

AD-A241 353



2

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

CONCEPTS HIERARCHIES
FOR
EXTENSIBLE DATABASES
by

Christopher A. Barnes

September, 1990

Thesis Advisor:

Daniel R. Dolk

Approved for public release; distribution is unlimited.

91-12534



91 10 14 1089

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution is unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b. OFFICE SYMBOL (If applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000			7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
			Program Element No	Project No	Task No Work Unit Accession Number
11. TITLE (Include Security Classification) CONCEPT HIERARCHIES FOR EXTENSIBLE DATABASES					
12. PERSONAL AUTHOR(S) BARNES, Christopher A.					
13a. TYPE OF REPORT Master's Thesis		13b. TIME COVERED From To		14. DATE OF REPORT (year, month, day) 1990 SEPTEMBER	
				15. PAGE COUNT 82	
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.					
17 COSATI CODES			18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUBGROUP	database design, data manipulation, semantic networks		
19. ABSTRACT (continue on reverse if necessary and identify by block number) This thesis presents a simple and efficient method of implementing a semantic type checking system for use with relational databases. Numeric data typically represent measures of a specific property or characteristic of a real world object. Computers manipulate only the numeric value. It is the responsibility of the user to ensure that the data are handled in a manner consistent with its meaning. If the semantics associated with the numbers are stored in a data dictionary, semantic consistency can be verified by the database system. This increases the integrity of data manipulation and helps ensure meaningful results. This thesis demonstrates a simple scheme of representing the property, or quantity, and unit of measure associated with numeric attributes. This information is then used to verify dimensional consistency of database queries and to automatically convert units across systems of measurement. Finally, a concept is defined for each relation in the database. These concepts can be used to build a concept hierarchy to help ensure queries are consistent with the semantics of the database design.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Daniel R. Dolk			22b. TELEPHONE (Include Area code) (408) 646-2260		22c. OFFICE SYMBOL AS/Dk

Approved for public release; distribution is unlimited.

Concept Hierarchies
for
Extensible Databases

by

Christopher A. Barnes
Lieutenant, United States Navy
B.S., Northwestern University

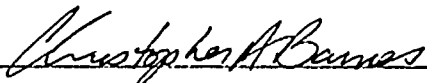
Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

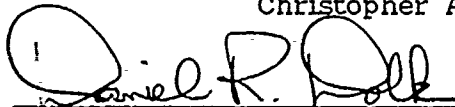
from the

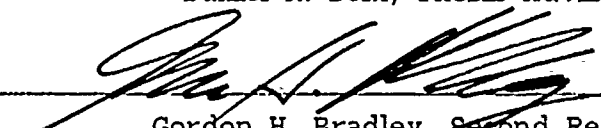
NAVAL POSTGRADUATE SCHOOL
September 1990

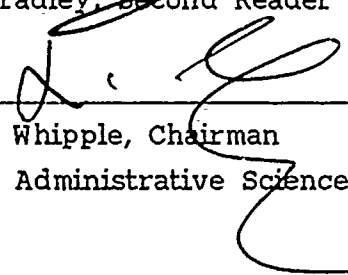
Author:


Christopher A. Barnes

Approved by:


Daniel R. Dolk, Thesis Advisor


Gordon H. Bradley, Second Reader


David R. Whipple, Chairman
Department of Administrative Sciences

ABSTRACT

This thesis presents a simple and efficient method of implementing a semantic type checking system for use with relational databases. Numeric data typically represent measures of a specific property or characteristic of a real world object. Computers manipulate only the numeric value. It is the responsibility of the user to ensure that the data are handled in a manner consistent with its meaning. If the semantics associated with the numbers are stored in a data dictionary, semantic consistency can be verified by the database system. This increases the integrity of data manipulation and helps ensure meaningful results. This thesis demonstrates a simple scheme of representing the property, or quantity, and unit of measure associated with numeric attributes. This information is then used to verify dimensional consistency of database queries and to automatically convert units across systems of measurement. Finally, a concept is defined for each relation in the database. These concepts can be used to build a concept hierarchy to help ensure queries are consistent with the semantics of the database design.

TABLE OF CONTENTS

I.	INTRODUCTION	1
	A. PROBLEM DESCRIPTION	1
	B. METHODOLOGY	3
	C. THESIS STRUCTURE	4
II.	THE SEMANTICS OF NUMERIC DATA	5
	A. DATA TYPING AND TYPE CHECKING	5
	B. NUMERIC DATA TYPES AND UNITS OF MEASURE	6
	C. DIMENSIONAL ANALYSIS	8
	D. THE SEMANTICS OF QUANTITY AND CONCEPT	11
	E. AN EXTENDED NUMERIC DATA TYPE	13
III.	NUMERIC DATA IN RELATIONAL DATABASES	15
	A. RELATIONAL DATA DEFINITION	15
	B. RELATIONAL DATA MANIPULATION	17
	C. ADDING CONCEPT, QUANTITY AND UNITS TO THE RELATIONAL MODEL	26
IV.	REPRESENTING THE SEMANTICS IN THE DATA DICTIONARY	30
	A. REPRESENTATION OF SEMANTICS	30
	B. DATA DICTIONARY REPRESENTATION	35

V. EVALUATING SQL EXPRESSIONS	40
A. ACCESSING THE DATA DICTIONARY	41
B. EVALUATING SEARCH CONDITIONS	42
C. AN EXAMPLE OF PREPROCESSING	47
D. CONVERSION OF COMMENSURATE UNITS	50
VI. CONCEPT HIERARCHIES	52
A. LINKING RELATIONS	53
B. DIRECTED GRAPHS	56
C. CONCEPT HIERARCHIES	58
VII. CONCLUSION	64
APPENDIX A: SAMPLE DATA DICTIONARY TABLES	67
APPENDIX B: POSTFIX ALGORITHMS	69
LIST OF REFERENCES	73
INITIAL DISTRIBUTION LIST	75



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

I. INTRODUCTION

This thesis addresses the problem of adding semantic information to numeric-valued data attributes stored in a database. Numeric data normally are associated with a characteristic or dimension of some real world object or event. The data typically reference a specific system of measurement through a particular unit of measure. Database management systems (DBMS), however, typically store only the numeric value without reference to this measurement. The values are then manipulated without regard to their real-world meaning. For example, a value measured in miles might be added to one measured in pounds. It is currently the responsibility of the user to ensure that data are manipulated in a manner consistent with its meaning. We propose a method of storing the meaning or semantics of the numeric data along with the actual data values. This will allow the system to verify automatically the semantic consistency of data manipulation.

A. PROBLEM DESCRIPTION

In physics and engineering, as well as other applications, including units of measure along with numerical values has a long tradition. The numbers by themselves have arbitrary

meaning. Only with a unit of measure is there a standard of comparison with which to determine the extent of something. Suppose X is measured in units of feet. We then know the value assigned to X represents the number of times one foot occurs in the object represented by X. Furthermore, we know X should not be added, subtracted, or compared with Y unless both are expressed in feet. (Clemence, 1987, p. 8-9)

In addition to units of measure, numeric data are associated with some object or event. Specifically, the data represent a measure of some quantifiable property of that object or event. This semantic information ensures data are manipulated in a meaningful way. Apples should not be added to oranges and the weight of a car should be subtracted from its length.

Verification of the semantic consistency of data manipulation is currently the responsibility of the user. The computer manipulates only the numbers. To verify semantic consistency, each numeric-valued symbol is replaced with its explanatory description. Two types of dimensional calculus are then applied. First, is the calculus of measurement units: units are multiplied or divided and an analysis is performed to ensure that the numbers being added, subtracted or compared all have the same scale of reference. The other calculus is concerned with what the numbers represent and the properties being measured. Again the calculus prescribes

certain rules for multiplication, division, addition, subtraction, and comparison. (Bradley and Clemence, 1987, p. 404)

The rules of semantic consistency also apply to the relational database environment. The relational model provides a powerful data access and manipulation capability. Data are stored in two dimensional tables, a structure easily understood by most users. Many updates and queries are accomplished by combining, or joining, multiple tables in various ways. This provides an extremely powerful access capability, particularly for ad hoc queries.

This ease of access and unlimited data manipulation capability does not come free. The user is still responsible for ensuring data are combined in a manner consistent with their real-world meaning. There is still nothing that prevents the weight of an apple from being added to the diameter of an orange. In small databases with relatively few tables, the user might be able to track the meaning of the data himself. However, this is unlikely in larger systems with multiple users.

B. METHODOLOGY

This thesis focuses on design and implementation of an SQL shell to enforce dimensional and unit consistency of database queries. The research includes the following steps:

1. Review of the semantics of numeric data and the application of concept hierarchies in model management as presented by Bradley and Clemence (Bradley and Clemence, 1988).
2. Application of the semantics of quantity and unit of measure to numeric data in the relational data model.
3. Design of data dictionary tables needed to incorporate semantic information.
4. Design and implementation of an SQL preprocessor which uses the semantic information to verify dimensional consistency of database queries.

C. THESIS STRUCTURE

Our research is presented in six chapters. Chapter II discusses the semantics of numeric data and presents an abstract data type consisting of a value description and a semantic description. Chapter III demonstrates the benefits of including the semantics in the relational data model. Chapter IV describes how to represent semantic information in the data dictionary. Chapter V presents a simple and efficient method of ensuring the dimensional consistency of database queries. Chapter VI shows how the semantics can be used to build concept hierarchies. These hierarchies are then used to enforce semantic integrity of database queries. Chapter VII presents our conclusions and recommendations for further research.

II. THE SEMANTICS OF NUMERIC DATA

Numeric data typically are associated with quantifiable characteristics or phenomena in the real world. The computer, however, manipulates only the numbers without regard to their meaning. The user has the responsibility of ensuring that numbers are manipulated in a way consistent with their meaning. In this chapter we discuss the semantics of numeric data and propose an extended data type for use in a more rigorous typing system. Data types and type checking will be discussed first in general and then in the context of units of measure and dimensional analysis. Finally, the semantics of quantity and concept are added and an abstract numeric data type is proposed.

A. DATA TYPING AND TYPE CHECKING

Computers store and manipulate data and programming constructs as combinations of zeros and ones. At the bit level, there is no distinction between a character and a number. Both are represented as binary numbers. Data typing provides the distinction and establishes how the contents of memory are to be interpreted by the computer.

