relational approach should tend to be more attractive to those who would prefer to formulate the algebraic linear program as it is given in Section 3, while the hierarchical approach should appeal to those who would prefer the formulation of Section 4. There are other important concerns, however, which we divide below into three categories; we argue that there are benefits either way in *ease of use,* but that there is a clear tradeoff between a hierarchical scheme's compact *data storage* and a relational scheme's flexible *data retrieval.*

5.1 Ease of use

In a realistic setting, the database scheme of Section 3 or 4 would be maintained by a database *administrator* who would also have responsibility for implementing displays and reports. A variety of database *users* would enter different collections of materials, facilities, bounds, yields, costs and so forth, for the purpose of modeling different plants and scenarios. Thus our proposed schemes are valuable only insofar as they can be implemented, within a database management system, to serve the needs of both administrators and users.

The hierarchical database diagrams in Appendix C may appear, at first glance, to be more complicated than their relational counterparts. Yet, on closer examination, the hierarchical scheme is seen to offer an especially simple and straightforward representation of the data. There are only two files, one for materials and one for facilities. In the materials file, each record contains all the purchase and sales data on one material, plus a list of any applicable conversions together with their costs and yields. In the facilities file, each record likewise specifies all information pertinent to a particular facility; and each activity subrecord gives all information pertinent to one activity at the facility. There is a certain intuitive arrangement to the hierarchical structure that is readily conveyed to database users.

The relational structure, by comparison, has a larger number of simpler files, and arguably requires the administrator to have a greater understanding of database principles in order to arrange for the extraction of desired information. The relational approach derives a significant advantage, however, from its solid foundation in the theory of relational algebra and relational calculus, and from its use of well-established concepts such as *key, query* and *view* in the relational context. Numerous implementations are based on the relational theory and concepts; many support a query language, SQL, for which there is a written standard. Popular textbooks [14, 45] also emphasize the relational approach. In contrast, implementations of hierarchical databases are less common and are minimally standardized. Thus an experienced database administrator is more likely to be familiar with the relational

structure of Section 3, and can choose from a wider range of software with which to implement it.

While these distinctions between hierarchical and relational are important, in practice they may be less than absolute. Relying on the features of a relational database management system, an administrator may be able to set up a relational scheme that supports certain hierarchical *views* for the user. Conversely, since hierarchical database schemes may be viewed as *nested* relational schemes, the popular relational algebra and relational query languages may be generalized for application to the hierarchical case; there has been extensive research along these lines in recent years, as the annotated bibliography in [14, Section 26.5] suggests.

5.2 Data storage

All of our proposed database schemes satisfy the normalization requirement that no piece of information be stored in more than one place. The hierarchical schemes do allow the data to be maintained more compactly, however.

The difference can be seen clearly in the data structures for the general LP model, which are shown side-by-side in Appendix C. There would be an exact correspondence between fields in the two structures, except that the *coeff_col* key field in the relational structure does not appear in the hierarchical one. This field is needed in the relational approach, because each coefficient is regarded as an independent entity, which can be identified only by giving the constraint (*coeff_row*) and variable (*coeff_col*) to which it belongs. In the hierarchical approach, however, a coefficient is considered part of the information about a variable; within the coefficient subrecords for a variable, it suffices to specify the constraint to which each coefficient belongs.

This difference of one field leads to several reductions in what must be stored and retrieved. The hierarchical structure saves one *coeff_col* entry for each of the many coefficients. In addition, any space that would be required by an *index* file for the *coeff_col* field—to facilitate searching or sorting on the field—is also saved. Finally, a lengthy index for the *coeff_row* field can also be avoided. Under the hierarchical structure, each coefficient subfile can be expected to have just a few records, because linear programs typically involve each variable in only a few constraints. Hence the subfile's *coeff_row* key field requires only a minimal form of indexing to permit efficient searches of subrecords. The extent of these savings necessarily depends on implementational details; in Section 6 we exhibit some examples based on experiments with realistic data.

Many opportunities for comparable savings can be found in our database for the multi-facility production problem. Corresponding to each of the four files having two key fields and the two files having three key fields in the relational structure, there is just one key field needed in the hierarchical structure. Thus there is a savings of the contents of 8 key fields and of as many as 14 indexes. Indeed, the only sizable indexes likely to be used by the hierarchical scheme are for the *mat_name* key field of the `MATERIALS` file, and the *fac_name* key field of the `FACILITIES` file.

5.3 Data retrieval

We have previously hinted that the hierarchical structure pays for its compactness with a loss of flexibility. In the case of the general LP formulation, the difference is essentially that the hierarchical scheme can only easily access coefficients "by column" whereas the relational scheme can also conveniently scan them "by row".

As a specific illustration, consider the problem of displaying all nonzero coefficients in a given constraint, together with the values of the corresponding variables. Such a request is readily filled by use of the select and join operations provided by any relational database management system. We would not expect to easily perform any comparable query on the hierarchical structure, however, because it does not conveniently collect the coefficients in a single file; they are scattered throughout the subfiles of different records in the `VARIABLES` file. This difficulty can be remedied in various ways, as described for example by [45, Section 2.6], but only by giving up some or all of the hierarchical scheme's advantage in data storage.

So long as we only want to access the nonzero coefficients of given variables, however, the hierarchical structure should be quite efficient. Indeed, all of the coefficient information for a variable can be retrieved directly as part of the record for the variable.

In the case of the multi-facility production model, the situation is predictably similar. Some queries are as easily performed on the hierarchical structure as on the relational one, because they involve just scanning a record and its subrecords (and possibly their sub-subrecords). Examples include the following:

- What materials can be converted from a given material, and at what cost and yield?
- What are all the materials used as input by a given facility, and how much of each is used in the optimal solution?
- What are the costs of all the activities at a given facility?

Where the desired information is scattered among subrecords, however, the relational model can be expected to be superior—as in these seemingly similar examples:

- What materials can be converted to a given material, and at what cost and yield?
- What are all the facilities that use a certain material as input, and how much do they use in the optimal solution?
- What are the costs of all the activities that make use of a given material?

These kinds of queries are useful not only for retrieving information of interest, but for diagnostic purposes. For instance, if the last query above shows that no activity makes use of a given material, then there is reason to expect an error or omission in the data.

6. Implementation

Our investigation of database structures was originally motivated by the intention of implementing them in an easy-to-use system for production planning. This section describes the system that has evolved. Our purpose is not to promote any particular database software or hardware, but rather to give the reader some feel for what is involved in building a working system around the ideas of the previous sections. Thus we emphasize the practical consequences of implementing database structures for linear programs, especially where difficult or unexpected design decisions have been required.

The operation of our system is quickly described in general terms. The principal steps of its use are as follows:

1. Collect data describing a production scenario, and store it according to one of the previously described database schemes for the multi-facility production model.
2. Extract the bounds and the nonzero coefficients of the associated linear program, and store them according to one of the previously described database schemes for the general linear programming model.
3. Solve the linear program, and enter the optimal values in the appropriate fields of the database files.
4. Display and print the results as required.

As a practical matter, we expect step 1 to be only rarely performed in full. Usually an existing scenario will be modified by changing only a few fields, and step 2 will modify the bounds and coefficients accordingly. Steps 3 and 4 will then be repeated to obtain and examine the new solution.

The presentation in this section has four parts that correspond to major concerns of the implementation: data management, optimization, reporting, and updating.

6.1 Data management

In preliminary discussions with potential users, we encountered a widespread perception of linear programming as a difficult modeling technique that could be applied only through a great investment in time and effort. We resolved to design a system that would tend to counteract this perception as much as possible.

Our first decision concerned the database management system software within which our database schemes would be implemented. We chose a package called *4th Dimension* [40], which offered especially powerful facilities for defining menus and graphical entry screens tailored to our application. Fortuitously, 4th Dimension also provided a stimulus to our research by supporting both relational and hierarchical models of data; but its selection was based more on its interface features than on any other factor. In more recent years, similar features have spread to popular database packages running under Microsoft Windows, so that if we were to start the project today we would have a much wider choice.

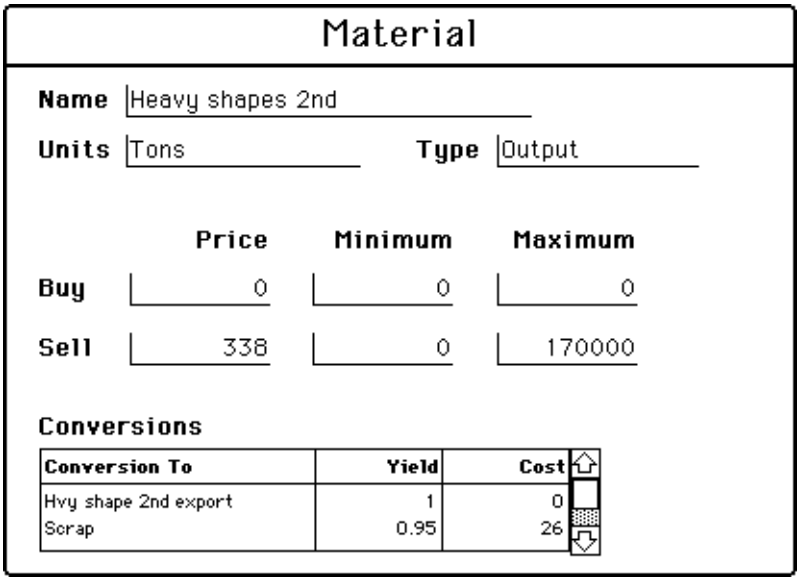


Figure 6–1. *A data entry layout for a record in the* `MATERIALS` *file.*

We next chose to implement the hierarchical scheme for the multi-facility production data. Our decision was based largely on the perception that, in the first version of the 4th Dimension software, hierarchical structures were best supported and easiest to work with. Subsequent versions have offered much stronger support for relational structures; our experiments with a relational scheme for the `VARIABLES`, `CONSTRAINTS` and `COEFFICIENTS` files are described later in this section.

