A KNOWLEDGE LEVEL USER INTERFACE USING THE ENTITY-RELATIONSHIP MODEL

by

HOCK CHUAN CHAN

B.A., University of Cambridge, 1979

M.A., University of Cambridge, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

Faculty of Commerce and Business Administration

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1989

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Commerce and Business Administration

The University of British Columbia
Vancouver, Canada

Date 14 April 89

ABSTRACT

The relational database system has achieved widespread popularity; however, it is still very difficult for users, even those trained in the relational model, to formulate relational queries. The major cause of the difficulties is the fact that the user and the database system communicate using constructs that are not closely related to the user's world.

This dissertation develops a new level of user-database interaction — the knowledge level (KL) interface — where the user and the database system exchange only knowledge of the domain. The data structure used in the database is fully hidden from the user. In this way, the query is very closely related to the user's world.

Under the new KL approach, the database system is no longer seen as a store of data. It is set up as an agent to know the domain knowledge told to it by the user. The system will then provide the knowledge required by the user during retrievals. It will use elements of the entity-relationship model for communicating knowledge about the real world with the user.

It is shown that the KL interface is in many ways better than the relational interface. Users of the KL interface need to know less and perform fewer data manipulation operations than users of the relational interface. The KL interface also achieves both physical and logical data independence, unlike the relational system which does not truly achieve logical data independence.

This dissertation also proposes a new approach to understanding the meaning of completeness of a query language, breaking away from the traditional calculus-based measure of completeness. This new approach is then applied to the development of the knowledge level interface.

The main contributions of this dissertation are the proposal and development of a knowledge level interface, the anaylsis to show that this interface is better than the relational interface, and the demonstration that such an interface is feasible even for large databases.

TABLE OF CONTENTS

List of Figures

ACKNOWLEDGEMENT

I will like to thank my supervisor Robert Goldstein for his very patient and fruitful guidance, and the other members of the supervising committee Yair Wand and Richard Rosenberg for their very helpful suggestions and questions. They have contributed much of their time to make this thesis possible.

I will also like to thank the many faculty members and students of the Management Information Systems Division and of the Computer Science Department who have in many ways contributed to the development of the ideas in this thesis.

I. INTRODUCTION

A. THE USER-DATABASE INTERFACE

The user interface to a database system is an important research area, since a better interface allows the user easier interaction with the database and allows the programmer/user to formulate queries faster and more accurately (Chamberlain 1980).

There have been numerous research efforts in this area. The more recent examples include Albano et al. 1985, Atzeni and Chen 1981, Brady and Dampney 1987, Campbell et al. 1985 and more than twenty others in the references. Some works on the user interface are more technical, such as testing for completeness and increasing the computational efficiency in processing queries (for example, Merrett 1978, Atzeni and Chen 1981, Klug 1982, Campbell et al. 1985, Sellis 1988). Some works are more behavioral, such as measuring the time taken by the users to learn a query language and the accuracy of the user formulated queries (for example, Welty 1985, Jarke and Vassiliou 1985, Reisner 1981, Welty and Stemple 1981, Chamberlain 1980, Greenbalt and Waxman 1978). Other works develop new query languages and query displays (for example, Codd 1978, Harris 1980, Burgess 1984, Flory and March 1984, Fogg 1984, Kaplan 1984, Vossen and Brosda 1985, White 1985, Templeton and Burger 1986).

Despite the considerable amount of research, the problem remains that user interfaces are complicated for users, even those who are trained (Tsur and

Zaniolo 1984, Welty 1985, Motro 1986).

B. THE LOGICAL DATA LEVEL INTERFACE

The major difficulties in the user-database interface arise because the user has to deal with unfamiliar data objects, such as relations and fields, in the database (Poonen 1979, Templeton and Burger 1986, Junet 1987), and because the user has to carefully manipulate many pieces of these data elements in order to specify simple real world concepts.

The query system for a relational data model (RDM) is the standard example of such an interface. The user has to know the relations in the database, and he has to perform operations on the relations and their columns.

We will use the term *logical data level interface* to refer to this level of interaction where the user needs to know and manipulate the data elements in the database.

C. OBJECTIVES OF THE DISSERTATION

This dissertation aims to develop a new level of interface — the knowledge level interface — that will eliminate the difficulties of the logical data level interface.

The main thrust of the dissertation will be the conceptual development of a new knowledge level approach to interacting with a database system. To provide support for the new approach, a comparative analysis will be done to show the advantages of the new approach over the logical data level approach. Additional

supportive work will be done to demonstrate the feasibility of the new approach.

In general, a user-database interface allows for the three main functions of insertion, deletion and retrieval of information. It also has many management functions such as locking, unlocking, triggers, security settings, backup, and recovery. The design and development of these management functions at the knowledge level will not be part of the dissertation.

D. CONTRIBUTIONS OF THE DISSERTATION

The theoretical contributions of this dissertation include:

1. The development of a clear knowledge level approach for the user to interact with the database system. This approach is distinctively new, and it solves many difficulties of the existing logical data level interface. Furthermore, this approach is not tied to any specific data model or conceptual model.
2. The development of a new approach to defining the completeness of a query language.
3. The development of a knowledge level interface using the elements of the entity relationship (ER) model. Quite a few languages have been proposed for the ER model. The one developed here is explicitly designed for the knowledge level and makes no reference to logical data constructs, unlike many of the existing ER languages. The interface is also more comprehensive, for example, in its treatment of IS_A relationships and inheritances. A more detailed comparison of this KL interface and the existing ER languages is given in Appendix F.

4. A clear analysis of problems associated with the logical data level, in terms of the knowledge and the data manipulation operation requirements, and how these difficulties are avoided.
5. A demonstration that the knowledge level approach is practically possible, even for large database systems.

On the practical side, the knowledge level interface can lead to:

1. Higher productivity in query writing. This will apply to both programmers and other users. Since they need to know less and do less, higher productivity can be expected.
2. Easier interfacing with the database for all users, through reduction of the knowledge and operations required of the user.
3. The possible elimination of the manual task involved in the logical design of a database, thus leading to higher designer productivity.

E. OUTLINE OF THE THESIS

The difficulties of the logical data level interface will be presented in detail in Chapter II. Chapter III discusses the research methodology, describing the hypotheses and the methods used to verify the hypotheses. Chapter IV gives a description of the abstraction levels commonly used in studies of information systems. Chapter V introduces the knowledge level and proposes a new way of viewing a database management system (DBMS). The new approach views a DBMS as a knowledge agent rather than a store of data.

Chapter VI develops a knowledge level interface using elements of the

entity-relationship model. The development is based on a new approach to the issue of language completeness. A specific language — KQL — is defined for the interface.

Chapter VII shows how the users of the knowledge level interface need to know less and perform fewer data manipulation operations than the users of the relational interface. It also shows how the difficulties of the logical data level interface shown in Chapter II are avoided.

The implementation of the knowledge level language using Prolog is described in Chapter VIII. Chapter IX shows a method for building the knowledge level system on top of a relational DBMS. Lastly, Chapter X concludes with the major contributions of the research and discusses the interesting future work.

II. QUERY DIFFICULTIES OF THE RELATIONAL DATA MODEL

During the development of the three classical data models — the relational model, the network model and the hierarchical model — there were many debates and much research on which was the best model. It has since been generally agreed that the relational model is the best of the three when it comes to the user interface (Lochovsky and Tsichritzis 1977, Date 1982, Goldstein 1985). Date (1982, p485) writes, "It is generally accepted that nonprogramming users, at least, will require a relational view of the database." The dominance of the relational model is also evident in the large number of commercial relational database management systems†. Hence we will limit the discussion of the logical data level query difficulties to the best data model. Since SQL is the ANSI standard for relational database query, we will use SQL to illustrate relational queries and as a benchmark against which to evaluate the new knowledge level interface.

The relational model has been widely criticized for many weaknesses in representing real world semantics (Codd 1979). The weaknesses include semantic overloading of the relation structure (that is, a relation represents anything and everything), atomic domains, orientation toward computer processing in requiring non-redundancy, and lack of good representations for special relationships such as specialization, generalization and aggregation (Xie 1986, Smith and Smith 1977).

A poor database design can make queries difficult or even impossible. However,

†These include relational database machines such as ADABAS Database Machine, Teradata machines and IDM 500, mainframe DBMSs such as DB2, INGRES and Oracle, and many microcomputer DBMSs such as the 31 DBMSs reviewed in PC Magazine, Vol. 7, No. 9, May 17, 1988.

a good design does not guarantee easy queries since the criteria for good designs and easy queries are very different. Design criteria emphasize the ability to represent the real world. Physical design and, sometimes, higher level designs emphasize computational efficiency. Query criteria emphasize the ease and ability to extract knowledge of the real world from the database, and these are more human-oriented.

There are many topics that will not concern a database designer but are very important to the user making the query. For example, a difficult topic in query writing is the use of aggregate functions. Queries often involve the use of functions such as ALL, COUNT, SUM, and AVERAGE. However, these are not considered during design.

We will focus on the major difficulties in querying a relational database.

1. Integrity and Consistency Maintenance

It is tedious for the user to maintain the integrity and consistency of a relational database. Consider, for example, the constraint that when an entity is deleted, all its relationships have to be deleted as well. When the user deletes a tuple (in a relation) that represents an entity, he must go through the other relations that represent relationships involving this entity. In addition, the entity may be represented by more than one relation. The user has to perform different actions, depending on how the entity and its relationships are being represented by the relations. The user may have to delete tuples in another relation, he may have to set a column of some tuples in another relation to

null, he may have to set multiple columns to null, or he may have to do all of these operations.

Consider the following hypothetical set of relations:

```
SUPPLIER(SNO, NAME, CITY)
PART(PNO, NAME, DESCRIPTION)
SUPPLY(SNO, PNO, PRICE, QTY)
DEPT(DNO, NAME, LOCATION, SNO)
CAR(CNO, TYPE, SNO, PRICE, QTY)
```

The relation SUPPLY represents the attributes PRICE and QTY of the relationship between a supplier and a part. A supplier can supply many parts, and a part can be supplied by many suppliers. The relation DEPT shows the attributes of a department as well as the SNO of the supplier that a department is in charge of. A department is only in charge of one supplier. The relation CAR shows the attributes CNO and TYPE of a car as well as the SNO of the supplier supplying the car and the attributes PRICE and QTY of this relationship. A car is supplied by only one supplier, and one supplier can only supply one car.

In this case, deleting a tuple in SUPPLIER with SNO value of, say, 'S1' will require deleting each tuple in SUPPLY that has SNO value of 'S1', finding each tuple in DEPT with SNO value of 'S1' and setting its SNO value to null, and also finding each tuple in CAR with SNO value of 'S1' and setting its SNO, PRICE and QTY values to null.

These actions are not only tedious, they also require the user to know the database structure thoroughly.

To alleviate the user's tasks, some relational DBMSs such as INGRES provide for integrity rules and triggers to be defined by the user/designer. These triggers will automatically perform the actions resulting from deleting a tuple in the relation SUPPLIER. However, rule and trigger definition is tedious as every integrity constraint has to be defined for each and every application and for every part of the database. The definition of triggers cannot be automated, at least not without considerable constraint on the naming of columns and the design of relations. This is because the relational model does not provide the database system with sufficient information on how the tables and columns are related. For example, there is no information for the system to decide if the columns PRICE and QTY in relation CAR are affected by the deletion of a tuple in SUPPLIER.

2. Inheritance of Attributes and Relationships

A group of entities is sometimes considered to be a subentity of another group of entities. For example, the group of engineers in a company is a subentity of the group of employees, and so is the group of managers. The subentity groups are often referred to as specializations, while the superentity is referred to as the generalization. There are general attributes such as name, address and employee number that all employees will have. There are also special attributes that not all subentities will have; for example, engineers have engineering professions such as electronics or mechanical, while managers have ranks such as

senior or junior.

Attribute inheritance refers to the ability of the system to retreive general attributes when the user specifies the special entities. For example, the user may ask for the names, addresses and employee numbers of engineers.

The relational model does not provide any standard mechanism for representing the specialization of entities. A possible representation following the example given by Alagic (1986) is this set of relations:

Real Relation

ENGINEER(NUMBER, NAME, ADDRESS, PROFESSION)

MANAGER(NUMBER, NAME, ADDRESS, RANK)

Virtual Relation

EMPLOYEE(NUMBER, NAME, ADDRESS)

A second possible representation is the following set of real relations:

EMPLOYEE(NUMBER, NAME, ADDRESS)

ENGINEER(NUMBER, PROFESSION)

MANAGER(NUMBER, RANK)

Both representations do not allow many convenient queries. In the first representation, the attribute inheritance is already designed into the real relations. The problem is that the virtual relation cannot be updated. The user cannot add a new employee tuple to the employee relation. One reason is that a relational DBMS does not have the information to decide which real relation to update.

Another reason is that updating of views has not been thoroughly researched and is little supported by existing relational DBMSs (Date 1987a). The second representation leaves the attribute inheritance to the user who will have to perform join operations based on the employee number.

Further difficulties will arise if an engineer can also be a manager; for example, the first representation will not show that an engineer can have a rank. Manual inheritance by join operations is again required. While this case may be rare in the real world, there are many cases where an instance of one subentity is also an instance of another subentity.

The same problems apply for inheritance of relationships where the relationships formed by the superentity are inherited by the subentities.

3. Data Oriented Retrievals

Before the user can retrieve information about the real world, he first has to understand the meanings represented by the data elements and by the arrangement of the data elements (for example, two columns placed in a relation may represent a relationship or may be attributes of an entity). He then has to manipulate the data elements and retrieve them. For example, a user may have to know that a particular relation represents a relationship, that the details of the participating entities are to be found in two other relations, and that the correct way to reshape the data is by performing joins based on the key columns.

For retrievals involving only one relation, the queries are usually very simple. But more difficult queries can be very complicated. Consider the SQL command to get suppliers who supply at least all the parts supplied by supplier 'S2'. This query can be found in many of Date's books, (for example, Date 1987a and Date 1987b). where it is used to illustrate the power of SQL. In this example, relation SP contains SNO, PNO and other columns; *SPX*, *SPY* and *SPZ* are variables standing for SP.

```
SQL:    SELECT DISTINCT SPX.SNO
        FROM SP SPX
        WHERE NOT EXISTS
            (SELECT *
            FROM SP SPY
            WHERE SPY.SNO='S2'
            AND NOT EXISTS
                (SELECT *
                FROM SP SPZ
                WHERE SPZ.SNO=SPX.SNO
                AND SPZ.PNO=SPY.PNO)).
```

Another example is the following query to retrieve the names of those suppliers who supply all parts. In this example, relation SP is as before, relation S contains SNO, SNAME and other details of the supplier, and relation P contains PNO and other details of the part.

```
SQL:    SELECT S.SNAME
        FROM S
        WHERE NOT EXISTS
            (SELECT *
            FROM P
            WHERE NOT EXISTS
                (SELECT *
                FROM SP
                WHERE SP.SNO=S.SNO
                AND SP.PNO=P.PNO))
```

If the relations representing the suppliers and parts are S'(SNO,SNAME) and P'(PNO,SNO,PNAME,PRICE,QTY) instead of S, P and SP above, then the SQL query to do the same real world query may be substantially different. A suitable version is the following:

```
SQL:    SELECT S'.SNAME
        FROM S'
        WHERE NOT EXISTS
            (SELECT *
            FROM P'
            WHERE P'.SNO ≠ S'.SNO).
```

It further shows that the SQL formulation of a real world query has to be done differently depending on the specific structure of the relations. In other words, the retrieval is data-oriented.

The primitive relational operations such as join, project and select are operations purely on the data. Many of these data manipulation operations are difficult for users to handle (Welty 1985, Welty and Stemple 1981, Greenbalt and Waxman 1978). For example, Junet (1987) builds a system that allows casual users to query only one relation: "Indeed we think that operations on several relations are too complex for casual end-users, because it needs a good knowledge of the join operator." It must be emphasized that joining of relations is an essential operation in the relational model. If this is difficult, it can be expected that the interface will be difficult.

Greenbatt and Waxman (1978) compared user performance in using three relational query languages — SQL, Query-By-Example (QBE), and a relational algebra. After being trained on 25 examples for one to two hours, subjects were asked to write queries. The time taken to write the queries and the correctness of the queries were measured. The results showed no statistical difference among the three languages. For SQL, the percentage of correct queries was 72.8%.

Welty and Stemple (1981) conducted a study where students attended 14 one-hour sessions to learn a relational query language (either SQL or TABLE) before they took two query writing tests, the second some time after the first. The result clearly shows the difficulty of writing queries that involve data manipulation. Even for the easy SQL queries, the subjects after the thorough 14-hour training could only write correctly 88% of these. Of the hard queries, only 41% were correct. The joining of relations was difficult, with only 43% correct at the first test and 29% at the subsequent test. Chaining, also known

as nesting or composition, gets 71% and 68%; the "group by" clause gets 39% and 14%; and combinations of these conditions get 42% and 37%.

Clearly, elimination of the data manipulation operations will be a big advantage for the users.

Existing systems resort to the provision of customized interfaces, such as menu interface with predefined sequences of prompts and inputs (for example, as allowed, with extra programming, in INGRES and dBASE III), or graphical displays to handle simple queries (Elmasri and Larsen 1985, Flogg 1984, Burgess 1984). Some relational systems have natural language interfaces, which require some customization to suit the set of relations and the actual data in the relations. Customized interfaces are good, except that they require skilled manpower and extra work, and they also suffer from limited functionality.

4. Data Oriented Updates

The updates to the relational database focus on columns and tables. A real world change may necessitate changes to many data elements. If an entity is represented by two or more relations, possibly as a result of normalization, deleting an entity instance will require changes to the tuples of all these relations. The user is expected to know the relevant relations.

In the case where there is a one-to-one relationship between two entities, the standard relational representation is one relation showing an entity and the relationship. When one relationship instance is to be deleted, the user has to be

careful to set certain columns, those representing the relationship, of one tuple to null values while keeping the other columns, those representing the entity, unchanged. The user has the responsibility of knowing exactly which are the appropriate columns.

5. Extra Mental Processing

It seems most probable that users do not think of the real world in terms of relations with tuples and columns. A strong piece of evidence is the process for designing the relational database. The relations are derived from other things such as entity-relationship models or functional dependencies. Nobody looks at the world and says, "Yes, these are the relations in this world." The relational user has to make a mental transformation between the real world and the relational elements. There is therefore extra mental processing demanded of the user. Poonen (1979) states that "one of the reasons that data base systems appear to be so complex is the mismatch between the way a user conceptualizes his data and the way the data bases choose to represent it, physically or logically". Similarly, Templeton and Burger (1986) write, "A large conceptual gap exists between the way data are seen by the programmers and database administrators who organize and maintain the database and by the end users of the data."

This emphasizes the need for data base languages to operate at a very high level using terms that are familiar to the user. This move towards the user's world is also reflected in Gaines' (1981) rules of dialogue programming, one of which is to use the user's model. In fact, in the area of computer programming, this movement towards the user's world has culminated with the idea of

knowledge programming (Abbott 1987). Knowledge programming produces programs that contain the domain knowledge explicitly.