A data type specifies the domain of the data value. Most programming languages support a few well-known base data types: character, real, integer and boolean. More complex types, such as arrays, records and sets, may be constructed from the base types. The type also identifies the operations which may be performed over its domain. The type domains, the operations and any rules for conversion or coercion between data types represent a type system. Type checking uses the semantics of the type system to ensure correct interpretation and meaningful manipulation of the data. (Aho, Sethi, and Ullman, 1986, p.343-347)

B. NUMERIC DATA TYPES AND UNITS OF MEASURE

Type checking ensures that expressions are consistent with the typing system. For example, only numeric data values (real or integer) may be operated on by the arithmetic operators of addition, subtraction, multiplication and division. This increases the security of the programming language by preventing meaningless or incorrect operations from being performed (Clemence, 1987, p. 5).

Current typing systems provide only a limited set of numeric data types. Only real and integer types are typically supported and most systems provide rules for conversion or coercion between the two types. However, numbers by themselves have no meaning. In the real world, numbers represent the quantification of some event or characteristic.

To have meaning, they must be associated with a unit of measure (Clemence, 1987, p. 8). The unit of measure provides a standard by which numbers may be compared and gives semantic value to the number.

Including a unit of measure with a numeric value in a data type increases the reliability and readability of mathematical calculations (Karr and Loveman, 1978, p. 386). Such a data type would also increase security of the language itself (Clemence, 1987). Gehani (1977), House (1983), Karr and Loveman (1978) have all proposed programming languages with units of measure. An extension of the PASCAL language to include units has been implemented by Dreiheller, Moerschbacker, and Mohr (1986). Clemence and Bradley contend that units alone do not increase security and recommend including the semantics of quantity and concept (Bradley and Clemence, 1987). This idea is also applicable to numeric data in relational database systems.

C. DIMENSIONAL ANALYSIS

1. Units of Measure

Each numeric data value represents the measure of some dimension in a particular unit of measure. While every unit of measure is associated with a specific dimension, each dimension may have several units of measure depending on the system of measurement being used. Each system recognizes certain fundamental dimensions with a base unit of measure and possibly several other units. Units can be converted within and across systems of measure according to specific laws of conversion. For example, in the standard metric system there are seven fundamental dimensions (Beyer, 1987):

Dimension	Base Unit	Other Units
Length	Meter	Kilometer, Centimeter
Mass	Kilogram	Grams, Milligrams
Time	Second	Hour, Minute
Electric Current	Ampere	
Temperature	Deg(Kelvin)	Deg(Celsius)
Luminous Intensity	Candela	
Amount of Substance	Mole	

To this we add the fundamental dimension of currency measured in dollars. There are also dimensionless numbers, such as ratios, which have no units and may be considered measures of a null dimension. This gives nine fundamental dimensions--length, mass, time, electric current, temperature, luminous

intensity, amount of substance, currency and the null dimension. There are also derived dimensions and associated units of measure. A derived dimension is obtained as a product of two or more fundamental dimensions or their inverses (Bhargava, 1990, p. 5). For example, volume is derived by multiplying length times length times length.

Units of measure are also fundamental or derived according to which dimension they measure. Thus, given a value's unit of measure we can easily determine the dimension. Each unit of measure is associated with only one dimension. For this reason, the distinction between the units of measure and the dimension being measured is often ignored. However, since each dimension may have many associated units, we retain the distinction to allow a check of dimensional consistency regardless of the units being used.

2. Dimensional Consistency

The addition of dimension and unit of measure to numeric data types provides a more rigorous and secure typing system. Type checking becomes a problem in dimensional analysis. Not only the consistency of types is checked but consistency of dimensions is verified as well. The laws of dimensional consistency are shown below (House, 1983, p. 366):

1. Values with similar dimensions may be added or subtracted, yielding a value possessing the same dimension.
2. Values with dissimilar or similar dimensions may be multiplied. The dimension of the result is the product of the dimensions.
3. Values with dissimilar or similar dimensions may be divided. The resulting dimension is the ratio of the original dimensions.
4. No other operations are allowed.

Practical application of Rule 1. requires that the units of measure be the same. If the values have the same dimension, the units of measure will differ only by a scalar conversion factor. Units that can be converted to one another are said to commensurate. The conversion of commensurate units must take place before the addition or subtraction of the values.

Checking an expression for dimensional consistency is essentially a problem of symbolic arithmetic. The arithmetic operations are performed on the non-numeric symbols of dimension and units of measure. For example, the expression $a + (b * c)$ might become length + (time * speed) or length + (time * [length/time]). According to Rule 2, dissimilar dimensions (time and length/time) may be multiplied with the resulting dimension equal to the product of the dimensions. Symbolically this multiplication results in a dimension of length. Rule 1 allows a length to be added to a length. The

expression is found to be dimensionally consistent and gives a result that the expression is of dimension length. If the expression is on the right side of an assignment, $X := a + (b*c)$, the object represented by X must be a number of dimension length to maintain consistency.

D. THE SEMANTICS OF QUANTITY AND CONCEPT

Now that we have seen how dimensions and units are manipulated we can extend the numeric data type to include additional information.

Unit of measurement [dimension] alone is insufficient to convey information accurately to someone else. Scientific observation requires two kinds of description: a quantitative description so that the observed phenomenon can be distinguished from other phenomena; and a unit of measurement to distinguish quantitatively similar occurrences of different magnitude. (Clemence, 1987, p. 11)

While a dimension is an abstract concept used to evaluate expressions and units of measure, a quantity is the actual attribute being measured. Numerical values of the same magnitude are not equal unless they describe the same quantity and measure the same dimension (Clemence, 1987, p. 14). The quantities *LENGTH*, *WIDTH* and *ALTITUDE* all are represented by the fundamental dimension of length but clearly represent different measurable phenomena.

Like dimensions, quantities may be either fundamental or derived. A fundamental quantity can not be expressed as a combination of quantities. A derived quantity is obtained by combining fundamental quantities. For example, the quantity AREA can be derived from the quantities LENGTH and WIDTH. The rules for combining quantities and checking consistency are the same as for dimensions:

1. Similar quantities may be added or subtracted, yielding a value possessing the same quantity. Dissimilar quantities may not be added or subtracted.
2. Dissimilar or similar quantities may be multiplied. The quantity of the result is the product of the quantities.
3. Dissimilar or similar quantities may be divided. The resulting quantity is the ratio of the original quantities.
4. No other operations are allowed.

These rules are applied in addition to the rules for dimensional arithmetic and consistency. Failure to meet any rule causes an inconsistency. Thus, an expression specifying the addition of LENGTH (measured in feet) to a WIDTH (measured in meters) would be consistent under the rules of dimensional arithmetic. (Feet and meters both measure the fundamental dimension of length and are therefore commensurate.) However, since the quantities are not equal, the expression is inconsistent under the rules of quantity arithmetic. "The operations of addition and subtraction, and the use of

relational operators (=, <>, <=, >=), are meaningful only when their operands can be reduced to a common quantity and dimension (Clemence, 1987, p. 14)."

Quantity and dimension ensure that values are associated with a measurable attribute and have a scale of reference in a measurement system. Quantities in turn are attributable to an object, or the occurrence of some event, in the real world. This object/event is called a *concept* (Bradley and Clemence, 1987, p. 4). Each concept has an associated set of quantities representing measurable attributes of the object. For example, the concept *Crate* has the attributes *LENGTH*, *WIDTH*, *HEIGHT*, and *WEIGHT*. Unlike quantities and dimensions, there are no derived concepts. Concepts cannot be multiplied or divided.

E. AN EXTENDED NUMERIC DATA TYPE

Armed with the semantics of concept, quantity and dimension, we can now define an extended numeric data type. The type consists of two components, a *value description* and a *semantic description*. The *value description* consists of the number and its unit of measure. Only one unit of measure may be associated with each number. The dimension of the value is determined by the unit of measure. The *semantic description*

consists of a quantity and a concept possessing that quantity.
Our extended data type is shown in Figure 2-1.

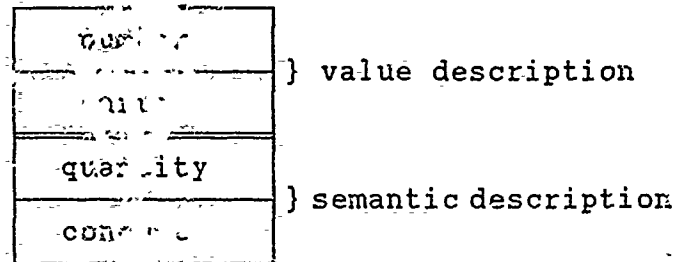


Figure 2-1. Numeric Data Type With Semantics

This abstract numeric data type can now be applied to the relational data model. In the next chapter, we present the relation as an abstract data type and demonstrate how the semantics of numeric valued data can benefit relational operations.

III. NUMERIC DATA IN RELATIONAL DATABASES

The semantics of numeric data are easily represented in the relational database environment. In this chapter we discuss the relational database as a collection of two dimensional tables where rows represent unique entities and columns represent attributes. Next we discuss three fundamental operations with relations: projection, selection and joining. Finally, we apply the semantics of concept, quantity and unit of measure to the relational model and demonstrate their use in relational operations.

A. RELATIONAL DATA DEFINITION

A relational database is a body of information stored in two-dimensional tables, or relations. Each relation consists of one or more rows, called tuples. Each tuple represents an instance of an object, or entity, about which data is collected. A relation has the properties shown below (Kroenke and Lolan, 1988, p 297):

1. The entries in the relation are single-valued. Repeating groups and arrays are not allowed.
2. The entries in any column are all of the same data type, or domain. Each column has a unique name and the order is immaterial. Columns of a relation are called attributes.
3. No two rows, or tuples, in the relation are identical and the order of rows is insignificant.

It may be helpful to think of a relation as a complex data type composed of attributes defined over a set of base data types. Commercially available relational database systems support only a limited set of base data types. For example, ORACLE provides a NUMBER type, a CHAR(i) VAR type for character strings, a DATE and a TIME type. The NUMBER type provides facilities for handling real and integer numeric data values. A set of functions and operations is specified for each domain type. More advanced systems allow abstract data types to be defined for domains and allow operations to be defined on them (Osborn and Heaven, 1986, p. 359). These systems are relatively complex and normally are designed to support specialized applications such as computer-aided design. Object-oriented DBMS also provide more robust types although such systems are still largely experimental and inefficient.

The description of the structure of a relation and the specification of its attribute domains makes up a relational

schema. The schema identifies the name of the relation, the names and domains of its attributes, and the generalized format or structure of the relation. It is in the schema that the semantics of concept, quantity and units of measure will be specified.

Relational database terminology is summarized in Figure 3-1.

