OPERATION OF A RELATIONAL ACCOUNTING SYSTEM

Graham Gal and William E. McCarthy

ABSTRACT

This paper describes the procedures needed to maintain a relational accounting information system and to retrieve from it normal types of financial reporting data in a computerized environment. Elementary concepts of relational databases are presented first, followed by a description of the Query-by-Example (QBE) system. QBE procedures needed to derive a general ledger from disaggregate records of economic resources, events, and agents for a sample company are illustrated and explained.

Following the description of general ledger processing via the QBE system, the paper discusses the difficulties inherent in the use of set-oriented languages with accounting data. These difficulties include, for example, problems with accounting conventions such as LIFO and FIFO inventory costing. The paper concludes with a list of recommendations for accountants who are considering the use of relational database management systems.

INTRODUCTION

In recent years, relational database management systems (DBMS) have become increasingly available on a wide variety of both large mainframe

Advances in Accounting, Volume 3, pages 83–112.
Copyright © 1986 JAI Press Inc.
All rights of reproduction in any form reserved.
ISBN: 0-89232-655-7.

computers and smaller microcomputers. This widespread commercial availability follows a decade or more of scientific research which has consistently demonstrated the conceptual primacy of the relational model in the design and maintenance of multipurpose data management systems.

The arguments for relational systems are discussed by Date [1981, Chap. 28], and they are both theoretical and practical in nature. On the theory side [Codd, 1970], relational systems have not only a sound basis in the mathematical theory of relations but also a well-developed body of computer science research which supplies formal criteria both for evaluation of database designs (the concept of normalization) and for determination of sufficiency for a database system's language features (the concept of relational completeness). Aspects of these formal theories are introduced in the context of managerial and financial accounting systems by Everest and Weber [1977].

On the practical side, the advantages of relational DBMS can be characterized quite succinctly: these systems are simple and easy to use. Data is displayed and manipulated in simple tabular form, and the number of new constructs to be learned by a novice user is relatively small. Indeed, while the theoretical considerations mentioned above will always remain important, it is this simplicity which has made relational systems especially appealing to the growing number of smaller computer users, a market segment which accounts for most of the increase in their commercial use [Kruglinski, 1983].

In this paper, we describe an implementation of a relational accounting system with a particular emphasis on its *operational* or *procedural* characteristics. Our intent here is twofold: (1) to demonstrate the simplicity of relational procedures and to show how such procedures can be used to construct an *events accounting system* [McCarthy, 1981; Sorter, 1969] and (2) to identify distinct limitations of *pure* relational implementations and to suggest alternatives which might need to be considered in future accounting uses of these systems. As a vehicle for explanation of our operations, we use the data for a simple retail enterprise over the course of one month. The relational DBMS used in our actual implementation was the commercial version of Zloof's Query-by-Example [Zloof, 1975; IBM, 1980], a system whose simplicity and ease of use has been demonstrated in an experimental setting by Greenblatt and Waxman [1978].

This paper is organized as follows. We first describe some fundamental concepts of relational systems with two simple examples, and we use these examples to describe the procedural mechanics of Query-by-Example (QBE). This introduction is followed by two sections which describe our use of QBE procedures and which overview the hierarchic structure of the operations we perform. This overview is followed by a more detailed description of the derivation of successively more aggregate accounting values such as an accounts-payable balance, a liabilities total, and a trial balance total. The paper's final section discusses limitations in our implementation, including the problems engendered by the sole use of specification operations [Tsichritzis and Lochovsky, 1982, pp. 74–82]. This last part also explains how elements of the REA accounting model [McCarthy, 1982] relate to this system and why we found it necessary to extend the basic QBE operational facilities to perform needed accounting calculations.

RELATIONAL DATABASES AND QUERY-BY-EXAMPLE

Query-by-Example derives its name from the method of retrieval it uses in selecting data from a relational system. Relational databases are presented to a user in tabular form (similar, for instance, to a tax table or a present value table), and QBE performs operations by:

1. having users compose an answer to their questions substituting "example" elements for columns with unknown variables, and
2. using those answers to link separate tables together and extract data.

In the next two subsections of the paper, we explain some simple relational concepts, and we demonstrate how a language like QBE can be used to retrieve data from tables.

QBE With a Single Table

Figure 1 illustrates a table (also called a relation) used to represent information about inventory items in a company's database. The name of this particular table is given on the top left as "INVENTORY." The column headings, such as "STOCK#" and "DESCRIPTION," represent attributes of inventory items, while the rows (also called tuples) in the body of the table represent actual values for each of 12 different inventory types. The boldface column "STOCK#" is this table's key attribute. Relational theory requires that each table represent just one concept, and a key attribute is one whose values uniquely identify all occurrences of that concept. In Figure 1, there are 12 different types of inventory; hence there are 12 rows or tuples, each with a unique value in the boldface column. For the sake of contrast with this small example, one could envision a table representing the employees of a large factory. Such a table might have 5,000 or more rows. Each row would represent an individual employee, and it probably would be keyed on that employee's social security number.

Table names and column headings constitute the *intension* or the type definitions of a certain database. Thus, "INVENTORY" is the name of a

INVENTORY	**STOCK #**	DESCRIPTION	PRICE	QOH	COLOR
	8555	TEDDY BEAR	$15.00	0	GREEN
	5510	MARBLE	.23	786	MIX
	1187	BALL	3.75	108	WHITE
	2751	FRISBEE	4.00	15	RED
	2456	TEDDY BEAR	15.00	15	YELLOW
	4310	PUPPET	8.00	6	BLUE
	5332	BALL	.50	190	RED
	8400	PUPPET	8.50	0	BLUE
	3721	TOPS	.60	84	RED
	5130	TEDDY BEAR	11.00	25	BROWN
	4563	RATTLE	.65	100	WHITE
	6964	FRISBEE	4.25	11	BLACK

Figure 1. Database Table (Relation) Representing Inventory Items.

concept of interest in this company, and "STOCK#," "DESCRIPTION," etc., are that concept's attributes of interest. The *extension* of a database consists of its occurrence definitions or its actual row values. The extension of the table portrayed in Figure 1 consists of its 12 rows or tuples.

Figure 2 illustrates how Query-by-Example can be used to extract tabular data. QBE users begin their query process by asking to see table headings for a concept in which they are interested. If the inventory example of Figure 1 were stored in a QBE database, such a request would be performed by titling a blank or skeleton table with the name "INVENTORY," and QBE would respond with the intensional features shown in Figure 2(a).

Once given a table's intension, users specify which rows are to be retrieved by typing in an example answer. This example answer uses *constant* elements for known attributes and *example* elements for unknown attributes. Example elements are typed in prefixed by a single underscore character, and they are intended to represent a typical instance of a column's value (for instance, "BANANA" might be put under a "FRUIT" column). Constant elements are simply typed in unadorned.

Figure 2(b) shows the QBE operators that a customer service clerk would fill into the "INVENTORY" table to find out which teddy bears are in

INVENTORY	**STOCK #**	DESCRIPTION	PRICE	QOH	COLOR

Figure 2 (a). Inventory Intension.

INVENTORY	**STOCK #**	DESCRIPTION	PRICE	QOH	COLOR
	P. _5555	P. TEDDY BEAR		P. > 0	P. _PINK

Figure 2 (b). Query for In-Stock Teddy Bears.