Also, Meersman (1987) states, "It is firmly believed by the author that our current way of handling relations as a 'natural' representation of information will be superseded by more conceptual and more elementary paradigms - once the requirement to group data elements artificially for purposes of rapid access only will have disappeared."

The relational model has achieved physical data independence, but not logical data independence (Staley and Anderson 1985, Vossen and Brosda 1985, Date 1982 p139). This imposes the need for users to logically navigate among relations. This navigation seems to be a very troublesome requirement for users, so there are many efforts to eliminate this, such as by having universal relations (Maier et al. 1984, Pahwa and Arora 1985). The universal relation will relieve users of the need to know the tables. However users still need to know the columns. There are also many other problems plaguing the universal relation method (Atzeni and Parker 82), such as difficulty in dealing with multiple relationships (direct or indirect) among the same attributes, requirement for unique column names, and likely user misunderstanding of the very big relation, such as a hundred column relation. Also, it does not solve the problems of semantic overloading; even worse, a single relation represents everything.

An interesting and highly-researched solution to avoid the logical navigation is the provision of natural language (NL) interfaces for relational databases (McCord

1987, Templeton and Burger 1986, Jarke et al. 1986, White 1985, Kaplan 1984, Wallace 1984, Harris 1980, Codd 1978). In fact, the user need not know the relations in the database. The NL interfaces (to relational databases) are excellent at handling flexible syntax; but they are usually limited in quite severe ways. They need skilled manual customization to suit the actual relations and data, often involving the end-users, for example, when eliciting a representative set of sample queries (Templeton and Burger 1986). NL interfaces are usually limited to retrievals. Updates, especially changes to the relations, are usually not allowed. The retrieval power is often not described. There is often some uncertainty as to the extent of the naturalness. A user cannot be certain without trying if his natural query will be understood. While this is acceptable for interactive query where the system can ask for clarification, and the user is there to answer, it is quite unacceptable for non-interactive query, for example, for queries embedded in some general programming language. Lastly, natural language can be quite messy in specifying the list of things to be printed. For example, it will be quite demanding for a user to express the following SQL query in clear natural language acceptable to the NL interface:

```
SELECT S1.SNAME, SP1.PRICE, SP1.QTY, P.PNAME,
       SP2.PRICE, SP2.QTY, S2.SNAME
FROM S S1, SP SP1, P, SP SP2, S S2
WHERE S1.SNO=SP1.SNO
  AND SP1.PNO=P.PNO
  AND P.PNO=SP2.PNO
  AND SP2.SNO=S2.SNO
  AND S1.SNO > S2.SNO.
```

This query, in real world terms, is to find the part that is supplied by two or more different suppliers, print the first supplier's name, the price and quantity that this supplier is supplying the part for, print the part name, print the price and quantity that the second supplier is suppling this part for, and lastly to print the name of the second supplier. If there are more than two suppliers supplying this part, then print a few lines, each line with two suppliers.

III. THE RESEARCH METHODOLOGY

There are three central hypotheses in this dissertation. The first is:

Hypothesis 1:

> It is possible to design a database system that will interact with the user using domain knowledge only and that, unlike the relational system, will not require the user to know the data structures used in the database.

The interaction here is not limited to retrieval queries. It includes insertions, deletions and changes. It also includes the definition of the real world by the user to the system; this definition corresponds to the data definitions of the relational system. The term *user* is not limited to the end-user. It means any person who needs to interact with the database system. It can include the end-user who wants to retrieve information, the designer who wants to define his world to the system, and also the programmer who writes programs to interact with the database system.

Hypothesis 1 will be verified by the following methods:

1. Conceptual development of a new approach — the knowledge level approach — to design the user-database interaction. This will incorporate ideas from the fields of artificial intelligence, computer programming, and database.
2. Systematic design of a knowledge level query language to accompany this new approach.
3. Implementation of the new approach by developing a computer program. The efficiency of the computation will be considered, but is not of importance to this dissertation. The aim is to provide additional support

that this approach is feasible.

The second and third hypotheses are:

Hypothesis 2:

The user interacting with the new system needs to know less than the user interacting with the relational system.

Hypothesis 3:

The user interacting with the new system needs to perform fewer data manipulation operations than the user interacting with the relational system.

The practical significance of these two hypotheses is clear when we consider the fact that individuals have limited information processing capacities (Simon 1957). The less the user needs to know, and the fewer data manipulation operations he needs to do, the better will be his information processing capability. This is even more so when the data manipulation operations are difficult, as shown in Chapter II.

These hypotheses do not quantify the advantages of the new interface over the relational interface. There are other comparison factors that are not formally measured; they include human factors like intuitiveness and complexity of the various operations.

Hypotheses 2 and 3 will be verified by the following method:

1. Comparative analysis of user interaction with the new system (the knowledge level interaction) and with the relational system (the data level

interaction). The analysis will cover the various types of interaction: definition of the data/world to the system, updates and retrievals.

The analysis will show that the user of the new system escapes having to know things that the user of the relational system must know. In contrast to some other query-comparison studies, we do not pre-select a set of questions and compare based on this set of questions, since the result (for example, that query language A has 20% fewer data manipulation operations than query language B) will be highly dependent on the choice of questions. Rather, the analysis will show the specific situations where data manipulation operations are avoided or reduced.

One might even claim that the definition of the new system logically implies hypotheses 2 and 3. The comparative analysis will, however, provide a better understanding of how the knowledge level interaction differs from the data level interaction.

The conceptual development, design and implementation are directed at verifying the possibility of the knowledge approach. The analysis is directed at verifying that the knowledge approach, as compared to the data level approach, requires the user to know less and do less.

IV. GENERAL ABSTRACTION LEVELS

A general description of the abstractions commonly used in information system and database research is given here. It will be related to recent attempts at separating knowledge from its representations. The knowledge abstraction level to be described later is another step in our continual effort at creating better abstractions.

Abstraction is the process of generalizing and discarding details that are not needed. After analysing a number of information systems design methodologies, Olive (1983) arrives at a list of five levels of abstraction which are likely to be the "most clear and important levels." These are the external, conceptual, logical, architectural and physical levels. His paper provides the following definitions:

1. external level of abstraction — this describes the function of the real world system and the information flow among the units; for example, this level can be described with the data flow diagrams of structured analysis.
2. conceptual level — this is a description of the possible states of the real world systems, and the events and derivation rules; that is, it includes the statics and the dynamics.
3. logical level — this is an operational description of the information system, separated from the technology that could be used to implement it.
4. architectural levels — this and the next level incorporate the technology aspects into the description of an information system. An architectural model describes the logical structure and the processes. It will specify the mode of the processes, such as manual or computerized, and interative or batch modes.

5. physical level – this describes the physical structure of the database and detailed structure of each process. According to Olive, the physical model is the basis of the programming activities.

In information systems and databases, it is usual to separate details of different levels. For example, *Structured Analysis* produces separate physical and logical data flow diagrams. And *Structured Design* produces the logical design before the physical design. In database technology, one of the principal goals is to provide data independence (Brodie 84). Physical data independence means a separation of data between the physical abstraction level and the other levels; logical data independence means a separation of data between the logical level and the higher levels. In effect, this means that changes at a lower level should not effect the higher levels.

There is an almost universal move towards higher levels of abstraction (Abbott 1987), not only in the analysis and design of information systems but also in the usage of such systems. McGee (1976) writes, "The higher the language level, the faster and more accurate the programming." This move is also shown by the continual emergence of new generations of programming languages.

It should be noted that there is not much global consensus on the definitions of the various abstraction levels. For instance, there are different interpretations for ANSI/SPARC's three schema levels, which are the external, conceptual and internal schema. Gray (1984) considers the conceptual schema as a description of

the logical relationship between data items, together with integrity constraints; the internal schema as details of allocation of records to storage area, indexes, pointers, etc.; and the external schema as subschemas with enhanced facilities. But Yang (1986) considers ANSI/SPARC's conceptual level as either the conceptual or the logical level; the internal level as the physical level; and the external level as concerned with the views created from the logical databases by users. The major difference in interpretation is about the conceptual schema.

The relational model is commonly classified at the logical level, though it is occasionally classified at the physical level or even the conceptual level. In any case, queries made of the relational model are queries that are formulated in terms of the information system's data elements, that is, the relations, records and columns. The queries are not formulated in terms of the real world concepts. Since the queries are based on the logical data, the relational model can never really provide logical data independence. Staley and Anderson (1985) state "there has been no model introduced to provide a logical data independence analogous to the physical data independence of the relational model. It is not that a need for logical data independence has not been recognized; just that research in this area has not yet resolved this issue." The new knowledge level approach to be proposed will provide for logical data independence.

V. THE KNOWLEDGE AGENT

A. SEPARATION OF KNOWLEDGE AND REPRESENTATION

In the artificial intelligence field, there is a growing literature on making a clear distinction between knowledge and its representation. Newell (1982) introduces the concept of a *knowledge level*, defining knowledge as whatever can be ascribed to an agent such that its behavior can be considered rational. The agent has goals and can act rationally to satisfy its goals. This is purely a functional definition of knowledge, with no implication of any structure, nor of inference or search strategies. The main point is that knowledge has no structure, whereas knowledge representations have structures. There are many papers that have adopted this abstraction of knowledge from its representation (Levesque 1984, Clancy 1984, Sloman 1985, Brachman and Levesque 1986†).

A complementary idea appears in the area of computer programming, where Abbott (1987) calls for a new way of programming — *knowledge programming* — which he defines as producing programs where the domain level knowledge is explicitly visible. With knowledge programming, the programmer should care very little, if at all, about the computer-level representation of the knowledge. In particular, the order that the data appear (that is, the data structure) should not have to be considered.

This separation of knowledge from its representation can be viewed as a further

†Brachman and Levesque (1986, p77) state, "We might even consider that the single most important contribution that AI can make to databases is the notion of the Knowledge Level itself."

split at the conceptual level. At the conceptual level, we are concerned about the real world and usually produce a specific model of the real world. The model is a representation of the real world (Brodie 1984); it embodies knowledge of the real world. There can be many possible representations of the same knowledge. For example, the same knowledge can be represented using logic, networks or frames. Even within the logic system, there are many ways of representing the same facts. Therefore we can distinguish between the conceptual knowledge level and the conceptual representation level.

A conceptual knowledge level may be viewed as the content of the ANSI/SPARC conceptual schema independent of the representation. An example will help illustrate the difference between the conceptual knowledge level and the conceptual representation level. Let us consider this small schema using the conceptual language Galileo (Albano et al. 1985).

```
Organization := (
     REC Departments CLASS
          Department <->
          (Name : string
          AND Employees : VAR SEQ employees)
     AND Employees CLASS
          Employee <->
          (Name : string
          AND Nameofdept := DERIVED Name of
                         get Departments with
                         this isin (at Employees))).
```

In Galileo, a derived attribute cannot be updated. This is an artificial constraint imposed by the definition. If an employee moves to another department, the user must update `Department.employees` and cannot update `Employee.nameofdept`. This is thus an update to a knowledge representation. A pure knowledge update should be free of the representational restrictions.

B. THE KNOWLEDGE AGENT

The two ideas of knowledge separation and knowledge programming can be incorporated into the database environment. *The new database system is seen as an agent whose goals are to keep track of knowledge and to accept and answer queries, including both retrievals and updates, that are formulated as knowledge programs.* Internally the database has to rely on data structures to represent the knowledge. The system will perform its own integrity maintenance on the data; and it will automatically derive knowledge from its store of data.

The user no longer performs operations on the data in the database. Rather, the user will now communicate with the database system through exchange of domain knowledge. For example, the user will tell the database system a piece of knowledge such as that the supplier with number 'S2' is providing the part with number 'P3', instead of telling the database to insert a record "'S2','P3'" in the table SUPPLY. This can be seen as programming using only domain knowledge.†

† At this point, the interaction may appear similar to that provided by a natural language interface. The differences between the knowledge level interface and a natural language interface are discussed in the later section on future research.

One way to visualize this new system is to think of it as a human assistant. You tell him what you know of the real world and allow him to choose the method for representing this knowledge. When you need to refresh your memory, you ask your assistant for the required knowledge rather than asking to see the files and pieces of paper that he uses to record the knowledge. You have to tell a relational DBMS how to join relations. But you do not have to tell your assistant how to operate on his files.

A substantially new way of looking at a DBMS is advocated. Traditionally, a DBMS helps the user to organize his data. The user knows what data are inside the database and has the responsibility for ensuring that the data remain consistent. In the new way, the user does not know what data are inside the database. An immediate benefit is that the user is relieved of the responsibility of keeping the data consistent.

The database is now a store of domain knowledge rather than a store of logical data elements.

This idea of a knowledge agent also has support from the emerging object-oriented programming systems. Nierstrasz (1987) concludes, "The most important concept in object-oriented approach is data abstraction. . . . We are interested in the behavior of an object rather than in its representation." And here we are interested in the behavior (the user-database interaction) of the database rather than in the representations used inside the database. The database system proposed here will not provide any method to manipulate the

variables inside the database. The internal variables do not concern the user. The system will separate knowledge from the data within the database.

Another advantage of not dealing with the representational details is the provision of entity identity. Entity identity is provided if the entity can be referenced directly as a unit, and the entity can appear in multiple places (that is, represented by many pieces of data) in the database without any pointer or indirect mechanism visible to the user. A data model that provides entity identity can avoid many inconveniences and avoid proliferation of constraints (Copeland and Maier 1984). An example of the inconveniences has been discussed in the section on preserving consistency in the relational database. Upon deletion of an entity, the user has to go through related relations and do appropriate updates. The alternative is to define many constraints/triggers to perform these follow-up actions.

The proposed knowledge exchange interface will fully provide for entity identity. Not only are there no pointers visible to the user, there is no need for the user to know that the entity appears at multiple places in the database.

C. THE KNOWLEDGE INTERFACE

The communication of knowledge between the user and the system must be based on some shared conceptual schema (Jardine 1985). The relational model having only a single construct, the relation, does not seem suitable for communication of domain knowledge between the system and the user. It will require too much transformation between the user's real world schema and the

relations.

The entity-relationship model† will be used as a basis for domain knowledge communication. The ER model has very few structural constraints, and its concepts are widely believed to be natural. Brodie (1984, p32) writes of "the popularity of the ER model [and] the widespread belief in entities and relationships as natural modelling concepts." Parent and Spaccapietra (1985) write, "Within this group [of semantic models], the Entity-Relationship — ER — model (Chen 1976) may actually be considered as the most popular one, mostly because of its widespread use as a design tool." Teorey et al. (1986) state that "The ER model has been most successful as a tool for communication between the designer and the end-user during the requirements analysis and conceptual design phases because of its ease and its convenience in representation." Brady and Dampney (1987, p287) also state, "The author has used the technique [the ER approach] in a number of large commercial, manufacturing and government organizations. The diagrams are readily understood, and users can easily verify their accuracy." Schuldt (1987) recorded many opinions from practitioners who found the ER model to be much better than the relational model. His record includes the following quotations: "The ER approach models the real world better than the other models"; "The ER approach has two major advantages over the relational approach"; and "The relational (tabular) representation . . . occupied a 15-foot wall, and the structure of the subject matter was not at all clear. When

†There are many versions of the entity-relationship model, as classified by Chen (1981). The ER model used in this interface includes entity types, entity attributes, relationship types, relationship attributes, roles, cardinalities, and IS_A relationships. More details are given in the various sections on the actual language and the implementation.

they created an ER model from the tables, it suddenly became obvious what was going on." Finally, Chen (1976, p9) proposed the ER model because "the ER model adopts the more natural view that the real world consists of entities and relationships".

In addition, the knowledge level ER model has many of the desirable features for knowledge level communication. It inherently contains simple integrity constraints such as the requirement that relationship instances can exist only if the participating entity instances exist. It allows for automatic inheritance of attributes and even relationships from parent entities. It provides entity identity.

For the knowledge interface, the ER model must be used at a correct level — the knowledge level. However, the ER model is currently used predominantly at the representation level. The best example of this is seen in the declaration of relationships. Most of the papers in the literature (Dogac et al. 1987, Chen 1977, Teorey et al. 1986, Staley and Anderson 1985, Nakano 1983) insist that a relationship specification includes the key attributes of the participating entities. This specification decides how the participating entities are to be linked to the relationship. It decides on the use of keys as logical pointers, as in the relational model. It therefore belongs to the logical level. It also unnecessarily tells the user that the entities' key attributes exist in multiple places in the system.

At the knowledge level, the user simply specifies that a relationship is between certain participating entities. For example, the specification may be: "New

relationship supply between entity supplier and entity part". The system decides whether to include the key attributes of the entities, to use pointers, or to use some other identification methods such as system surrogates. Whatever the method used, it is internal to the system. The user need not have to know anything about it.

The ER model also allows easy two-way descriptions. For an ER model with a relationship SUPPLY between entities SUPPLIER and PART whose roles are SUPPLIES and SUPPLIED_BY respectively, the user can say "supplier supplies part", "part supplied-by supplier",† "supplier supply part" or "part supply supplier". The model does not require any entity to be the first in the description. In contrast, the previous Galileo example is more restrictive. Also, in some of the ER research (Hwang and Dayal 1981, Nakano 1983, Markowitz and Ray 1983, Tsur and Zaniolo 1984), some ordering is imposed so that, for instance, "supply(supplier,part)" may be allowed while "supply(part,supplier)" is not. These imposed orders are not necessitated by the domain, and they therefore unnecessarily restrict the communication between the user and the system. Besides, the user now has to remember some arbitrary ordering. At the knowledge level, we will not impose such orderings in the ER model.

Database designers commonly use the ER model to design the conceptual schema, which is then used to design the logical schema (usually the relational schema). This can be seen in the many papers on converting an ER model to a relational

†It is assumed that the role names between any two entities are unique; otherwise, these expressions may be ambiguous.

model, for example, in Teorey et al. (1986), Braind et al. (1985) and Staley and Anderson (1985). The conceptual schema is then discarded or at least shelved away; and the users query the logical schema. Adopting the ER model for the knowledge level interface will not require any changes to the conceptual design process. Thus the new knowledge level system can be easily merged into existing design practices.

The basic conceptual schema shared between the user and the system are the elements of the ER model; these are the concepts of entity type, relationship type, role and attribute. With this shared schema, the user can then proceed to tell the system of the existence of specific entity types such as supplier, part and department; and specific relationship types. After the types have been defined (that is, the shared conceptual schema has been expanded), the user can proceed to tell the system of specific entity instances and relationship instances.

From the above discussions, we will propose these basic principles for the design of a knowledge level interface:

1. Principle 1:- The user-database communication should use only domain knowledge, with some appropriate syntax.
2. Principle 2:- The communication should not be restricted by any peculiarity of the data structure (or knowledge representation) used in the database. In particular, knowledge of the order of the data in the data structure should not be required in the communication.
3. Principle 3:- The interface should avoid all operations that are not required by the domain. An example of such an operation is the joining of relations.

Another important example is the manipulation or definition of physical or logical pointers.

The following chapter will describe the development of an interface and a specific language for knowledge communication with the system. The system will require full explicit specification of all knowledge elements.