Term	Meaning
Relation	Two-dimensional table.
Attribute	Column in a relation.
Domain	Set of values an attribute can have.
Tuple	Row in a relation.
Relational Schema	The structure and domain constraints of a relation.

Figure 3-1. Relational Database Terminology

B. RELATIONAL DATA MANIPULATION

The power of the relational data model lies in its flexibility and ease of use. Storing data in tables is intuitive and easily understood by most users. Relationships between tables, represented by common data values, are recognizable and readily established. Finally, because the structure is so simple and elegant only a small set of operations is needed to process it. There are three

fundamental operations used to process relations (Kroenke and Dolan, 1988, p. 313-317):

1. Projection selects specified attributes from a relation to form a new relation.
2. Selection creates a new relation by selecting rows that satisfy certain conditions.
3. Join creates a new relation from the rows in two or more relations that have attributes satisfying one or more conditions. Join is a Cartesian product of relations coupled with projection and/or selection.

Operations on relations are specified using a data manipulation language. Most commercially available relational database systems support ANSI SQL (Structured Query Language) or an extended version of the standard. The nonprocedural nature of the language allows easy access to the data without requiring knowledge of the mathematical foundations of the relational model. In fact, a single SQL construct performs all three of the fundamental relational operations. The basic syntax of the SELECT statement is:

```
SELECT [ALL | DISTINCT] select-list
FROM {{table-name | view-name}[correlation-name]},...
[WHERE search-condition];
```

Additional clauses may specify ordering or grouping, but the syntax shown is sufficient to illustrate the three relational operations.

1. Projection

The projection operation selects specific attributes, or columns, from a relation. The result is a logical relation containing the selected attributes. In the SELECT statement, the attributes named in the select-list identify the columns to be projected from one relation to the relation being formed. The FROM clause specifies the relation being used. For example, consider a relation containing distances between U.S. cities (Figure 3-2).

Origin	Destination	Mileage
New York	Seattle	2408
New York	Chicago	713
Chicago	Los Angeles	1745
Los Angeles	Denver	831
Los Angeles	New York	2451
New York	Atlanta	748

Figure 3-2. US_ROUTES Relation

The projection of the relation US_ROUTES on the attribute ORIGIN is specified by the statement:

```
SELECT origin
FROM us_routes;
```

The result of the projection is shown in Figure 3-3. Note that the original relation US_ROUTES contains six tuples, or rows. The result of the projection contains only three. Since the result of a projection is a relation, the third property of relations prevents duplicate rows from appearing in a projection. Three tuples were eliminated after the projection was done.

Origin
New York
Chicago
Los Angeles

Figure 3-3. Projection of US_ROUTES on ORIGIN

2. Selection

Projection identifies the attributes to be contained in the new relation. Selection identifies the tuples to be included. Rows are specified by the search-condition in the WHERE clause of the SELECT statement. The search-condition describes a simple or compound predicate that evaluates as true, false or unknown about a given row (Viescas, 1989, p. 63). If the condition is true when applied to a row, the row

is included in the resulting relation. For example, the statement

```
SELECT origin, destination, mileage
FROM us_routes
WHERE mileage > 1000;
```

results in the relation shown in Figure 3-4.

Origin	Destination	Mileage
New York	Seattle	2408
Chicago	Los Angeles	1745
Los Angeles	New York	2451

Figure 3-4. Selection of US_ROUTES where MILEAGE > 1000

3. Join

The relational data model derives much of its power from the ability to combine information from, or join, two or more relations. A join is specified by including more than one relation in the FROM clause of the SELECT statement. The join operation requires three logical steps. First, the Cartesian product of the specified relations is determined. This results in a logical relation which pairs all the tuples of the first relation with all the tuples of the second relation paired with all the tuples of the third relation and so on. A selection operation is then performed to eliminate some of the tuples. (In practice, the selection takes place before the Cartesian product is formed. This eliminates many

rows from the product and saves time.) Finally, a projection may take place to remove certain attributes.

To illustrate, consider a join between the US_ROUTES relation and a relation containing data on international routes (Figure 3-5). A SELECT statement with the two tables listed in the FROM clause and with no selection criteria specified in a WHERE clause will return the complete Cartesian product. This product will have six attributes (Origin, Destination, Mileage, Source, Terminal, Distance) and will contain 54 tuples.

Source	Terminal	Distance
New York	London	5589
Seattle	Tokyo	7714
London	Berlin	933
London	Rome	1436
London	Paris	346
London	Moscow	2504
Tokyo	Singapore	5317
Tokyo	Peking	2100
New York	Rio De Janeiro	7733

Figure 3-5. INTERNATL_ROUTES Relation

To obtain a more meaningful result, we need to include join criteria in the WHERE clause of the SELECT statement.

This eliminates rows from the Cartesian product that contain superfluous information. For example, including

WHERE US_ROUTES.DESTINATION = INTERNATL_ROUTES.SOURCE
in the SELECT statement results in the relation shown in Figure 3-6.

Origin	Destination	Mileage	Source	Terminal	Distance
New York	Seattle	2408	Seattle	Tokyo	7714
LA	New York	2451	New York	London	5589
LA	New York	2451	New York	Rio	7733

Figure 3-6. A Join Between US_ROUTES and INTERNATL_ROUTES

Because this join includes only rows that have matching attribute values from both tables, it is called an Inner Join. If we do not provide a select-list, or specify that all columns be selected, an inner join returns the matched data values twice (once in the DESTINATION column and one in the SOURCE column). We can eliminate this duplication by select only one of the columns identified in the join criteria:

```
SELECT origin, destination, distance, terminal, mileage
FROM us_routes, internatl_routes
WHERE us_routes.destination =
      internatl_routes.source;
```

This results in a relation that does not contain the duplicate attribute values (Figure 3-7). This is a Natural Inner Join.

Origin	Destination	Mileage	Terminal	Distance
New York	Seattle	2408	Tokyo	7714
LA	New York	2451	London	5589
LA	New York	2451	Rio	7733

Figure 3-7. Natural Inner Join

Relations also can be joined using more complex criteria. The search_condition of the WHERE clause may include multiple predicates of varying degrees of complexity. Our focus is on predicates involving numeric valued data attributes. These may be as simple as a relational comparison between two attributes or between an attribute and a numeric constant. Predicates also may include more complex mathematical expressions combining many attributes, constants and relational operators.

For example, we formed a natural inner join of the relations US_ROUTES (Figure 3-2) and INTERNATL_ROUTES (Figure 3-5). The resulting relation (Figure 3-7) contained information about U.S. shipping routes with connections to international cities. We can go from New York to Seattle and then on to Tokyo. If we start in Los Angeles we can reach

either London or Rio De Janeiro via New York. Now suppose we want to limit our trip to less than 9,000 miles. The SELECT statement

```
SELECT origin, destination, terminal
FROM us_routes, internatl_routes
WHERE us_routes.destination = internatl.source AND
      us_routes.mileage + internatl.distance < 9000;
```

returns the relation shown in Figure 3-8. Only one of the three tuples in the original join meets the second part of our join criteria. The mileage from Los Angeles to New York is 2451 and the distance from New York to London is 5589, giving a total trip of only 8040 miles. (Los Angeles to Rio: 2451 + 7733 = 10184; New York to Tokyo: 2408 + 7714 = 10122)

Origin	Destination	Terminal
Los Angeles	New York	London

Figure 3.8. Join Using a Compound Search Condition

This result depends on several assumptions.. First, we assume that US_ROUTES and INTERNATL_ROUTES represent similar entities or concepts. Next, we assume that the attributes MILEAGE and DISTANCE both represent identical quantities. Finally, we treat the values contained in the MILEAGE and DISTANCE columns as if they were derived from the same unit of measure, miles.

What if the values in one table are measured in miles, while kilometers are used in the other? The result of our query then has no meaning, or at least may not answer the question being asked. The computer manipulates only the numbers not the semantics. Dimensional consistency of the query is left to the user. This responsibility of checking semantic consistency can be shifted to the database system, however, if the semantics are stored along with the data values.

C. ADDING CONCEPT, QUANTITY AND UNITS TO THE RELATIONAL MODEL

Application of our extended numeric data type to the relational model is fairly straightforward. A concept represents a type of object or event in the real world which has one or more measurable attributes or quantities. Thus, a concept specifies an entity type. A relation is simply an aggregation of specific instances of entities (tuples). Since each tuple represents an instance of the same entity type, one concept is sufficient for each relation. A concept may be associated with more than one relation but a relation represents only one concept. For example, two relations, one containing information about automobiles and the other data about trucks, may both represent the single concept Vehicle.

As defined earlier, a quantity is a measurable attribute or characteristic of an object. This relates directly to the column in the relational model. Every column in a relation represents a unique characteristic or attribute. To allow for meaningful use of data, each column defined over a numeric domain should have an associated quantity and a unit of measure. The second property of relations requires that all data values in a column be of the same data type. Therefore, only one quantity and one unit of measure apply to all data in the column. The same quantity, however, may be associated with multiple attributes both within a single relation and across other relations.

Returning to our example, suppose the semantics of concept, quantity and unit of measure are added to the relations, US_ROUTES and INTERNATL_ROUTES. The schema, or structure, of the relations are shown below:

```

US_ROUTES (Concept = AIR_ROUTE)
  Origin      Character String
  Destination  Character String
  Mileage      Number (Quantity = Distance,
                      Units = Miles)

INTERNATL_ROUTES (Concept = AIR_ROUTE)
  Source      Character String
  Terminal     Character String
  Distance     Number (Quantity = Distance,
                      Units = Kilometers)

```

We now can check our earlier query

```
SELECT origin, destination, terminal
FROM us_routes, internatl_routes
WHERE us_routes.destination = internatl_routes.source AND
      us_routes.mileage + internatl_routes.distance < 9000
```

for semantic and dimensional consistency. We focus on join criteria specified in the WHERE clause, particularly the portion dealing with numeric data attributes. The first portion of the predicate specifies a relational comparison of non-numeric attributes. This is outside the scope of our numerical dimensional check.

The second predicate is a mathematical expression involving two numeric attributes (us_routes.mileage, internatl_routes.distance) and one numeric constant (9000). For the time being, we will assume the data type of the constant specifies Air_Route as its concept, Distance as the quantity and Miles as the unit of measure. Both operands on the left side of the expression are associated with the same concept, Air_Route, and quantity, Distance. The units of measure are different. Both Miles and Kilometers measure the fundamental dimension of length, however, and are therefore commensurate. Thus only a conversion factor (0.62137

Miles/Kilometer) need be applied to the expression to make it dimensionally consistent. The corrected query

```
SELECT origin, destination, terminal
FROM us_routes, internatl_routes
WHERE us_routes.destination = internatl_routes.source AND
      us_routes.mileage +
      0.62137*internatl_routes.distance < 9000;
```

returns the relation in Figure 3-9. Note that all three of the trips contained in the original join (Figure 3-7) are included in the new relation. All three have a total distance under 9000 miles, a fact we could not determine without performing a dimensional check on the query and applying the necessary conversion factor.