The "design environment" of 4th Dimension is employed to define the underlying database scheme, as well as to design layouts through which the user can enter, examine or modify data in individual records. The layout for materials, presented in Figure 6–1, is seen to provide entry areas for seven of the fields of the hierarchical `MATERIALS` file:

`mat_name`
`buy_cost`, `buy_min`, `buy_max`
`sell_cost`, `sell_min`, `sell_max`

Two additional fields have been added for informational purposes: one to describe the units in which the material is measured, and one to classify the material as an input, intermediate or output. This sort of information can be valuable to users even though it has no bearing on the linear program.

The scrollable area at the bottom of the layout presents the records of the `CONVERSIONS` subfile, one to a line. Each line contains what we have called the *`to_mat`*, `conv_yield` and `conv_cost` records of a subfile field. To enter a new conversion, the user double-clicks at the top of the area, bringing up the subsidiary layout of Figure 6–2. A menu of material names for the *`to_mat`* field can be pulled down as illustrated. We find this kind of menu to be an essential practical feature, because users do not remember the exact names of all materials they have defined.

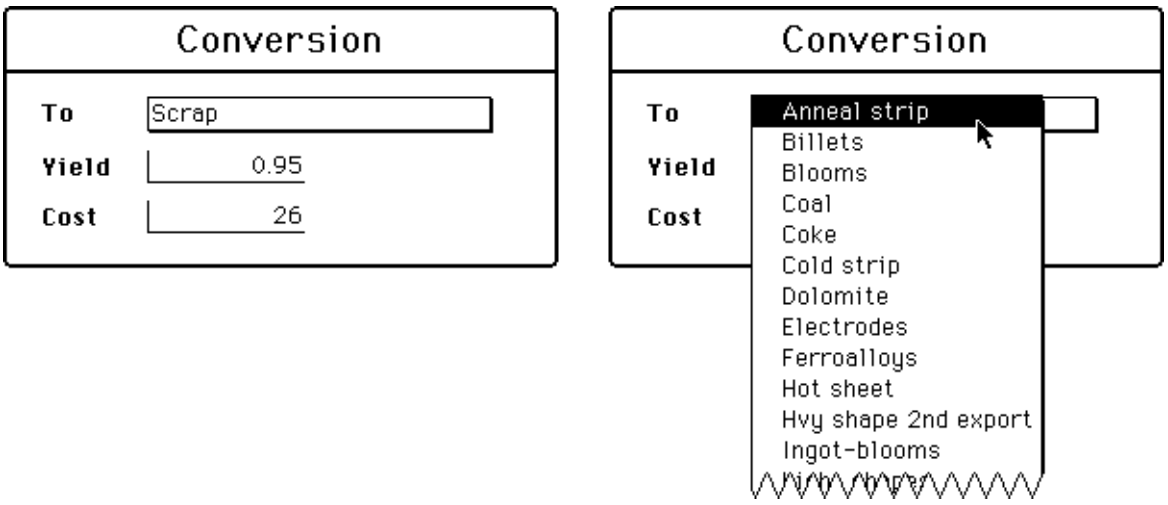


Figure 6–2. *Two views of a data entry layout for a subrecord in the* `CONVERSIONS` *subfile of the* `MATERIALS` *file.* At left a typical subrecord is displayed. At right the user has pulled down a menu of materials that are eligible targets for conversion.

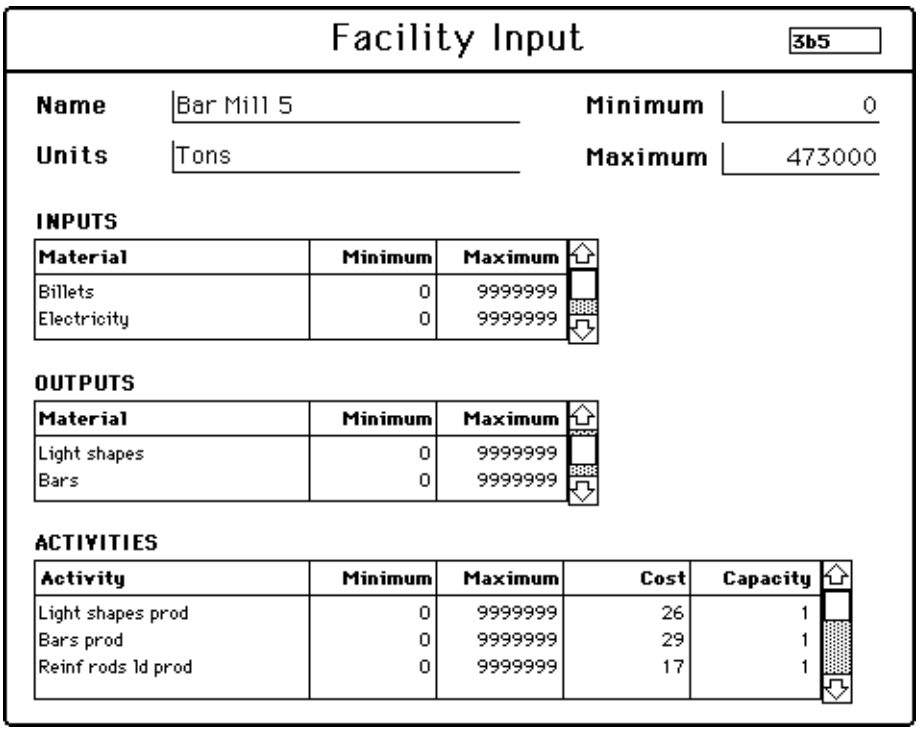


Figure 6–3. *A data entry layout for a record in the* `FACILITIES` *file.*

The same approaches are employed for the facilities data. A record of the hierarchical `FACILITIES` file is entered, displayed or modified through the layout shown in Figure 6–3. Toward the top are areas for the *`fac_name`*, `cap_min` and `cap_max` fields. A small area at the top right holds a "tag" field on which reports are sorted;

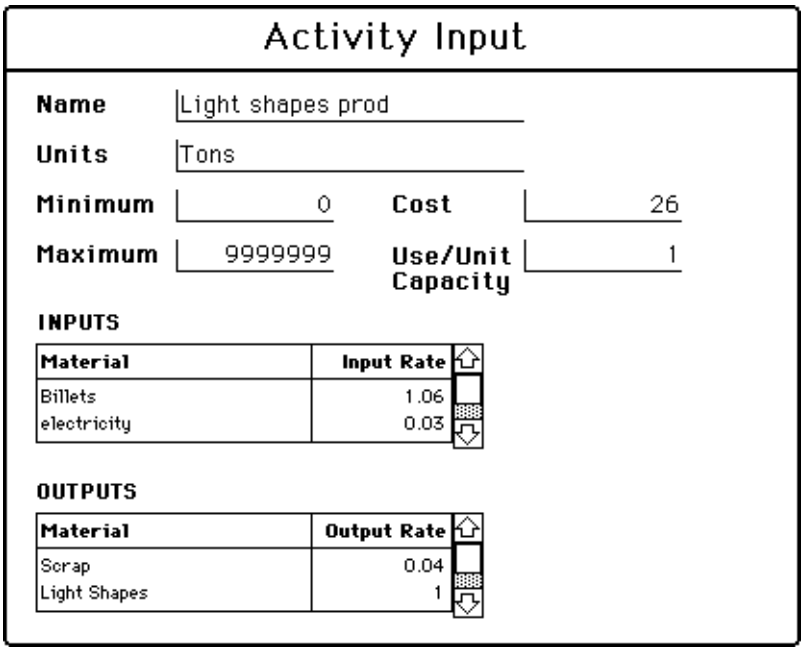


Figure 6–4. *A data entry layout for a subrecord in the* `ACTIVITIES` *subfile of the* `FACILITIES` *file.*

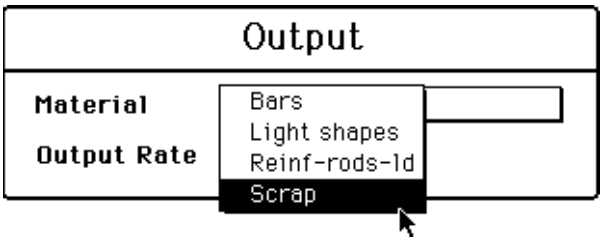


Figure 6–5. *A data entry layout for a sub-subrecord in the* `ACT_OUTPUTS` *sub-subfile of the* `ACTIVITIES` *subfile (of the* `FACILITIES` *file).* The user has pulled down a menu of possible output materials at the facility that contains this sub-subrecord.

properly chosen, the entries in this field cause the facilities to appear in some physically meaningful order. The lower part of the layout contains the scrollable areas that display records from the three subfiles. The `INPUTS` and `OUTPUTS` subfiles are managed in much the same way as the `CONVERSIONS` subfile described above. The `ACTIVITIES` subfile is a more complex case, because it has sub-subfiles that cannot be summarized in the scrollable area.

To work on the sub-subfiles of an activity subrecord, the user double-clicks on a line in the activity area to bring up the activity layout shown in Figure 6–4. Areas for the activity's *`act_name`*, `act_min`, `act_max`, `act_cost` and `act_cap_rate` fields are seen at the top. The two scrollable areas display records from the `ACT_INPUTS` and `ACT_OUTPUTS` sub-subfiles; that is, they show the activity's inputs and outputs, and their rates of use. Double-clicking on these brings up a small layout, like the one in Figure 6–5, that works much like the small layout for conversions (Figure 6–2).