INVENTORY	**STOCK #**	DESCRIPTION	QOH	COLOR
	2456	TEDDY BEAR	15	YELLOW
	5130	TEDDY BEAR	25	BROWN

Figure 2 (c). Results of the QBE Operation.

stock. This query ranges over the table's entire extension, and it returns tuples meeting two conditions: (1) rows whose description value is "TEDDY BEAR" and (2) rows whose QOH value is ">0" (greater than zero). This clerk also wishes to see the "STOCK#" and "COLOR" columns for in-stock teddy bears, but not their "PRICE" values. This last condition is specified by filling in example elements (which again are prefixed by an underscore character) in the first and fifth columns and by leaving the third column blank. The "P." or print operator specifies which columns are to be used in displaying the answer.

Figure 2(c) illustrates the result that would be displayed when the clerk's query is executed. A comparison of this output with Figure 1 illustrates that operations have been performed which (1) reduced the number of rows from twelve to two and (2) reduced the number of columns from five to four.[1]

The full data tables, such as the one shown in Figure 1, which are actually defined and stored in a QBE database are called *base* tables or relations. They are the fundamental building blocks from which all answers are ultimately derived. The reduced tables, such as the one shown in Figure 2(c), which represent the output from a QBE operation are called derived relations or *virtual* tables. Applying QBE operations to a base relation always produces a derived result which is itself a relation. This important property of *closure* [Date, 1981, p. 203] is one which we use extensively in our accounting implementation.

QBE with Multiple Tables

A sample QBE retrieval that ranges across four base tables is illustrated in Figure 3(a). This is a more complex query than our first one, and it is performed on tables which represent the interactions of customers and salespeople in sales transactions. Again, column headings represent attributes, and boldface columns signify key attributes or combinations of attributes that uniquely identify each sale, each customer, each salesperson, and each instance of a relationship among those three entities.

In the body of the tables, we have inserted the QBE operators needed to answer the question, "Who are our salespeople who made large (greater than $10,000) sales to customers from Boston?" This particular query needs four different example elements (each again prefixed by a single underscore character) to link data in the various tables together. Starting with the table at the top, we can see that sales over $10,000 are first identified. These sales (represented by "_INV1") are then linked to their appropriate salespeople and customer (represented by "_SP1" and "_C1" respectively), and a check is then made to restrict the answer to the salespeople who deal with Boston customers. The example element "_SP1" represents all salespeople who meet both criteria. We then link "_SP1" to its appropriate salesperson name (example element "_SMITH") and designate that both pieces of data are to be placed in the "OUTPUT" table shown at the bottom Figure 3(a).[2] The "OUTPUT" table is not an actual part of the database (that is, it is not a base table) but is instead a user-defined or derived table designed only to hold the results of a query. The "P." operator placed under the relation name here causes both columns to be displayed.

Figure 3(b) illustrates a possible answer to this entire second query. In this case, it shows that only four people—Bob, Ann, Dick, and Kathy—made large sales to Boston customers.

This multiple table query finishes our introduction to relational databases and QBE. We proceed now to an overview of our programming methods.

PROCEDURAL OVERVIEW

QBE is a graphical language based upon the domain calculus [Ullman, 1983], and it is most commonly used interactively with a video terminal as a stand-alone retrieval facility [IBM, 1980]. In our accounting work, we will limit procedure specifications to those which can be effected entirely within the QBE system. As we explain later, this narrows the scope of our work somewhat. However, it also allows us to demonstrate how *set* operations alone can be used to derive general ledger balances from a group of relations which do not contain debits, credits, or accounts. Because of its proven user-friendliness, QBE makes these set operations relatively easy to explain and understand, as well as to implement.

In this section of the paper, we discuss the general operational features of our relational accounting implementation, and we give simple examples of the programming methods that we used (such as modularization, nesting, and view construction).

The Structure of a QBE Programming Module

A *program* in QBE is usually defined with a single screen on the video display terminal. Thus, Figure 2(b) can be considered a program with the following parts:

1. an input component consisting of the full intension and extension of the "INVENTORY" table;
2. a process component consisting of the various relational operations (such as selection, projection, and join) that are effected by the QBE operators, constant elements, and example elements; and

3. an output component consisting of the reduced table shown in Figure 2(c).

In a similar manner, a screen which contains the operators illustrated in Figure 3(a) would be a program which transforms four tables of input into one table of output.

Each QBE program can be given a name, and we tried in our implementation to use program names which closely described the conclusion being materialized [McCarthy, 1982, pp. 567–70] from the database by that particular QBE procedure. Thus, a program which uses data concerning customers, sales, and cash receipts as inputs and which produces a subsidiary accounts-receivable listing as an output would be called "accounts-receivable materialization." We also use the term *module* [Yourdon and Constantine, 1979; Turner, 1980] to describe a QBE procedure or set of procedures that has just one functional purpose.

Program Hierarchies

It is possible in QBE to nest modules by having them call and execute each other in much the same manner as subroutines are invoked in programming languages like BASIC. For example, a procedure for materialization of costed inventory items in a manufacturing company might consist of three sub-tasks which produce in turn the relevant information for raw-material items, work-in-process jobs, and finished-good products. In each case, a program could be written which would accumulate cost information from base tables representing both the inventory types and the inventory transactions (such as purchases, factory operations, and sales). This new data would be put into derived tables to be passed to a consolidating module which materializes the costed data for all inventory types.

A common method for showing such subordinate-superior program relationships is the *structure chart* or *hierarchy* [Yourdon and Constantine, 1979] illustrated in Figure 4. Each box on this chart represents a program module for the inventory materialization example just discussed. The arrows with blank tails that go from the second level modules to the top or control module simply illustrate the data that is passed from lower to higher processes. At the bottom level, we can see that various base tables of the manufacturer's database would provide the essential disaggregate data needed for costed inventory calculations.

View Construction

In our accounting implementation, we encountered difficulty in nesting large groups of modules, because the operational version of QBE [IBM, 1980] did not fully support the database concept of "views" [Date, 1981,