VI. THE KNOWLEDGE INTERFACE LANGUAGE

A. A FUNCTIONAL DEFINITION OF KNOWLEDGE LEVEL COMPLETENESS

1. The Existing Definition of Completeness

The retrieval power of a query language or system is commonly measured by its completeness. In the relational model, completeness is based on the power of relational calculus. A language is relationally complete if it can retrieve any relation derivable from the database of relations by means of an expression of relational calculus (Date 1982). Usually, a stricter definition is applied: that the retrieval can be done in a single self-contained request. To meet this stricter definition, the single request will usually be a multi-level query; that is, the query will include subqueries.

Completeness for a query language based on the ER model has not been well defined. Many designers of ER query languages totally adopt relational completeness. Dogac et al. (1987) reflect this group's view when they state that when relations are used in representing entities and relationships, there is no need for a definition of ER completeness since relational completeness is sufficient. This group therefore prove that their ER languages are relationally complete (Campbell et al. 1985, Dogac et al. 1987, Markowitz and Raz 1983). The proof, for example that by Dogac et al. (1987), is usually done by expressing the basic relational algebra operators (for example, union, difference, projection, selection and cartesian-product) in terms of the ER language. However,

relational completeness is not ER completeness.

Other ER language designers define ER completeness using an approach similar to the definition of relational completeness. They first define an ER calculus; then they define ER completeness as the ability to do what the calculus can do. For example, Atzeni and Chen (1981) propose an ER calculus and define completeness based on this calculus. They define completeness as the ability to extract data from any number of different entity sets and/or relationship sets; and simplified ER completeness as the ability to select instances in a single entity set or relationship set, with conditions that may involve any object in the schema. Their calculus only allows three types of conditions — the attribute-constant comparison, the attribute-attribute comparison within an object instance, and the relationship-existence condition. Atzeni and Chen (1981) state that "all the high-level languages presented until now for the ER model satisfy neither of the definitions of completeness". They then propose a language and prove it satisfies the definition of simplified completeness. However, this "complete" language has very little power; for example, it cannot retrieve details of suppliers and the parts supplied by them.

The above types of completeness definition are calculus-based. The calculus-based approach to a definition of ER completeness, while aiming for a good mathematical grounding, is not very satisfactory. Firstly, it depends on the specific ER calculus. Atzeni and Chen (1981) proposed two versions of the calculus, the simplified and the non-simplified versions. There are many other ER algebras and calculi being proposed in the literature, for example, by Parent and

Spaccapietra (1984 and 1985), Chen (1984), and Campbell et al. (1985). The completeness defined on a calculus is limited by the power of that calculus. Secondly, there are many different versions of the ER model, as analyzed by Chen (1981). Some versions have no attributes for the relationships, some have very limited form of relationships, some have no concept of roles, some have no 'IS_A' relationships, and some do not allow an entity type to have relationships with itself. Every slight variation in the model will require a different calculus and consequently a different definition of completeness.

We shall reexamine the concept of completeness, and propose an alternative approach to defining completeness.

2. What Exactly is Completeness?

What exactly is meant by completeness? It appears to include at least three criteria which are more fundamental for a query language: coverage, ease and conciseness. Coverage is how much the language can do, such as count, average and arithmetic operations. Ease is how easy it is for users to formulate queries with that language. Conciseness refers to the number of operations needed to make the query. There is probably a close relationship between ease and conciseness. Something that is concise is probably easy to do. However, a distinction can be made. Ease refers to the user's behavior; conciseness refers to the actual query statement. Let us consider relational completeness. It includes coverage: a relationally complete language must be able to do what relational calculus can do. But relational completeness includes more than coverage, otherwise any general programming language would be relationally complete since

the user can use it to program a relational system. Relational completeness also includes ease and conciseness in that the retrieval should be done in a single self-contained query, possibly including subqueries, that is relatively easy for the user to formulate.

On the criterion of coverage, relational completeness is not good enough for practical usage (Tsur and Zaniola 1984, Subieta and Missala 1987). Probably the most convincing evidence is that no practical query language has been designed that meets only the requirement of relational completeness. All practical query languages provide more coverage than that of relational completeness. They have extra features, such as simple statistical functions and even some report formatting capabilities. Recognizing this inadequacy, Merrett (1978) and Klug (1982) propose extensions to relational algebra to include some aggregate functions. In addition, some rules in relational calculus are not even desirable. For example, the user may want to retrieve duplicate tuples, which is strictly not allowed in relational calculus.

The coverage of relational completeness is actually very limited. Subieta and Missala (1987) have harsh words for the concept of relational completeness: "Relational completeness is a consequence of improper understanding of semantic domains for database query. . . . It is an arbitrary, unmotivated point on the scale of universality of query languages." In rejecting relational completeness, they propose that "the only proper measure of a query language's universality is the power of algorithmic programming languages." Their concept of universality appears to be the same as what we call coverage. This universality can be

easily achieved by embedding any relational query language in a general programming language. However universality (or coverage) by itself is not a good measure of a query language. The ease and conciseness of a query language have to be considered.

Relational completeness insists on ease and conciseness for queries within its coverage only. It does not extend to extra-relational features, such as statistical functions.

Designers of practical query languages usually build extra-relational features into their languages to allow easy formulation of a wider range of queries; that is, they extend coverage while maintaining ease and conciseness. For instance, SQL includes many built-in functions to reduce the amount of programming needed of the user. For example, SQL provides the AVG function which allows a user to get the average salary of all employees with a short SQL query: `SELECT AVG(SALARY) FROM EMPLOYEE`. Without the AVG function, the user would have to write an SQL query to extract the list of salaries, and he would have to use another programming language (such as COBOL) to calculate the average of the salaries. Thus, Chamberlain (1980, p183) points out that an important advantage of SQL is that "having the power and flexibility of a high level query language available at an application programming interface made a significant improvement in productivity of application developers, compared to lower level 'navigational' languages."

For a query language that manipulates relations, relational completeness seems to

be a good property to have. It shows some basic coverage with a certain amount of ease and conciseness for queries within this core coverage. Of course, the ease and conciseness vary from one relational language to another.

It would appear that the same ideas are behind the calculus-based approach to defining ER completeness: the language has to be easy and concise for a certain coverage defined by the ER calculus.

3. Knowledge Level Completeness

Relational completeness belongs to the logical data level since it is concerned with manipulating logical data elements. This logical level completeness is defined without any reference to the physical level "completeness". There are many things that can be done at the physical level that cannot be done by relational algebra. For example, relational algebra cannot transpose or sort a table. Relational completeness is also defined without any reference to implementation details, which is as it should be.

Some researchers believe that ER completeness must be defined based on the data structures used. Dogac et al. (1987) write, "The definition of ER completeness depends on the data structure chosen to represent the entities and relationships". Subieta and Missala (1987) also write that it is impossible to formally define manipulation languages without the formalization of data structures which are to be manipulated. These views undesirably mix the logical data level with the knowledge level.

At the knowledge level, it will not be meaningful to define ER completeness based on any data structures, such as relations. The simple reason is that the data structures are hidden from the interface. When the user does not know anything about the relations, relational completeness will not mean anything to him. Another reason is to allow a clear separation between the knowledge level and the logical data level. The knowledge level does not prescribe the use of any particular data structure in the database. Different systems of the same knowledge level can use different data structures depending on the choice of the systems designers. To define completeness based on data structures will unnecessarily lead to a proliferation of definitions, and consequent confusion.

Therefore, to define knowledge level completeness based on data structures will be to confuse two distinct levels of abstraction. The query language at the knowledge level is not a manipulation language to manipulate data structures. In fact, the term *Data Manipulation Language (DML)* does not apply at the knowledge level.

We will therefore propose a functional approach to defining ER completeness. The definition is based on what can be done at the knowledge level.

> **Completeness of an ER query (retrieval) system is the ability of the system to derive all the domain knowledge that a human being not using additional knowledge about the entities, relationships or their attributes can derive from the same ER model.**

This definition of completeness is an ideal: it makes ultimate use of all the knowledge that has been put inside the ER model. The system is compared with a human being using the same amount of knowledge about the world. Specifically the human being cannot use "commonsense" knowledge about the entities and relationships that is not contained in the ER model. The definition is also universal in the sense that it is not limited to any specific version of the ER model. No real system, with its limited memory space, will be able to reach this ideal level of ER completeness.†

With this definition in mind, we now proceed to describe the various operations and conditions that the retrieval system must provide.

4. Knowledge Level Operations

A retrieval is an operation for the system to perform.‡ This operation may be composed of other operations and/or conditions. An operation is a process that accepts some input and produces some output. A condition can be considered as an operation that returns a Boolean value of TRUE or FALSE. For practical purposes, these operations may be grouped into a few categories based on the operands.

Before we describe the categories, a short note on the notation is necessary.

- O stands for an operation as well as the result of the operation. For

†A restricted version of ER completeness — fundamental ER completeness — is offered later.

‡In fact, any command to the system is an operation for the system to perform. A retrieval usually results in some visible output. Other commands may change the system's knowledge of the domain.

example, +(1,2) stands for the addition operation as well as the result, which is the number 3.

- C stands for a condition as well as the result of the condition. C is either true or false.
- S_i stands for type i. If we want to restrict consideration to an entity type, "E" will be used in place of "S". Similarly, to specify a relationship type, we will use "R" in place of "S".
- $s_{(i,j)}$ stands for instance j of type i. If we want to specify an entity instance, we will use "e" in place of "s". Similarly, we will use "r" in place of "s" if we want to specify a relationship instance.
- $V_k(S_i)$ stands for the set of values of attribute k of the type i.
- $v_k(s_{(i,j)})$ stands for the value of the attribute k of the instance j of type i.
- c stands for a value. This may be a value that is not from any instance or type.

The operations can be divided into the following categories based on the kind of operands.

1. Logical combination of conditions
2. Nesting of operations
3. Operation on values
4. Operation on instances
5. Operation on attribute values of instances
6. Operation on entity/relationship types
7. Operation on the sets of attribute values of the instances in the types

8. Operation on one or more instances and one or more types.

In addition, sub-categories can be derived based on the output of the operations.

The main categories will now be described in detail. The description shows what kind of operations the interface can have. It is not the purpose at this point to show how these operations can be expressed. Examples of the operations will be given, but these do not necessarily show the preferred syntax. Many different query languages can be designed to express these operations.

a. Logical Combination of Conditions

If "C_1" and "C_2" are conditions, then "NOT(C_1)", "NOT(C_2)", "C_1 OR C_2" and "C_1 AND C_2" are conditions.

b. Nesting of Operations

We first define the various kinds of "things" that the system can accept or produce. At the knowledge level, a "thing" can be a list of values, a list of instances, or a list of types. A list can have a single member. A value may be of a certain datatype, such as date, integer or character. A value can be a list of single values, or a list of lists of single values. The instances are of various entity/relationship types; and the types are entity or relationship types. For example, a set of supplier instances can be a thing; a supplier instance can also be a thing.

An operation always operates on certain things and produces other things. In general, an operation may be written as:

$O(X_1, X_2, \ldots, X_n)$, where X_i is a thing of kind "i".

The output of the operation is itself a thing of some kind.

The nesting of operations is defined as follows:

given $O(X_1, X_2, \ldots, X_n)$, where X_i is a thing of kind "i",

and given $O'(\ldots)$ which is a thing of kind "j",

then $O(X_1, X_2, \ldots, X_{j-1}, O'(\ldots), X_{j+1} \ldots, X_n)$ is also a valid operation.

For example, if +(X,Y) operates on two numbers X and Y, and if *(A,B) is a number, then +(X,*(A,B)) is a valid operation.

c. Operation on Values

$O(c_1, c_2, \ldots, c_n)$.

When we consider values, there is not much difference whether we consider these at the knowledge level or the logical data level. A "1" remains a "1" at either level. Therefore these operations are the usual ones applicable to values. The common operations are arithmetic operations, string operations and other mathematical operations such as logarithms, exponentials, set operations on set(s) of values, (for example, membership, subset and set equality conditions), statistical operations on lists of values (for example, minimum, maximum, average, sum and count operations), and the common magnitude comparisons: "=", "<", ">", "≤" and "≥". These operations are well known and need no further description. In addition, since values can be of certain abstract datatypes, operations specialized to these datatypes can be included. For example, if c_1 and c_2 are dates, then $c_1 - c_2$ can produce the number of days between the two dates.

d. Operation About One or More Instances

$O(s_{(i_1,j_1)}, s_{(i_2,j_2)}, \ldots, s_{(i_n,j_n)})$.

Some examples are:

1. The relationship amongst a few instances:
   a. A condition that a list of entity instances are related through a relationship instance. This condition may be written as:

      $\mathtt{related}(r_{(i_1,j_1)}, (e_{(i_2,j_2)}, e_{(i_3,j_3)}, \ldots))$.

   b. A condition that a list of entity instances are related through a relationship type. It does not care about the particular instance of that relationship. This condition may be written as:

      $\mathtt{related}(R_{i_1}, (e_{(i_2,j_2)}, e_{(i_3,j_3)}, \ldots))$.

   c. Another condition is that two entity instances are related through some relationship instance of any relationship type. This condition may be written as:

      $\mathtt{related}(e_{(i_1,j_1)}, e_{(i_2,j_2)})$.

   d. A further possible relationship condition is that two entity instances are related possibly remotely through any number of other relationship instances and entity instances. This condition may be written as:

      $\mathtt{remote\text{-}related}(e_{(i_1,j_1)}, e_{(i_2,j_2)})$.

2. The equality of two instances. This may be written as

   $=(s_{(i_1,j_1)}, s_{(i_2,j_2)})$.

e. Operation About the Attribute Values of the Instances

$O(v_{k_1}(s_{(i_1,j_1)}), v_{k_2}(s_{(i_2,j_2)}), \ldots, v_{k_n}(s_{(n_1,j_n)}))$.

The term $v_{k_l}(s_{(i_l,j_l)})$ stands for the value of attribute k_l of instance j_l of type i_l. We can substitute these symbols with terms from a real ER model. For example, we can substitute "v_{k_l}" with "name", and "$s_{(i_l,j_l)}$" with "employee_1"; thus, $\text{name}(\text{employee}_1)$ may stand for the value "John" of the attribute "name" of the instance 1 of the employee type. The notation above simply says that the operation can involve any number of attribute values from any number of instances.

We can consider a sub-category where the result is a Boolean value. The operations in the category compare the attribute values with other values. Some examples are:

$2 < \text{price}(\text{supply}_1)$,

$\text{price}(\text{supply}_1) = \text{price}(\text{supply}_2)$,

$\text{price}(\text{supply}_1)$ IN (10,12,14), or

substring('Ltd',$\text{name}(\text{supplier}_1)$)

where IN is the membership test, and the last example is a string condition that the name contains the string Ltd. Examples of common conditions that fall under this category are "=", "<", ">", "≤", "≥", and membership condition of an attribute value in a list of values.

Another sub-category is where the result is a list of values. Some examples are:

1. Arithmetic and string operations, for example,

   $2 * \text{price}(\text{supply}_1)$, or

   substring(1, 3, $\text{address}(\text{employee}_1)$),

which can be an operation to get the first three characters of the address.

2. An operation, essential for retrievals, to display these attribute values; the operation may be written as:

   display(name($supplier_i$), address($supplier_i$)).

f. Operation on One or More Types

$O(S_{i_1}, S_{i_2}, \ldots, S_{i_n})$.

We can consider a sub-category where the result is a Boolean value. Some examples are:

1. the subset condition that one type is a subset of another type. This condition may be written as:

   subset(S_{i_1}, S_{i_2}).

2. the proper subset condition, which may be written as:

   proper-subset(S_{i_1}, S_{i_2}).

Another sub-category is where the result is a value. An example of such an operation is counting the number of instances in a type.

Another sub-category is where the result is an instance. An example is the extraction of an instance from a type.

The last sub-category is where the result is another type. Some examples are: union and intersection operations, and the extraction of a subset of instances from a type.

For some complex operations, the output may be a list of values, a list of instances or another list of types.

g. Operation on the Set(s) of Attribute Values of the Instances in the Type(s)

$O(V_{k_1}(S_{i_1}), V_{k_2}(S_{i_2}), . . ., V_{k_n}(S_{i_n}))$.

The term $V_{k_1}(S_{i_1})$ stands for the set of values of attribute k_1 of all the instances of the type i_1. We can substitute these symbols with terms from a real ER model. For example, we can substitute "V_{k_1}" with "Salary", and "S_{i_1}" with "Employee". Thus, Salary(Employee) is a set of the salary values of all the employees.

The output of the operation may be a single value or a list of values. Some examples are: getting the average, total, minimum, maximum or standard deviation of the attribute values. One can think of other statistical analyses such as regression analysis between multiple sets of attribute values within the same set of instances, or other operations on attribute values of different sets of instances.

h. Operation on One or More Instances and One or More Types

$O(s_{(i_1,j_1)}, s_{(i_2,j_2)}, . . ., s_{(i_m,j_m)}, S_{k_1}, S_{k_2}, . . ., S_{k_n})$.

Examples are:

1. A condition that an instance of one entity type is related through a relationship type to N (or more than N, or less than N) number of instances of another entity type. This may be written as:

   $related(R_{j_1}, e_{(i_2,j_2)}, E_{j_3}, =, N)$.

2. Another condition is that an entity instance is IS_A-linked to some instance of an entity type. This condition may be written as:

   $IS\_A(e_{(i_1,j_1)}, E_{j_2})$.

It is not claimed that the above categories of conditions and operations cover all the possibilities. However it can be claimed that these categories cover the common conditions and operations that are found in popular database query languages. For example, SQL provides these conditions/operations: arithmetic operations +,-,*,/; simple statistical functions COUNT, SUM, AVG, MAX, MIN; and comparison operations =, <, >, ≤, ≥ and the special operations SORT BY, GROUP BY, IN, EXISTS, DISTINCT and UNION. All these belong to one category: the operations on values. Note that a value may be a list of values, and therefore a relation is a value. This is perhaps to be expected, since the relational model has no concept of entity/relationship instances or types.

We can now offer an alternative, more concrete, definition of ER completeness: it is the ability of the system to perform all the possible operations in each of the categories. This is an ideal level of completeness, since no real system will have all the possible operations. In contrast, relational completeness which requires only a few operations is less complete than many actual query languages.

Closure of the Operations

We will conclude this section on ER completeness with a discussion on the closure of the operations. The closure of a query language refers to the ability of one query to accept as input the result from another query. In relational languages, the output of a query is a relation (however big or small) which can be used in subsequent queries. But note that not all relational operations will accept any relations: the union operator requires relations to be union compatible. SQL also treats some results as simply a list of values for use in the "IN"

function. Having the property of closure will allow for nesting; that is, a query can be composed of subqueries. It will also allow a query to be broken into two or more queries. The user can choose whichever approach is easier.

In a more mathematical sense, closure implies that all operations produce "things" of the kinds already in existence. As long as the operations allowable in the ER language are restricted to those that produce "things" of the kinds already described, closure is maintained. The language to be described later allows only operations that produce values, sets of values, instances, set of instances and types. There are no other possible outputs.