The main difference is that the pull-down menu shows only the materials currently entered in the facility's `INPUTS` or `OUTPUTS` subfile, since these are the only materials available for the facility to use. (In the algebraic model of Appendix B, this is the restriction expressed as $\mathcal{A}_{ik}^{\text{in}} \subseteq \mathcal{F}_i^{\text{in}}$ and $\mathcal{A}_{ik}^{\text{out}} \subseteq \mathcal{F}_i^{\text{out}}$.)

6.2 Optimization

To find profit-maximizing levels of operation, our implementation executes a series of programs to perform the following operations:

1. Determine the constraints of the associated linear program. Extract constraint-related data values (l_i^{row} and u_i^{row}) and store them in a separate `CONSTRAINTS` file of the database.
2. Determine the variables of the associated linear program. Extract variable-related data values (c_j, l_j^{col}, u_j^{col} and nonzero coefficients a_{ij}) and store them in a separate `VARIABLES` file of the database.
3. Scan the `CONSTRAINTS` and `VARIABLES` files, and write all of the essential information about the linear program to an ordinary text file in a compact format.
4. Read the text file, solve the indicated linear program, and write the optimal values of the variables to a second text file.
5. Read the second text file, and place the optimal values in appropriate fields of the `MATERIALS` and `FACILITIES` files.

Steps 1–3 and 5 are implemented in 4th Dimension's database programming language, which can specify the actions associated with menu items and layouts, and with objects (such as buttons and fields) within layouts. This language resembles a cross between Pascal and HyperTalk, with numerous built-in functions for working with database files, fields, and records. Step 4 uses separate, general-purpose linear programming software. We comment below on the major design issues posed by individual steps, and then address overall limitations on problem size.

For steps 1 and 2, the structures of the `CONSTRAINTS` and `VARIABLES` files may be either the relational ones of Section 3 or the hierarchical ones of Section 4. Table 6–6 compares the performance of these alternatives on a moderately large example in 286 materials and 19 facilities. The hierarchical structure, which has been used in our implementation, requires somewhat less disk space and computation time, as the discussion in Section 5 has predicted. The relational structure is still efficient enough to be practical, however.

Step 2 generates variables x_j^{buy} (or x_j^{sell}) only for those materials that have a positive upper limit u_j^{buy} (or u_j^{sell}). Since typically few materials are both bought and sold, and many intermediates are neither bought nor sold, this refinement can substantially reduce the number of `VARIABLES` records that must be generated. The other variables, which are much less likely to have upper limits of zero, are always generated.

For step 4, we employ a general-purpose programming language to create a short driver that reads the first text file, calls an algorithm from an optimization subroutine library, and writes the optimal values to the second text file. We use Fortran

	HIER	RELA
Materials	286	
Facilities	19	
Constraints	853	
Variables	847	
Nonzeroes	2337	
Steel data	399K	
LP data	557K	740K
Gen const	1:39	
Gen var	7:21	9:01
Write constr	:55	:59
Write var	1:19	1:33
Solve (XMP)	1:59	
Read var	1:38	
Read constr	1:25	

Table 6–6. *A comparison of the hierarchical and relational structures for the* `VARIABLES` *and* `CONSTRAINTS` *files.* Centered quantities are the same for both alternatives. Timings are in minutes and seconds on a Macintosh IIci with a 50 MHz 68030 accelerator, using 4th Dimension 3.0.1 and 4D Compiler 2.0.2; the four groups of times (Gen, Write, Solve, Read) correspond to steps 1–2, 3, 4, and 5, respectively, as defined in the text.

and the primal simplex routines from the XMP library [32], but our arrangement could easily be adapted to use any collection of routines that apply an algorithm for linear programming. Since XMP, like most linear programming codes, requires only a column-wise representation of the coefficient matrix, we find it convenient to place the subfile of nonzero coefficients within the `VARIABLES` file.

It is not hard to coordinate steps 3–5 so that each optimal value is placed in the appropriate `col_optimal` record of the `VARIABLES` file. The user wants to look at materials and facilities, however, not at constraints and variables. Thus step 5 must be able to determine, by looking at a `VARIABLES` record, where the associated optimal value ought to go in the `MATERIALS` or `FACILITIES` files. At the cost of some extra processing, this information could be encoded in the *`col_name`* field. At the cost of some extra space instead, we place the information in three fields, one that gives the variable's type (material bought, input to facility, ...) and two that specify the pertinent material, facility or activity as appropriate. The *`col_name`* key field holds a unique record number that has no intrinsic meaning.

There is no inherent limit on the size of optimization problems that can be generated and solved by this kind of arrangement. When our our first prototype was completed in 1987, the speed of available Macintosh hardware and the 4th Dimension software imposed a practical limit of about 1000 variables. Subsequent advances have pushed the limit much higher, so that—for single-period planning in steel mills—the true practical limit at this point is determined by the level of detail that is worth modeling, rather than by the capabilities of the implementation. A linear program of even 2500 variables would represent a very high level of detail.

6.3 Reporting

A modified collection of layouts present the optimal values along with the input data. Figure 6–7 shows the optimality layout for a representative facility. The columns labeled "actual" give the optimal levels of inputs used, outputs produced and activities operated; these amounts are precisely the optimal values of the x_{ij}^{in}, x_{ij}^{out} and x_{ik}^{act} variables.

The number labeled "capacity used" is $\sum_{k \in \mathcal{F}_i^{\text{act}}} x_{ik}^{\text{act}} / r_{ik}^{\text{act}}$, the total capacity required to run all of the facility's activities at their optimal levels. This derived value, taken from the body of the capacity constraint, shows the user whether the facility is a bottleneck. Functions of data values are easily added to a layout by specifying a short "script", in the 4th Dimension programming language, that is executed every time a new record is displayed.

The availability of optimal values alongside data values is in fact a key feature of any reporting environment for linear programming. The facilities of 4th Dimension permit elaborate transformations of the results, to put them in forms that people want to see. As one example, our implementation provides a summary layout (Figure 6–8) that breaks the different kinds of cost out of the net profit. A variety of printed reports are also available through selection of menu items.

Another kind of reporting is provided by a sequence of diagnostic tests imple-

Figure 6–7. *An analogue of the layout in Figure 6–3, but showing optimal values as well as the facility data.*

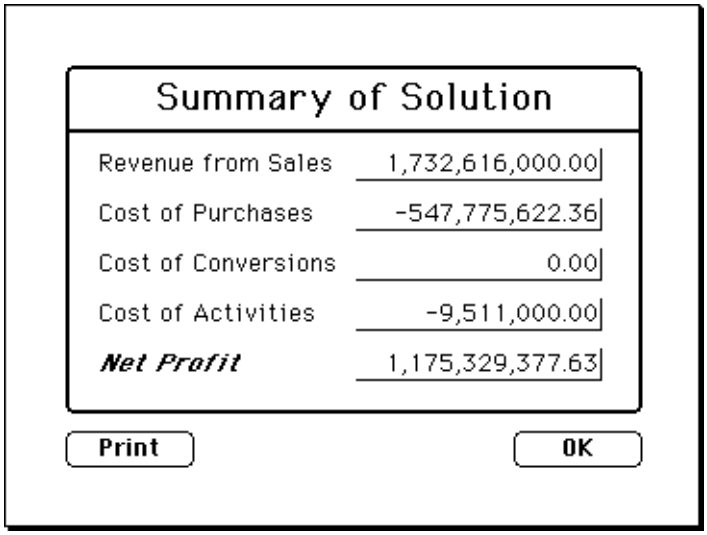


Figure 6–8. *A summary layout showing revenue and cost totals.*

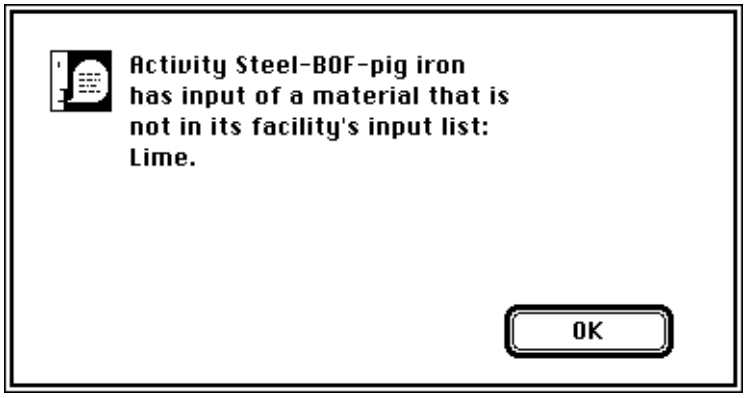


Figure 6–9. *A representative message produced by the optional diagnostic routines.*

mented in the 4th Dimension language. These tests scan the data for inconsistencies, such as materials that must be sold but that cannot be created or purchased in any way. The diagnostics are run only when the user requests them, through a menu selection; they produce brief messages identifying any problems found (Figure 6–9).

Most of the inconsistencies detected by our diagnostic tests would also be revealed by a general-purpose diagnostic tool such as ANALYZE [25, 26]. By detecting these errors though a scan of the database, before any linear program is generated, we can save the user some time and can provide error messages more closely tailored to the implementation. On the other hand, we have had to write and maintain some fairly intricate and specialized routines in the database language to carry out our diagnostics, whereas ANALYZE would apply equally well to any linear program that we happened to generate.

6.4 Updating

In our experiments with diverse steel-production scenarios, the effort of generating the constraints and variables has invariably dominated the effort of solving the resulting linear program; the timings reported by Table 6–6 are typical. This is not a hindrance initially, since both generation and solution seem fast compared to the work of entering all the database records. In typical continued use, however, the difference between one scenario and the next is often just a few quick changes to the data. A substantial part of the user's time thus ends up being spent in waiting for the generation stage to be completed.

Fortunately, when successive scenarios are quite similar, so are the successive linear programs. In most cases, changing one value in the database has the effect of changing just one lower bound, upper bound, or coefficient in the linear program.