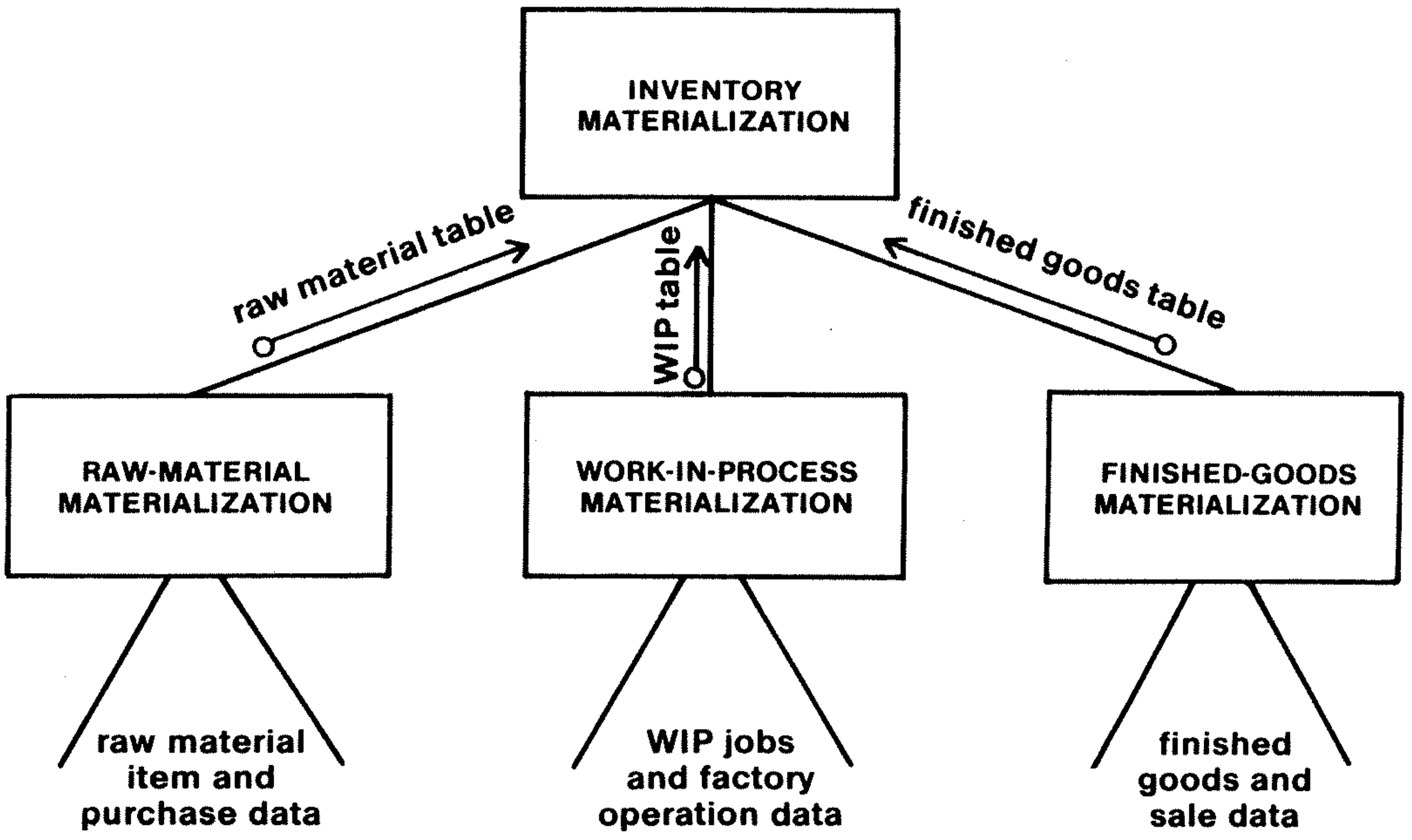
INVENTORY MATERIALIZATION
raw material table
WIP table
finished goods table
RAW-MATERIAL MATERIALIZATION
WORK-IN-PROCESS MATERIALIZATION
FINISHED-GOODS MATERIALIZATION
raw material item and purchase data
WIP jobs and factory operation data
finished goods and sale data

Figure 4. A Structure Chart.

pp. 159-162; McCarthy, 1982, p. 569]. More specifically, QBE did not allow the output from one program to be passed to another unless that output was made part of the permanent database. We did not want our derived tables to become base elements in the system; we only wanted them to be virtual elements which would cease to exist once a procedure was finished. On the other hand, we also wanted to be able to build multilevel module structures.

To overcome this problem, we did two things. We first defined intensions for all of our derived tables; these were stored permanently. We then defined procedures for materialization of the extensions of our derived tables by using the two-level module structure shown in Figure 5. Each materialization module consisted of three subprograms:

1. a delete module which removed the entire previous extension of the derived table;
2. a populate module which created a new extension from the current version of the base tables; and
3. a display module which printed out the results of the populate operations.

For explanation purposes, we assume in the rest of this paper that these materialization procedures work like views in the strictest sense of the word. That is, we assume that the only way to access a virtual table is to invoke the procedure which produces it.

This explanation of view construction finishes our procedural overview. The paper's next section details the specifics of our retail implementation.

RELATIONAL IMPLEMENTATION OF AN EVENTS ACCOUNTING DATABASE

At the logical design level, an accounting database has three primary components [Tsichritzis and Lochovsky, 1982, Chaps. 1–4]:

1. A structural or declarative component which uses the concept of abstraction to categorize accounting phenomena into sets of objects with similar properties;
2. a constraint component which restricts both the values used to characterize accounting objects and the structures that can be used to relate these objects together; and
3. an operation or procedural component which specifies the retrievals and actions which may be performed on the database model.