We can distinguish two kinds of ER retrieval queries. The first kind includes the entity-subset/relationship-subset query where the user selects only one or more entity/relationship types without selecting any attributes. The conditions can involve any number of other types and instances. Some examples of subset queries, using the KQL syntax to be described later, are:

i.
```
select supplier
where supplier supplies part
```

and

ii.
```
select bigsupply = supply
where supply quantity > 1000.
```

For this kind of query, the system will not produce any visible output to the user since the "output" is meant to be used in subsequent queries. Thus, the first example should be used only as the inner query in a nested query; the second example may be used as a query before another query.

The property of closure applies to this kind of output since the output is merely another entity/relationship type. The new types can be used in subsequent queries wherever the original types can be used. The subset queries resemble the defined views of SQL.

The second kind of retrieval query is the value query where the user can combine any attributes from any entity/relationship types into a value. This arbitrary combination of attributes has no meaning in the ER model; it is neither an entity nor a relationship. It is just a value, which may be a list of values, to be presented to the user in the manner chosen by the user. An example query that will give this type of output is:

```
select department name, supplier name, supplier city
where department city = supplier city.
```

The outputs from the value queries can be subdivided into three collectively exhaustive classes: a single value output, a one-column output, and a multi-column output. These outputs can also be treated as values that can be used for subsequent operations or comparisons. To be fully flexible, a query that produces a single value output should be able to appear anywhere that a single value (of the same datatype) is expected, for example,

```
select supplier name
where supplier supply part,
      supply quantity > 2 * (sp is an instance of supply
                             select min(sp quantity)).
```

(The comma in the where-clause denotes the AND operation.)

Similarly, a query that produces a list of single values such as (12,43,4) should

be able to appear anywhere that a list of values is expected. Thus the query may appear during a set operation such as membership testing, for example,

```
select supplier name
where supplier city IN (select department city).
```

Even a multi-column table output may be used in subsequent queries if there are operations that accept such a table.

The treatment of a query output as a value is similar to SQL. Although a relational query language produces relations, the user has the choice to use some outputs as variables, such as in the following SQL query:

```
SELECT *
FROM S
WHERE S CITY IN
     (SELECT CITY
     FROM D).
```

Since the output of an ER query can be used as input for another ER query, closure is maintained for the ER language.

B. THE KQL LANGUAGE

The previous section describes the knowledge level interface in a way that is independent of any specific language. One can now proceed to design different languages, including graphical languages, to enable the user to describe the various operations and conditions.

We will describe the syntax of a specific language for knowledge level query – KQL.†

1. The General Format

The general format of a query is described here. The use of specific conditions and operations will be described in the later sections. The syntax for the numerous conditions/operations has to be carefully designed so as to provide natural, unambiguous, easy, and concise expressions.

Many guidelines for query language design are considered in the design of KQL. For example, KQL has a clear separation of outputs and conditions. This clear separation has worked very well in relational query languages such as SQL and QUEL. Also considered are guidelines such as minimum syntax, consistency and flexibility as proposed by Zloof (1978). The syntax must be simple and straightforward. Admittedly this is quite a subjective criterion. The KQL syntax synthesizes from those of SQL and QUEL, the two major relational query languages. Many of the keywords from these two languages are considered redundant and not used in KQL; for example, keywords like FROM, GROUP BY, HAVING, and RANGE OF are excluded. Consistency means that the operations must have the same meanings in all situations. Flexibility means that a query can be formulated in different ways depending on the user's choice. KQL provides the primitive conditions as well as certain common combination conditions so that the user need not decompose a common condition into many primitive

†Even though relational completeness is not acceptable as a definition of ER completeness, it is nevertheless a common, and thus useful, yardstick for comparing the power of retrieval languages. It is shown in Appendix A that KQL is relationally complete.

conditions. To maximize the flexibility, all operations with suitable outputs can be nested.

The general structure of a query is given by the following BNF grammar:†

<query> ::= [<instance-clause>]
<action-clause>
[<report-formatting-clause>]
[<where-clause>].

The general interpretation of a query is as follows: the system will find all possible instances defined in the <instance-clause> and meeting all the conditions specified in the <where-clause>; the system then performs the actions specified in the <action-clause> in accordance with any additional instructions specified in the <report-formatting-clause>.

The instance-clause declares the instances. The user must provide a name for the instance and declare the type that the instance belongs to.

<instance-clause> ::=
<instance-identifier> <instance-type-connection> <type>
{,<instance-identifier> <instance-type-connection> <type>}

<instance-identifier> ::=
<entity-instance-identifier>
| <relationship-instance-identifier>

†A pair of square brackets in the grammar means that the content inside the brackets is optional. Curly brackets mean that the content can be listed zero or more times.

| <role-instance-identifier>

These instance-identifiers are terminal. They are names given by the user at the time the query is made.

<instance-type-connection> ::=

IS | IS A | IS AN

| IS AN INSTANCE OF

<type> ::= <entity-type-name>

| <relationship-type-name>

| <role-name>

All names are terminal.

<action-clause> ::=

<select-clause> [<delete-clause>] [<change-clause>]

| <delete-clause> [<change-clause>]

| <change-clause>

The action-clause can be all of the three clauses allowing the user to perform all three functions in a single query. For now, we will only discuss the select-clause.

<select-clause> ::=

<entity-relationship-subset-select-clause>

| <value-select-clause>

<entity-relationship-subset-select-clause> ::=

SELECT [<type-name> =] <instance-identifier>

{, [<type-name> =] <instance-identifier>}

The type-name is a user-given name, and is terminal.

<value-select-clause> ::=

SELECT <select-item> {,<select-item>}

<select-item> ::=

<instance-identifier> <attribute-name>

| <instance-identifier> *

| <operation>

The "*" means to select all attributes of the instance. An <operation> is an operation on values. It includes the arithmetic and statistical operations to be described later.

The report-formatting-clause contains instructions for organizing the output of the query. The instructions may include the common sorting commands and more advanced commands such as sub-totals, totals, titles and even graphics. In general, any command that works on a single table can be included here. The exact syntax depends on the specific instructions. Though these commands will be very useful to the users, they are not directly related to the ER model and will not be discussed much in this paper.

<report-formatting-clause> ::=

REPORT <formatting-instructions>

The where-clause contains the conditions that the instances specified in the query must meet.

<where-clause> ::=

WHERE <condition-list>

<condition-list> ::=

<condition-andlist> [OR <condition-list>]

<condition-andlist> ::=

<condition> [, <condition-andlist>]

<condition> ::= NOT (<condition-list>)

A <condition> can also be one of the many conditions in the various categories described previously. It can include any appropriate operations and values; for example, "60 < max(employee age)" can be a condition. The exact syntax can only be given for specific conditions and operations. Since a retrieval is an operation, a suitable condition can include a sub-query.

2. Fundamental Specific Operations

Obviously the number of possible operations for the knowledge level interface will be very large, if not infinite. Each query can be considered as an operation. Some restrictions have to be applied. We will first discuss the fundamental conditions in an ER model. In addition, we will consider the operations that are provided in our benchmark language SQL.

The fundamental operations in the ER model are defined as those operations that we can express in the ER model. These are the facts that the ER model can contain. These include only the following:†

1. That an instance belongs to a certain entity type
2. That an instance belongs to a certain relationship type
3. That two (or more) entity instances are related through a certain

†The inheritance for is-a relationships is generally considered as an extension to the ER model, and it will be treated in a later section.

relationship instance. In addition, the instances may have certain roles in the relationship.

4. That the value of a certain attribute of a certain instance equals a certain value.

It is debatable whether the cardinalities (that is, the lower and upper degrees) of a relationship can be counted as being contained in the ER model. If this is considered as contained in the ER model, then the fundamental operations should include an operation to count the number of entity instances (of a certain type) that are related (through a certain relationship) to an entity instance. The cardinalities seem to be contraints on the behavior of the entity instances. Also, many discussions of the ER model omit the cardinalities, or simply uses "one" or "many" without specific numbers. It seems reasonable to exclude this operation from the list of fundamental operations.

All the operations in the above list are conditions.

We may define a **fundamental level of ER completeness** as the ability to specify all the conditions that we can express in the ER model. Thus any ER language that allows the conditions in the above list will meet the fundamental level of ER completeness.

We will now define the syntax and semantics of each of the fundamental conditions.

a. Entity/Relationship Type Conditions

This is the condition that declares an instance to be of a certain entity/relationship type. The condition is stated in the <instance-clause> of the query. The syntax has been described in the previous section on the general format of a query. Some examples are:

```
1.  s is a supplier
2.  sp is a supply
3.  sp is a supplies
```

To reduce the number of instance-of declarations, the user may omit the instance declaration and use the actual entity type names, relationship type names or role names in the <select-clause> and <where-clause> wherever there are no ambiguities. The actual type name will be understood as the name for an instance of that type. However, the "redundant" instance-of statements may enhanced the clarity of the query. For example, both of the following queries are acceptable:

```
s is a supplier
select s name.
```

and

```
select supplier name.
```

b. Relationship Existence Condition

This is a condition relating two entity instances and either a relationship instance or a role instance. The syntax is:

<relationship-existence-condition> ::=

<entity-instance-identifier>

[(<role-name>)]

<related-to-instance>

[(<role-name>)]

<entity-instance-identifier>

<related-to-instance> ::=

<relationship-instance-identifier>

| <role-instance-identifier>

The first <role-name> refers to the role of the first entity instance in the relationship, the second <role-name> refers to the role of the second entity instance in the relationship. In most cases, these role-names can be omitted without any ambiguities. In rare cases, these are needed. One such case is when two entity types each has two roles in the same relationship.

This condition is true if the two entity instances are related through the relationship instance, with the stated roles. It is false if the two entity instances are not related through this relationship instance with the stated roles; the two entity instances may be related through this relationship instance but with different roles, they may be related through another relationship instance, or one of them may be related through this relationship instance to some other entity instance.

A <role-instance-identifier> is a one-sided synonym for a <relationship-instance-identifier>. It can be used only in a certain sequence.

Consider the ER model where there is a SUPPLY relationship between entities SUPPLIER and PART, and the role of SUPPLIER is SUPPLIES while the role of PART is SUPPLIED_BY, then "supplier supplies part" is a valid condition, but "supplier supplied_by part" is not. In the condition "supplier supplies part", the identifier "supplies" represents the relationship instance that involves the instances "supplier" and "part".

Example queries with relationship existence conditions are:

```
i.   s is an instance of supplier,
     p is an instance of part,
     sp is an instance of supply
     select s name, p name, sp price
     where s sp p.
ii.  s is an instance of supplier,
     p is an instance of part,
     sp is an instance of supplies
     select s name, p name, sp price
     where s sp p.
```

c. Equality Comparison of Attribute Values

The comparison operator to be described is '='. The syntax is:

<equality-attribute-value-condition> ::=

<attribute-value> = <attribute-value>

| <attribute-value> = <user-given-value>

| <user-given-value> = <attribute-value>

<attribute-value> ::=

<instance-identifier> <attribute-name>

Both <attribute-name> and <user-given-value> are terminal. A <user-given-value> is any value such as an integer or a character string that is accepted by the system.

This concludes the description of the fundamental conditions.

3. Additional Specific Operations

When we consider the additional operations, we are faced with a large number of possibilities. Many of these have been described under the various categories of operations. It does not serve much purpose here to describe the syntax of every possible operation. Since we envisage that the new system will be built on top of a relational system and will interface with the relational system through SQL, and since we are using SQL as a benchmark language, we will consider the additional operations that are available in SQL. The syntax for the "missing" operations, especially the well-known operations such as string operations and set operations, can be easily added when necessary.

The following additional operations will be considered:

1. Arithmetic operations on attribute values. These include multiplication, addition, subtraction and division.
2. More comparison conditions: ">" and "<". The equality "=" comparison described under the fundamental conditions will be extended. The usual combinations "≤" and "≥" will also be included.

3. Simple statistical operations: COUNT, SUM, AVG, MAX and MIN.
4. Miscellaneous other functions that are available in SQL, such as IN, EXISTS and UNION. SQL does not provide explicit intersection and difference functions, which can be built by using the EXISTS function. However for ease of use, a fully implemented KQL should provide explicit intersection and difference functions so that users can use these if they choose not to use the EXISTS function. The syntax for these should be similar to that for the UNION function.

a. Arithmetic Operations

The arithmetic operations of addition, subtraction, multiplication and division will be described here. These operations are applied to attribute values and/or other values provided by the user. The syntax is as follows:

<arithmetic-operation> ::=

[<sign>] <arithmetic-term>

{<add-minus> <arithmetic-term>}

<arithmetic-term> ::=

<arithmetic-subterm>

{<time-divide> <arithmetic-subterm>}

<arithmetic-subterm> ::=

<simple-value> | (<arithmetic-operation>)

<sign> ::= + | −

<add-minus> ::= + | −

<time-divide> ::= * | /

<simple-value> ::=

<attribute-value>

| <user-given-value>

| <statistical-operation>

The term <statistical-operation> will be described later.

An example of query with arithmetic is:

```
e1 is an instance of employee,
e2 is an instance of employee
select e1 name
where e1 salary = 0.5 * e2 salary,
      e2 name = 'MARY'
```

b. Additional Comparisons

This section describes the syntax for the comparison conditions. The arithmetic operations are incorporated.

<comparison-condition> ::=

<value> <comparison-condition-name> <value>

<comparison-condition-name> ::=

= | < | > | <= | =>

<value> ::= <arithmetic-operation> | <simple-value>

c. Statistical Operations

The syntax for statistical operations is:

<statistical-operation> ::=

<instance-statistical-operation>

| <attribute-statistical-operation>

<instance-statistical-operation> ::=

COUNT

[UNIQUE]

(<entity-instance-identifier>

[FOR EACH <grouping-values>])

<attribute-statistical-operation> ::=

<attribute-statistical-operator>

[UNIQUE]

(<instance-identifier> <attribute-name>

[FOR EACH <grouping-values>])

<attribute-statistical-operator> ::=

COUNT | MAX | MIN | AVG | SUM

<grouping-values> ::=

<grouping-value> {, <grouping-values>}

<grouping-value> ::=

<entity-instance-identifier>

| <instance-identifier> <attribute-name>

The interpretation of a statistical operation is: for each combination of the <grouping-values>, perform the operation on the instances (or the attribute values of the instances), provided the <grouping-values> and the instances must

satisfy the where-conditions.

An example of a query with statistical operation is:

```
select part number, part name, avg(supply price for each part)
where supplier supply part.
```

This query gets the part's name and number, and the average price that it is supplied by its suppliers.

d. Some other operations

Here we will consider the equivalent of some miscellaneous functions that are available in SQL. Specifically, we will consider the IN, EXISTS and UNION functions.

The IN function is a simple test of membership of one value against a list of values. The KQL syntax is similar to SQL's.

<membership-condition> ::=
    <value>
    IN
    (<list-of-values>)
<list-of-values> ::=
    <value> {, <value>}

<list-of-values> ::=

[<instance-clause>]

SELECT <value>

[<where-clause>]

The EXISTS function simply checks for the presence or absence of output in a retrieval. The KQL syntax is similar to SQL's.

<exists-condition> ::=

EXISTS

(<retrieval-query>)

<retrieval-query> ::=

[<instance-clause>]

<select-clause>

[<report-formatting-clause>]

[<where-clause>]

The UNION function in KQL can be divided into two kinds. One is the union of sets of entity instances, where the instances must all be of the same entity type. The second is the union of tables of attribute values. This is similar to the union of two relations. The syntax is:

<union-retrieval-query> ::=

<retrieval-query>

[UNION <union-retrieval-query>]

C. EXTENSIONS TO THE KQL LANGUAGE

This section will discuss some useful extensions to the KQL language described previously. One extension is the incorporation of inclusion (IS_A) relationships into the ER model and how this is handled by KQL. The second extension describes some useful conditions that are combinations of the specific conditions described previously. The addition of these combination conditions may make certain queries easier to write.

1. Special Relationships

Some entity types may be considered to be subtypes of other entity types. For example, ENGINEER entity type can be a subtype of EMPLOYEE entity type. It is usual to say that ENGINEER is IS_A related to EMPLOYEE, or more simply that ENGINEER IS_A EMPLOYEE.

The IS_A relationship means that an instance of the subentity can exist only if it also exists at the superentity. We do not consider many variations of IS_A discussed in the literature, such as an IS_A where a superentity instance must exist as one of the subentities, or an IS_A where the subentities are mutually exclusive. A common property applies to all these variations: attributes are inherited. Following the common version discussed in the literature, the IS_A relationship used in this research has no attributes of its own.

The attributes of the superentity type (EMPLOYEE in this case) can be inherited by the subentity type (ENGINEER). This is the usual attribute inheritance discussed in the literature (Buneman 1986). When formulating queries, the user

can treat the attributes of the superentity as if they already belong to the subentity. Suppose EMPLOYEE has attributes NAME, NUMBER and ADDRESS, and ENGINEER has attribute PROFESSION. The user can ask the following queries:

i. `select engineer name`
`where engineer number = 12354`

and ii. `select engineer name`
`where engineer profession = electronics`

When it comes to database usage at the knowledge level, we would like "inheritance" in every direction. We will propose a *knowledge level principle of attribute and relationship inheritance*:

> By refering to an entity E, it should be possible to refer directly to attributes and relationships that are not directly of E but are of entities that are IS_A-linked to E, however remote the linkage. If E_1 and E_2 are subentities of E, then E_1 is IS_A-linked to E, to E_2, and to any other entity that is IS_A-linked to E or E_2.

This principle may be viewed as a further application of the concept of entity identity. All the instances which are IS_A-linked are actually different manifestations of a single entity instance.