To take advantage of this situation, we allow the user to switch to an updating mode. The layouts in this mode look almost the same as those exhibited previously, but they support only the simplest and most straightforward updating operations. For example, the materials update layout (Figure 6–10) allows changes to the costs, limits and yields, but not additions or deletions of records or changes to material names. When one of the allowed changes is made, a corresponding change is quickly made to the `VARIABLES` file, so that its representation is kept up to date.

When all updates are completed, it remains only to run the steps 3–5 defined in Section 6.2. Since step 2 is usually the most time-consuming, the savings afforded by updating are often considerable.

There are many other ways in which we might allocate the work of generating a

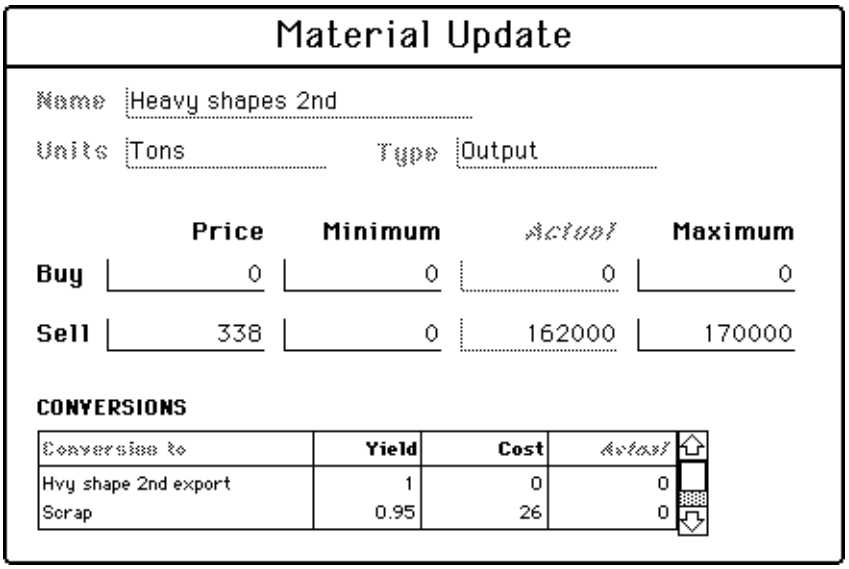


Figure 6–10. *An analogue of the layout in Figure 6–1, used for making quick modifications to data for materials.* Changes are reflected immediately in the `VARIABLES` file as well as in the `MATERIALS` file.

linear program. The following represent perhaps the two most extreme possibilities:

- The coefficients and bounds could be generated directly from the `MATERIALS` and `FACILITIES` files, without the intermediate creation of records in a `CONSTRAINTS` file or a `VARIABLES` file.
- The `CONSTRAINTS` and `VARIABLES` files could be automatically built up, step by step, along with the `MATERIALS` and `FACILITIES` files—so that data entry would always be in a powerful update mode that could handle changes of any kind.

We have adopted one of the many intermediates between these possibilities, in the hope of achieving a good compromise between ease of implementation, speed and reliability.

The influence of reliability on our design decision is perhaps surprising, but should not be underestimated. So long as the programs implementing steps 1 and 2 are subject to development and modification, displays of the `CONSTRAINTS` and `VARIABLES` files are a valuable tool for validation. If the output of step 3—essentially, a list of numbers—were instead generated directly from the the `MATERIALS` and `FACILITIES` files, our programs would be a lot harder to debug reliably. Hence even if direct generation were the preferred option, we might want to implement it only when further modifications were no longer anticipated.

7. Extensions and Generalizations

This paper's presentation has relied on examples motivated by one steelmaking application and implementation. The same ideas could readily be applied, however, in other applications to multi-facility production planning. More broadly, we have exhibited general principles that relate the customary algebraic description of linear programming data to standard relational and hierarchical structures for databases.

As a complement to Section 6, which has presented a particular implementation in detail, this section surveys a range of alternatives for the integration of database management and linear programming. We first consider varied ideas for the addition of linear programming features to database management systems, and for the incorporation of database features into modeling systems for linear programming. We then describe several promising options for intermediate levels of integration between database and LP software. In classifying integration strategies, we observe that much depends on how the modeling system's symbolic description of data is coordinated with the database management system's data scheme.

7.1 Linear programming in database management systems

More people are familiar with database management than with linear programming, and much of a planning model's data may already be maintained in corporate databases. Thus there is an argument to be made for implementing LP models entirely within the framework of a database system. Indeed, once the general structure of a model is decided upon, we can expect the users' attention to be fixed almost entirely on database activities such as entry of updated values and design of reports.

Popular database management systems support specialized programming languages and environments that can manipulate records and fields in almost any desired way. Thus, in particular, database systems can serve to implement "matrix generators" that write out LP data in the form required by an optimization algorithm, and that produce reports from the resulting optimal solution. The use of the dBASE II language for such a purpose has been described by Pasquier *et al.* [39]. The implementation described in Section 6 also has at its core a matrix generator, written in the 4th Dimension language.

Because the languages employed by database management systems were not designed with mathematical programming in mind, they lack high-level tools for purposes of communicating with an LP optimizer. The 4th Dimension programs that perform steps 1–3 and 5 described in the previous section, for example, rely upon functions such as SORT SELECTION, NEXT RECORD and SEARCH SUBRECORDS to individually extract the constraint coefficients or store solution values. These programs are reasonably short and easy to modify; an option for "soft capacities" at the facilities was added in an hour, for example, in response to one company's request at a meeting. Nevertheless, in the long run, such programs pose the same problems of understandability, verifiability and maintainability as other matrix generators [17]. One remedy for this difficulty is to extend a familiar database language with statements that specifically refer to variables, constraints, objectives and other constructs of optimization. Extensions to the widely-used SQL query language have been described by Choobineh [11], by Lenard [31], and by Savage and Baker [42].

7.2 Database management in linear programming systems

There is also an argument to be made for using software that has been specifically designed to support linear programming applications. People who build and maintain LP models are already comfortable with standard ways of representing them; the algebraic approach of Section 2 is just one example. A system that employs a standard and familiar LP representation has a built-in advantage, especially if changes often have to be made to the structure (not just the data) of the objective and constraints.

As we remarked at the outset of this paper, successful mathematical programming systems have always addressed the issue of data management. An ability to define data tables, to manipulate data, and to generate reports is built into commercial systems as diverse as AIMMS [5], AMPL [18, 19], DATAFORM [13, 38], GAMS [8], MathPro [27], MGG [43] and PAM [47]; Palmer [37] describes database management as a key aspect of the PLATOFORM system developed at Exxon. Nevertheless, mathematical programming remains the focus of these systems; the associated database features do not provide the rich variety of relational operations and application development tools that are commonplace in commercial systems intended for database management alone. As an example, the 4th Dimension package (employed by the Section 6 implementation) offers direct support for such concepts as many-to-one relationships, custom menus, and scrolling windows of subrecords, and provides a database design environment to help developers combine these elements into applications.

The data management features of mathematical programming systems have also been strongly influenced by their reliance upon sequential text files to store set members and numerical values. This approach has the advantage of allowing data to be entered and maintained using any text editor, whereas the data files created by an efficient database management system cannot be easily examined or changed except by further use of that system. The simplicity of a text file also imposes significant limitations, however, on how data may be conveniently formatted and displayed. Typically there is a separate table, or series of tables, for each indexed collection of values, with perhaps some limited provision for tabulating similarly indexed values together. By contrast, Section 6's database implementation lets the user enter and view in a single layout *all* the data relevant to one material, facility or activity. As an example, a text data file for the model of Appendix B would include a table of l^{act}_{ik} values, a table of u^{act}_{ik} values, a collection of tables giving the a^{in}_{ijk} values, and so forth; using our hierarchical database for the same model, the layout shown by Figure 6–4 gathers all of the data (l^{act}_{ik}, u^{act}_{ik}, c^{act}_{ik}, r^{act}_{ik}, $\mathcal{A}^{\text{in}}_{ik}$, a^{in}_{ijk}, $\mathcal{A}^{\text{out}}_{ik}$, a^{out}_{ijk}) relevant to one activity k.

Other, less traditional designs for mathematical programming systems place a stronger emphasis on the database aspects. Examples are found in the projects of Bonczek, Holsapple and Whinston [7], of Bürger [9], and of Stohr and Tanniru [44]. Among current commercial implementations, MIMI/LP [10, 12] is notable for allowing optimizers to request values directly from its database, so as to eliminate much of the usual work of translation between hierarchical or relational data structures and the optimizer's data format. This high level of integration is facilitated by

the design of MIMI's model description language, which maps blocks of constraint coefficients directly to tables in the database. Kristjansson *et al.* [30] describe a different approach based on MPL, an algebraic modeling language; they extend the language so that each declaration of an indexed collection of data may specify a relational database file from which the data values are to be retrieved.

7.3 Intermediate levels of integration

It is not easy to design or to implement a system that is ideal for both database management and linear programming. Even if such a system were available, moreover, it would tend to duplicate the facilities already available in dedicated database management and linear programming software. One is thus led to consider the alternative in which separate systems are linked. Various designs, differing in the degree and method of integration, can result from this approach; we consider first some general principles of integration, and then describe three specific possibilities.

Database systems make a distinction between the data scheme, which describes the structure of files and fields, and the data values that might be stored according to the scheme. In linear programming formulations there is a similar distinction between the abstract description of the data, and the actual values that specify a particular case to be solved. Several LP modeling languages, notably AMPL [18, 19] and SML [22, 23], are designed to support this distinction explicitly. They describe the form of the data in much the same way that we present it in Appendix A or B, while leaving the explicit data values to be provided in separate tables.