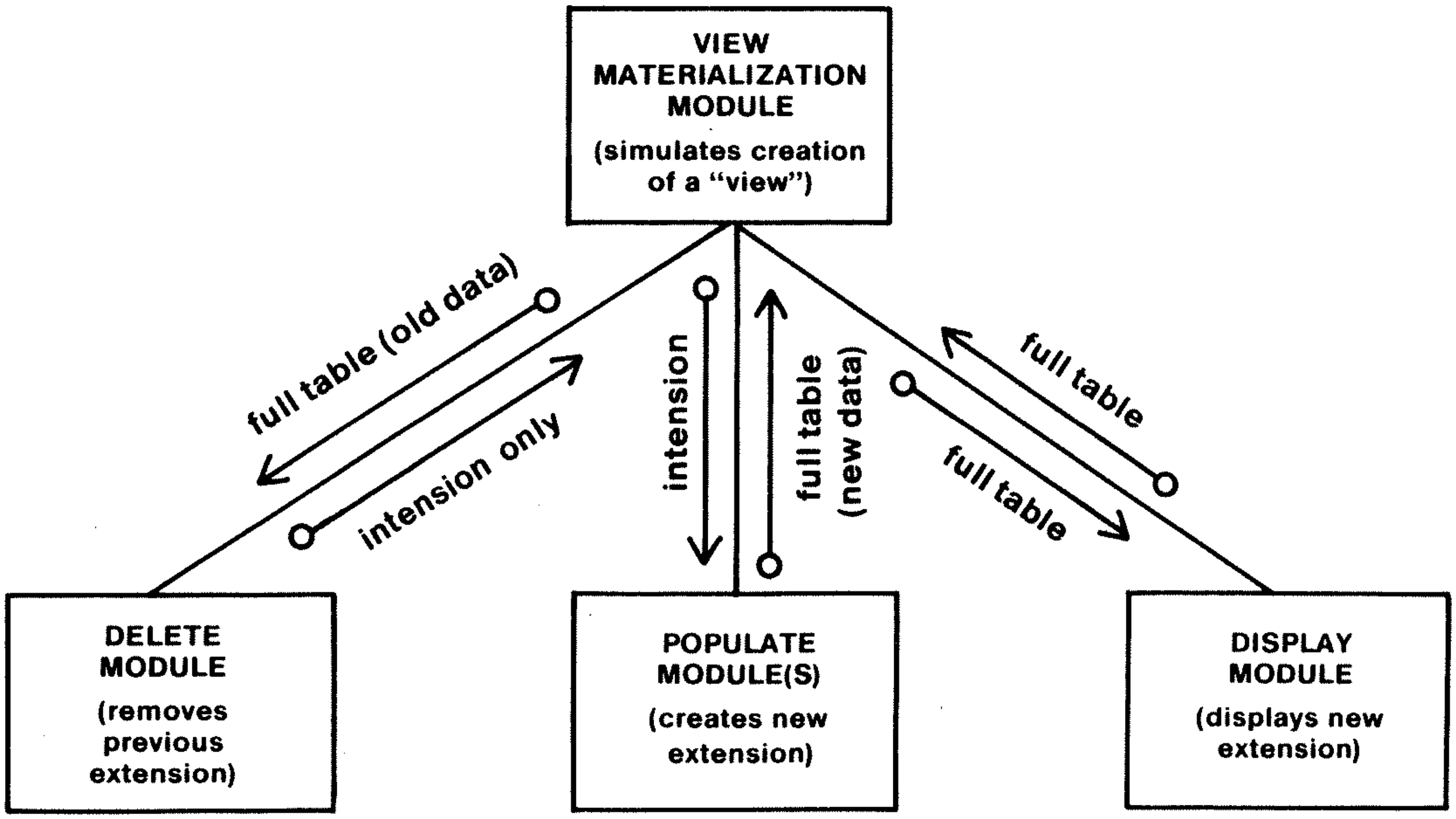

Figure 5. Two-Level Module Structure for View Materialization.

As mentioned previously, we concentrate in this paper on the third of these components: that is, the sets of QBE procedures needed to *maintain* an accounting system and to *retrieve* from it normal types of financial reporting data. Our relational implementation models a small retail enterprise with 41 base tables and a well-developed set of QBE procedures.[3] This section of the paper first overviews that implementation. We then proceed with a thorough explanation of a single accounting example—materialization of accounts-payable for inventory purchases—and we finish with less-detailed expositions of some more aggregate accounting elements.

Procedural Overview

Figure 6 portrays the procedural specification of a general ledger for a simple retail enterprise. Each box on the structure chart represents a QBE module which materializes certain accounting elements and which then passes those elements (in the form of relations) to a higher level in the hierarchy. The data items shown on the connecting lines are views or virtual relations; thus, each one of the materialization modules has the delete, populate, and display subcomponents explained in Figure 5. We also note that base relations provide program input at both the lower level and the second level of the structure chart.

In the two subsections which follow, we explain the detailed mechanisms of those modules surrounded by dotted lines in Figure 6.

Accounts-Payable for Inventory

As mentioned above, the declarative component of a database uses the concept of abstraction [Tsichritzis and Lochovsky, 1982] to categorize phenomena into sets of objects with similar properties. In our accounting example, the particular abstraction method used was Entity-Relationship (E-R) modeling [Chen, 1976], and application of this method to an example retail enterprise resulted in an E-R model similar to that illustrated by McCarthy [1979, p. 675].

Figure 7 portrays the portion of our E-R model which deals specifically with the calculation of accounts-payable for inventory purchases. Each of the three rectangles shown represents an entity set, while the diamonds represent relationship sets. Attributes of entities and relationships are represented above the rectangles and diamonds by lines with circles on the end. Circles that are filled in signify key attributes [Atzeni et al., 1983].

Mapping an E-R model into a relational database is a relatively straightforward task which uses one table (or base relation) to represent each entity set and one table to represent each relationship set.[4] Column headings represent attributes. Thus, a user could work with the data model of Figure 7 by creating five tables and by designating keys properly. As the

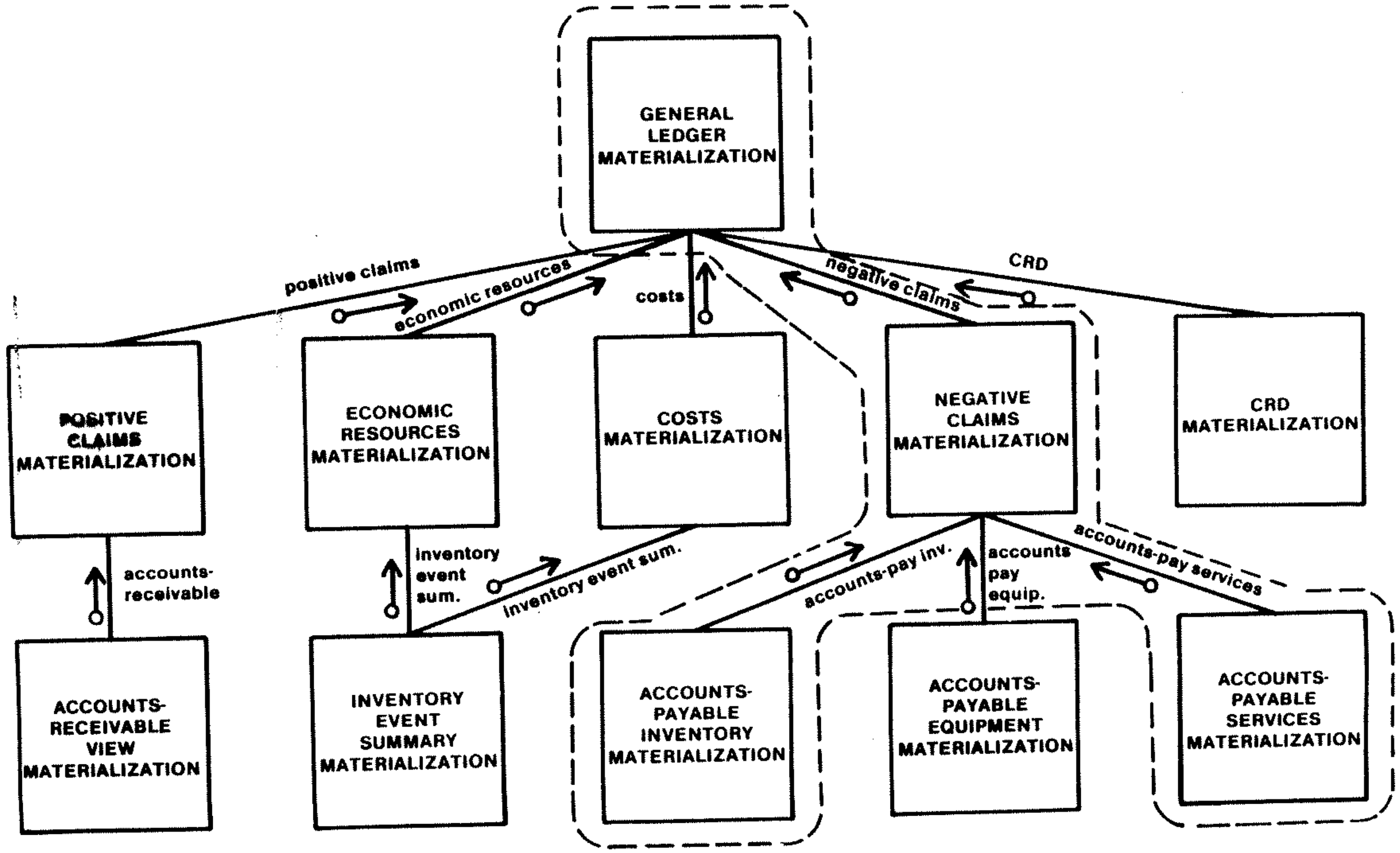
GENERAL LEDGER MATERIALIZATION
positive claims
economic resources
costs
negative claims
CRD
POSITIVE CLAIMS MATERIALIZATION
ECONOMIC RESOURCES MATERIALIZATION
COSTS MATERIALIZATION
NEGATIVE CLAIMS MATERIALIZATION
CRD MATERIALIZATION
accounts-receivable
inventory event sum.
inventory event sum.
accounts-pay inv.
accounts pay equip.
accounts-pay services
ACCOUNTS-RECEIVABLE VIEW MATERIALIZATION
INVENTORY EVENT SUMMARY MATERIALIZATION
ACCOUNTS-PAYABLE INVENTORY MATERIALIZATION
ACCOUNTS-PAYABLE EQUIPMENT MATERIALIZATION
ACCOUNTS-PAYABLE SERVICES MATERIALIZATION

Figure 6. Structure Chart for Materialization of General Ledger.