Consider the IS_A relationships shown in the ER model in Figure 1. The user should be able to make the following queries:

1. `select engineer name`

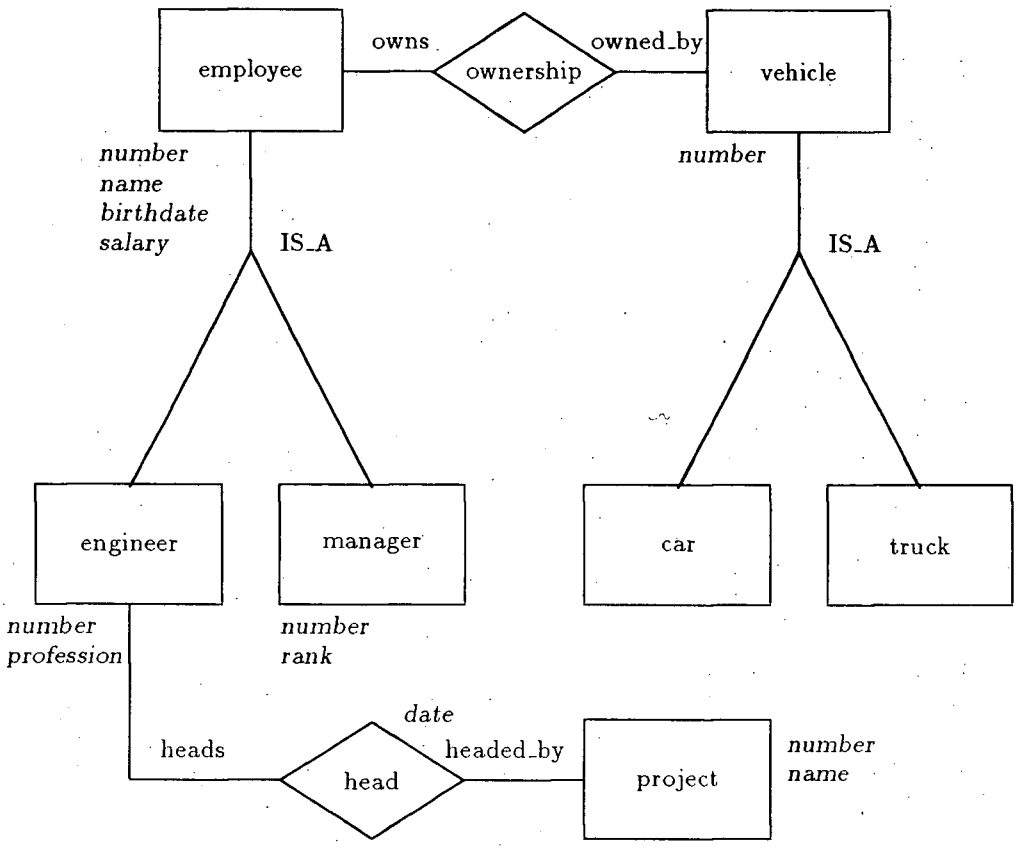

Figure 1: IS_A Relationships in an ER Model

```
2.  select engineer rank
3.  select employee rank, employee profession
4.  select employee name where employee heads project
5.  select manager name where manager owns vehicle
6.  select manager name where manager owns car
```

Note that inheritance can occur at both ends of the relationship, as shown in the last example.

With the introduction of IS_A relationships, we want to allow new conditions about these. The first condition is about an entity instance and an entity type. The syntax for the condition is:

<isa-relationship-condition> ::=

<entity-instance-identifier> IS_A <entity-type-name>

This condition is true if the entity instance is a member of the entity type. That is, there is an instance in the entity type that has the same key values as the entity instance in the condition, and the entity type is IS_A-linked to the entity type of the entity instance in the condition. An example query with this condition is:

```
e is an instance of engineer
select e name
where e IS_A manager
```

This query will select the engineers who are also managers.

The second condition is the equality of two entity instances. This is true if the two entity instances are of entity types that are IS_A-linked and their key

values are the same. The syntax for this condition is:

<entity-equality-condition> ::=

<entity-instance-identifier> = <entity-instance-identifier>

An example query retrieving the same information as the previous query is:

```
e is an instance of engineer,
m is an instance of manager
select e name
where e = m
```

2. Combination Conditions

In real life, we are often interested in knowing whether two entity instances are related but we may not be interested in the particular relationship instance. This condition and its negation can be expressed using a combination of the relationship-existence condition and the EXIST condition.† However it will be harder for the user. The syntax for this condition is:

<combination-relationship-condition-1> ::=

<entity-instance-identifier>

<relationship-related>

<entity-instance-identifier>

<relationship-related> ::=

<relationship-type-name>-RELATED

| <role-name>-RELATED

---

†How this combination condition and others can be decomposed into more primitive conditions is shown in the later chapter on Prolog implementation. However these combination conditions can be understood without knowing the decompositions.

This condition is true if the two entity instances are related through some instance of the relationship type. It is false if there are no instances of the relationship type that relates the two entity instances. An example is:

```
s is an instance of supplier,
p is an instance of part
select s name, p name
where s supply-related p
```

This condition is extended further to allow easier writing of more complex conditions which we believe are used frequently, for example, in the query "Find students who take more than 4 courses." It also allows the counting of instances to be done without explicitly using the COUNT operation. The syntax of the new condition is:

<combination-relationship-condition-2> ::=

<entity-instance-identifier>

<relationship-related>

<number-specification>

<entity-type-name>

<number-specification> ::=

ALL | NO

| [<comparsion-condition-name>] <integer>

The condition is true if the entity instance is related through the relationship type to the specified number of instances of the specified entity type. Some examples of queries with this combination condition are:

i. `select supplier name`

`where supplier supply-related > 5 part`

(This gets the supplier who supplies more than 5 part instances.)

and ii. `select supplier name`

`where supplier supply-related no part`

(This gets the supplier who does not supply any parts.)

D. KQL UPDATES AND DEFINITIONS

KQL allows the user to define his world of entities and relationships to the system. This corresponds to the data definition process for relational DBMSs. The language must also allow modifications to the entity/relationship types. The syntax for these is given in Appendix B.

The creation of an entity instance is very simple, requiring only the values of the attributes. The creation of a relationship instance involves specifying (retrieving) the participating entity instances.

Deletion of an instance or changes to its attributes can only be done if the instance is first retrieved. Thus a good retrieval language is half the solution for deletions and changes. The syntax for creation and modification of instances are shown in Appendix C.

Appendix B and C also include details on the creation and use of IS_A-linked entities.

VII. COMPARATIVE ANALYSIS OF KNOWLEDGE LEVEL VERSUS LOGICAL DATA LEVEL INTERFACES

This part will show that the knowledge level interface requires less knowledge of the user and reduces the number of data manipulation operations that the user has to perform, as compared to the SQL interface, which is an example of the logical data level interface. In the process, it will also be shown how the major relational difficulties described in Chapter II will be avoided when using KQL. A further comparison of KQL and SQL based on the query language comparison frameworks in the literature is done in Appendix D.

A. KNOWLEDGE REQUIREMENT

To allow simple reference, we will use the term *K-users* to denote users using a knowledge level interface and the term *D-users* to denote users using a data level interface.

Both D-users and K-users need to know the domain. They both need to know the syntax of some interface language. In addition D-users need to know the data elements in the database and the correspondence between these data elements and the domain. Pahwa and Arora (1985) write, "Relational database systems impose on the user the responsibility of specifying logical level access paths in order to retrieve information. As a result, the user must be aware of the relevant relations and all possible interconnections among these relations . . . this assumed level of user sophistication is a constant source of difficulties."

It is acknowledged that not everybody sees the world in the same way. While

some may see the world in terms of entities and relationships, others may see it in some radically different terms. We can divide the users into two groups: the ER group who sees the world in ER terms, and the non-ER group who sees the world in other terms. But nobody sees the world as a set of relations.

The ER group is of considerable size, as evident by the widespread use of the ER model. Let us consider this group when they use the ER−>Relational† approach to design the database (Chen 1976, Poonen 1979, Braind et al. 1985, Ling 1985, Parent and Spaccapietra 1985, Storey 1986, Teorey, Yang and Fry 1986, Azar and Pichat 1987, Reiner et al. 1987, Schuldt 1987, Brady and Dampney 1987). K-users in this group will know everything they need to know about the database by the time the conceptual ER model is designed. D-users also know as much as the K-users. However, D-users need to know more. They must know the relations, how to map from the relations to their ER world, and how to map from their ER world to the relations.

Thus for this group of ER users who use the ER−>Relational approach, what K-users need to know is strictly a subset of what D-users need to know. The additional amount of knowledge required of D-users is not trivial. The logical conclusion, assuming that users have limited information processing capability, is that these users will find the knowledge level system less taxing than the data level system.

Let us consider those in the ER group who do not use the ER−>Relational

†This is used to refer to the database design approach where a conceptual ER model is first designed, and it is then used to design the relations.

approach. K-users by definition of the ER group will be designing the database using the ER model. D-users who have to use the relations may design these using some other approach, such as the functional dependency approach. In this case, it would appear that D-users need to map the relations to their world, which they see as consisting of entities and relationships. K-users, on the other hand, have no need to make any transformation; the database interaction uses the same terms as they think about the world. Again, K-users need to know only a subset of what D-users need to know.

Now we consider the non-ER group. Suppose this group thinks of the world in terms of an arbitrary X model. D-users in this group need to map relations into the terms of the X model. K-users also need to map the ER terms into the X model terms. Given that the X model is not specified, it is not clear whether K-users or D-users have the advantage.

For the non-ER group, future research may try to design a knowledge level interface based on the X model. If this is successful then again K-users in this group will need to know only a subset of what D-users need to know.

Nevertheless, even K-users in the non-ER group may have an advantage in using the ER model over D-users using the relational model. It would appear that K-users need to know less, since knowledge is not duplicated but logical data elements are. For example, the relations have to duplicate key fields to use as logical pointers. Furthermore, the ER model contains more relevant knowledge which will reduce ambiguities. In contrast, a relation, suffering from semantic

overloading, is an oversimplified representation that omits relevant details. The result is that a relation is open to multiple interpretations (Borkin 1980). For example, the user will not know whose age is represented in the relation R(SUPERVISOR, AGE, EMPLOYEE). A more complex example is the relation R(PERSON, WINE, MEAT) which may represent that certain persons like certain wines and certain meats, or that only certain combinations of wine and meat are liked by certain persons.

In concluding this section, we see that there are three distinct types of knowledge distributed among the users and the database systems. These are domain knowledge, data knowledge and knowledge of the transformation between the first two types. A knowledge level system knows all three types, so that K-users only need to know the domain knowledge. The data level system only knows the data knowledge; as a result, D-users have to know all three types of knowledge. This is illustrated in Figure 2.

B. DATA MANIPULATION OPERATION REQUIREMENT

The previous section shows that K-users do not need to know the data elements. It follows from logic that K-users will do fewer data manipulation operations than D-users. Data manipulation operations include the actual operation performed on the given data as well as the process of finding and deciding the relevant data. This section examines the specific situations that K-users can avoid the data manipulation operations. It also shows how the difficulties of the relational queries are avoided.

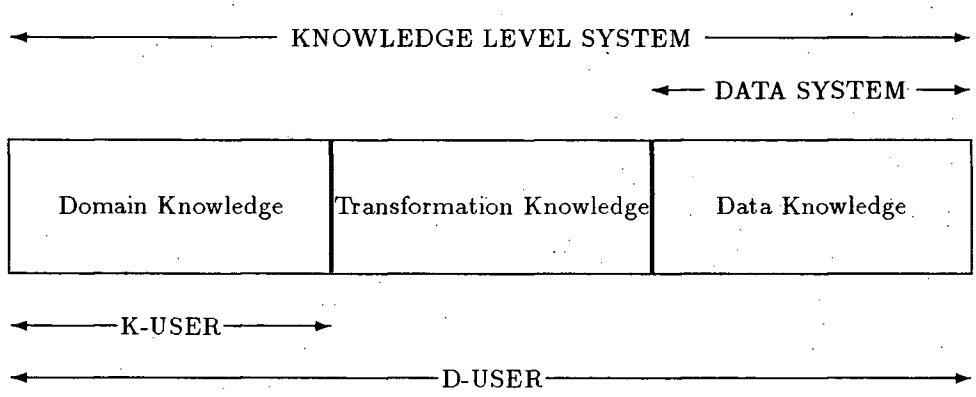

Figure 2: Knowledge Known by Systems and Users

As stated with the hypotheses, we are interested not only in the end-users making retrieval queries but also in designers and programmers interacting with the database system. Hence we will examine the three main types of interaction: definition of the world, updates and retrievals.

1. Data Definition

In defining the real world to the database system, K-users need to define the entities, the relationships, and their attributes. They do not have to decide how to link the entity instances in the relationships.

D-users, however, need to do the linkages by defining foreign keys. They also have to decide whether or not to put the relationship attributes together with the entity attributes in a single relation. These are logical data manipulation operations that simply do not concern K-users.

With the IS_A relationships, D-users need to decide how to represent these IS_A-linked entities with some relations. As seen in Chapter II there are a few possible relational representations, each with some disadvantages. D-users have to evaluate and choose one representation. Again, K-users need not perform this data manipulation operation.

2. Updates

a. Deletions

K-users do not delete records. They delete instances, with commands such as "`delete supplier where supplier name='IBM'.`" Since the database system understands the entity supplier and the relationships that involve supplier, it can automatically delete the appropriate relationship instances so as to maintain the integrity of the database.

D-users, on the other hand, have to do the follow-up deletes of records. They have to decide on the appropriate relations and either delete some tuples or set some columns of some tuples to null. Alternatively, D-users have to define and maintain triggers to do the follow-up deletions. These data manipulation operations are avoided by K-users.

In deleting a relationship instance, D-users need to find out where the data are stored, and then either delete a tuple or set certain columns to null. A real world deletion is translated into one of two data manipulation operations.

Where IS_A-linked entities are involved, deleting a superentity instance will mean the deletion of the subentity instances of the same key values. Again, the KL interface can take care of these follow-up deletion of instances.

D-users have even more data manipulation operations to perform. They need to know where to "post" the deletion of a superentity instance. Some representations

of IS_A-linked entities do not have a relation for superentities; some do. D-users need to examine the relations to decide what to do.

b. Changes and Insertions

D-users have to find out where to post the changes or insert the data. This search for the relevant relations and columns is similar to those required for deletions, as discussed above.

In addition, when inserting a relationship D-users need to check that entity keys are not non-existent. They either retrieve the entity keys from some relation or define some constraints in terms of the columns of some relations. These data manipulation operations are avoided by K-users.

3. Retrievals

SQL retrievals are usually classified based on the type of data manipulation operations. This is a consequence of the fact that SQL is meant for manipulating data elements. These are the common classes:

1. simple one table retrieval
2. join operations, where two or more relations are joined
3. chaining, also known as nesting, composition, combination, or queries with sub-queries (the terminology varies among researchers)
4. grouping of items
5. specification of variables to range over the same relation
6. set operations
7. univeral quantification

8. combination, where a few of the above operations are done.

a. Join Operations

There are two real world purposes for performing joins on relations. One is to re-create the relationship between entities. This involves joins based on the keys and foreign keys. The other is to compare two attribute values, and it may involve the non-key columns of the relations.

For the second purpose, K-users simply compare attribute values of the instances without bothering about which columns of which relations. For the first purpose, K-users simply specify the existence of the relationship, for example,

K-users:
```
select supplier name, part name
where supplier supplies part
```

D-users:
```
SELECT S.SNAME, P.PNAME
FROM S, SP, P
WHERE S.SNO = SP.SNO
   AND P.PNO = SP.PNO
```

D-users have to know that these two conditions performing joins are needed to form the relationship. More fundamentally, D-users must be aware that there is a need for the join operation.

In cases where the key consists of multiple columns, D-users have to specify a few equality conditions just to perform a single join operation. For example, they may have to specify that TABLE1.COLUMN1 equals TABLE2.COLUMN1 and

TABLE1.COLUMN2 equals TABLE2.COLUMN2. Recognizing this difficulty, Chamberlain (1978) suggested an additional function in SQL to reduce these multiple conditions. The function allows D-users to simply say "TABLE1 MATCH TABLE2". However, this function has its limitations. Every column to be matched must have the same name in both tables. Furthermore, any columns with the same name in both tables will be matched; thus, the user may unintentionally specify more equality conditions than he wants.

We see that K-users need not explicity perform any join operations.

b. Grouping by Instances

The number and order of keywords needed for some operations can be confusing. For example, Welty (1985) found that subjects who had gone through 14 sessions learning SQL had a lot of difficulty writing queries that involved the "WHERE . . . GROUP BY . . . HAVING . . ." statement. These keywords are unavoidable in SQL; the improvement suggested by Welty (1985) is only to permit flexible sequencing of these three keywords. The knowledge level interface can help to reduce the number of keywords. For example:

D-users:
```
SELECT S.SNAME
FROM S , SP
WHERE S.SNO = SP.SNO
GROUP BY S.SNO
HAVING COUNT(*) > 1
```

K-users: `select supplier name`

`where supplier supplies-related >1 part`

Alternatively,

```
select supplier name
where supplier supply part,
    count(part for each supplier) > 1
```

The improvement is in the reduction of key phrases, which will reduce possible confusion of the key phrase sequence.

c. Universal Quantification

The knowledge level also allows more direct specifications for queries involving universal quantification of instances. Comparatively, the SQL query uses very roundabout ways. Consider the query to retrieve the suppliers who supply all parts. K-users give a very direct specification:

K-users: `select supplier name`

`where supplier supplies-related all parts.`

D-users, on the other hand, have to specify the selection of a supplier such that there are no parts that are not supplied by the supplier. They use the double negative. In addition, the need to carefully manipulate three relations complicates the process:

D-users:
```
SELECT S.SNAME
FROM S
WHERE NOT EXISTS
    (SELECT *
    FROM P
    WHERE NOT EXISTS
        (SELECT *
        FROM SP
        WHERE SP.PNO=P.PNO
          AND SP.SNO=S.SNO))
```

Another similar example is the query to retrieve the suppliers who do not supply any parts. K-users simply state that condition directly. D-users have to be more procedural. They have to specify finding a tuple in relation S such that there is no tuple in relation SP with the same SNO value. Thus the knowledge level query is less procedural than the SQL query.

D-users:
```
SELECT S.SNAME
FROM S
WHERE NOT EXISTS
    (SELECT *
    FROM SP
    WHERE S.SNO=SP.SNO)
```

K-users:
```
select supplier name
where supplier supplies-related no part.
```

We see that in these cases K-users need fewer subqueries, nestings and EXISTS conditions.

d. Specification of Multiple Instances of the Same Type

Greenbatt and Waxman (1978) found that subjects had considerable difficulty in cross-linking in one table. Cross-linking involves using the same relation more than once in the query. Subjects had to perform this operation for queries such as "Who is younger than John's manager?" Consider the following relation:

```
EMP(NAME, SALARY, AGE, MANAGER).
```

Some possible SQL formulations of the above query are given below:

Version 1:

```
SELECT NAME
FROM EMP
WHERE AGE <
     (SELECT EMP2.AGE
     FROM EMP EMP1, EMP EMP2
     WHERE EMP1.NAME='JOHN'
       AND EMP1.MANAGER=EMP2.NAME).
```

Version 2:

```
SELECT NAME
FROM EMP
WHERE AGE <
     (SELECT AGE
     FROM EMP
```

```
WHERE NAME=
     (SELECT MANAGER
     FROM EMP
     WHERE NAME='JOHN')).
```

Version 3:

```
SELECT EMP3.NAME
FROM EMP EMP1, EMP EMP2, EMP EMP3
WHERE EMP1.NAME='JOHN'
  AND EMP1.MANAGER=EMP2.NAME
   AND EMP3.AGE < EMP2.AGE
```

Assume that the real world has an entity type EMPLOYEE that has a relationship MANAGER with itself, and the roles of EMPLOYEE in this relationship are MANAGES and MANAGED_BY. The knowledge level query for the same question is:

```
e1 is an employee,
e2 is an employee,
e3 is an employee
select e3 name
where e3 age < e1 age,
     e1 manages e2,
     e2 name = 'John'.
```

At the knowledge level, defining multiple instances of the same type is similar to defining multiple instances of different types. No extra difficulty is expected.

Further simplification can be expected for K-users in that they avoid the join operations that almost invariably accompany the multiple use of one relation.

e. Nesting

As seen in the previous examples, many KQL queries do not use nesting while the corresponding SQL queries do use nesting. Nesting of queries have been found to be difficult (Welty and Stemple 1981).