These considerations suggest a three-point division of responsibilities between database and linear programming systems:

- The abstract structure of the objective and constraints is read and processed by an LP modeling system.
- The actual data values are entered, changed and viewed through a database management system.
- The abstract structure of the sets, parameters and variables—that is, the data scheme—is specified by one or both of the two systems.

The logic of the first two points is evident; in each case, the work is given to the system that is designed to deal with it. The third point leaves several options, which lead to distinctly different modes of integration. We consider first the situation in which both systems maintain their own data schemes, and then the possibilities for one or the other system's scheme to take precedence.

Separate data schemes. The most loosely integrated option gives precedence to neither the LP nor the database system. Under such an arrangement,

- ▷ a data scheme is developed, and data values are stored and viewed, in the context of a database system;
- ▷ an analogous data scheme is created, along with a description of the abstract objective and constraint structure, in the context of an LP system;

- ▷ programs in the database language scan the data, and convert it to tables in the format required by the LP system;
- ▷ the LP system reads the tables together with the abstract model structure, and writes the LP matrix for the optimizing algorithm; and eventually,
- ▷ programs in the database language read the listing of optimal values, and store selected values for retrieval from the database.

An implementation along these lines is described by Kendrick, Krishnan and Carl-Mitchell [28]. The attractiveness of such an approach lies in the simplicity of the integration; if suitable database and LP systems are already available, then no further system development is necessary, aside from the writing of the necessary programs in the database language.

The price that is paid for this loose integration, however, is the lack of any automatic relationship between the data schemes employed by the two systems. For each different application, a different collection of programs must be written in the database language; and subsequently, any change to the structure of the data requires compensating changes to the data schemes and to the database programs. As a consequence, the development and maintenance of a reliable application can be difficult, even when the modeler is well versed in both database design and linear programming.

One possibility for a more tightly integrated arrangement, described by Bhargava, Krishnan and Mukherjee [3], introduces a third *embedding* language in which the correspondence between the LP system's data scheme and the database system's data scheme can be formally specified. Under most current approaches to tight integration, however, there is one data scheme that dominates. Changes to the structure of the data require only that changes be made to this one scheme. Naturally there are two possibilities, depending on whether the data scheme of the database system or of the LP system is the dominant one.

LP system dominant. If the linear programming system's data scheme takes precedence, then we are essentially automating the kind of analysis that was done in Sections 3 and 4. In this case,

- ▷ the abstract structure of the data, objective and constraints is developed in a form natural to LP modelers;
- ▷ a corresponding data scheme is *automatically* set up, in the context of a database system, to receive the actual data values;
- ▷ an LP system reads the values from the database together with the description of the abstract model structure, and writes the LP matrix for the optimizing algorithm; and eventually,
- ▷ the optimal values are automatically retrieved and stored in the appropriate fields within the database.

This option gives precedence to an abstract structure that is designed for modeling, not just for database management. Indeed, it does not require that the modeler

invest any work in database design, programming and maintenance. A good example is provided by aspects of the prototype system FW/SM [21, 36]. The abstract structure of FW/SM data is described by use of the SML language [22, 23], which embodies the principles that have been codified as *structured modeling* [20]. The database management component is supplied by Framework III, an integrated software system for MS-DOS computers.

This is an appealing arrangement, but it does confront some substantial design issues. Certainly, the class of schemes "natural to LP modelers" is too broad and vague to be supported in any literal way. We have already noted in Section 2, as an example, that modelers may want to define parameters that are indexed over a join of two indexing sets, whereas for purposes of database definition it is desirable to limit indexing to fundamental sets and subsets of their cartesian products. Bisschop and Kuip [6] describe several other instances of particularly intricate data indexing requirements that may be impossible to support in any conventional database. To address concerns in this area, the design of an LP system must reflect a judicious tradeoff: it must incorporate a data description that is restrictive enough to support automatic creation of corresponding database schemes, yet that is intuitive and general enough to allow for the models that people want to develop. Two contrasting approaches to the design of such a system may be identified.

A *descriptive* approach determines what is "natural to LP modelers" by examining the data representations that existing LP systems make available to users. The AMPL language [19], for instance, allows for indexing over subsets of cartesian products as in Section 3, and over indexed collections of subsets as in Section 4, but also offers indexing over sets defined in a variety of other ways: by unions, intersections and other set operations; by "slicing" or "projection" of multi-dimensional sets; and by selection of members that satisfy logical conditions. Since these features were provided (as explained in [18]) to satisfy people's perceptions that different models were expressed most naturally in different ways, it would make sense to consider how all of these features could be systematically mapped to the structures of a database management system. Such an effort would in effect address all of linear programming, rather than any particular class of applications as in Sections 2–4, and would represent a significant extension of present research.

A contrasting *prescriptive* approach seeks to design an LP system's data description conventions through reference to design principles of existing database systems. The LP system thus rules out certain common but dubious modeling constructs, such as representations that force a set member or number to be supplied in more than one place, or indexing sets whose membership is unnecessarily conditioned on the values of numerical parameters. The range of representations available to the modeler is narrowed as a result, but in a systematic way that promotes good modeling practice. This approach has been adopted in studies by Geoffrion [24] and Neustadter [35].

Under either approach, the linear programming system must be designed to cope with changes to the LP data scheme. Most seriously, if the LP data scheme is changed after many records have been created in the corresponding database, then some automatic adjustments to the data may be necessary to maintain consistency.

On the whole, the automated maintenance of the database is substantially more challenging to implement than the simpler, looser integration described previously.

Database system dominant. If instead the database's scheme takes precedence, then we are faced with a somewhat different arrangement:

- ▷ an appropriate data scheme for the modeling project is developed, and data values are stored and viewed, in the context of a database system;
- ▷ the abstract structure of the objective and constraints is expressed in a form natural to LP modelers, under the requirement that each reference to a parameter or variable must correspond to some field in the database;
- ▷ an LP system reads the values from the database together with the abstract objective and constraints, and writes the LP matrix for the optimizing algorithm; and eventually,
- ▷ the optimal values are automatically retrieved and stored in the appropriate fields within the database.

Although this alternative superficially resembles the loose integration described previously, it offers fundamentally stronger support for the connection between the database and LP systems. Creation of the database scheme automatically defines the set, parameter and variable references that will be recognized in the LP formulation. Data values can then be communicated automatically to the LP system, and optimal values can be communicated automatically back to the database; there are no application-specific table generating and result retrieving programs to be written, as in the loose integration approach.

The problems of coordination under this arrangement are likely to be less serious than under an arrangement in which the LP system dominates. In particular, changes to the data scheme are more easily accommodated, because both the scheme and its associated data values are maintained by the database system. Some coordination between the database and LP system is still necessary, but only to ensure consistency between the data scheme and the abstract model structure; the model cannot refer, say, to a set `MATERIALS` and to a parameter `buy_min` indexed over it, unless the database has a file `MATERIALS` in which there is a field `buy_min`. This sort of consistency needs to be enforced in any case, however, no matter how the model and data are represented.

A further advantage might be achieved by storing the objective and constraint descriptions within text fields of the database. The facilities of the database system might then be used to manage scenarios of both the model and the data. This would be a step toward the "model management" or "model base" systems described by Dolk [15] and Muhanna [33].

A significant hurdle remains, however, between the idea of this approach and its successful realization. Any implementation must incorporate a workable mapping from the fields of database schemes to the descriptions of sets, parameters and variables in linear programs—essentially the reverse of what we developed in Sections 3

and 4. The rules of the mapping must be theoretically sound, must be reasonably easy for a modeler to understand, and must be capable of automatic application by computer software. The case study in this paper can suggest many of the principles of a suitable mapping, but the details remain to be worked out in subsequent research.

Acknowledgements

Valuable comments on earlier versions of this paper were provided by A.M. Geoffrion, by reviewers for the 24th Hawaii International Conference on System Sciences, and by referees for *ORSA Journal on Computing*.

Appendix A.
Formulation Using Ordered Pairs and Triples

Materials data

$\mathcal{M}$ is the set of materials

l_j^{buy} = the lower limit on purchases of material j, for each $j \in \mathcal{M}$

u_j^{buy} = the upper limit on purchases of material j, for each $j \in \mathcal{M}$

c_j^{buy} = the cost per unit of material j purchased, for each $j \in \mathcal{M}$

l_j^{sell} = the lower limit on sales of material j, for each $j \in \mathcal{M}$

u_j^{sell} = the upper limit on sales of material j, for each $j \in \mathcal{M}$

c_j^{sell} = the revenue per unit of material j sold, for each $j \in \mathcal{M}$

$\mathcal{M}^{\text{conv}} \subseteq \mathcal{M} \times \mathcal{M}$ *is the set of conversions:*
$(j, j') \in \mathcal{M}^{\text{conv}}$ means that material j can be converted to material j'

$a_{jj'}^{\text{conv}}$ = number of units of material j' that result from converting one unit of material j, for each $(j, j') \in \mathcal{M}^{\text{conv}}$

$c_{jj'}^{\text{conv}}$ = cost per unit of material j of the conversion from j to j', for each $(j, j') \in \mathcal{M}^{\text{conv}}$

Facilities data

$\mathcal{F}$ is the set of facilities

l_i^{cap} = the minimum amount of the capacity of facility i that must be used, for each $i \in \mathcal{F}$

u_i^{cap} = the capacity of facility i, for each $i \in \mathcal{F}$

$\mathcal{F}^{\text{in}} \subseteq \mathcal{F} \times \mathcal{M}$ *is the set of facility inputs:*
$(i, j) \in \mathcal{F}^{\text{in}}$ means that material j is used as an input at facility i

l_{ij}^{in} = the minimum amount of material j that must be used as input at facility i, for each $(i, j) \in \mathcal{F}^{\text{in}}$

u_{ij}^{in} = the maximum amount of material j that may be used as input at facility i, for each $(i, j) \in \mathcal{F}^{\text{in}}$