Fig ...ucture Chart for Materialization of General Ledger.

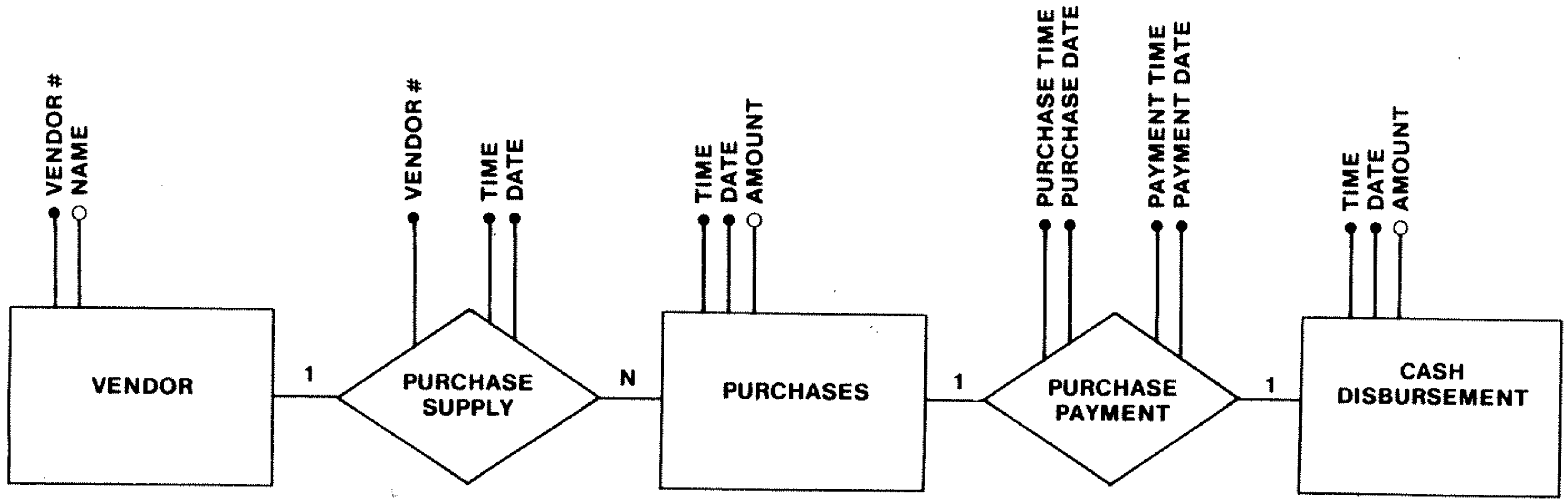

Figure 7. Entity-Relationship Model (Partial) of Inventory Purchases.

actual purchases and cash disbursements for the retail company occur and as vendors are added to our list of suppliers, new rows would be inserted into the database for all of our example tables.

If we assume that relations such as those described above have been created and maintained in the retail company for a month, the QBE program of Figure 8 could be used to produce a listing of subsidiary accounts-payable for vendors who supply inventory. The intensions of the first five tables shown in that program are mapped directly from the E-R diagram of Figure 7. The sixth table shown—"ACCOUNTS PAYABLE INVENTORY"—is a virtual relation into which the results of the QBE program are inserted and with which those results are passed to a higher level module.

The calculation of accounts-payable for purchases is a good example of the *set* orientation of relational database systems, and we highlight this orientation by portraying the QBE operations graphically in Figure 9. This particular calculation can be best understood by viewing both Figure 8 and Figure 9 together and by following the explanations given below.

a. First, the set of unpaid purchases is identified for the retail enterprise by performing a set difference operation which subtracts paid-for purchases from all purchases. In QBE terms, this operation is performed by identifying all purchase events (that is, extensions of the "PURCHASES" table as specified by example elements in its key columns) which do *not* also participate in the purchase relationship (as specified by the *not*–"¬"–operator in the "PURCHASE PAYMENT" table).
b. Second, elements of this set of unpaid purchases are linked individually to their respective vendors and then grouped into subsets on the basis of vendor identification. In QBE terms, this linking and grouping is done by using example elements and by inserting the *group-by* operator—"G."—in the key column of the "VENDOR" table.
c. Third, the summed dollar amount for each subset of unpaid purchases, along with its vendor number and name, is identified and inserted into the database as accounts-payable. This last operation is effected by QBE with the *insert* operation—"I."—in the "ACCOUNTS PAYABLE INVENTORY" table.

The set-oriented illustrations of Figure 9 portray graphically an advantageous feature of relational systems which is their ability to have operations range over an entire table at a time. This feature allows users simply to specify operators appropriate for a given set of things without having to

VENDOR	**VENDOR #**	NAME
	G. _ VN	_ VM

PURCHASE SUPPLY	**VENDOR #**	**TIME**	**DATE**
	_ VN	_ T	_ D

PURCHASES	**TIME**	**DATE**	AMOUNT
	_ T	_ D	ALL. _ AM

PURCHASE PAYMENT	**PURCHASE TIME**	**PURCHASE DATE**	**PAYMENT TIME**	**PAYMENT DATE**
¬	_ T	_ D		

CASH DISBURSEMENT	**TIME**	**DATE**	AMOUNT

ACCOUNTS PAYABLE INVENTORY	VENDOR #	NAME	AMOUNT OWED
I.	_ VN	_ VM	SUM. ALL. _ AM

Figure 8. Populate Module for Materialization of Accounts-Payable Inventory.