Origin	Destination	Terminal
New York	Seattle	Tokyo
Los Angeles	New York	London
Los Angeles	New York	Rio De Janeiro

Figure 3-9. Join After Check for Semantic Consistency

In the next two chapters, we discuss the implementation of a type system shell for SQL which allows the database system to perform the dimensional analysis and apply the conversion automatically.

IV. REPRESENTING THE SEMANTICS IN THE DATA DICTIONARY

Implementation of an SQL type checking shell involves two primary functions. First, the semantics of concept, quantity and unit of measure must be incorporated in the relational database. This is accomplished in the design of the data dictionary, which contains the relational schema for the database. Also included in the data dictionary are the scalar factors needed for conversion of commensurate units of measure.

In this chapter we discuss incorporation of the semantics into the data dictionary. First, we propose a prime number encoding scheme that allows numeric representation of the semantics. This transforms the problem of symbolic semantic manipulation to a much simpler numeric problem. Finally, we discuss the data dictionary tables needed to represent the semantics of concept, quantity and unit of measure.

A. REPRESENTATION OF SEMANTICS

Semantic manipulation to support type checking involves three primary operations (Bhargava, 1990, p. 5):

1. Verification of equivalence. For example, $\text{kg} \cdot \text{m/s}$ is equivalent to $(1/\text{s}) \cdot \text{m} \cdot \text{kg}$.
2. Simplification. For example, $(\text{m/s}^2) \cdot \text{s} + \text{m/s}$ simplifies to m/s .
3. Transformation of Commensurate Units.

The first two operations are related and are the most complicated to automate symbolically. Bhargava proposes a prime encoding scheme which transforms the problem from one of symbolic arithmetic to a much simpler numeric problem.

1. Verification of Equivalence

The semantics of quantity and dimension are either fundamental or derived. Verification of the equivalence of fundamental semantics is easily implemented symbolically. There are no arithmetic operations required. If the semantics are represented by identical character strings, they are equivalent. If the string representations are different, the semantics are not equal.

Derived semantics pose a more difficult problem of verification. Equivalent semantics can be derived by combining fundamental semantics in different ways. For example, the dimension $(1/\text{time}) \cdot \text{length}$ is equivalent to $\text{length}/\text{time}$. A simple comparison of the symbolic representation of the two dimensions does not readily reveal

this equivalence. In fact, more complex derived semantics can easily become a significant problem of symbolic mathematics.

While there are programs capable of evaluating symbolic expressions, the problem can be transformed into one of simple numeric arithmetic by representing fundamental dimensions with prime numbers. The product (or quotient) of prime numbers is always unique. No matter how many ways fundamental semantics are combined, two combinations resulting in the same numeric value are equivalent. Semantic equality is equivalent to numeric equality. (Bhargava, 1990)

The nine fundamental dimensions and their prime number representations are shown in Figure 4-1. Fundamental quantities can be represented in the same manner. Notice that the null dimension (for use with unitless numbers) is represented by the number one (1), an identity for multiplication.

The numeric representation of derived semantics is obtained by arithmetic combination of the prime number representations of the fundamental semantics. For example, the dimension volume is derived from the fundamental dimension of length according to the formula $\text{length} \times \text{length} \times \text{length}$ or $(\text{length})^3$. Substituting the prime number representation for length into either of the formulas and carrying out the arithmetic operations gives us the numeric representation for

volume: $3*3*3 = (3)^3 = 27$. This representation can then be used to derive the representation of the dimension density:
density = mass / volume $[(5)/(27) = 0.185185185]$.

Fundamental Dimension	Prime Number Representation
null dimension	1
currency	2
length	3
mass	5
time	7
electric current	11
temperature	13
luminous intensity	17
amount of substance	19

Figure 4-1. Representation of Fundamental Dimensions

Dimensions derived through division can result in floating point numeric representations. Such representations complicate verification of equivalence. The limitation on the number of significant digits which can be represented on a given computer gives rise to precision errors in floating point arithmetic. The order in which values are multiplied or divided can affect the final internal representation of the result. Thus, equivalent semantics derived via a different order of multiplication and division may not have identical numeric representations.

We can avoid floating point precision errors by eliminating the division of the semantic representations. Each quantity and dimension can be represented by two integer

values, one for the prime product in the numerator (d_1) and the other for the denominator (d_2). Equivalence of two dimensions can be verified by cross multiplying their d_1 's and d_2 's and checking for numeric equality. Figure 4-2 shows a revised encoding of the fundamental dimensions and the derived dimensions of volume and density.

Fundamental Dimension	Prime Number Representation	
	d_1	d_2
null dimension	1	1
currency	2	1
length	3	1
mass	5	1
time	7	1
electric current	11	1
temperature	13	1
luminous intensity	17	1
amount of substance	19	1
volume	9	1
density	5	9

Figure 4-2. Two-part Representation of Dimensions

2. Simplification

Simplification of semantic expressions is now straightforward. The laws of consistency are readily implemented since semantic equality is easily tested. Multiplication of semantic terms requires multiplication of the d_1 's and d_2 's. Division requires cross multiplication of d_1 's and d_2 's. For addition and subtraction, check semantic equivalence. (Bhargava, 1990, p. 10-11)

B. DATA DICTIONARY REPRESENTATION

Information on the design and structure of a database is normally stored in a central repository known as the data dictionary. Because the information contained in the dictionary describes other data, it is often termed metadata. In relational systems, metadata typically is stored in relations, or dictionary tables. For example, ORACLE uses 39 dictionary tables to store metadata in its data dictionary. Information contained in the tables includes system administration data as well as data about user-defined relations and their attributes.

Data dictionaries are classified as either passive or active in nature (Dolk and Kirsch, 1987, p. 49). A passive dictionary is similar to a language dictionary found in the library. A passive dictionary merely documents metadata. The database system does not rely on the dictionary to control and process data. In fact, the system may obtain its metadata from other sources. On the other hand, an active data dictionary is the sole source of metadata for the DBMS. The dictionary must be accessed for each process or transaction allowing for implementation of much more powerful control mechanisms. There is a performance penalty, however, since the system must access the dictionary for every transaction.

Implementation of a type analysis shell requires an active data dictionary. In addition to the standard dictionary tables, three tables are needed to contain the semantic information associated with numeric valued data. One table holds the data needed for automatic conversion of commensurate units. The second table associates a concept with each relation. Finally, a third dictionary table contains the quantity and unit of measure associated with each numeric valued attribute or column.

1. The UNITS Dictionary Table

The UNITS dictionary table is used to identify commensurate units and the conversion factors needed to make units equivalent. Recall that each unit of measure is associated with a dimension. Commensurate units are associated with the same dimension and differ only by a scalar conversion factor. Thus, our UNITS dictionary table must contain the unit of measure, its associated dimension and the conversion factors. A relational schema for the table is shown in Figure 4-3.

<u>Unit</u>	Dimension	d1	d2	Conversion Factor
-------------	-----------	----	----	-------------------

Figure 4-3. UNITS Dictionary Table

The unit and dimension attributes are stored as character strings. The d_1 and d_2 columns contain the numeric

representation of the dimension. Commensurate units are identified by cross multiplying the d_1 's and d_2 's and checking for numeric equality. The conversion factors are stored as character strings to allow insertion into the SQL statement during preprocessing.

Notice there is only one conversion factor per unit of measure. The system recognizes a specific base unit for each of the nine fundamental dimensions (see Figure 4-4). All other units are defined in terms of the base units.

Dimension	d1	d2	Base Unit
(null)	1	1	(unitless)
currency	2	1	Dollar(US)
length	3	1	Meter
mass	5	1	Kilogram
time	7	1	Second
electric current	11	1	Ampere
temperature	13	1	Degrees(Kelvin)
luminous intensity	17	1	Candela
amount of substance	19	1	Mole

Figure 4-4. Base Units of Measure

The factor stored in the dictionary allows conversion of units to the base. Thus, commensurate units are converted into identical base units for use in SQL expressions. Extensibility is supported by allowing the user to define units using the base units or any unit of measure already defined.

2. The CONCEPTS Dictionary Table

The CONCEPTS dictionary table associates a semantic concept with each relation. Concepts cannot be derived through multiplication or division, therefore a numeric representation is not necessary. The relational schema for the CONCEPTS dictionary table is shown in Figure 4-5. Both attributes are stored as character strings.

<u>Table-Name</u>	Concept
-------------------	---------

Figure 4-5. CONCEPTS Dictionary Table

3. The QUANTITIES Dictionary Table

The QUANTITIES dictionary table contains the quantity and unit of measure associated with each numeric valued attribute. Like dimensions, quantities can be either fundamental or derived from other quantities. The same numeric scheme is used to represent quantities in the data dictionary. The relational schema is shown in Figure 4-6.

<u>Table-name</u>	<u>Column-name</u>	Quantity	q1	q2	Units
-------------------	--------------------	----------	----	----	-------

Figure 4-6. QUANTITIES Dictionary Table

Once the semantics have been added to the database, we must provide a mechanism for their use in the type checking of

expressions. In the next chapter we discuss implementation of a preprocessor which evaluates SQL statements for semantic consistency before they reach the DBMS for processing.

V. EVALUATING SQL EXPRESSIONS

Storing semantic information in a data dictionary allows us to check automatically for semantic consistency of SQL SELECT statements. This is easily accomplished by an SQL preprocessor before the statement is passed to the DBMS for routine processing. The preprocessor is concerned only with SELECT statements. All other SQL statements are passed directly to the DBMS for routine processing. The preprocessor must perform three primary operations:

1. Retrieve the required semantic information from the data dictionary tables.
2. Evaluate the search conditions in the WHERE and HAVING clauses for semantic consistency.
3. Convert commensurate units into a common unit of measure.

In this chapter, we discuss implementation of the SQL preprocessor. The discussion focuses on verification of quantity consistency and conversion of commensurate units. Evaluating concept consistency is more difficult and is discussed in the next chapter.

A. ACCESSING THE DATA DICTIONARY

The data dictionary tables are updated each time a CREATE TABLE or CREATE VIEW command is entered by the user. The user must define a concept for each table created and a quantity and unit of measure for each numeric-valued attribute. Quantities may be defined in terms of known quantities or may be fundamental. Units of measure must be defined in terms of known units of measure.