$\mathcal{F}^{\text{out}} \subseteq \mathcal{F} \times \mathcal{M}$ *is the set of facility outputs:*
$(i, j) \in \mathcal{F}^{\text{out}}$ means that material j is produced as an output at facility i

l_{ij}^{out} = the minimum amount of material j that must be produced as output at facility i, for each $(i, j) \in \mathcal{F}^{\text{out}}$

u_{ij}^{out} = the maximum amount of material j that may be produced as output at facility i, for each $(i, j) \in \mathcal{F}^{\text{out}}$

Activities data

$\mathcal{F}^{\text{act}} \subseteq$ –$(i,k) : i \in \mathcal{F}''$ *is the set of activities:*
$(i,k) \in \mathcal{F}^{\text{act}}$ means that k is an activity available at facility i

l_{ik}^{act} = the minimum number of units of activity k that may be run at facility i, for each $(i,k) \in \mathcal{F}^{\text{act}}$

u_{ik}^{act} = the maximum number of units of activity k that may be run at facility i, for each $(i,k) \in \mathcal{F}^{\text{act}}$

c_{ik}^{act} = the cost per unit of running activity k at facility i, for each $(i,k) \in \mathcal{F}^{\text{act}}$

r_{ik}^{act} = the number of units of activity k that can be accommodated in one unit of capacity of facility i, for each $(i,k) \in \mathcal{F}^{\text{act}}$

$\mathcal{A}^{\text{in}} \subseteq$ –$(i,j,k) : (i,j) \in \mathcal{F}^{\text{in}}$ and $(i,k) \in \mathcal{F}^{\text{act}\,\prime\prime}$ *is the set of activity inputs:*
$(i,j,k) \in \mathcal{A}^{\text{in}}$ means that input material j is used by activity k at facility i

a_{ijk}^{in} = units of input material j required by one unit of activity k at facility i, for each $(i,j,k) \in \mathcal{A}^{\text{in}}$

$\mathcal{A}^{\text{out}} \subseteq$ –$(i,j,k) : (i,j) \in \mathcal{F}^{\text{out}}$ and $(i,k) \in \mathcal{F}^{\text{act}\,\prime\prime}$ *is the set of activity outputs:*
$(i,j,k) \in \mathcal{A}^{\text{out}}$ means that output material j is produced by activity k at facility i

a_{ijk}^{out} = units of output material j produced by one unit of activity k at facility i, for each $(i,j,k) \in \mathcal{A}^{\text{out}}$

Variables

x_j^{buy} = units of material j bought, for each $j \in \mathcal{M}$

x_j^{sell} = units of material j sold, for each $j \in \mathcal{M}$

$x_{jj'}^{\text{conv}}$ = units of material j converted to material j', for each $(j,j') \in \mathcal{M}^{\text{conv}}$

x_{ij}^{in} = units of material j used as input by facility i, for each $(i,j) \in \mathcal{F}^{\text{in}}$

x_{ij}^{out} = units of material j produced as output by facility i, for each $(i,j) \in \mathcal{F}^{\text{out}}$

x_{ik}^{act} = units of activity k operated at facility i, for each $(i,k) \in \mathcal{F}^{\text{act}}$

Objective

Maximize revenue from sales, less the costs of purchasing, converting and running activities:

$$\sum_{j\in\mathcal{M}} c_j^{\text{sell}} x_j^{\text{sell}} - \sum_{j\in\mathcal{M}} c_j^{\text{buy}} x_j^{\text{buy}} - \sum_{(j,j')\in\mathcal{M}^{\text{conv}}} c_{jj'}^{\text{conv}} x_{jj'}^{\text{conv}} - \sum_{(i,k)\in\mathcal{F}^{\text{act}}} c_{ik}^{\text{act}} x_{ik}^{\text{act}}$$

Constraints

For all $j \in \mathcal{M}$, the amount of material j made available by purchases, production and conversions must equal the amount used for sales, production and conversions:

$$x_j^{\text{buy}} + \sum_{(i,j)\in\mathcal{F}^{\text{out}}} x_{ij}^{\text{out}} + \sum_{(j',j)\in\mathcal{M}^{\text{conv}}} a_{j'j}^{\text{conv}} x_{j'j}^{\text{conv}} = x_j^{\text{sell}} + \sum_{(i,j)\in\mathcal{F}^{\text{in}}} x_{ij}^{\text{in}} + \sum_{(j,j')\in\mathcal{M}^{\text{conv}}} x_{jj'}^{\text{conv}}$$

For each $(i,j) \in \mathcal{F}^{\text{in}}$, the amount of input j used at facility i must equal the total consumption by all activities at facility i:

$$x_{ij}^{\text{in}} = \sum_{(i,j,k)\in\mathcal{A}^{\text{in}}} a_{ijk}^{\text{in}} x_{ik}^{\text{act}}$$

For each $(i,j) \in \mathcal{F}^{\text{out}}$, the amount of output j produced at facility i must equal the total production by all activities at facility i:

$$x_{ij}^{\text{out}} = \sum_{(i,j,k)\in\mathcal{A}^{\text{out}}} a_{ijk}^{\text{out}} x_{ik}^{\text{act}}$$

For each $i \in \mathcal{F}$, the capacity used by all activities at facility i must be within the specified limits:

$$l_i^{\text{cap}} \le \sum_{(i,k)\in\mathcal{F}^{\text{act}}} x_{ik}^{\text{act}} / r_{ik}^{\text{act}} \le u_i^{\text{cap}}$$

All variables must lie within the relevant limits defined by the data:

$$\begin{aligned}
&l_j^{\text{buy}} \le x_j^{\text{buy}} \le u_j^{\text{buy}}, && \text{for each } j \in \mathcal{M}\\
&l_j^{\text{sell}} \le x_j^{\text{sell}} \le u_j^{\text{sell}}, && \text{for each } j \in \mathcal{M}\\
&0 \le x_{jj'}^{\text{conv}}, && \text{for each } (j,j') \in \mathcal{M}^{\text{conv}}\\
&l_{ij}^{\text{in}} \le x_{ij}^{\text{in}} \le u_{ij}^{\text{in}}, && \text{for each } (i,j) \in \mathcal{F}^{\text{in}}\\
&l_{ij}^{\text{out}} \le x_{ij}^{\text{out}} \le u_{ij}^{\text{out}}, && \text{for each } (i,j) \in \mathcal{F}^{\text{out}}\\
&l_{ik}^{\text{act}} \le x_{ik}^{\text{act}} \le u_{ik}^{\text{act}}, && \text{for each } (i,k) \in \mathcal{F}^{\text{act}}
\end{aligned}$$

Appendix B.
Formulation Using Indexed Subsets

Materials data

$\mathcal{M}$ is the set of materials

l_j^{buy} = the lower limit on purchases of material j, for each $j \in \mathcal{M}$

u_j^{buy} = the upper limit on purchases of material j, for each $j \in \mathcal{M}$

c_j^{buy} = the cost per unit of material j purchased, for each $j \in \mathcal{M}$

l_j^{sell} = the lower limit on sales of material j, for each $j \in \mathcal{M}$

u_j^{sell} = the upper limit on sales of material j, for each $j \in \mathcal{M}$

c_j^{sell} = the revenue per unit of material j sold, for each $j \in \mathcal{M}$

$\mathcal{M}_j^{\text{conv}} \subseteq \mathcal{M}$ *is a subset of conversions, for each* $j \in \mathcal{M}$*:*
$j' \in \mathcal{M}_j^{\text{conv}}$ means that material j can be converted to material j'

$a_{jj'}^{\text{conv}}$ = number of units of material j' that result from converting one unit of material j, for each $j \in \mathcal{M}$ and $j' \in \mathcal{M}_j^{\text{conv}}$

$c_{jj'}^{\text{conv}}$ = cost per unit of material j of the conversion from j to j', for each $j \in \mathcal{M}$ and $j' \in \mathcal{M}_j^{\text{conv}}$

Facilities data

$\mathcal{F}$ is the set of facilities

l_i^{cap} = the minimum amount of the capacity of facility i that must be used, for each $i \in \mathcal{F}$

u_i^{cap} = the capacity of facility i, for each $i \in \mathcal{F}$

$\mathcal{F}_i^{\text{in}} \subseteq \mathcal{M}$ *is a subset of facility inputs, for each* $i \in \mathcal{F}$*:*
$j \in \mathcal{F}_i^{\text{in}}$ means that material j is used as an input at facility i

l_{ij}^{in} = the minimum amount of material j that must be used as input at facility i, for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{in}}$

u_{ij}^{in} = the maximum amount of material j that may be used as input at facility i, for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{in}}$

$\mathcal{F}_i^{\text{out}} \subseteq \mathcal{M}$ *is a subset of facility outputs, for each* $i \in \mathcal{F}$*:*
$j \in \mathcal{F}_i^{\text{out}}$ means that material j is produced as an output at facility i

l_{ij}^{out} = the minimum amount of material j that must be produced as output at facility i, for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{out}}$

u_{ij}^{out} = the maximum amount of material j that may be produced as output at facility i, for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{out}}$

Activities data

$\mathcal{F}_i^{\text{act}}$ *is a subset of activities, for each* $i \in \mathcal{F}$*:*
$k \in \mathcal{F}_i^{\text{act}}$ means that k is an activity available at facility i

l_{ik}^{act} = the minimum number of units of activity k that may be run at facility i, for each $i \in \mathcal{F}$ and $k \in \mathcal{F}_i^{\text{act}}$

u_{ik}^{act} = the maximum number of units of activity k that may be run at facility i, for each $i \in \mathcal{F}$ and $k \in \mathcal{F}_i^{\text{act}}$

c_{ik}^{act} = the cost per unit of running activity k at facility i, for each $i \in \mathcal{F}$ and $k \in \mathcal{F}_i^{\text{act}}$

r_{ik}^{act} = the number of units of activity k that can be accommodated in one unit of capacity of facility i, for each $i \in \mathcal{F}$ and $k \in \mathcal{F}_i^{\text{act}}$