There are a few different occasions that require nesting SQL queries. One is for queries involving the logical quantifiers. These SQL queries are formulated roughly along this structure: find A such that A (and/or B) cannot be found under some other conditions. This is nesting. The KQL queries provide more direct expressions which avoid the nesting operation.

Sometimes, the nesting operation is used in place of the join operation, for example, writing

```
SELECT SNAME
FROM S
WHERE S.SNO IN
     (SELECT SNO
     FROM SP)
```

instead of writing

```
SELECT SNAME
FROM S, SP
WHERE S.SNO = SPS.SNO.
```

K-users, being able to avoid join operations, are also able to avoid this type of nesting operations. Also, the nesting required in quantified SQL queries are avoided by K-users, as seen previously.

f. Retrievals Involving Inheritance

Inheritance by IS_A-linked entities is done by the system for K-users. D-users, however, need to choose and perform joins on the appropriate relations and columns to create the inheritance. For example, consider this set of relations to represent the IS_A-linked entities `employee`, `engineer` and `manager`:

```
EMPLOYEE(NUMBER, NAME, ADDRESS)
ENGINEER(NUMBER, PROFESSION)
MANAGER(NUMBER, RANK)
```

To get the names of the engineers, K-users simply say `select engineer name` while D-users have to choose the two relations `EMPLOYEE` and `ENGINEER` and perform a join:

```
SELECT NAME
FROM EMPLOYEE, ENGINEER
WHERE EMPLOYEE.NUMBER = ENGINEER.NUMBER
```

Conclusion on Data Manipulation Operations

To conclude this section on data manipulation operation requirement, we will review the list of manipulation operations that are avoided or reduced. It is a long list:

1. Definition of foreign keys as logical pointers.
2. Deciding whether to put the relationship attributes and the entity attributes in a single relation or to put these in two relations.
3. Deciding on the relations to represent the IS_A-linked entities.
4. Doing follow-up deletion of relationship relation, or setting to 'null' the relationship attributes in a mixed entity/relationship relation, after an entity instance has been deleted.
5. Deciding which relations to change when a relationship instance is to be deleted. This involves further decision as to whether to delete a whole tuple or just to set some columns to 'null'.
6. Doing follow-up deletions of IS_A-linked entity instances when a higher entity instance is deleted.
7. Finding out which relation to delete when a top entity is deleted. There may be no real relation representing the top entity.
8. Checking that the entity keys exist in the relevant relations when inserting a relationship instance.
9. Deciding which relations and columns to update when attribute values change or when new instances have to be added.
10. Performing join operations, and deciding which are the relevant relations and columns to join.
11. Grouping occurrences by the key values.

12. Doing nesting and subqueries.
13. Manipulating multiple cases of the same relation.
14. Manipulating relations to create attribute and relationship inheritances.

In general, we can classify the first three manipulations as the designer's job and the remainder as the programmer's job. The last five can also be classified as the end-user's job. Hence KQL will benefit all three types of user.

VIII. PROLOG IMPLEMENTATION

A. CHOICE OF IMPLEMENTATION LANGUAGE

Prolog has been chosen to implement the knowledge level system. Prolog programming is very similar to logic, and the final program can therefore serve as a formal logic specification of the system.

There are also other favourable considerations. There is considerable research linking Prolog with some database system; for example, the European research project ESPRIT has ongoing research on the translation and optimization of a Prolog program to a series of data base queries expressed as a formal database language (Haass 1987), and an integration of Prolog and SQL for DB2 is described by Chang and Walker (1986). Some simple integration of Prolog and the relational database langauge SQL are also commercially available, for example, ARITY Prolog/SQL, as described in Rettig (1987), and IF/Prolog by Interface Computer GmbH has an SQL interface to Oracle.

Besides this approach of linking a Prolog program with a database system, there is also the alternative approach of designing computationally more efficient database systems within Prolog; for example, Li (1984) describes a relational DBMS with query optimization written totally in Prolog, with future direct linkage with some other relational DBMS.

It seems a real possibility that writing the knowledge level system in Prolog will allow for easy extensions which will be able to make use of the existing

research effort on making Prolog a practical database system.

B. THE SYSTEM ARCHITECTURE

The main components of the knowledge level system are the parser, the compiler, the executor and the knowledge bases. A general description of their main functions are given below.

The parser accepts the query from the user and verifies that the syntax is correct. It makes use of the knowledge bases to check for meaningful queries. Thus, queries on non-existent entities or relationships will be rejected.

The compiler accepts the verified query, performs certain processes, and produces an executable version of the query.

The executor accepts the query from the compiler, executes the query by assessing the knowledge bases, and returns the result to the user.

The domain knowledge is stored in two knowledge bases. One contains knowledge of entity types and relationship types together with their attributes. The other contains knowledge of all the entity instances and relationship instances.

C. THE KNOWLEDGE BASES

The knowledge bases are composed of Prolog predicates which are used to represent the domain knowledge. The domain knowledge stated by the user is transformed into these predicates. During retrievals, these predicates are

manipulated by the system — not by the user — to produce the answers matching the KQL queries.

1. Entity/Relationship Types Knowledge Base

This knowledge base contains knowledge of the entity types and relationship types. Each entity type and each relationship type has its own attributes. Each attribute is associated with a datatype. For entity types, the attributes are classified as key or non-key attributes. A relationship type has a set of participating entity types and the roles these entities play in the relationship. In addition, knowledge of mappings are also included.

The knowledge of entity types is represented by the following kind of Prolog predicates:

```
entity(entity-type-name,
     [[attribute-name_1, datatype_1, key/non-key],
      . . .,
      [attribute-name_n, datatype_n, key/non-key]]).
```

The knowledge of relationship types is represented by the kind of Prolog predicates shown below. The lower degree and upper degree show the minimum and maximum number of instances of this relationship that the entity instance can have. The lower degree can be either 0 or 1. The upper degree is either 1 or '*', which denotes more than one instances. A question mark can be used to denote an unknown degree, for example, [?,?].

```
rel(relationship-type-name,
     [[attribute-name_1, datatype_1],
      . . .,
      [attribute-name_m, datatype_m]],
     [[entity-name_1, role_1, [lowerdegree_1, upperdegree_1]],
      . . .,
      [entity-name_n, role_n, [lowerdegree_n, upperdegree_n]]]).
```

A relationship type can have any number of participating entity types, and each entity type can have multiple roles in the relationship. If there is only one participating entity type, then this entity type must have multiple roles in the relationship.

An example of the representation for a simple world with only suppliers and parts that are related by a relationship "supply" is given below:

```
entity(supplier, [[name,char,n],[city,char,n],[number,char,y]]).
entity(part, [[number,char,y],[name,char,n]]).
rel(supply, [[price,numeric],[quantity,numeric]],
            [[supplier, supplies, [0,*]],
             [part, supplied_by, [1,*]]]).
```

The IS_A relationship and the entities are represented as follows:

1. each entity type has the usual entity representation showing the key and the attributes special to it. The top entity type representation shows the key and the general attributes.

2. the IS_A relationship is represented by the following kind of Prolog predicates:

```
isa(superentity,
    [subentity1, subentity2, . . ., subentityn]).
```

2. Entity/Relationship Instances Knowledge Base

This knowledge base contains knowledge of the entity and relationship instances. Each instance has values for its attributes. A relationship instance also has references to the involved entity instances.

The representation used for entity instances is the following kind of Prolog predicates:

```
inst(entity-type-name,
     [[attribute-name1, value1],
      . . .,
      [attribute-namen, valuen]]).
```

The representation used for relationship instances is the following type of Prolog predicates:

```
inst(relationship-type-name,
    [[[attribute-name1, value1],
      . . .,
      [attribute-namem, valuem]],
     [[entity-type-name1, role1, [[key-attribute-name1, value1],
                                  . . .,
```

```
                                [key-attribute-name_n, value_n]]],
    . . .,
  [entity-type-name_p, role_p, [[key-attribute-name_1, value_1],
                                . . .,
                                [key-attribute-name_q, value_q]]]]]).
```

Below is a partial list of predicates to represent the instances of the simple world of suppliers and parts:

```
inst(supplier, [[name, ibm],[city,paris],[number,'S1']]).
inst(part,[[number,'P1'],[name,bolt]]).
inst(supply,[[[price,70],[quantity,200]],
             [[supplier, supplies, [[number,'S1']]],
              [part, supplied_by, [[number,'P1']]]]]).
```

D. THE PARSER, THE COMPILER AND THE EXECUTOR

The parser takes the input from the user, checking that the syntax is correct. It also fills in the missing instance declarations, since users are allowed to use the actual type names without declaring these as instance variables if there is no ambiguity. This step simply declares the type name as an instance of itself, such as "supplier is an instance of supplier."

The compiler has two distinct components: compiler-1 and compiler-2. Compiler-1 takes the query from the parser and perform certain things to it. It will

1. process inheritance conditions. This makes the inheritance explicit.
2. process combination conditions. This breaks combination conditions into

specific conditions.

3. process subset conditions. This replaces the subsets with the original set and the subset query.

The processes are described in detail in Appendix G. The output of compiler-1 is a query with only "primitive" conditions.

Compiler-2 accepts the output of compiler-1 and produces a set of prolog predicates which are data representations of the operations in the KQL query. These predicates necessarily depends on the other Prolog predicates used to manipulate the ER representations.

The executor takes the output of compiler-2 and executes it. It finds every set of instances that matches the conditions and perform the necessary operations with them, such as displaying the attribute values.

E. THE ACTUAL IMPLEMENTATION

The parser and compiler-1 have been nearly fully implemented. These will accept any of the conditions/operations specified for KQL in Chapter VI, except for entity/relationship subset retrievals. The implementation is not nearly as flexible as desired in terms of the nesting of operations; for example, "`s salary IN (2*1000, 300+3000)`" will not be accepted as a valid condition.

Compiler-2 and the executor can accept simple queries with relationship existence conditions and simple comparison of attribute values. Work on these was

suspended as we explore the more interesting alternative of running the KQL queries on relational DBMSs. This is the topic described in the next chapter.

IX. SQL IMPLEMENTATION

This chapter shows how KQL may be constructed on top of an SQL interface. This will allow KQL to be used widely, making full use of the many existing relational DBMSs.

A. SYSTEM CONFIGURATION

This system uses many of the components in the Prolog implementation. The parser, the types knowledge base and compiler-1 carry over unchanged. The knowledge base of instances is transferred to the relational system; that is, the instances are stored as tuples in the relations. The output of compiler-1 is fed to a KQL->SQL translator that produces SQL queries to be executed by the relational system.

The KQL->SQL translator translates one KQL retrieval directly into exactly one SQL retrieval. This allows very simple implementation, with no need to further process the results returned by SQL. This preserves the speed of the relational DBMSs. On the other hand, the acceptable KQL queries become limited by the SQL capabilities. The flexibility of KQL in nesting operations is also severely limited by the flexibiltiy of SQL.

The translator basically translates the instance-clause in the KQL query to the FROM statement in SQL, the select-clause in KQL to the SELECT statement in SQL, and the where-clause in KQL to the WHERE statement in SQL. In order to perform the translation, the translator needs to know the ER model and the relations representing the entities and relationships. The full algorithm for the

translation is shown in Appendix H.

B. THE ACTUAL IMPLEMENTATION

The translation algorithm shown in the appendix translates from the KQL query as entered by the user. In the actual implementation, the user's KQL query is first transformed into some internal representation by the parser and compiler-1. This internal form is then translated into the SQL query. The algorithm is essentially the same.

The translator has been nearly fully implemented. It will accept anything from compiler-1 and produce the SQL query, subject to the restrictions noted in the algorithm. It is written in Arity Prolog. Appendix I shows some examples of KQL queries and the SQL queries generated by the translator.

X. CONCLUSION AND FURTHER RESEARCH

A. CONCLUSION

It has been shown that it is possible for the user to communicate with a database system by using only domain knowledge. This has been done through the conceptual development of a knowledge level query language and additionally through the partial implementation of a computer system to handle this language.

It has also been shown that users of the knowledge level interface need to know less and perform fewer data manipulation operations than users of the logical data level interface. This has been done through a comparative analysis of the knowledge and operations required of the users at the two levels. In addition, specific logical data level difficulties are shown to have disappeared at the knowledge level.

Thus the hypotheses of this dissertation have been verified.

The main contribution of the dissertation is the proposal and demonstration of a knowledge level approach to designing the user-database interface. This approach emphasizes that the user-database exchange should be in terms of the domain knowledge rather than knowledge of the data elements in the database. In this way the user is totally cushioned from the logical and physical data.

The dissertation also makes the following contributions:

1. An analysis and definition of the meaning of completeness for a knowledge

level query language.

2. A demonstration that the entity-relationship model can be used at the knowledge level.
3. The development of a knowledge level interface for the entity-relationship model. This is followed with the design of a concrete language for the interface.
4. A clear analysis of how the knowledge level interface avoids the many difficulties of the logical data level interface.
5. Proof that relational completeness is easily achieved with a small subset of the possible KQL conditions and operations.
6. A demonstration that KQL can be implemented on top of a relational DBMS, thus making KQL practical for large systems.

There are potential practical benefits, such as higher programmer productivity and better database access that can result from the better approach. Also, the database designer can now skip the design of relations, thus saving time. In fact, new designers need not be trained in the relational model.

B. FUTURE EXTENSIONS TO THIS RESEARCH TOPIC

The following topics are suggested:

1. Experimental comparison of user's performance at the knowledge level and at the logical data level. The results can be used to confirm or modify the syntax of KQL. The experiment can also provide quantification of the improvement of KQL over SQL, for example, in terms of increase in accuracy of queries and time saved by users.

2. Further development of the SQL linkage between the knowledge level module and the relational DBMS. This will make KQL practical for large databases by using the computational power of relational systems such as relational database machines.

3. Examine the best way to apply KQL to existing relational databases, assuming that the SQL linkage is completed. An ER model has to be derived from the relations, and possibly new virtual and actual relations have to be defined while existing relations are deleted. This can have big impacts on the other existing programs that are accessing the database.

   The derivation of the ER model should be relatively easy if the relations had been designed using the ER->Relational approach, since the ER model is already there and probably only minor updates are needed.

   A simple method with no impact on existing programs is to define a set of virtual relations on top of the existing relations. These virtual relations will correspond to the entities and relationships of the ER model. However, since most relational DBMSs support very limited updates for virtual relations (Date 1987), this method will give a KQL interface mainly for retrievals only. More research is necessary, especially on the updating of virtual relations.

4. Design of an interactive graphical interface. Although there are some graphical query languages for the ER model, these are rather limited in the retrieval power.

5. Application of the knowledge level approach to conceptual models other than the entity-relationship model.

6. Application of the knowledge level system to systems development. Systems

development usually involves three stages of transformation: the real world is analyzed to get the model/requirement, the requirement is used to design a logical system, and the logical system is coded to produce the physical system. When the knowledge level system is available, systems development may only need to concentrate on the first stage of understanding the world. Coding in some sense will always be necessary, but it will be knowledge programming rather than data programming.

For a long time, it is considered that the ER model does not capture the behaviour of the real world. With a practical query language for the ER model, it may be possible to model events as well as statics. It appears that any event that causes changes in the database or uses information from the database can be modelled as one or more queries. Triggers and constraints can be added to model chain-reactions and other retrictions.

7. Development of add-on modules to handle poorly specified, incomplete or ambiguous queries. For example, instead of writing `"select supplier name where supplier supplies part, part number='P2'"`, the user may be able to write `"select supplier name where supplier supplies 'P2'"`, or `"select supplier name where part = 'P2'"`, or even `"give name of supplier of P2"`, which is getting close to the natural language interface.

8. Use ER/KQL as the foundation for a portable NL interface. As for human users, an NL processor interacting with KQL will need to know less and perform fewer data manipulation operations than if it has to interact directly with SQL.

The research on NL interfaces to databases share an important objective with the KL approach. Both try to buffer the user from the data

structures in the database. A big difference is that the NL interface emphasizes the goal of allowing flexible natural syntax, while KL interface emphasizes the power of the interface in terms of knowledge definition, updates and retrieval completeness. KQL may in some cases appear natural; however, the naturalness does not come from the syntax, it comes from using the real world concepts.

Another big difference, at least for the ER version of the knowledge level approach, is that the KL interface assumes that the world consists of certain basic concepts, while the NL interface does not formalize clearly its view of the world. The NL interface usually focuses on the nouns and the verbs linking these nouns. These nouns and verbs resemble the elements of the ER model (Chen 1983).

The NL designer does reverse engineering on the relations. He takes the relations, which are data structures designed from some real world model, and tries to re-create the real world concepts from the relations.

This reverse engineering process is a major obstacle to the domain portability of NL interfaces. Besides, it seems a waste of effort that some database designer designs the relations from a real world model, and then some other designer tries to re-create the real world model from the relations. It therefore appears that if the NL interface is based on a KL model rather than the relational model, it's portability across domains will improve dramatically. It will also be easier to extend NL interfaces to include updates and knowledge definitions, since the gap between the domain knowledge and the relations has been bridged by the KQL module.

There are important ideas in KQL that will benefit an NL interface. An

example is the inheritance of attributes and relationships.

It is interesting to observe that the SQL interface expects database designers and end-users to think the same way: the designer's way of data structures; the NL interface expects the designers to continue thinking the data structure way and the end-users to revert to thinking the real world way, with the NL expert bridging the gap; and the KL interface expects designers and users to think the same way using knowledge of the real world.

REFERENCE

Abbott, R.J., "Knowledge Abstraction", *Communications of the ACM*, Vol. 30, No. 8, pp664-671, August 1987.

Alagic, S., *Relational Database Technology*, Springer-Verlag, USA, 1986.

Albano, A., L. Cardelli and R. Orsini, "Galileo: A Strongly Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, June 1985.

Albrecht, A.J. and J Gaffney Jr., "Software Functions, Source Lines of Code and Development Effort Prediction: A Software Science Validation", *IEEE Transactions on Software Engineering*, November 1983, SE-9(6), pp639-648.

Astrahan, M.M. et. al., "System R: Relational Approach to Database Management", pp97-137, *ACM Trans. Database Systems*, Vol. 1, No. 2, June 1976.

Atzeni, P., C. Batini, M. Lenzerini and F. Villanelli, "INCOD: A System for Conceptual Design of Data and Transactions in the Entity-Relationship Model", pp375-411, *Entity-Relationship Approach to Information Modeling and Analysis*, ed. P.P. Chen, North-Holland, 1981.

Atzeni, P. and E. Carboni, "INCOD (A System for Interactive Conceptual Design) Revisited After the Implementation of a Prototype", pp449-464, *Entity-Relationship Approach to Software Engineering*, ed. C.G.Davis, S.Jajodia, P.A.B.Ng and R.T.Yah, North-Holland, 1983

Atzeni, P. and P.P. Chen, "Completeness of Query languages for the Entity-Relationship Model" pp109-122, *Entity-Relationship Approach to Information Modeling and Analysis*, ed. P.P. Chen, North-Holland, 1981.