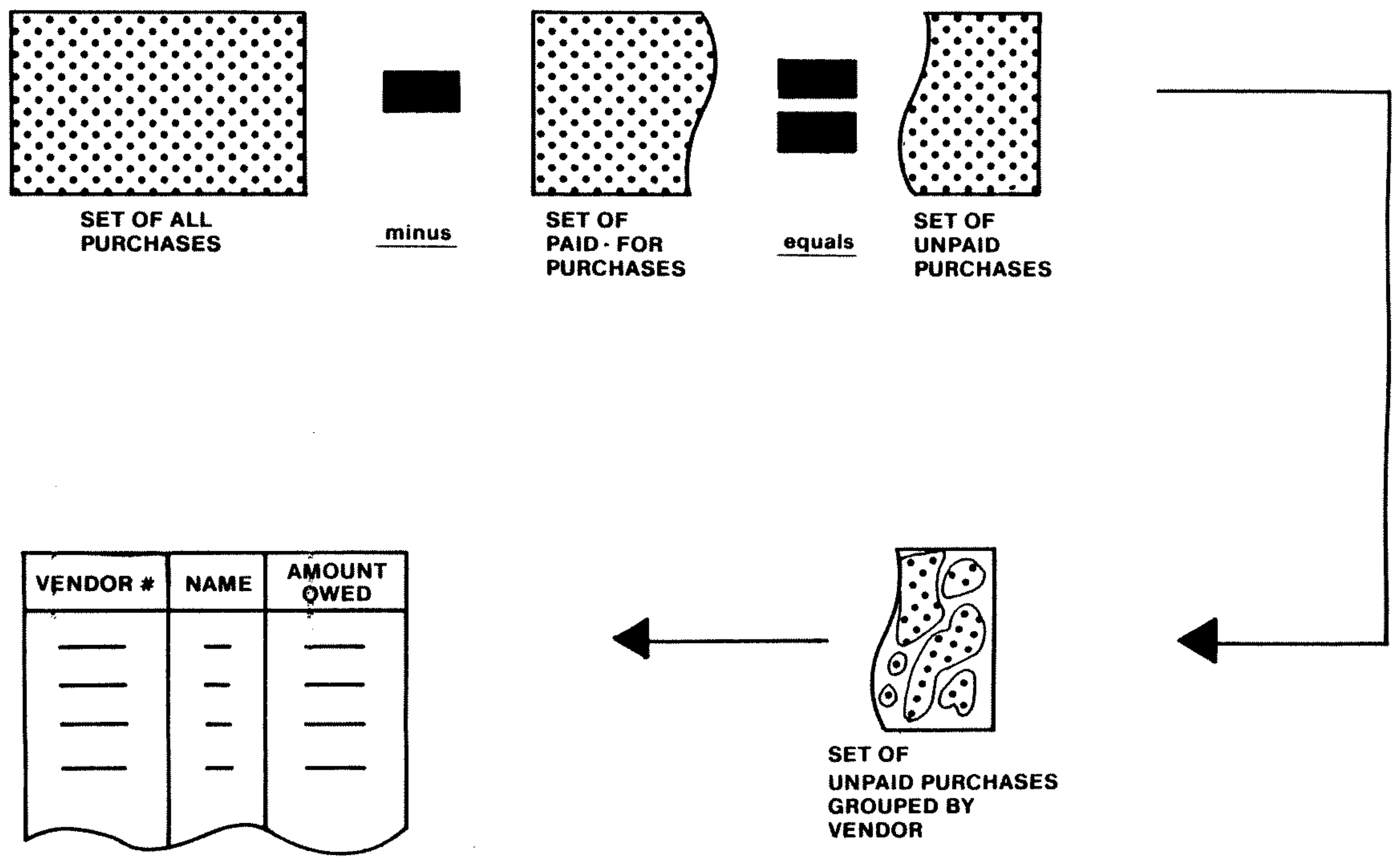

Figure 9. Graphic Illustration of Accounts-Payable Derivation.

lay out algorithmically the element-by-element iterations needed for individual application.

If we now return to the structure chart of Figure 6, we see that the materialization of accounts-payable for inventory is a program module located at the third hierarchical level and that it produces as a result of its invocation both the intension and extension of the last table shown in Figure 8. This table is a virtual relation or view which is always current (the information is never stale) and which can always be accessed on-line by anyone who is given the authority to look at it (a disbursements clerk, for example). For purposes of this paper however, we are more interested in this module's role in the overall scheme of Figure 6—that is, in its use in the overall materialization of the general ledger. Seen in that light, the payables for inventory simply become a component of a more aggregate set of accounting numbers. In the section that follows, we explain how this and other components are used for this larger purpose.

Other Materialization Modules

This section discusses the derivation of the rest of the selected components surrounded by dotted lines in Figure 6. Because the explanations of previous examples have been very detailed, we will assume that the rudiments of QBE operations are now understood. Explanations here will proceed more quickly.

Figure 10 illustrates the populate program for the materialization of accounts-payable for services, a procedure which produces a list of vendors who do such things as rent the company office-space or provide it with advertising. This module operates analogously to the module shown earlier for inventory payables with one exception: no set difference operation is performed. Instead, payables are calculated by summing up expense transactions and then by subtracting related disbursements. To perform such an operation, we identify each vendor (using example element "_VN") and then link through to obtain the dollar amounts of both "GENERAL AND ADMINISTRATIVE SERVICE" and "CASH DISBURSEMENT." The Group-by operator ("G.") in the top table again partitions the dollar amounts by individual vendor, while the derived table "ACCOUNTS PAYABLE SERVICES" at the bottom of the figure is the repository for the final arithmetic calculation which subtracts summarized disbursements from summarized services.

The method shown above must be used with relational systems to materialize claims when the *duality* relationships between economic events is not functional (that is, when they are not "l-to-l" or "n-to-l" [McCarthy, 1979, pp. 671–73]). In our modeled example, we assumed that a service could be paid for with many disbursements (such as a big freight bill being

VENDOR	VENDOR #	NAME
	G._VN	_VM

GEN ADMIN SUPPLY	VENDOR #	TIME	DATE
	_VN _VN	_T _T3	_D _D3

GENERAL AND ADMINISTRATIVE SERVICE	TIME	DATE	AMOUNT
	_T3	_D3	ALL._A1

GEN ADMIN PAYMENT	SERVICE TIME	SERVICE DATE	PAYMENT TIME	PAYMEN DATE
	_T	_D	_T1	_D1

CASH DISBURSEMENT	TIME	DATE	AMOUNT
	_T1	_D1	ALL._A2

ACCOUNTS PAYABLE SERVICES	VENDOR	NAME	AMOUNT OWED
I.	_VN	_VM	(SUM.ALL._A1-SUM.ALL._

Figure 10. Populate Module for Materialization of Accounts-Payal Services.

paid in monthly installments), but we also assumed that many services could be paid for with one disbursement (such as monthly advertising services being paid with a yearly check). This relationship was many-to-many ("m-to-n") and non-functional.

Figure 11 portrays the materialization of the negative claims (a term explained by Ijiri [1975, p. 65]) for the entire enterprise. For simplicity sake, we used two populate modules in this illustration. The first one is shown as Figure 11(a), and it simply aggregates (across both source and vendor) the various accounts-payable. The second populate program—Figure 11(b)—calculates other claims against the company. After this set of programs is executed, there is another view available for use in the database. That view is a table called "NEGATIVE CLAIMS," and its extension consists of three rows, one each for accounts-payable, accrued wages, and dividends-payable.

Finally, Figure 12 illustrates the controlling module of the program hierarchy: the materialization of the general ledger. The first half—Figure 12(a)—portrays the insert operations which first take data from the various

ACCOUNTS PAYABLE INVENTORY	VENDOR #	AMOUNT OWED
		ALL. _A1

ACCOUNTS PAYABLE EQUIPMENT	VENDOR #	AMOUNT OWED
		ALL. _A2

ACCOUNTS PAYABLE SERVICES	VENDOR #	AMOUNT OWED
		ALL. _A3

NEGATIVE CLAIMS	NAME	AMOUNT
I.	ACCOUNTS PAYABLE	(SUM. ALL. _A1 + SUM. ALL. _A2 + SUM. ALL. _A3)

Figure 11 (a). First Populate Module for Negative Claims.