If a table or view is dropped from the database, the corresponding rows in the CONCEPTS and QUANTITIES dictionary are deleted. Similarly, the dictionary tables are updated each time a database table is altered by adding or deleting an attribute.

The FROM clause of the SELECT statement identifies the relation or relations to be queried by the statement. The table-names specified in the FROM clause can be used to generate three data dictionary queries described below. The results of the three data dictionary queries should be stored in temporary data structures in computer memory. This minimizes the number of database accesses and speeds preprocessing.

The first query retrieves the concepts associated with each table from the CONCEPTS dictionary table. Entries are made in the CONCEPTS table only if a relation contains numeric

valued attributes. Thus, if there are no entries in the CONCEPTS table corresponding to the tables named in the FROM clause, there will be no numeric valued attributes involved in the query and no need for further preprocessing. The SELECT statement can be passed directly to the DBMS for normal query processing.

The second query to the data dictionary, retrieves the quantities and units of measure from the QUANTITIES table. Entries from the QUANTITIES table identify the numeric valued attributes in each table and the quantity and units of measure associated with each attribute. The results of this query subsequently are used to retrieve the appropriate conversion factors from the UNITS table. We now have all the information needed to evaluate search conditions for semantic consistency.

B. EVALUATING SEARCH CONDITIONS

Search conditions are used in the WHERE clause and in the HAVING clause of the SELECT statement, as shown below

(Viescas, 1989, p. 64):

```
SELECT [ALL | DISTINCT] select-list
FROM {{table-name | view-name} [correlation-name]},...
[WHERE search-condition]
[GROUP BY {column-name | column-number},...]
[HAVING search-condition]
[{{INTERSECT | MINUS | UNION [ALL]} select_statement]
[[ORDER BY {{column-name | column-number}[ASC
|DESC]],...]]
[FOR UPDATE OF {column-name},...]]
```

The search condition identifies a simple or compound predicate that is true, false, or unknown about a given row. The condition defines which rows will appear in the resulting logical table. If the condition is true when applied to the row, the row is included in the result table. Application of a search condition does not change the value of the attributes contained in a row. (Viescas, 1989, p. 63) Syntax for the search condition is shown here:

```
[NOT] {predicate | (search-condition)}  
[AND | OR] [NOT] {predicate | (search-condition)}]...
```

The standard Boolean operators--AND, OR, and NOT--combine individual predicates into compound search conditions. The Boolean operators accept only three values--true, false, and unknown--and are not subject to the rules of semantic consistency applied to numeric expressions.

There are two types of predicates which involve numeric valued attributes and require semantic evaluation. The first compares an expression to a range of values. The second compares the values of two expressions using standard relational operators.

The BETWEEN predicate compares a value with a range of values.

```
expression [NOT] BETWEEN expression AND expression
```

The data types of all expressions must be compatible and each expression must be semantically consistent.

The comparison predicate compares values of two expressions using the standard relational operators. The semantics of the first expression must be the same as those of second expression.

`expression { = | <> | > | < | >= | <= } expression`

To verify the semantic consistency of the BETWEEN or the comparison predicate, we must first evaluate their component expressions. Numeric-valued expressions obey the following syntax:

`[+|-] { (expression) | literal | column-name}
[+|-|*|/] { (expression) | literal | column-name}]...`

Expressions are evaluated for correctness according to the laws of semantic consistency. This is easily accomplished by first converting the expression to postfix form and then conducting operations on a simple stack. As each operand is identified, its semantics are retrieved from the temporary dictionary tables created earlier. The semantics are then concatenated to the operand token and pushed onto the stack.

1. Quantity Consistency

As the quantity of each operand is retrieved, its two-part prime product representation (q_1, q_2) is pushed onto the stack. When an arithmetic operator is read, its operands will be the two top elements of the stack. These are popped from the stack, the indicated operation is performed and the resulting quantity is returned to the stack.

a. Multiplication and Division

When a multiplication operator is encountered, the top two quantities $(q_1, q_2)^2$ and $(q_1, q_2)^1$ are popped from the stack. The two q_1 's are multiplied to give the resulting representation $(q_1)'$. The product of the q_2 's gives the resulting $(q_2)'$ value. The new $(q_1, q_2)'$ is returned to the stack.

Division requires a similar though slightly more complicated procedure. The first quantity popped from the stack $(q_1, q_2)^2$ represents the divisor, the second quantity $(q_1, q_2)^1$, the dividend. The resulting quantity $(q_1, q_2)'$ is determined by cross multiplying:

$$(q_1)' = (q_2)^2 * (q_1)^1 \quad \text{and} \quad (q_2)' = (q_1)^2 * (q_2)^1$$

b. Addition and Subtraction

Addition and subtraction require that both operands have the same quantity. If $(q_1, q_2)^2$ and $(q_1, q_2)^1$ differ, the

expression is inconsistent. The quantity resulting from an addition or subtraction is the same as that of the operands $((q_1, q_2)' = (q_1, q_2)^2 = (q_1, q_2)^1)$.

Unary plus and minus operators involve only one operand. However, they do not affect the quantity or unit of measure of the operand. Thus, unary operators are ignored by the preprocessor.

c. Relational Operators

Like addition and subtraction, the relational operators (=, <>, >, <, >=, <=) require that both operands have the same quantity. The result of a relational operation is either true or false. It does not have a quantity.

2. Handling Literals

Expressions in SQL may contain numeric constants or literals. These are known to the system only at the time the query is being processed. The data dictionary holds no concept, quantity or units description for the literals. However, the semantics of these constants must be considered when evaluating the expression for consistency.

There are two ways to deal with literals: The first is to require all literals to be declared as defined constants and given a variable name. A quantity and unit of measure is assigned by the user when the constant is defined.

The second method assumes the user intends the literal to have a quantity and unit of measure consistent with the context of the expression. In this case, we must introduce a universal type, (u1,u2). It is equivalent to all other quantities for addition, subtraction and comparison. For multiplication and division, it is equivalent to a dimensionless quantity where (q1,q2) = (1,1). (Bradley and Clemence, p. 47)

C. AN EXAMPLE OF PREPROCESSING

Evaluation of an SQL SELECT statement is demonstrated in the following example. Consider the following relations:

SHIP (concept = SHIP)

SHIP_NAME

LENGTH (Quantity = length, Units = Ft)

BEAM (Quantity = width, Units = Ft)

DRAFT (Quantity = depth, Units = Ft)

HOLD_LENGTH (Quantity = length, Units = Ft)

HOLD_WIDTH (Quantity = width, Units = Ft)

HOLD_HEIGHT (Quantity = height, Units = Ft)

HATCH_AREA (Quantity = area, Units = SqFt)

CARGO_CAPACITY (Quantity = weight, Units = Tons)

CRATE (concept = CRATE)

CRATE_ID

LENGTH (Quantity = length, Units = Meters)

WIDTH (Quantity = width, Units = Meters)

HEIGHT (Quantity = height, Units = Meters)

WEIGHT (Quantity = weight, Units = Kilograms)

For a crate to be loaded onto a ship it must fit through the ship's hatch. Crates are stacked five high inside the

ship's hold. The following SELECT statement will identify those crates that can be loaded on a specific ship:

```
SELECT CRATE_ID
FROM CRATE, SHIP
WHERE SHIP_NAME = "TITANIC" AND
      CRATE.LENGTH*CRATE.WIDTH < HATCH_AREA AND
      CRATE.HEIGHT*5 < HOLD_HEIGHT;
```

The FROM clause provides the table names needed to query the data dictionary. Three temporary dictionary tables are established by the preprocessor:

CONCEPTS

<u>Table Name</u>	<u>Concept</u>
CRATE	Crate
SHIP	Ship

QUANTITIES

<u>Table-name</u>	<u>Column-name</u>	<u>Quantity</u>	<u>q1</u>	<u>q2</u>	<u>Units</u>
CRATE	LENGTH	length	2	1	Meters
CRATE	WIDTH	width	3	1	Meters
CRATE	HEIGHT	height	5	1	Meters
CRATE	WEIGHT	weight	7	1	Kilograms
SHIP	LENGTH	length	2	1	Ft
SHIP	BEAM	width	3	1	Ft
SHIP	DRAFT	depth	11	1	Ft
SHIP	HOLD_LENGTH	length	2	1	Ft
SHIP	HOLD_WIDTH	width	3	1	Ft
SHIP	HOLD_HEIGHT	height	5	1	Ft
SHIP	HATCH_AREA	area	6	1	SqFt
SHIP	CARGO_CAPACITY	weight	7	1	Tons

UNITS

<u>Units</u>	<u>Dimension</u>	<u>d₁</u>	<u>d₂</u>	<u>Conversion Factor</u>
Meters	length	3	1	
Kilograms	mass	5	1	
Ft	length	3	1	0.3048
SqFt	length*length	9	1	0.09290304
Ton	mass	5	1	(.45359/2000)

Notice that the derived quantity area is defined in terms of the quantities length and width.

The WHERE clause contains a compound search condition with three distinct predicates. The first, SHIP_NAME = "TITANIC" involves non-numeric attributes and is ignored by the preprocessor. The remaining predicates involve numeric expressions and must be evaluated for semantic consistency.

Evaluation begins by converting the expressions into postfix form.

<u>Expression</u>	<u>Postfix Representation</u>
CRATE.LENGTH*CRATE.WIDTH < HATCH_AREA	CRATE.LENGTH,CRATE.WIDTH,*,HATCH_AREA,<
CRATE.HEIGHT*5 < HOLD_HEIGHT	CRATE.HEIGHT,5,*,HOLD_HEIGHT,<

Figure 5-1 shows the contents of the postfix stack as the expressions are evaluated.

Symbol	Quantity	Oper1	Oper2	Result	Stack
CRATE.LENGTH	(2,1)				(2,1)
CRATE.WIDTH	(3,1)				(2,1)(3,1)
*		(2,1)	(3,1)	(6,1)	(6,1)
HATCH_AREA	(6,1)				(6,1)(6,1)
<		(6,1)	(6,1)	Logical	Consistent
CRATE.HEIGHT	(5,1)				(5,1)
5	(u1,u2)				(5,1)(u1,u2)
*		(5,1)	(u1,u2)	(5,1)	(5,1)
HOLD_HEIGHT	(5,1)				(5,1)(5,1)
<		(5,1)	(5,1)	Logical	Consistent

Figure 5-1. Evaluation of Postfix Expressions

Because the system has no way of converting quantities into one another, inconsistent expressions cause the SELECT

statement to be returned to the user for correction. Figure 5-2 shows the progress of the stack for:

```
SELECT CRATE-ID
FROM CRATE, SHIP
WHERE SHIP-ID = "TITANIC" AND
      CRATE.LENGTH <= CRATE.WEIGHT;
```

The expression `CRATE.LENGTH <= CRATE.WEIGHT` tries to compare an attribute representing a length to one representing a weight. This is inconsistent with respect to quantity.