Atzeni, P. and D.S. Parker, "Assumptions in Relational Database Theory", *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

Azar, N. and E. Pichat, "Translation of an Extended Entity-Relationship Model into the Universal Relations with Inclusions Formalism", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Bachman, C.W., "The Role Concept in Data Models", p464-476, *Proceedings, Third International Conference on Very Large Databases*, 1977. A>

Berghel, H.L., "Simplified Integration of Prolog with RDBMS", *DATA BASE*, Vol. 16, No. 3, Spring 1985.

Borgida, A., "Features of Languages for the Development of Information Systems at the Conceptual Level", pp63-72, *IEEE Software*, Vol. 2, No. 1, January 1985.

Borkin, S.A., "The Semantic Relation Data Model: Foundation for a User Interface", p47-64, *Proceedings of the International Conference on Data Bases*, July 1980.

Brachman, R.J. and H.J. Levesque, "What Makes a Knowledge Base Knowledgeable? A View of Databases from the Knowledge Level", *Expert Database Systems*, ed. Larry Kerschberg, The Benjamin/Cummings Publishing Company, Inc., USA, 1986.

Brady, L.I. and C.N.G. Dampney, "The Semantics of Relational Database Functions", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Braind, H., H. Habrias, J. Hue and Y. Simon, "Expert System for Translating an ER Diagram into Databases", p199-206, *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Brodie, M.L., "On Modelling Behavior Semantics of Databases", *Proceedings of the 7th International Conference on Very Large Databases*, Cannes, France, September 1981.

Brodie, M.L., "On the Development of Data Models", in *On Conceptual Modelling*, ed. M.L. Brodie, J. Mylopoulos and J.W. Schmidt, Springer-Verlag, 1984.

Buneman, P., "Inheritance and Persistence in Database Programming Languages", *Proceedings of the ACM SIGMOD '86, International Conference on Management of Data*, 1986.

Burgess, C. G., "A Database Interface Aid", *IEEE Computer Society Workshop on Visual Languages* 1984, pp72-76.

Campbell, D.M., D.W. Embley and B. Czejdo, "A Relationally Complete Query Language for an Entity-Relationship Model", *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Capaccioli, M. and M.E. Occhinto, "A Workbench for Conceptual Design in Galileo", *Computer-aided Database Design*, ed. A. Albano, V. de Antonellis and A. di Leva, Elsevier Science Publishers, The Netherlands, 1985.

Cardenas, A.F. and G.R. Wang, "Translation of SQL/DS Data Access/Update into Entity-Relationship Data Access/Update", pp256-267, *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Chamberlain, D.D., "A Summary of User Experience with the SQL Data Sublanguage", *Proceedings of the International Conference on Data Bases*, July 1980, p181-203.

Chang, C.L. and A. Walker, "PROSQL: A Prolog Programming Interface with SQL/DS", *Expert Database Systems*, ed. Larry Kerschberg, The Benjamin/Cummings Publishing Company, Inc., USA, 1986.

Chen, P.P., "The Entity-Relationship Model: Towards a Unified View of Data", pp9-36, *ACM Trans. Database Systems*, Vol. 1, No. 1, March 1976.

Chen, P.P., "A Preliminary Framework for Entity-Relationship Models", *ER Approach to Information modeling and analysis*, ed. P.P. chen, North-Holland, 1981.

Chen, P.P., "English Sentence Structure and Entity Relationship Diagrams", *Information Sciences*, Vol. 29, pp127-149, Elsevier Science Pub. Co., New York, 1983.

Chen, P.P., "An Algebra for a Directional Binary Entity-Relationship Model", *First International Conference on Data Engineering*, pp37-41, 1984.

Clancey, W.J., "Knowledge Acquisition for Classification Expert Systems", *ACM '84 Annual Conference, The Fifth Generation Challenge*, p11-14.

Codd, E.F., "How About Recently? (English Dialog with Relational Data Bases Using Rendezvous", in *Databases: Improving Usability and Representativeness*, ed. B.Shneiderman, New York, Academic Press, 1978.

Codd, E.F., "Extending the Database Relational Model to Capture more Meaning", *ACM Transactions on Database Systems*, December 1979.

Copeland, G. and D. Maier, "Making Smalltalk a Database System", *ACM SIGMOD Conference* 1984.

Date, C.J., *An Introduction to Database Systems*, Addison-Wesley, 1982.

Date, C.J., *A Guide to INGRES*, USA, 1987a.

Date, C.J., *A Guide to the SQL Standard*, Addison-Wesley, USA, 1987b.

Dayal, U. and P.A. Bernstein, "On the Updatability of Relational Views", *Proc. 4th Very Large Databases Conference*, Berlin, 368-377, 1978.

Dogac, A., F. Eyupoglu and E. Arkun, "VERS - A Vector Based Entity Relationship Database Management System", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Dumpala, S.R. and S.K. Arora, "Schema Translation Using the Entity-Relationship Approach", *Entity-Relationship Approach to Information Modeling and Analysis*, ed. P.P. Chen, North-Holland, 1981.

Ehrenreich, S.L., "Query Languages: Design Recommendations Derived from the Human Factors Literature", *Human Factors*, Vol. 23, 1981, p709-725.

Elmasri, R.A. and J.A. Larson, "A Graphical Query Facility for ER Databases", pp236-245, *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Elmasri, R. and G. Wiederhold, "GORDAS: A Formal High-level Query Language for the Entity-Relationship Model", pp49-72, *Entity-Relationship Approach to Information Modeling and Analysis*, ed. P.P. Chen, North-Holland, 1981.

Finkelstein, S., M. Schkolnick and P. Tiberio, "Physical Database Design for Relational Databases", p91-128, *ACM Transactions on Database Systems*, Vol. 13, No. 1, March 1988.

Flory, A. and S.T. March, "SCRABBLE: A Local Database Management System", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Fogg, Dennis, "Lessons from a 'Living in a Database' Graphical Query Interface", p100-106, *SIGMOD 84, Proceedings of Annual Meeting, SIGMOD Record Vol. 14, No. 2*, ACM, USA, 1984.

Furtado, A.L. and E.J. Neuhold, *Formal Techniques for Data Base Design*, Springer-Verlag, Germany, 1986.

Gaines, B., "The Technology of Interaction - Dialogue Programming Rules", *International Journal of Man-Machine Studies*, Vol. 14, 1981, 133-180.

Ganski, R.A. and H.K.T. Wong, "Optimization of Nested SQL Queries Revisited", p23-33, *Proceedings of ACM SIGMOD 1987 Annual Conference.*

Gilberg, R.F., "A Schema Methodology for Large Entity-Relationship Diagrams", pp320-325, *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Goldstein, R.C., *Database Technology and Management*, J. Wiley, USA, 1985.

Gray, Peter, *Logic, Algebra and Databases*, Ellis Horwood Ltd., UK, 1984.

Greenbalt, D. and Waxman, "A Study of Three Database Query Languages", in *Databases: Improving Usability and Representativeness*, ed. B.Shneiderman, New York, Academic Press, 1978.

Haass, U.L., "The Integration of Data and Knowledge Base Systems, Activities in ESPRIT", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Harris, L.R., "ROBOT: A High Performance Natural Language Interface for Data Base Query", *Natural Language Based Computer Systems*, ed. Leonard Bolc, Carl Hanser Verlag, Germany, 1980.

Hitchcock, P. et al. "The Use of databases for Software Engineering", pp. 55-70, *Proceedings of the Fifth British National Conference on Databases*, ed. E.A. Oxborrow, Cambridge University Press, UK, 1986.

Hwang, H.Y. and U. Dayal, "Using the Entity-Relationship Model for Implementing Multi-model Database Systems", pp235-256, *Entity-Relationship Approach to Information Modeling and Analysis*, ed. P.P. Chen, North-Holland, 1981.

Janas, J.M., "The Semantics-Based Natural Language Interface to Relational Databases", in *Cooperative Interfaces to Information Systems*,ed. Bolc L. and M. Jarke, Springer-Verlag, Germany, 1986.

Jardine, D.A., "Semantic Agreement and the Communication of Knowledge", *Data Semantics (DS-1)*, ed. T.B.Steel Jr. and R.Meersman, Elsevier Science Publishers, The Netherlands, 1985.

Jarke, M., J. Krause and Y. Vassiliou, "Studies in the Evaluation of a Domain-Independent Natural Language Query System", in *Cooperative Interfaces to Information Systems*,ed. Bolc L. and M. Jarke, Springer-Verlag, Germany, 1986.

Jarke, M. and Y. Vassiliou, "A Framework for Choosing a Database Query Language", *Computing Surveys*, Vol. 17, No. 3, September 1985.

Junet, Marc, "Design and Implementation of an Extended Entity-Relationship Data Base Management System (ECRINS/86)", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Kaplan, S.J., "Designing a Portable Natural Language Database Query System", *ACM Transactions on Database Systems*, Vol. 9, No. 1, March 1984.

Kim, W., D.S. Reiner and D.S. Batory, (Ed), *Query Processing in Database Systems*, Springer-Verlag, Germany, 1985.

Klug, A., "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *Journal of the ACM*, 29, 3(July 1982), 699-717.

Korth, H.F. and A. Silberschatz, *Database System Concepts*, McGraw-Hill Book Company, USA, 1986.

Kluzniak, F. and S. Szpakowicz, *Prolog for Programmers*, Academic Press, USA, 1985.

Levesque, H.J., "Foundations of a Functional Approach to Knowledge Representation", *Artificial Intelligence*, 23(1984), 155-212.

Li, Deyi. *A PROLOG Database System*, Research Studies Press Ltd, England, U.K., 1984.

Ling, T.W., "A Normal Form for Entity-Relationship Diagrams", *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Lochovsky, F.H. and D.C. Tsichritzis, "User Performance Considerations in DBMS Selection", *Proceedings, ACM SIGMOD*, 1977, p128-134.

Lucas, R., "An Expert System to Detect Burglars Using a Logic Language and a Relational Database", *Proceedings of the Fifth British National Conference on Databases*, ed. E.A. Oxborrow, Cambridge University Press, UK, 1986.

Maier, D., *The Theory of Relational Databases*, Computer Science Press, USA, 1983.

Maier, D. and J.D. Ullman, Vardi, M.Y., "On the Foundations of the Universal Relation Model", *ACM Transactions on Database Systems*, June 1984.

Markowitz, V.M. and Y. Raz, "ERROL : An Entity-Relationship, Role-oriented, Query Language", pp329-345, *Entity-Relationship Approach to Software Engineering*, ed. C.G.Davis, S.Jajodia, P.A.B.Ng and R.T.Yah, North-Holland, 1983

Markowitz, V.M. and Y. Raz, "A Modified Relational Algebra and its Use in an Entity-Relationship Environment", pp315-328, *Entity-Relationship Approach to Software Engineering*, ed. C.G.Davis, S.Jajodia, P.A.B.Ng and R.T.Yah, North-Holland, 1983

Martin, J., *Managing the Date-Base Environment*, Prentice-Hall, USA, 1983.

McCord, M., "Natural Language Processing in Prolog", *Knowledge Systems and Prolog*, ed. A Walker, Addison-Wesley, USA, 1987.

McGee, W.C., "On User Criteria for Data Model Evaluation", ACM Transactions on Database Systems December 1976.

Meersman, R., "Knowledge and Data: A Survey in the Margin of the IFIP DS-2 conference", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Merrett, T.H., "The Extended Relational Algebra, a Basis for Query Languages", in *Databases: Improving Usability and Representativeness*, ed. B.Shneiderman, New York, Academic Press, 1978.

Motro, A., "Constructing Queries from Tokens", Proceedings of ACM SIGMOD, 1986, SIGMOD Record Vol 15, No. 2, June 1986.

Nakano, R., "Integrity Checking in a Logic Oriented ER Model", pp551-564, *Entity-Relationship Approach to Software Engineering*, ed. C.G.Davis, S.Jajodia, P.A.B.Ng and R.T.Yah, North-Holland, 1983

Newell, A., "The Knowledge Level", *Artificial Intelligence*, 18(1982)87-127.

Nierstrasz, O.M., "What is the 'Object' in Object-Oriented Programming", in *Objects and Things* ed. D.C. Tsichritzis, University of Geneve, pp1-13, March 1987.

Nilsson, M., "A Logical Model of Knowledge", *International Joint Conference on Artificial Intelligence*, 1983.

Ogden, W.C. and S.R. Brooks, "Query Languages for the Casual User: Exploring the Middle Ground Between Formal and Natural Languages", CHI'83 Proceedings.

Olive, Antoni, "Analysis of Conceptual and Logical Models in Information Systems Design Methodologies", *Information Systems Design Methodologies*, ed. T.W. Olle, H.G. Sol and C.J. Tully, Elsevier Science Publishers, IFIP 1983.

Palwa, A. and A.K. Arora, "Automatic Database Navigation: Towards a High Level User Interface", *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Parent, C. and S. Spaccapietra, "An Entity-Relationship Algebra", *First International Conference on Data Engineering*, pp500-509, 1984.

Parent, C. and S. Spaccapietra, "Enhancing the Operational Semantics of the Entity-Relationship Model", *Data Semantics (DS-1)*, ed. T.B.Steel Jr. and R.Meersman, Elsevier Science Publishers, The Netherlands, 1985.

Poonen, G., "CLEAR: A Conceptual Language for Entities and Relationships", pp194-215, *Centralized and Distributed Data Base Systems*, ed. W.W. Chu and P.P. Chen, IEEE Computer Society, 1979.

Reiner, D. et al., "A Database Designer's Workbench", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Reisner, P., "Human Factors Studies of Database Query Languages, a Survey and Assessment", *Computing Surveys*, 13.1, p13-31, March 1981.

Rettig, M., "Prolog and SQL: A Happy Union", *AI Expert*, Vol. 2, No. 7, July 1987, p19-24.

Roesner, W., "DESPATH: An Entity-Relationship Manipulation Language", *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Rosenberg, R.S., "Approaching Discourse Computationally: A Review", in *Representation and Processing of Natural Language*, Leonard Bolc Ed. Carl Hanser Verlag, Europe, 1980.

Schuldt, G., "User Experience with the ER Approach, Report on the Panel Session", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Sellis, T.K., "Multiple-Query Optimization", p23-52, *ACM Transactions on Database Systems*, Vol. 13, No. 1, March 1988.

Shneiderman, B., "Improving the Human Factors Aspect of Database Interactions", *ACM Transactions on Database Systems*, Vol. 3, No. 4, Dec. 1978, 417-439.

Shoshani, A., *CABLE: A Language Based on the Entity-Relationship Model*, Lawrence Berkeley Lab., Berkeley, CA, 1978.

Simon, H.A., *Administrative Behavior*, MacMillian, New York, 1957.

Sloman, A., "Why We need Many Knowledge Representation Formalisms", in *Research and Development in Expert Systems*, ed. M.A. Bramer, Cambridge University Press, Great Britain, 1985.

Smith, J.M. and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", *ACM Transactions on Database Systems*, Vol. 2, No. 2, June 1977.

Staley, S.M. and D.C. Anderson, "Executable E-R specifications for Database Schema Design", *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Steel Jr., T.B. and R. Meersman, *Data Semantics (DS-1)*, Proceedings of the IFIP WG 2.6 Working Conference on Data Semantics (DS-1), Belgium, January 1985.

Stonebraker, M., "Triggers and Inference in Database Systems", in *On Knowledge Base Management Systems*, ed. M.L. Brodie and J. Mylopoulos, Springer-Verlag, USA, 1986.

Storey, V.C., *An Expert View Creation System for Database Design*, Ph.D. Dissertation, Faculty of Commerce and Business Administration, University of British Columbia, 1986.

Subieta, K. and M. Missala, "Semantics of Query languages for the Entity Relationship Model", *Entity-Relationship Approach*, ed. S. Spaccapietra, Elsevier Sciences Publishers, 1987.

Templeton, M. and J. Burger, "Considerations for the Development of Natural Language Interfaces to Database Management Systems", in *Cooperative Interfaces to Information Systems*,ed. Bolc L. and M. Jarke, Springer-Verlag, Germany, 1986.

Teorey, T.J., D. Yang and J.P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", *Computing Surveys*, Vol. 18, No. 2, June 1986.

Tsichritzis, D.C. and F.H. Lochovsky, *Data Models*, Prentice-Hall, USA, 1982.

Tsur, S. and C. Zaniolo, "An Implementation of GEM - Supporting a Semantic Data Model on a Relational Back-end", p286-295, *SIGMOD 84, Proceedings of Annual Meeting, SIGMOD Record Vol. 14, No. 2*, ACM, USA, 1984.

Velez, F., "LAMBDA: An Entity-Relationship Based Query Language for the Retrieval of Structured Documents", *The 4th International Conference on Entity-Relationship Approach* , IEEE Computer Society Press, 1985.

Vossen, G. and V. Brosda, "A High-level User Interface for Update and Retrieval in Relational Databases - Language Aspects", *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data.*

Walker, A. et al., *Knowledge Systems and Prolog*, Addison-Wesley, USA, 1987.

Wallace, M., *Communicating with Databases in Natural Language*, Ellis Horwood Limited, England, 1984.

Welty C. and D.W. Stemple, "Human Factors Comparison of a Procedural and a Nonprocedural Query Language", *ACM Transactions on Database Systems*, Vol. 6, No. 4, December 1981, 626-649.

Welty, C., "Correcting User Errors in SQL", *International Journal of Man-Machine Studies*, Vol. 22, 1985, p463-477.

White, S.J., *A Portable Natural Language Database Query System*, Master Thesis, Department of Computer Science, University of British Columbia, 1985.

Wong, H.K.T. and I. Kuo, "GUIDE: Graphical User Interface for Database Exploration", *Proceedings of the 8th International Conference on Very Large Data Bases*, Mexico City, Mexico, September 1982.

Xie, Linchi, *Semantic Database Model SDBM, Its Design and Strategy of Implementation*, Master Thesis, Faculty of Commerce and Business Administration, University of British Columbia, 1986.

Yang, Chao-Chih, *Relational Databases*, Prentice-Hall, 1986.

Yeo, C., J. Thorpe and J. Longstuff, "Knowledge Base Enhancements to Relational databases", *Proceedings of the Fifth British National Conference on Databases*, ed. E.A. Oxborrow, Cambridge University Press, UK, 1986.

Zhang, Z.Q. and A.O. Mendelzon, "A Graphical Query Language for Entity-Relationship Databases", pp441-448, *Entity-Relationship Approach to Software Engineering*, ed. C.G.Davis, S.Jajodia, P.A.B.Ng and R.T.Yah, North-Holland, 1983

Zloof, M.M., "Design Aspects of the Query-By-Example Data Base Manipulation Language", in *Databases: Improving Usability and Representativeness*, ed. B.Shneiderman, New York, Academic Press, 1978.

APPENDIX A - RELATIONAL COMPLETENESS OF KQL

While it is not meaningful to define completeness of the knowledge level based on relational completeness, it is nevertheless worthwhile to ask: if the ER model is represented by relations, can KQL produce results that can be produced by relational operations on the relations? The practical version of this question is: can the user with direct access (through a relational algebra) to the set of relations retrieve more results than the user who is using KQL on the ER model? The answers to these questions are "yes" and "no" respectively. It will be shown that KQL is relationally complete.