PERSONNEL SERVICE	TIME	DATE	GROSS PAY
			ALL. _PSAMT

EMPLOYEE PAYMENT	RECORD TIME	RECORD DATE	PAYMENT TIME	PAYMENT DATE
			_PST	_PSD

CASH DISBURSEMENT	TIME	DATE	AMOUNT
	_PST _DT	_PSD _DD	ALL. _PSCD ALL. _DCD

CAPITAL PAYMENT	RECORD TIME	RECORD DATE	PAYMENT TIME	PAYMENT DATE
			_DT	_DD

DIVIDEND DECLARATION	TIME	DATE	AMOUNT
			ALL. _DDAMT

NEGATIVE CLAIMS	NAME	AMOUNT
I.	ACCRUED WAGES	(SUM. ALL. _PSAMT - SUM. ALL. _PSCD)
I.	DIVIDENDS PAYABLE	(SUM. ALL. _DDAMT - SUM. ALL. _DCD)

Figure 11 (b). Second Populate Module for Negative Claims.

second levels modules and which then place that data under the appropriate debit or credit heading. The second half—Figure 12(b)—illustrates the actual extension of the general ledger for the set of transactions that we used.

DISCUSSION

We have now completed an overview of the operation of an events-based relational system. Our examples and discussion have shown how relational

CLAIMS	NAME	AMOUNT
	_PCNAME	_PCA

NEGATIVE CLAIMS	NAME	AMOUNT
	_NCNAME	_NCA

ECONOMIC RESOURCES	NAME	AMOUNT
	_ERNAME	_ERA

CRD	NAME	AMOUNT
	_CRDNAME	_CRDA

COSTS	NAME	AMOUNT
	_COSTNAME	_COSTA

GENERAL LEDGER	ACCOUNT	DEBIT	CREDIT
I.	_PCNAME	_PCA	
I.	_NCNAME		_NCA
I.	_ERNAME	_ERA	
I.	_CRDNAME		_CRDA
I.	_COSTNAME	_COSTA	

Figure 12 (a). Populate Module for General Ledger Materialization.

GENERAL LEDGER	ACCOUNT	DEBIT	CREDIT
	ACCOUNTS RECEIVABLE	$70,100.00	—
	SUBSCRIPTIONS RECVBL	$0.00	—
	ACCOUNTS PAYABLE	—	$17,327.00
	ACCRUED WAGES	—	$656.00
	DIVIDENDS PAYABLE	—	$0.00
	INVENTORY	$37,290.00	—
	EQUIPMENT	$13,100.00	—
	CASH	$57,290.00	—
	ACCUMULATED DEPREC	—	$145.00
	COMMON STOCK		$150,000.00
	SALES		$124,000.00
	TRANSPORTATION	$5,782.00	—
	ADVERTISING	$1,000.00	—
	CLEANING	$650.00	—
	RENT	$1,600.00	—
	COST OF GOODS SOLD	$91,955.00	—
	DIVIDENDS	$6,000.00	—
	WAGES	$7,216.00	—
	DEPRECIATION	$145.00	—

Figure 12 (b). General Ledger Table.

systems can be used to support innovative methods of tracking economic events while simultaneously being used to meet the traditional general ledger needs of accountants. The capability to support financial reporting needs in accordance with generally accepted accounting principles is certainly the prime requirement of any computerized accounting system. However, it is a decided advantage for both accountants and non-accountants if the database that meets that requirement is also available for the support of less precise and less predictable managerial decision making needs. It is a further advantage if that database is presented to all users in the simple tabular forms shown here and if its data can be manipulated with powerful yet friendly languages like Query-by-Example. An implementation patterned after the one illustrated in this paper would clearly provide a sound foundation for an information system designed to facilitate the development and use of decision support systems [Sprague and Carlson, 1982].

Limitations in Scope

Caution should be exercised, however, in generalizing extensively from our implementation experience, primarily because of its limited scope.

Our retail model was small scale and deliberately chosen for simplicity of implementation. The quantity of transaction data (one month for a small company) was very low, and both the financial structure and the organizational structure of the company was simple. If we had chosen, for example, to include more complex equity structures or to include a manufacturing component in our company, the structure of modules illustrated in Figure 6 would have become both wider and deeper.

Additionally, no cost-benefit decisions were made concerning maintenance of data, physical storage, and program execution costs. We treated all of these elements as free goods, and quite obviously, a large-scale system would have to evaluate them in terms of normal database design tradeoffs [Auerbach Editorial Staff, 1981].

Finally in terms of limited scope, we omitted some programmed maintenance operations which would be needed for an actual company. We loaded in our data at month's end instead of day-by-day, thus avoiding the task of writing consistency maintenance mechanisms such as triggers [Eswaran, 1976]. The commercial version of QBE does not support triggers (which would, for example, decrease inventory quantity when a sale was made), and our consistency maintenance mechanisms would have had to be implemented via a programming language. We also would have needed programmed formatting and manipulation assistance if we had chosen to provide all the input-output enhancements (such as menu-driven entry of data and variable format report writing) that are usually furnished with commercial general ledger packages ([Yoder and Knight, 1984; Post, 1984]). Most of these maintenance tasks, however, were ones that we had implemented routinely with a programming language (FORTRAN) in some previous database work [Gal and McCarthy, 1983a].

Insights Provided

Our project's limited scope did not prevent us from encountering and solving problems which we believe provide valuable insights to accounting systems designers in a database environment. Two of these insights deserve note.

First, the project demonstrated the utility of the REA framework in the design and use of a computerized accounting system. We did not choose to use the full REA event template [McCarthy, 1982, p. 564] for every economic event tracked, because the information needs of such a small company did not necessitate it. In particular, we did not build in a responsibility accounting system, and we therefore had no need to identify inside economic agents. We also chose not to treat claims as base objects. However, either of these cases could have been implemented differently if we perceived that managers of our example company needed different types of information. This particular case study did show that the REA framework works well in producing a general ledger without first subjecting transactions to classification based on ledger accounts. It also provided very interesting examples of some of the design tradeoffs discussed by McCarthy [1982, pp. 565–75]. For example, system designers may gain considerable insight into the very difficult area of procedural-declarative tradeoffs by first viewing our method of materializing accounts-payable and by then speculating under what circumstances it might be appropriate to include this claim as a separate attribute of a base relation (instead of using a program as we did).

Second, this project did demonstrate that a relational database could be used effectively to implement an accounting information system. However, we did encounter some problems in our work which might cause other system designers to hesitate on choosing a *pure* relation system (like QBE) for implementation. Most of these concerned the "set-at-a-time" orientation of relational operators which on the one hand is a very appealing theoretical feature but which on the other hand limits the possible performance options and which also causes possible misunderstandings among users. For example, we found the following problems in our work.

A. In calculating our inventory, we had to use a weighted average cost scheme rather than FIFO or LIFO, because these latter methods require calculations that use set elements one at a time (or tuple-by-tuple in relational terms). To compute LIFO or FIFO, one needs to cost remaining inventory units against past purchases one-by-one. By contrast, weighted average uses the whole set of purchase events. The commercial version of QBE is limited to specification operations [Tsichritzis and Lochovsky, 1982, p. 74] which work on whole relations very well but which need programming language assistance for tuple-by-tuple computations.