Symbol	Quantity	Oper1	Oper2	Result	Stack
CRATE.LENGTH	(2,1)				(2,1)
CRATE.WEIGHT	(7,1)				(2,1)(7,1)
<=		(2,1)	(7,1)	Logical	INCONSISTENT

Figure 5-2. Stack Contents For Inconsistent Expression

D. CONVERSION OF COMMENSURATE UNITS

An expression that is consistent in quantity will also be dimensionally consistent. All that remains before the statement is passed to the DBMS is the conversion of commensurate units of measure. To simplify the conversion, all commensurate units are converted into the base units of measure. For example, both feet and inches will be converted into the base unit of measure for length, meters.

Every numeric-valued operand in an expression will have a conversion factor applied before the statement is passed to the DBMS. The conversion factor is stored in the UNITS

dictionary table as a character string which is inserted prior to the operand in the SQL statement. Conversion factors for base units are null strings and will not change the original SQL statement. In our example, the SELECT statement is modified as follows:

```
SELECT CRATE_ID
FROM CRATE, SHIP
WHERE SHIP_NAME = "TITANIC" AND
      CRATE.LENGTH*CRATE.WIDTH < 0.09290304*HATCH_AREA AND
      CRATE.HEIGHT*5 < 0.3048*HOLD_HEIGHT;
```

Literals are assigned the conversion factor of the operand preceding them in the postfix expression. The user is responsible for ensuring that the value represented by the literal is expressed in the appropriate units.

We have demonstrated a simple scheme for automatic verification of quantity consistency and conversion of commensurate units in database systems. Responsibility for ensuring dimensional consistency can now be shifted from the user to the database system. Verifying concept consistency poses a more difficult challenge, however. Assigning semantics to a relation resulting from a join also poses a problem. In the next chapter, we address these problems and introduce the concept hierarchy as a possible approach to solving them.

VI. CONCEPT HIERARCHIES

Verification of concept consistency in relational operations is a difficult problem. An SQL search condition will be consistent only if the concepts associated with each operand are identical. Since attributes inherit the concept associated with the relation, strict consistency would allow expressions to contain only attributes drawn from relations having the same concept. There is no way to combine concepts to create new concepts. Most database queries, however, involve more than one relation and most likely more than one concept. Search conditions rarely involve attributes from just one of the relations being joined.

If there is a way of relating concepts to one another, it becomes possible to coerce concept consistency by substituting related concepts when evaluating expressions. In this chapter, we introduce *concept hierarchies* as a way of determining the relationships between concepts. We begin with a discussion of the binary relationships which link database tables. These relationships can be combined into a variety of larger structures represented by directed graphs. A concept hierarchy is realized when concepts are included in the graph. This hierarchy reflects the semantic intent of the database

design and can be used to ensure queries are consistent with the design.

A. LINKING RELATIONS

In the relational data model each relation represents a collection of instances of a specific entity type. Specific instances (rows) are identified by a unique value in at least one column of the relation. This is known as the primary key for the relation. The key is used to identify each row and to keep all rows distinct.

Primary keys from one relation will often appear as columns in another relation. This indicates a relationship between the two tables. When a primary key is included as an attribute of a second relation, it is called a foreign key. A value of a foreign key refers to a row in the original relation. Referential integrity is achieved when each value (i.e., row) of the foreign key specifically refers to one and only one row of the parent relation. Foreign keys can be used to represent three primary types of relationships between tables: one-to-one, one-to-many, and many-to-many. These relationships are considered binary because they link only two entity types. Two or more binary relationships can be used to create more complicated structures.

1. One-To-One Relationships (1:1)

In a one-to-one (1:1) relationship, an instance of one relation is related to no more than one instance of another relation. This is represented by the placement of the key of either relation in the other relation. This is the simplest form of a binary relationship. (Kroenke and Dolan, 1988, p. 169-174)

Suppose a shipping crate can contain only one item. There is a one-to-one relationship between the crate and its item. Figure 6-1 shows the two relations. Primary keys are underlined. Foreign keys are marked with an asterisk. Notice that the primary key (ITEM-ID) of the ITEM relation is placed as a foreign key in the CRATE relation.

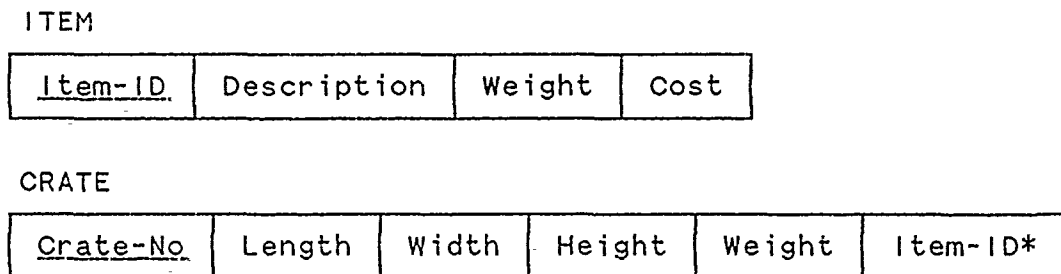


Figure 6-1. Representing a One-to-One Relationship

2. One-To-Many Relationships (1:N)

In a one-to-many (1:N) relationship, a row of one relation is related to potentially many rows of another relation. The relation on the one side of a 1:N relationship

is said to be the *parent*. The other relation is the *child*. To represent a 1:N relationship, the key of the parent relation is placed as a foreign key in the child. (Kroenke and Dolan, 1988, p. 174-178)

A 1:N relationship is illustrated in Figure 6-2. A ship can carry many crates but each crate can be on at most one ship. This relationship is represented by placing the key of the parent relation (Ship-Name) as a foreign key in the child relation.

SHIP

<u>Ship-Name</u>	Length	Beam	Draft	...
------------------	--------	------	-------	-----

CRATE

<u>Crate-No</u>	Length	...	Item-ID*	Ship-Name*
-----------------	--------	-----	----------	------------

Figure 6-2. Representing a One-To-Many Relationship

3. Many-To-Many Relationship (M:N)

A many-to-many (M:N) relationship cannot be directly represented in the relational model. In an M:N relationship, a row of one relation corresponds to many rows of another relation. A row in the second relation also corresponds to many rows of the first relation. To represent this, we need a third relation called an *intersection relation*. An intersection relation is composed of the keys of each of the

related relations. In this case the intersection relation is a child to both relations. (Kroenke and Dolan, 1988, p. 178-182)

A single ship may call at many ports and each port can be a destination for many ships. This many-to-many relationship is represented by the relations in Figure 6-3. The keys of the SHIP relation (Ship-Name) and of the PORT relation (Port-Name) are combined to form the SHIP-PORT relation. The key of this intersection relation is the combination of the keys of its parents (Ship-Name*+Port-Name*).

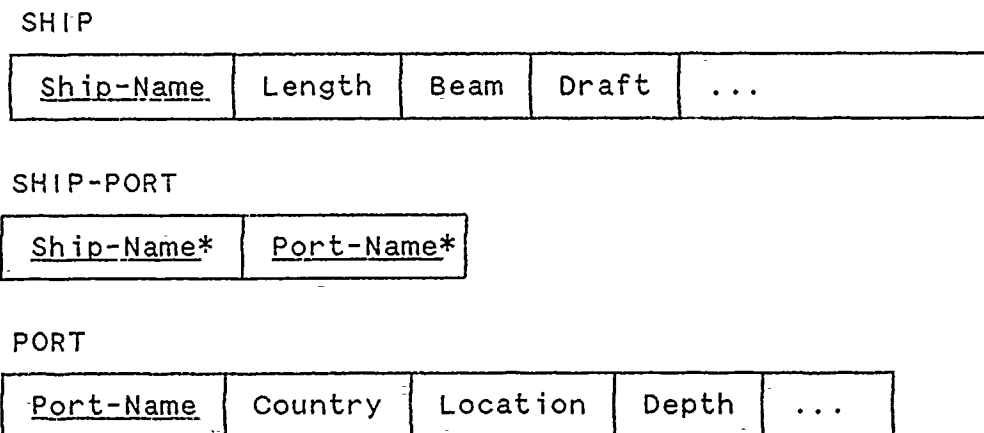


Figure 6-3. Representing a Many-To-Many Relationship

B. DIRECTED GRAPHS

Binary relationships can be combined into a variety of larger structures. It is often useful to represent these

structures graphically. A *graph* can be thought of as a set of nodes (or vertices) and arrows (or arcs) between the nodes. Each arc is specified by a pair of nodes. If the pairs of nodes that make up the arcs are ordered pairs, the graph is said to be a *directed graph*. (Tenebaum, Langsam, and Augenstein, 1990, p. 503) Since a binary relationship always identifies a parent and child, it can always be represented as a directed graph.

If the primary key of a relation is included as a foreign key in a second relation, the first relation will be a parent to the second. Thus, identifying the foreign keys in a table will identify the table's parent relations. This, in effect, orders pairs of relations and provides a means of building a directed graph. Figure 6-4 shows a directed graph representing the relationships between the ITEM, CRATE, SHIP and PORT relations.

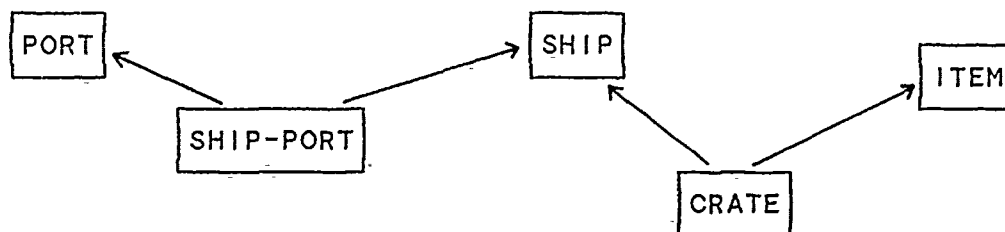


Figure 6-4. Directed Graph of Binary Relationships

Each relation represents a node in the graph. Links between relations are represented as arrows between nodes.