$\mathcal{A}_{ik}^{\text{in}} \subseteq \mathcal{F}_i^{\text{in}}$ *is a set of activity inputs, for each* $i \in \mathcal{F}$ *and* $k \in \mathcal{F}_i^{\text{act}}$*:*
$j \in \mathcal{A}_{ik}^{\text{in}}$ means that input material j is used by activity k at facility i

a_{ijk}^{in} = units of input material j required by one unit of activity k at facility i, for each $i \in \mathcal{F}$, $k \in \mathcal{F}_i^{\text{act}}$, and $j \in \mathcal{A}_{ik}^{\text{in}}$

$\mathcal{A}_{ik}^{\text{out}} \subseteq \mathcal{F}_i^{\text{out}}$ *is a set of activity outputs, for each* $i \in \mathcal{F}$ *and* $k \in \mathcal{F}_i^{\text{act}}$*:*
$j \in \mathcal{A}_{ik}^{\text{out}}$ means that output material j is produced by activity k at facility i

a_{ijk}^{out} = units of output material j produced by one unit of activity k at facility i, for each $i \in \mathcal{F}$, $k \in \mathcal{F}_i^{\text{act}}$, and $j \in \mathcal{A}_{ik}^{\text{out}}$

Variables

x_j^{buy} = units of material j bought, for each $j \in \mathcal{M}$

x_j^{sell} = units of material j sold, for each $j \in \mathcal{M}$

$x_{jj'}^{\text{conv}}$ = units of material j converted to material j', for each $j \in \mathcal{M}$ and $j' \in \mathcal{M}_j^{\text{conv}}$

x_{ij}^{in} = units of material j used as input by facility i, for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{in}}$

x_{ij}^{out} = units of material j produced as output by facility i, for each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{out}}$

x_{ik}^{act} = units of activity k operated at facility i, for each $i \in \mathcal{F}$ and $k \in \mathcal{F}_i^{\text{act}}$

Objective

Maximize revenue from sales, less the costs of purchasing, converting and running activities:

$$\sum_{j\in\mathcal{M}} c_j^{\text{sell}} x_j^{\text{sell}} - \sum_{j\in\mathcal{M}} c_j^{\text{buy}} x_j^{\text{buy}} - \sum_{j\in\mathcal{M}} \sum_{j'\in\mathcal{M}_j^{\text{conv}}} c_{jj'}^{\text{conv}} x_{jj'}^{\text{conv}} - \sum_{i\in\mathcal{F}} \sum_{k\in\mathcal{F}_i^{\text{act}}} c_{ik}^{\text{act}} x_{ik}^{\text{act}}$$

Constraints

For all $j \in \mathcal{M}$, the amount of material j made available by purchases, production and conversions must equal the amount used for sales, production and conversions:

$$x_j^{\text{buy}} + \sum_{i\in\mathcal{F}:j\in\mathcal{F}_i^{\text{out}}} x_{ij}^{\text{out}} + \sum_{j'\in\mathcal{M}:j\in\mathcal{M}_{j'}^{\text{conv}}} a_{j'j}^{\text{conv}} x_{j'j}^{\text{conv}} = x_j^{\text{sell}} + \sum_{i\in\mathcal{F}:j\in\mathcal{F}_i^{\text{in}}} x_{ij}^{\text{in}} + \sum_{j'\in\mathcal{M}_j^{\text{conv}}} x_{jj'}^{\text{conv}}$$

For each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{in}}$, the amount of input j used at facility i must equal the total consumption by all activities at facility i:

$$x_{ij}^{\text{in}} = \sum_{k\in\mathcal{F}_i^{\text{act}}:j\in\mathcal{A}_{ik}^{\text{in}}} a_{ijk}^{\text{in}} x_{ik}^{\text{act}}$$

For each $i \in \mathcal{F}$ and $j \in \mathcal{F}_i^{\text{out}}$, the amount of output j produced at facility i must equal the total production by all activities at facility i:

$$x_{ij}^{\text{out}} = \sum_{k\in\mathcal{F}_i^{\text{act}}:j\in\mathcal{A}_{ik}^{\text{out}}} a_{ijk}^{\text{out}} x_{ik}^{\text{act}}$$

For each $i \in \mathcal{F}$, the capacity used by all activities at facility i must be within the specified limits:

$$l_i^{\text{cap}} \le \sum_{k\in\mathcal{F}_i^{\text{act}}} x_{ik}^{\text{act}} / r_{ik}^{\text{act}} \le u_i^{\text{cap}}$$

All variables must lie within the relevant limits defined by the data:

$$\begin{aligned}
&l_j^{\text{buy}} \le x_j^{\text{buy}} \le u_j^{\text{buy}}, && \text{for each } j \in \mathcal{M}\\
&l_j^{\text{sell}} \le x_j^{\text{sell}} \le u_j^{\text{sell}}, && \text{for each } j \in \mathcal{M}\\
&0 \le x_{jj'}^{\text{conv}}, && \text{for each } j \in \mathcal{M} \text{ and } j' \in \mathcal{M}_j^{\text{conv}}\\
&l_{ij}^{\text{in}} \le x_{ij}^{\text{in}} \le u_{ij}^{\text{in}}, && \text{for each } i \in \mathcal{F} \text{ and } j \in \mathcal{F}_i^{\text{in}}\\
&l_{ij}^{\text{out}} \le x_{ij}^{\text{out}} \le u_{ij}^{\text{out}}, && \text{for each } i \in \mathcal{F} \text{ and } j \in \mathcal{F}_i^{\text{out}}\\
&l_{ik}^{\text{act}} \le x_{ik}^{\text{act}} \le u_{ik}^{\text{act}}, && \text{for each } i \in \mathcal{F} \text{ and } k \in \mathcal{F}_i^{\text{act}}
\end{aligned}$$

Appendix C.
Summary of Database Schemes

General model: Relational

CONSTRAINTS
row_name
row_min
row_max

VARIABLES
col_name
col_profit
col_min
col_optimal
col_max

COEFFICIENTS
coeff_row → CONSTRAINTS
coeff_col → VARIABLES
coeff_value

General model: Hierarchical

CONSTRAINTS
row_name
row_min
row_max

VARIABLES
col_name
col_profit
col_min
col_optimal
col_max

COEFFICIENTS
coeff_row → CONSTRAINTS
coeff_value

Multi-facility production model: Relational

MATERIALS
mat_name
buy_min
buy_opt
buy_max
buy_cost
sell_min
sell_opt
sell_max
sell_cost

FACILITIES
fac_name
cap_min
cap_max

MATERIAL_CONVERSIONS
from_mat → MATERIALS
to_mat → MATERIALS
conv_yield
conv_cost
conv_opt

FACILITY_INPUTS
in_fac → FACILITIES
in_mat → MATERIALS
in_min
in_opt
in_max

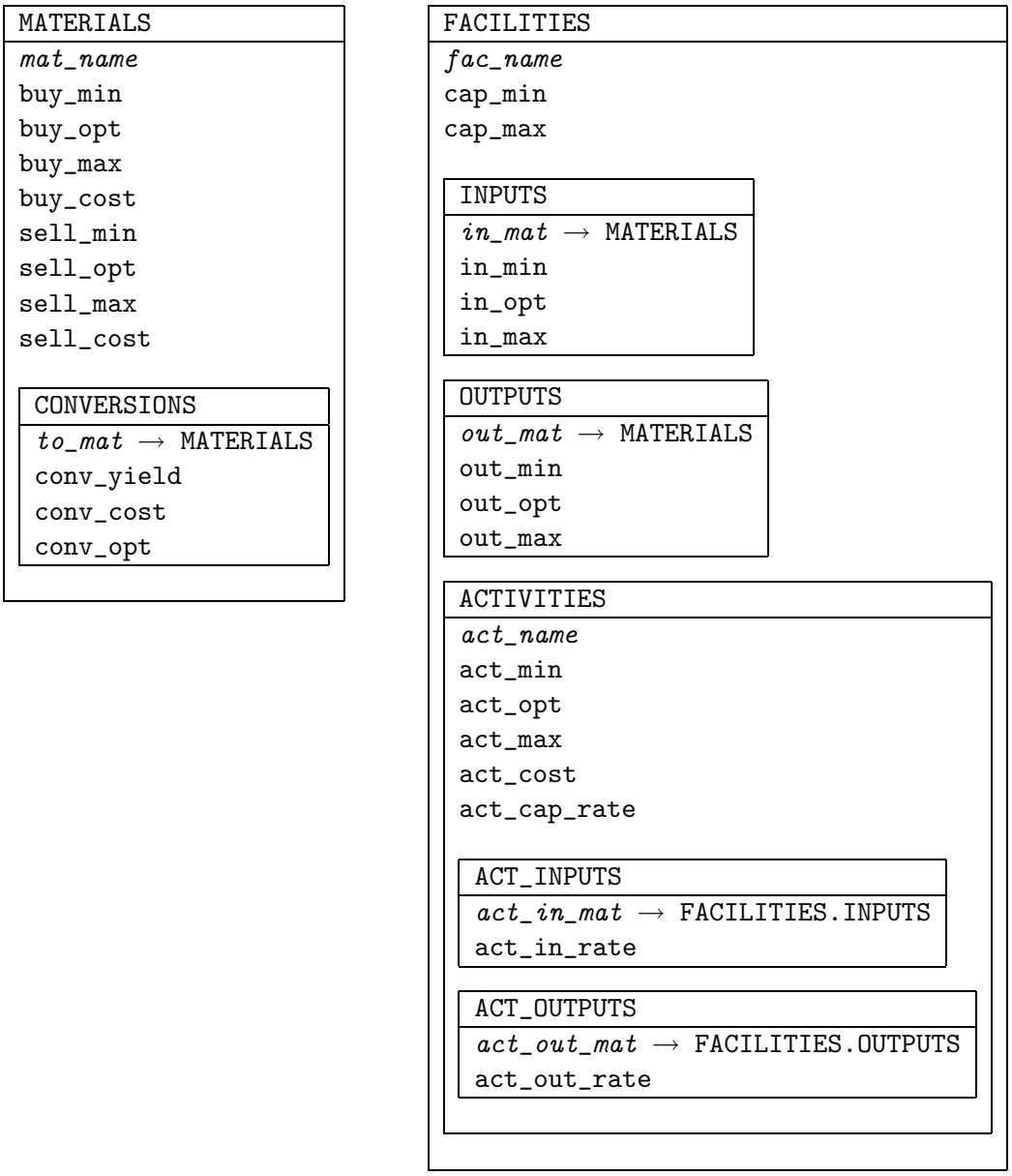
FACILITY_OUTPUTS
out_fac → FACILITIES
out_mat → MATERIALS
out_min
out_opt
out_max