B. We discovered that certain relational operations have very limited applicability to a realistic range of accounting entities. These operations include set divisions and set differences. For example, we materialized one set of claims (accounts-payable for inventory) with a set difference operation, but we noted that such a computation would only work if the duality relationship between our increment transaction (purchases) and our decrement transactions (cash disbursements) was functional ("n-to-l" or "l-to-l"). This is somewhat of an unrealistic assumption because it excludes "l-to-n" and "m-to-n" relationships [McCarthy, 1979, pp. 671-73]. It also precludes modeling of situations where the second duality transaction precedes the first in time (such as prepaid expenses). Given this limited applicability, most system designers may choose to discard these operations and to use the simple arithmetic methods that we used for service payables (see Figure 10 and accompanying discussion). These arithmetic methods retain some set-at-a-time flavor, because they range over all tuples in a specified relation. However, they are much more general in their application.

C. Finally, with regard to the set orientation of pure relational systems, we discovered a semantic mismatch between the accounting concept of zero (that is $0.00) and the relational concept of a null set. For example, if we had never received any cash receipts from particular customers, the correct accounting action is to subtract zero from their individual accounts-receivable when those claims are materialized. In relational or QBE terms however, the customers' null participation in the set of cash receipt transactions results in them being excluded entirely from the listing of subsidiary accounts. We grant that this is a relatively minor problem that occurs infrequently, but it does give one an appreciation for how carefully specified relational semantics must be.

We believe that the second and third of these problems are not significant, because either they occur very infrequently or they can be anticipated and dealt with by a knowledgeable designer at the early stages of implementation. The tuple-by-tuple access problem *is* significant. However, most relational DBMS available today have recognized the necessity of augmenting their basic capabilities with sequence, decision, and iteration constructs.[5] We also believe that most of the non-set or navigational processing can be done by professional programmers, leaving non-programmers with the simpler relational features to use in their decision support work.

Recommendations

From our implementation experience and analysis, we have the following recommendations for accountants who are attracted by the simplicity of relational systems and who believe that the environment in which their accounting systems will operate would benefit from an integrated database approach.[6]

First, relational systems do have conceptual advantages, but users must make sure they avail themselves of those advantages by careful choice of a software package. When companies purchase a relational DBMS for either a large computer system [Schmidt and Brodie, 1983] or a small computer system [PC Magazine Editorial Staff, 1984], they should insure that the set-oriented operations which link tables (join) and which reduce tables (selection and projection) are fully supported in a manner that non-programmers will find easy to use. Most commercial systems will *not* be as good as QBE in this regard, but there exists a variety of other carefully designed methods for interacting with relational systems. Some of these alternative systems have also been tested psychologically [Reisner, 1977, 1981; Welty and Stemple, 1981; Shneiderman, 1980], and many of the newer microcomputer packages have based their approaches on such established methods.

Second, in contrast to the ease-of-use advantages discussed above which are provided within the relational software one purchases, the *design* advantages of relational systems must be provided by the users themselves through a formal approach to data analysis. In other words, a company may use a relational DBMS and still have a database characterized by inconsistency and redundancy. It is not necessary that designers of accounting systems be intimately aware of all normalization research—much of which has evolved beyond the point of practical use—it is only necessary that those designers be aware of a few elementary rules [Kent, 1983] for avoiding maintenance anomalies in a database. A simple way to incorporate such rules (and to gain other advantages as well) is to use a semantic approach [Tsichritzis and Lochovsky, 1982] to design. Potential users who are interested in database analysis methodologies which combine the use of semantic models and normal forms may consult Martin [1983], Howe [1983], or Hawryszkiewycz [1984].

Third, as we discussed in our choice of inventory valuation methods, the set operations of a relational DBMS must be augmented by a capacity to access tables one row at a time in sequence. Most commercial systems will provide this capability via programming language extensions. The consolidated use of both navigation and specification languages will be best supported in an environment which delegates the operation of large application systems (such as payroll and general ledger) to professional programmers and which provides those programmers with both relational and non-relational capabilities. Managers in need of decision support facilities can be provided with relational views and languages in much the same manner as we derived our "views" in this paper. An alternative method for supporting this same environment would be to purchase software which provides preprogrammed modules for the routine applications and relational capabilities for the non-routine ones. This last approach is one on which many software vendors are currently basing their database services.

Finally, we would recommend that users consider fully the REA approach to database design used in this paper. Because of our small-scale example and because we ignored machine costs, we were able to support a full *events* approach to our accounting system development. With the present state of computer technology, users will have to consider such a system as a theoretical ideal which must be adjusted when the task of detailed analysis starts. However, this is how good database design methodologies work. Conceptual integrity is considered first, and realistic concessions are made to implementation cost with a careful evaluation of what is being lost in the compromise. Its semantic base insures that an REA accounting system will be well-designed, and its emphasis on integration will insure that maximum opportunity is afforded to use accounting data in a wide variety of decision settings in an organization.

ACKNOWLEDGMENTS

Research support for this work was provided by the Peat Marwick Mitchell Foundation through its Research Opportunities in Auditing Program. We would like to thank the editors and reviewers for their assistance in revising this paper.

NOTES

1. The QBE operators which identify certain rows (i.e., they isolate a horizontal subset of a table) and which identify certain columns (i.e., they isolate a vertical subset of a table) correspond to the relational algebra operations of "select" and "project" respectively [Date, 1981, Chap. 12].
2. This linking of data across multiple tables using example elements is the QBE equivalent of the relational algebra "join" operation [Date, 1981, Chap. 12].
3. The full transaction history, the full layout of all base tables, and the full specification for all procedures is available in Gal and McCarthy [1983b]. The constraint component of this implementation is outlined in Gal and McCarthy [1985].
4. Our data model had 18 entities and 23 relationships which resulted in 41 base tables.
5. Query-by-Example itself can be augmented with these capabilities by linking it with a programming language. However, the *linear syntax* of QBE [IBM, 1980] which must be used to perform this linking is not screen-oriented, and it does not possess any of the proven ease-of-use capabilities that we have mentioned previously.
6. A beginning book which might help users decide whether or not their companies ought to be using a database management system has been published by Date [1983].

REFERENCES

Auerbach Editorial Staff, "Trade-offs in Data Base Design," in *Practical Data Base Management* (Reston, 1981).

Atzeni, P., C. Batini, M. Lenzerini, and F. Villanelli, "INCOD: A System for Conceptual Design of Data and Transactions in the Entity-Relationship Model," in P. Chen, ed., *Entity-Relationship Approach to Information Modeling and Analysis* (North-Holland, 1983).

Chen, P. P., "The Entity-Relationship Model — Toward a Unified View of Data," *ACM Transactions on Database Systems* (March 1976), pp. 9–36.

Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* (June 1970), pp. 377–87.

Date, C. J., *An Introduction to Database Systems*, 3rd ed. (Addison-Wesley, 1981).

———, *Database: A Primer* (Addison-Wesley, 1983).

Eswaran, K. P., "Aspects of a Trigger Subsystem in an Integrated Database System," *Proceedings of Second International Conference on Software Engineering* (IEEE, 1976), pp. 243–50.

Everest, G. C. and R. Weber, "A Relational Approach to Accounting Models," *The Accounting Review* (April 1977), pp. 340–59.

Gal, G. and W. E. McCarthy, "Declarative and Procedural Features of a CODASYL Ac-