C. CONCEPT HIERARCHIES

Each relation in the directed graph will have an associated concept. If these concepts are substituted at each node of the graph (Figure 6-5) we establish a *concept hierarchy* representing the relationships between concepts. The term hierarchy is used here rather loosely to convey the ordered parent-child structure of the graph. Intersection relations used to represent M:N relationships will not normally have a unique concept, so for the time being, we simply label the intersect nodes. To complete the hierarchy we include a "dummy" node or universal concept (*) at the root of the graph. All other nodes are descended from this root.

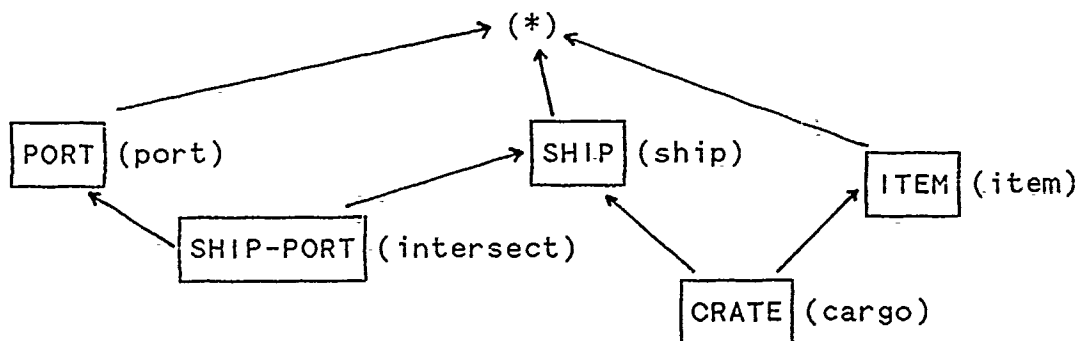


Figure 6-5. Concept Hierarchy

Since the concept hierarchy is based on the links between relations, it reflects the semantic intent of the database design. Thus, the graph can be used to ensure database queries are semantically consistent with the design of the

database. Expressions are evaluated by first determining the concept associated with each of the operands. If the concepts differ, we then begin at the lowest occurrence of each concept on the concept graph and search upward until a common node is reached. If the lowest common concept is (*), the expression is inconsistent. Otherwise, the operands share a common concept through inheritance and the expression is consistent with the semantic design of the database.

For example, suppose the crates described above can be shipped by either air or sea. Like a ship, an airplane can carry more than one crate (1:N between AIRCRAFT and CRATE) and can land at more than one city (M:N between AIRCRAFT and PORT). The AIRPORT relation represents the intersection of AIRCRAFT and PORT. The concept hierarchy is shown in Figure 6-6.

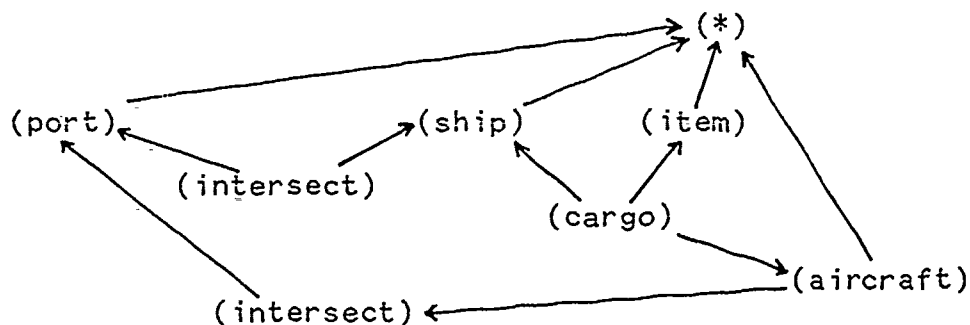


Figure 6-6. Concept Hierarchy

If we wanted to know which crates were too wide to load on a specific aircraft, the query might look like:

```

SELECT Crate-ID
FROM   Crate, Aircraft
WHERE  Aircraft-ID = "DC-10" AND
      Crate.Width < Aircraft.Width;

```

The query involves a compound search condition containing two predicates. The first, Aircraft-ID = "DC-10" does not involve numeric-valued attributes. The second, Crate.Width < Aircraft.Width, does involve numeric attributes with different concepts:

```

Crate.width    --> Concept = Cargo
Aircraft.Width --> Concept = Aircraft

```

The concept hierarchy in Figure 6-6 shows the two concepts share a common descendant, in this case "aircraft." Thus, the expression "Crate.width < Aircraft.Width" is consistent with the semantic intent of the database design and can be allowed. The expression must now be checked for dimensional consistency and, if required, commensurate units of measure must be reconciled.

Now suppose we wanted know which ships were wider than a DC-10.

```

SELECT Ship-Name
FROM   Ship, Aircraft
WHERE  Aircraft-ID = "DC-10" AND
      Ship.beam > Aircraft.Width;

```

Again the query has a compound search condition and the second predicate involves operands having different concepts.

```
Ship.width      --> Concept = ship
Aircraft.width  --> Concept = aircraft
```

A search of the concept graph returns (*) as the only common concept. Since there is no traceable relationship between ship and aircraft, the expression "Ship.beam > Aircraft.width" has no meaning in the context of the database design.

Many-to-many relationships are handled differently. The intersection relation used to represent these relationships is fundamentally different from other child relations. The intersection will not normally have meaning independent of its parent relations and thus will not have a concept. Technically, the intersection inherits the concept of each of the parents.

Until this problem of multiple inheritance is resolved, we assign a place holding concept, "(intersect)", to these relations in order to complete the concept hierarchy. We then allow the (intersect) concept to reverse the direction of its associated arcs, as shown in Figure 6-7. This has the effect of placing the intersection above its parent nodes on the concept graph. Our search algorithm will then recognize the

relationship as valid and allow queries across both parent relations.

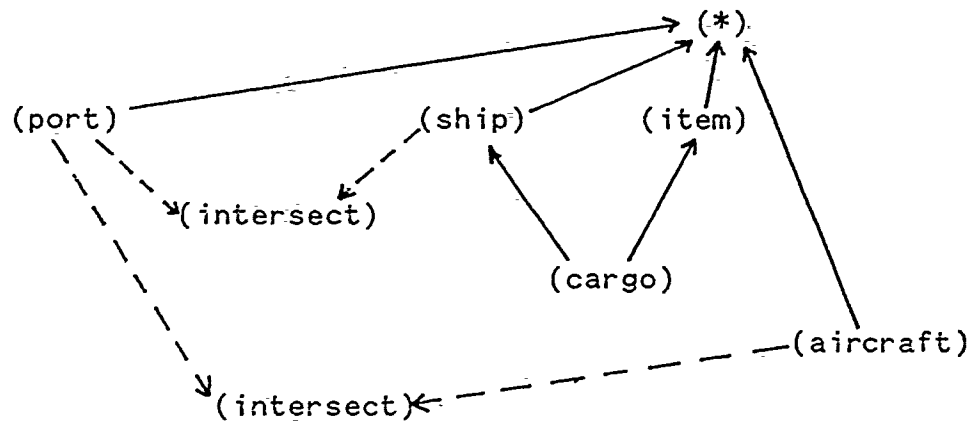


Figure 6-6. Revised Concept Hierarchy

We have indicated how a concept hierarchy might be used to ensure that queries are consistent with a specific database design. (The hierarchy also might be used to determine a concept of the relation returned by a query.) By identifying the concepts associated with each of the attributes in the select-list, we can search the hierarchy for the lowest common concept and assign it to the resulting relation. When more than one common concept is found the user will be required to reconcile the ambiguity.

The idea of concept complements the notions of quantity and unit to enforce a higher level of semantic consistency than is currently available in commercial relational database systems. However, the use of concept hierarchy in this chapter differs from the original notion introduced by Bradley

and Clemence. The ideas presented in this section are preliminary in nature. More work needs to be done investigating the ramifications of this approach as indicated in the concluding section.

VII. CONCLUSION

This thesis has presented a simple and efficient method of implementing a semantic type checking system for use with relational databases. By defining a concept, quantity and unit of measure for each numeric-valued data attribute, we increase the integrity of data manipulation. We can ensure queries are consistent with the design of the database. Only a human can determine the significance and meaning of data. We have shown, however, that the database system can share in the responsibility of handling the data in a manner consistent with its assigned meaning.

Manipulation of numeric data must conform to the laws of dimensional consistency. This is well known in the scientific and technical communities which deal with mostly numeric data measured across a variety of systems of measurement. There are obvious benefits to be gained by automating the dimensional analysis and conversion of commensurate units in the databases which support these communities.

Our research provides benefits outside the scientific community as well. As businesses gain greater access to international databases, the need for an automatic conversion of financial data becomes obvious. For example, including the

semantics of the data in the data dictionary would facilitate automatic tracking of currency fluctuations. This guarantees an accurate reflection of current market values.

There are also benefits to be gained by applying concept hierarchies to non-numeric data. A concept represents a real world object or event. Every table in a relational database has an associated concept regardless of the data types of its attributes. By identifying the foreign keys in each table we can build a concept hierarchy for any database. The hierarchy is guaranteed to reflect the semantic intent of the database design. For queries to be meaningful within the context of the design, they must be consistent with the concept hierarchy.

Our proposed system is not without its limitations. It has no provision for conversion or coercion between quantities. Thus, a height cannot be compared to a length or a width, even when this may make sense in a query. This problem could be addressed by defining relationships between quantities similar to those linking concepts. A quantity hierarchy could then be established and used to coerce related quantities. Determining whether such a scheme would provide sufficient benefits is an area for future research.

We have also largely ignored the problem of inheritance. We have assumed that a relation's attributes inherit the

concept associated with the relation. Foreign keys then have more than one concept. This presents no problem for semantic evaluation of SQL expressions, however. Concept consistency depends only on the relationships between database tables, not their attributes.

The relationships between concepts and quantities require further research. If a concept is assigned to a relation, must the relation contain all the attributes associated with the concept? Which concept and quantities should be applied to the relation produced by joining conceptually different relations?

Relational databases provide a powerful and easily understood data access capability. Some might argue that enforcing semantic consistency restricts and limits this capability. As databases grow larger and more integrated, however, unrestricted data manipulation increases the likelihood of improper handling of data and misinterpretation of the query results. Only by enforcing the semantic intent of the database design can we guarantee consistent and meaningful interpretations. The results of this thesis demonstrate a technique for enforcing semantic consistency which extends features currently existing in relational database systems.