ACTIVITIES
act_fac → FACILITIES
act_name
act_min
act_opt
act_max
act_cost
act_cap_rate

ACTIVITY_INPUTS
act_in_fac → FACILITIES
act_in_mat → MATERIALS
act_in
act_in_rate

ACTIVITY_OUTPUTS
act_out_fac → FACILITIES
act_out_mat → MATERIALS
act_out
act_out_rate

Multi-facility production model: Hierarchical



References

[1] T.E. Baker, 1986. A Hierarchical/Relational Approach to Modeling. Chesapeake Decision Sciences, New Providence, NJ.

[2] E.M.L. Beale, 1970. Matrix Generators and Output Analyzers. In H.W. Kuhn (ed.), *Proceedings of the Princeton Symposium on Mathematical Programming,* Princeton University Press, Princeton, NJ, pp. 25–36.

[3] H.K. Bhargava, R. Krishnan and S. Mukherjee, 1992. On the Integration of Data and Mathematical Modeling Languages. *Annals of Operations Research* **38,** 69–95.

[4] F.W. Bielefeld, K.-D. Walter and R. Wartmann, 1986. A Computer-Based Strategic Planning System for Steel Production. *Interfaces* **16:**4, 41–46.

[5] J. Bisschop and R. Entriken, 1993. *AIMMS: The Modeling System.* Paragon Decision Technology, Haarlem, The Netherlands.

[6] J.J. Bisschop and C.A.C. Kuip, 1993. Compound Sets in Mathematical Programming Modeling Languages. *Management Science* **39,** 746–756.

[7] R. Bonczek, C. Holsapple and A. Whinston, 1978. Mathematical Programming Within the Context of a Generalized Data Base Management System. *R.A.I.R.O. Recherche Opérationnelle/Operations Research* **12,** 117–139.

[8] A. Brooke, D. Kendrick and A. Meeraus, 1992. *GAMS: A User's Guide, Release 2.25.* The Scientific Press, South San Francisco, CA.

[9] W.F. Bürger, 1982. MLD: A Language and Data Base for Modeling. Research Report RC 9639, IBM Research Division, Yorktown Heights, NY.

[10] Chesapeake Decision Sciences, 1993. *Manager for Interactive Modeling Interfaces: MIMI User Manual,* version 3.40. Chesapeake Decision Sciences, New Providence, NJ.

[11] J. Choobineh, 1991. SQLMP: A Data Sublanguage for Representation and Formulation of Linear Mathematical Models. *ORSA Journal on Computing* **3,** 358–375.

[12] G.W. Cleaves and T.E. Baker, 1990. Chesapeake R&D Sponsor Groups. *Interfaces* **20:**6, 83–87.

[13] J.B. Creegan, 1985. DATAFORM, a Model Management System. Ketron, Inc., Arlington, VA.

[14] C.J. Date, 1990. *An Introduction to Database Systems.* Addison-Wesley Publishing Company, Reading, MA.

[15] D.R. Dolk, 1986. A Generalized Model Management System for Mathematical Programming. *ACM Transactions on Mathematical Software* **12,** 92–126.

[16] T. Fabian, 1958. A Linear Programming Model of Integrated Iron and Steel Production. *Management Science* **4,** 415–449.

[17] R. Fourer, 1983. Modeling Languages versus Matrix Generators for Linear Programming. *ACM Transactions on Mathematical Software* **9,** 143–183.

[18] R. Fourer, D.M. Gay and B.W. Kernighan, 1990. A Modeling Language for Mathematical Programming. *Management Science* **36,** 519–554.

[19] R. Fourer, D.M. Gay and B.W. Kernighan, 1992. *AMPL: A Modeling Language for Mathematical Programming.* Scientific Press, South San Francisco, CA.

[20] A.M. Geoffrion, 1987. An Introduction to Structured Modeling. *Management Science* **33,** 547–588.

[21] A.M. Geoffrion, 1991. FW/SM: A Prototype Structured Modeling Environment. *Management Science* **37,** 1513–1538.

[22] A.M. Geoffrion, 1992. The SML Language for Structured Modeling: Levels 1 and 2. *Operations Research* **40,** 38–57.

[23] A.M. Geoffrion, 1992. The SML Language for Structured Modeling: Levels 3 and 4. *Operations Research* **40,** 58–75.

[24] A.M. Geoffrion, 1992. Indexing in Modeling Languages for Mathematical Programming. *Management Science* **38,** 325–344.

[25] H.J. Greenberg, 1993. Enhancements to ANALYZE: A Computer-Assisted Analysis System for Linear Programming. *ACM Transactions on Mathematical Software* **19,** 233–256.

[26] H.J. Greenberg, 1993. *A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide for ANALYZE.* Kluwer Academic Publishers, Boston.

[27] D.S. Hirshfeld, 1993. Scenario Management, with Illustrations Using MathPro. IMPS Roundtable Discussion #14: Integration of Modeling, Optimization and Analysis, University of Colorado, Denver.

[28] D. Kendrick, R. Krishnan and S. Carl-Mitchell, 1984. Interfaces Between Database and Modeling Systems. Paper no. 84-12, Center for Economic Research, Department of Economics, University of Texas, Austin, TX.

[29] D.A. Kendrick, A. Meeraus and J. Alatorre, 1984. *The Planning of Investment Programs in the Steel Industry.* Johns Hopkins University Press, Baltimore.

[30] B. Kristjansson, C. Lucas, G. Mitra and S. Moody, 1993. Sets and Indices in Linear Programming Modelling and their Integration with Relational Data Models. Technical report, Maximal Software, Iceland, and Brunel University, Uxbridge, Middlesex, United Kingdom.

[31] M.L. Lenard, 1990. A Data-Model-Solver Interface Written in SQL. MD21.2, program of the 29th TIMS/ORSA Joint National Meeting, Las Vegas.

[32] R.E. Marsten, 1981. The Design of the XMP Linear Programming Library. *ACM Transactions on Mathematical Software* **7,** 481–497.

[33] W.A. Muhanna, 1992. On the Organization of Large Shared Model Bases. *Annals of Operations Research* **38,** 359–396.

[34] H. Müller-Merbach, 1990. Database-Oriented Design of Planning Models. *IMA Journal of Mathematics Applied in Business and Industry* **2,** 141–155.

[35] L. Neustadter, 1989. Value-Driven Sets in Modeling Languages: An Analysis. Anderson Graduate School of Management, University of California, Los Angeles.

[36] L. Neustadter, A. Geoffrion, S. Maturana, Y. Tsai and F. Vicuña, 1990. The Design and Implementation of a Prototype Structured Modeling Environment. Working Paper No. 380, Western Management Science Institute, University of California, Los Angeles.

[37] K.H. Palmer, 1984. Data and File Structure. In K.H. Palmer *et al.,* eds., *A Model-Management Framework for Mathematical Programming,* John Wiley & Sons, New York, pp. 63–95.

[38] K.H. Palmer, N.K. Boudwin, H.A. Patton, A.J. Rowland, J.D. Sammes and D.M. Smith, 1984. *A Model-Management Framework for Mathematical Programming.* John Wiley & Sons, New York.

[39] J. Pasquier, P. Hättenschwiler, T. Hürlimann and B. Sudan, 1985. A Convenient Technique for Constructing Your Own MPSX Generator Using dBASE II. Institute for Automation and Operations Research, University of Fribourg, Switzerland.

[40] G. Perlman, 1993. *Inside 4th Dimension.* Sybex, San Francisco.

[41] M.A. Roth, H.F. Korth and A. Silberschatz, 1988. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems* **13,** 389–417.

[42] S.L. Savage and M.D. Baker, 1990. Mathematical Modeling Databases. MD21.1, program of the 29th TIMS/ORSA Joint National Meeting, Las Vegas.

[43] R.V. Simons, 1987. Mathematical Programming Modeling Using MGG. *IMA Journal of Mathematics in Management* **1,** 267–276.

[44] E.A. Stohr and M.R. Tanniru, 1980. A Database for Operations Research Models. *International Journal of Policy Analysis and Information Systems* **4,** 105–121.

[45] J.D. Ullman, 1988. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, Rockville, MD.

[46] J.S. Welch, Jr., 1987. The Data Management Needs of Mathematical Programming Applications. *IMA Journal of Mathematics in Management* **1,** 237–250.

[47] J.S. Welch, Jr., 1987. PAM—A Practitioner's Approach to Modeling. *Management Science* **33,** 610–625.