APPENDIX A: SAMPLE DATA DICTIONARY TABLES

Schema:

CONCEPTS Dictionary Table

TABLE_NAME	CHAR(20)
CONCEPT	CHAR(20)

QUANTITIES Dictionary Table

TABLE_NAME	CHAR(20)
COLUMN_NAME	CHAR(20)
QUANTITY	CHAR(20)
Q1	NUMBER(4)
Q2	NUMBER(4)
UNITS	CHAR(20)

UNITS Dictionary Table

UNIT	CHAR(20)
DIMENSION	CHAR(20)
D1	NUMBER(4)
D2	NUMBER(4)
CONVERSION_STRING	CHAR(20)

Sample Dictionary Tables:

CONCEPTS:

TABLE_NAME	CONCEPT
US_ROUTES	AIR_ROUTE
INTERNATL_ROUTES	AIR_ROUTE
SHIP	SHIP
CRATE	CARGO
ITEM	ITEM
PORT	CITY
SHIP_PORT	(INTERSECT)
AIRCRAFT	AIRCRAFT
AIRPORT	(INTERSECT)

QUANTITIES:

TABLE_NAME	COLUMN_NAME	QUANTITY	Q1	Q2	UNITS
US_ROUTES	MILEAGE	DISTANCE	13	1	MILES
INTERNATL_ROUTES	DISTANCE	DISTANCE	13	1	KILOMETERS
SHIP	LENGTH	LENGTH	2	1	FT
SHIP	BEAM	WIDTH	3	1	FT
SHIP	DRAFT	DEPTH	11	1	FT
SHIP	HOLD_LENGTH	LENGTH	2	1	FT
SHIP	HOLD_WIDTH	WIDTH	3	1	FT
SHIP	HOLD_HEIGHT	HEIGHT	5	1	FT
SHIP	HATCH_AREA	AREA	6	1	SQFT
SHIP	CARGO_CAPACITY	WEIGHT	7	1	TONS
CRATE	LENGTH	LENGTH	2	1	METERS
CRATE	WIDTH	WIDTH	3	1	METERS
CRATE	HEIGHT	HEIGHT	5	1	METERS
CRATE	WEIGHT	WEIGHT	7	1	KILOGRAMS
PORT	DEPTH	DEPTH	11	1	FT

UNITS:

UNIT	DIMENSION	D1	D2	CONVERSION_STRING
UNITLESS	(NULL)	1	1	(Base Unit)
USDOLLAR	CURRENCY	2	1	(Base Unit)
METERS	LENGTH	3	1	(Base Unit)
KILOGRAMS	MASS	5	1	(Base Unit)
SECONDS	TIME	7	1	(Base Unit)
AMPS	ELEC_CURRENT	11	1	(Base Unit)
DEG_K	TEMPERATURE	13	1	(Base Unit)
CANDELA	LUMINOUS_INTENSITY	17	1	(Base Unit)
MOLE	AMT_SUBSTANCE	19	1	(Base Unit)
FT	LENGTH	3	1	0.3048*
SQFT	LENGTH*LENGTH	9	1	0.09290304*
TONS	MASS	5	1	(.45359/2000)*
MILES	LENGTH	3	1	1.609344*
CU_METERS	LENGTH*LENGTH*LENGTH	27	1	(Base Unit)

APPENDIX B: POSTFIX ALGORITHMS

The algorithms presented here represent the major functions of the preprocessor. The routines needed to access the data dictionary tables, store the results of queries, and the search routines have been omitted.

Initial implementation of the preprocessor was concerned only with evaluating SQL SELECT statements. Entries to the dictionary tables were made manually.

The `get_token()` function returns the next token from the SQL command. The preprocessor recognizes the following token types: DELIMITER, OPERATOR, BOOLEAN_OP, IDENTIFIER, LITERAL, NUMBER, and COMMAND. OPERATOR includes both the arithmetic and relational operators. BOOLEAN_OP is either AND, OR, or NOT. A LITERAL is a string enclosed by single quotes. NUMBER is for numbers and COMMAND is assigned when an SQL keyword is encountered.

IDENTIFIER and NUMBER represent complex structures consisting of: TOKEN_STRING, TABLE_NAME, COLUMN_NAME, CONCEPT, Q1, Q2, and CONVERSION_STRING. For NUMBER tokens only the TOKEN_STRING, Q1 and Q2 fields are used. Q1 and Q2 are assigned the value -1. This flags the token for special treatment during evaluation.

CONVERTING INFIX EXPRESSIONS TO POSTFIX

The following algorithm converts an expression from infix to postfix notation. The algorithm requires a function `prcd(op1,op2)` where `op1` and `op2` are OPERATOR tokens. This function returns TRUE if `op1` has precedence over `op2` when `op1` appears to the left of `op2` in an infix expression. Precedence of operators is shown below:

Highest	<code>*</code> , <code>/</code>
	<code>+</code> , <code>-</code>
Lowest	<code>=</code> , <code><></code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code>

For parentheses, `prcd()` returns the following results:

```
prcd( '(', op ) = FALSE
prcd( op, '(' ) = FALSE
prcd( op, ')' ) = TRUE
prcd( ')', op ) = undefined (an attempt to compare the
                             two indicates an error)
```

The algorithm is shown here:

```
while (not end of expression) {
    get_token();
    if (token is an identifier)
        add token to postfix string
    else { /* token is an operator */
        while (operator stack is not empty) and
            (top operator in stack has precedence over
             token) {
            pop top operator from stack and add to
            postfix string
        } /* end while */

        if (stack is empty OR token != ')')
            push token onto operator stack
        else /* top operator is open parenthesis */
            pop top operator from stack and discard;

    } /* end else */
} /* end while */
```

```

while (operator stack not empty) {
    pop the stack, add to postfix string
} /* end while */

```

EVALUATING THE POSTFIX EXPRESSION

Each operator in a postfix string refers to the previous two operands in the string. Each time we read an operand (i.e., IDENTIFIER or NUMBER) we push it onto a stack. When we reach an OPERATOR, its operands will be the top two elements of the stack. We are concerned only with the Q1 and Q2 elements of each operand. The following algorithm evaluates an expression in postfix using this method:

```

/* scan the postfix string reading one
   token at a time */
while (not end of postfix string) {
    get_token() from postfix string
    if (token is an IDENTIFIER or NUMBER)
        push token onto stack
    else {
        /* token is an operator */
        pop stack and assign to opnd2
        pop stack and assign to opnd1
        switch token {
            case '*':
                /* multiply Q1's and Q2's */
                result.Q1 = absvalue((opnd1.Q1)*(opnd2.Q1))
                result.Q2 = absvalue((opnd1.Q2)*(opnd2.Q2))
                push result onto stack
                break

            case '/':
                /* cross multiply Q1's and Q2's */
                result.Q1 = absvalue((opnd2.Q2)*(opnd1.Q1))
                result.Q2 = absvalue((opnd2.Q1)*(opnd1.Q2))
                push result onto stack
                break

            case '+':
            case '-':

```

```

/* Q1's and Q2's must be equal */
/* if one operand is a NUMBER */
/* it takes the quantity of */
/* the other operand */
if (opnd1.Q1 = -1 )
    result.Q1 = opnd2.Q1
    result.Q2 = opnd2.Q2
    push result;
if (opnd2.Q1 = -1)
    result.Q1 = opnd1.Q1
    result.Q2 = opnd1.Q2
    push result;

if (opnd1.Q1 = opnd2.Q1 and
    opnd1.Q2 = opnd2.Q2)
    /* Operation allowed */
    push opnd1
else {
    /* Quantity Inconsistency */
    expression is inconsistency
    end evaluation
} /* end else */

case '=':
case '<':
case '<=':
case '>':
case '>=':
/* Q1's and Q2's must be equal */
/* if one operand is a NUMBER */
/* it takes the quantity of */
/* the other operand and the */
/* expression is allowed */
if (opnd1.Q1 = -1 or
    opnd2.Q1 = -1 or
    opnd1.Q1 = opnd2.Q1 and
    opnd1.Q2 = opnd2.Q2)
    /* Operation allowed */
    expression is consistent
    end evaluation
else {
    /* Quantity Inconsistency */
    expression is inconsistent
    end evaluation
} /* end else */
} /* end while */

```

LIST OF REFERENCES

Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

Bradley, G.H. and Clemence, R.D., "A Type Calculus for Executable Modeling Languages," *IMA Journal of Mathematics in Management*, v. 1, p. 277-291, 1988.

Beyer, William H., *CRC Standard Mathematical Tables*, 28th Edition, CRC Press, 1987.

Bhargava, Hemant K., *A Simple and Fast Numerical Method For Dimensional Arithmetic*, Working Paper 90-01, Department of Administrative Sciences, Naval Postgraduate School, Monterey, California, March 1990.

Clemence, Robert D., *A Type Calculus for Mathematical Programming Modelling Languages*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, September 1990.

Dolk, Daniel R. and Kirsch, Robert A., "A Relational Information Resource Dictionary System," *Communications of the ACM*, v. 30, p. 48-61, January 1987.

Dreiheller, A., Moerschbacher, M., and Mohr, B., "Programming Pascal with Physical Units," *SIGPLAN Notices*, v. 21, p. 114-123, December 1986.

Gehani, N.H., "Units of Measure as a Data Attribute," *Computer Languages*, v. 2, p. 93-111, 1977.

House, R.T., "A Proposal for an Extended Form of Type Checking of Expressions," *The Computer Journal*, v. 26, p. 366-374, November 4, 1983.

Karr, Michael, and Loveman, David B., "Incorporation of Units into Programming Languages," *Communications of the ACM*, v. 21, p. 385-391, May 1978.

Kroenke, David M. and Dolan, Kathleen A., *Database Processing*, 3rd Edition, Science Research Associates, Inc., 1988.

Osborn, Sylvia L. and Heaven, T.E., "The Design of a Relational Database System with Abstract Data Types for Domains," *ACM Transactions on Database Systems*, v. 11, p. 357-373, September 1986.

Tenenbaum, Aaron M., Langsam, Yedidyah, and Augenstein, Moshe J., *Data Structures Using C*, Prentice Hall, 1990.

Viescas, John, *Quick Reference Guide to SQL*, Microsoft Press, 1989.

INITIAL DISTRIBUTION LIST

- | | | |
|----|--|---|
| 1. | Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 52
Naval Postgraduate School
Monterey, California 93943-5002 | 2 |
| 3. | Professor Daniel R. Dolk, Code AS/Dk
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 4. | Professor Gordon H. Bradley, Code OA/Bz
Department of Operations Research
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 5. | Professor Hemant K. Bhargava, Code AS/BH
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943 | 1 |
| 6. | Lieutenant Christopher A. Barnes
P.O. Box 1596
FPO New York, New York 09539-5000 | 2 |