In order to prove the relational completeness of KQL, we first have to describe the relational representation of an ER model. An entity E with key attribute k and other attributes a_1, a_2, . . ., a_n is represented by the relation:

$$E(k, a_1, a_2, \ldots, a_n).$$

The entity name is the same as the name of its relation. Whether something is an entity or a relation depends on the context. This also applies to relationships and their relations.

For example, the entity SUPPLIER with attributes NUMBER, NAME and CITY will be represented by the relation:

SUPPLIER(NUMBER, NAME, CITY),

A relationship R with attributes a_1, a_2, . . ., a_n, and involving entities E_1, E_2, . . ., E_m with roles $role_1$, $role_2$, . . ., $role_m$ respectively and keys k_1, k_2, . . ., k_n respectively, is represented by the relation:

$$R(E_1\_role_1\_k_1,\ E_2\_role_2\_k_2,\ .\ .\ .,\ E_m\_role_m\_k_m,$$
$$a_1,\ a_2,\ .\ .\ .,\ a_n).$$

For example, the relationship SUPPLY with attributes PRICE and QUANTITY involving the entity SUPPLIER with role SUPPLIES and the entity PART with role SUPPLIED_BY will be represented by the relation:

```
SUPPLY(SUPPLIER_SUPPLIES_NUMBER,
       PART_SUPPLIED_BY_NUMBER,
       PRICE, QUANTITY).
```

These relations representing the entities and the relationships are called the base relations.

The proof of relational completeness is done as follows:

1. Step 1 shows that the base relations can be retrieved by KQL.
2. Step 2 shows that KQL can produce relations derived by a single relational operation on the base relations.
3. Step 3 and 4 show that if KQL can produce relations R_1 and R_2, then KQL can produce relations that are derived by a relational operation on R_1 and/or R_2.
4. Step 5 shows that by induction, KQL can produce any relation that can be derived by an arbitrary number of relational operations on the base relations.

a. Step 1

1.1. Consider an entity E with key attribute k and attributes a_1, a_2, . . ., a_n.

KQL(E)† =

e is an instance of E

SELECT e k, e a_1, e a_2, . . ., e a_n

This query will be repeated many times in the proof. To make it more concise, we will write it as

e is an instance of E

SELECT list_for_E

1.2 Consider a relationship R involving entities E_1 to E_n with roles $role_1$ to $role_n$ and where the key of E_i is k_i with i ranging from 1 to n. The attributes of the relationship are a_1, a_2, . . ., a_m.

KQL(R) =

e_1 is an instance of E_1,

e_2 is an instance of E_2,

. . .,

e_n is an instance of E_n,

r is an instance of R

SELECT e_1 k_1, e_2 k_2, . . ., e_n k_n,

r a_1, r a_2, . . ., r a_m

†KQL(X) denotes the KQL query equivalent in result to the relational expression X.

WHERE e_1 ($role_1$) r e_2,

e_2 ($role_2$) r e_3,

. . .,

e_n ($role_n$) r e_1.

The roles are needed only if the relationship has the same entity type in multiple roles.

This query will be repeated many times in the proof. To make it more concise, we will write this query as

instance_clause_for_R

SELECT list_for_R

WHERE conditions_for_R

b. Step 2

In this step, we need to show the KQL queries to derive the relations that will result from a projection of a base relation, a selection on a single base relation, the cartesian product of two base relations, the union of two base relations and the difference of two base relations. We show these in steps 2.1 to 2.5.

2.1 Projection (denoted by [])

2.1.1 KQL(E[x]) =

e is an instance of E

SELECT e x

2.1.2 Making the same assumptions for the relationship R as made in step 1.2,

KQL(R[x]) =

instance_clause_for_R

SELECT EE(x)

WHERE conditions_for_R

EE(x) denotes the equivalent expression of x. If x is an attribute of the relationship R then EE(x) is "r x" where "r" has been declared as an instance of R. If x refers to the key attribute of entity E (that is, x is of the form E_role_k) then EE(x) is "e k" where e has been declared as an instance of E.

2.2 Selection (denoted by σ_{sp} where sp is the selection predicate)

2.2.1 KQL(σ_{sp}E) =

e is an instance of E

SELECT list_for_E

WHERE sp

2.2.2 KQL(σ_{sp}R) =

instance_clause_for_R

SELECT list_for_R

WHERE conditions_for_R,

EE(sp).

EE(sp) refers to the equivalent expression of sp. Let sp be "$y_1 \ \theta \ y_2$". If y is an attribute of the relationship then y is replaced by "r y" in the equivalent expression. If y refers to the key attribute of an entity, (that is, y is of the form E_role_k) then y is replaced by "e k" where e has been declared as an instance of E. If

y is a constant, it is not replaced.

2.3 Cartesian Product (denoted by **X**)

2.3.1 KQL(E_i **X** E_j) =

e_i is an instance of E_i,

e_j is an instance of E_j

SELECT list_for_E_i, list_for_E_j

2.3.2 KQL(R_i **X** R_j) =

instance_clause_for_R_i,

instance_clause_for_R_j

SELECT list_for_R_i, list_for_R_j

WHERE conditions_for_R_i, conditions_for_R_j

The instances declared for R_i must not have the same name as the instances declared for R_j. For example, if "e_1" is already declared as an instance in the instance_clause of R_i, then "e_1" should not be used for the declarations of R_j.

2.3.3 KQL(E **X** R) =

e is an instance of E,

instance_clause_for_R

SELECT list_for_E, list_for_R

WHERE conditions_for_R

2.3.4 KQL(R **X** E)

This is the same as for 2.3.3, with the two lists in the SELECT

clause reversed.

2.4 Union (for compatible relations/entities/relationships)

2.4.1 KQL(E_i U E_j) =

e_i is an instance of E_i

SELECT list_for_E_i

UNION

e_j is an instance of E_j

SELECT list_for_E_j

2.4.2 KQL(R_i U R_j) =

instance_clause_for_R_i

SELECT list_for_R_i

WHERE conditions_for_R_i

UNION

instance_clause_for_R_j

SELECT list_for_R_j

WHERE conditions_for_R_j

2.4.3 KQL(E U R) =

e is an instance of E

SELECT list_for_E

UNION

instance_clause_for_R

SELECT list_for_R

WHERE conditions_for_R

2.4.4 KQL(R U E)

This is the same as 2.4.3 above.

2.5 Set Difference (denoted by '−')

2.5.1 KQL(E_i − E_j) =

e_i is an instance of E_i

SELECT list_for_E_i

WHERE not exists

(e_j is an instance of E_j

SELECT list_for_E_j

WHERE list_for_E_i = list_for_E_j)

The condition "list_for_E_i=list_for_E_j" means that each member of the first list equals the corresponding member in the second list.

2.5.2 KQL(R_i − R_j) =

instance_clause_for_R_i

SELECT list_for_R_i

WHERE conditions_for_R_i,

not exists

(instance_clause_for_R_j

SELECT list_for_R_j

WHERE conditions_for_R_j,

list_for_R_i = list_for_R_j)

Again, the equality of two lists means the equality of corresponding members.

2.5.3 KQL(E − R) =

e is an instance of E

SELECT list_for_E

WHERE not exists

(instance_clause_for_R

SELECT list_for_R

WHERE conditions_for_R,

list_for_E) = list_for_R)

2.5.4 KQL(R − E) =

instance_clause_for_R

SELECT list_for_R

WHERE conditions_for_R,

not exists

(e is an instance of E

SELECT list_for_E

WHERE list_for_R = list_for_E)

c. Step 3

In this step, we assume that two arbitrary relations R_A and R_B are derivable by KQL queries. We then show that any relation resulting from a single relational operation on R_A and/or R_B is also derivable by a KQL query.

We assume that the KQL query for R_A is of the following form:

instance_clause(A)

SELECT list(A)

WHERE condition(A).

The KQL query for R_B is assumed to be of the same form:

instance_clause(B)

SELECT list(B)

WHERE condition(B).

3.1 Projection: KQL(R_A[x]) =

instance_clause(A)

SELECT EE(x)

WHERE condition(A).

(EE(x) is as defined in 2.1; it is a sublist of list(A))

The KQL query for a projection of R_B is similar.

3.2 Selection: KQL($\sigma_{sp}R_A$) =

instance_clause(A)

SELECT list(A)

WHERE condition(A), EE(sp).

(EE(sp) is as defined in 2.2)

The KQL query for a selection on R_B is similar.

3.3 Cartesian Product: KQL(R_A **X** R_B) =

instance_clause(A),

instance_clause(B)

SELECT list(A), list(B)

WHERE condition(A), condition(B).

3.4 Union: KQL(R_A U R_B) =

instance_clause(A)

SELECT list(A)

WHERE condition(A)

UNION

instance_clause(B)

SELECT list(B)

WHERE condition(B)

3.5 Set Difference: KQL(R_A − R_B) =

instance_clause(A)

SELECT list(A)

WHERE condition(A),

not exist

(instance_clause(B)

SELECT list(B)

WHERE condition(B),

list(A) = list(B) .)

Again, the equality of two list means the equality of all the corresponding members in the two list.

d. Step 4

In step 3, it is assumed that a relation R can be produced by a KQL query Q of the form

instance_clause

SELECT list

WHERE condition.

However, as seen in step 2 and 3, some relations can be produced only by a query of the form:

Q_1 U Q_2 U . . . U Q_n

where each Q_i, $1 \leq i \leq n$, has the form assumed for Q.

It will be shown that this violation of the assumption in step 3 does not alter the result in step 3. Assume that R_A and R_B are produced by the following KQL queries:

Q_{A1} U Q_{A2} U . . . U Q_{An},

and

Q_{B1} U Q_{B2} U . . . U Q_{Bm}

Each query Q_{Ai} or Q_{Bj} will produce a relation R_{Ai} or R_{Bj}. In other words, R_A equals (R_{A1} U R_{A2} U . . . U R_{An}), and similarly for R_B.

4.1 Projection: $R_A[x]$

$R_A[x] = R_{A1}[x] \cup R_{A2}[x] \cup \ldots \cup R_{An}[x]$

Each of the sub-relations can be expressed by a KQL query, as shown in step 3.1, and by 3.4 the unions of the sub-relations can be done by a KQL expression.

4.2 Selection: σR_A

$\sigma R_A = \sigma R_{A1} \cup \sigma R_{A2} \cup \ldots \cup \sigma R_{An}$

An argument similar to 4.1 applies here, showing that σR_A can be done by a KQL query.

4.3 Cartesian Product: $R_A \times R_B$

This is equal to

$R_{A1} \times R_{B1} \cup R_{A1} \times R_{B2} \cup \ldots \cup R_{A1} \times R_{Bm}$

$\cup$

$R_{A2} \times R_{B1} \cup R_{A2} \times R_{B2} \cup \ldots \cup R_{A2} \times R_{Bm}$

$\cup$

. . .

$\cup$

$R_{An} \times R_{B1} \cup R_{An} \times R_{B2} \cup \ldots \cup R_{Am} \times R_{Bm}$

An argument similar to 4.1 applies here, showing that $R_A \times R_B$ can be produced by a KQL query.

4.4 Union $R_A \cup R_B$

This is equal to

$R_{A1} \cup R_{A2} \cup \ldots \cup R_{An} \cup R_{B1} \cup R_{B2} \cup \ldots \cup R_{Bm}$.

An argument similar to 4.1 applies here, showing that the long chain of unions can be produced by a KQL query.

4.5 Set Difference: $R_A - R_B$

This is equal to

$$(((R_{A1} - R_{B1}) - R_{B2}) - \ldots - R_{Bm})$$
$$\mathbf{U}\ (((R_{A2} - R_{B1}) - R_{B2}) - \ldots - R_{Bm})$$
$$\mathbf{U}\ \ldots$$
$$\mathbf{U}\ (((R_{An} - R_{B1}) - R_{B2}) - \ldots - R_{Bm})$$

An argument similar to 4.1 applies here, showing that the long chain of unions and set differences can be produced by a KQL query.

e. Step 5

It has been shown that it is possible to use a single KQL query to produce any of the base relations, and that a single KQL query can produce any relation that can be derived by a single relational operation on the base relations. It has also been shown that if two relations R_A and R_B can be produced by queries Q_A and Q_B, then any relation resulting from a single relational operation on R_A and/or R_B can be produced by a KQL query.

It therefore follows by induction that the KQL query can derive relations involving any number of relational operations on the base relations. Hence, KQL is relationally complete.

Relational completeness is achieved with only a small subset of the possible KQL operations. Specifically, only these operations are needed: instance-clause, select-clause, relationship existence condition, equality conditions (<,=,>), union and exists.

APPENDIX B - CREATION AND UPDATE OF ENTITY AND RELATIONSHIP TYPES

This appendix describes how the user can provide knowledge about his world to the system (Of course, the system will only accept knowledge about entities and relationships). Before the user can talk of instances, he first has to tell the system about the entity types and the relationship types.

To describe an entity type, the user has to tell the system the name of the entity, the key attributes, the non-key attributes, and the datatype of the attributes. The syntax is:

<new-type> ::=

NEW ENTITY <entity-type-name>,
KEY ATTRIBUTE <attribute-name>
WITH DATATYPE <datatype-name>
{, KEY ATTRIBUTE <attribute-name>
WITH DATATYPE <datatype-name>}
{, ATTRIBUTE <attribute-name>
WITH DATATYPE <datatype-name>}.

The datatype-name is terminal. The choices depend on the particular implementation. Also depending on the implementation, the user may have to specify the "length" of the datatype; for example, CHAR(10) may mean a datatype with up to ten characters.

For example, the following describes a new entity supplier:

```
new entity supplier,
key attribute number with datatype character,
attribute name with datatype character,
attribute city with datatype character.
```

To describe a relationship type, the user has to include the name of the relationship, the attributes and their datatypes, and the entity types involved, with their roles and mapping degrees. The syntax is:

<new-type> ::=

NEW RELATIONSHIP <relationship-type-name>
{, ATTRIBUTE <attribute-name>
WITH DATATYPE <datatype-name>},
ENTITY <entity-type-name> WITH ROLE <role-name>
AND MAPPING <mapping-degrees>,
ENTITY <entity-type-name> WITH ROLE <role-name>
AND MAPPING <mapping-degrees>
{, ENTITY <entity-type-name> WITH ROLE <role-name>
AND MAPPING <mapping-degrees>}.

<mapping-degrees> ::= (<degree>, <degree>)

<degree>° ::= integer | * | ?

The mapping degrees consists of two 'numbers' to represent the lower and upper degrees. The lower degree denotes the minimum number of instances of the specified relationship type that any instance of the entity type must have. The

upper degree denotes the maximum number of the relationship instances that the entity instance can have.

An example to describe a relationship between supplier and part is as follows:

```
new relationship supply,
attribute price with datatype numerics,
attribute quantity with datatype numerics,
entity supplier with role supplies and mapping (0,*),
entity part with role supplied_by and mapping (0,*).
```

The user can create IS_A relationships by using these steps:

Step 1

Create the superentity as for normal entities, with key and non-key attributes.

Step 2

Create subentities listing the special attributes that belong there as well as the key attributes.

Step 3

Create an IS_A relationship between the superentity and the subentities.

The syntax of the command needed is:

<new-isa> ::=

NEW IS_A SUPERENTITY = <entity-type-name>,
SUBENTITY = <entity-type-name>
{, <entity-type-name> }.

An example of the creation of IS_A-linked entities is:

```
1.  new entity employee,
        key attribute number with datatype character,
        attribute name with datatype character.
2a. new entity engineer,
        key attribute number with datatype character,
        attribute profession with datatype character.
2b. new entity manager,
        key attribute number with datatype character,
        attribute rank with datatype character.
3.  new IS_A superentity = employee,
        subentity= engineer, manager.
```

Now we describe updates to the types. An entity or a relationship type may be deleted; a type's attribute may be deleted; or more attributes may be added to a type.

The syntax for <type-update> is one of the following:

1. DELETE ENTITY <entity-type-name>.

   This will result in the deletion of this entity type and all relationship types involving this entity type. Naturally, all the instances of the deleted types will also be deleted.

2. DELETE RELATIONSHIP <relationship-type-name>.

   This will result in deletion of this relationship type and all its instances.

3. DELETE ATTRIBUTE <attribute-name>
   OF ENTITY <entity-type-name>.
   This entity type will lose the specified attribute, and all its instances will lose the values of this attribute.
4. DELETE ATTRIBUTE <attribute-name>
   OF RELATIONSHIP <relationship-type-name>.
   This relationship type will lose the specified attribute, and all its instances will lose the values of this attribute.
5. ADD ATTRIBUTE <attribute-name>
   WITH DATATYPE <datatype-name>
   TO ENTITY <entity-type-name>.
   This entity type will gain the attribute. All its instances will have null values for this new attribute.
6. ADD ATTRIBUTE <attribute-name>
   WITH DATATYPE <datatype-name>
   TO RELATIONSHIP <relationship-type-name>.
   This relationship type will gain the attribute. All its instances will have null values for this new attribute.

Some potential ambiguities may arise during the descriptions. The rules that the system should follow are:

1. names for entity/relationship types are unique. No two types can have the same name.
2. names for attributes are unique within a type.
3. role names are unique between any two entities.

APPENDIX C - CREATION AND UPDATE OF INSTANCES

This section describes how instances can be created and updated. To specify entity instances, the syntax is:

<new-instance> ::=

NEW <entity-type-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>}.

The values for all the key attributes must be given.

An example to describe a supplier instance is:

```
new supplier,
number = 'S1',
name='Wiley',
city='London'.
```

An example to describe a part instance is:

```
new part,
number = 'P1',
name='tire'.
```

To describe a relationship instance, the user has to provide values for the attributes of the relationship, and specify the entity instances involved in the relationship instance. The syntax is:

<new-instance> ::=

NEW <relationship-type-name>

{, <attribute-name> = <value>},

ENTITY <entity-type-name> WITH ROLE <role-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>},

ENTITY <entity-type-name> WITH ROLE <role-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>}

{, ENTITY <entity-type-name> WITH ROLE <role-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>} }.

An example to describe the relationship between a supplier instance and a part instance is:

```
new supply,
price=100, quantity=1000,
entity supplier with role supplies,
number='S1',
entity part with role supplied_by,
number='P1'.
```

Now we describe updates: either deleting an instance, or changing certain attribute values of the instance. The format is similar to retrieval commands. In place of the select clause, the user puts in the change/delete clause. The syntax is:

<change-clause> ::=

CHANGE <attribute-value> TO <value>

{, <attribute-value> TO <value>}

<delete-clause> ::=

DELETE <instance-identifier>

{, <instance-identifier> }

Multiple instances of different entity/relationship types may be deleted or changed in one command.

KQL allows the user to combine the select, delete and change clauses in a single command. With this flexibility, a user can change some attribute values, delete some instances, and print some attribute values, all in a single command.

Some potential ambiguities may arise during the descriptions. The rules that the system should follow are:

1. Non-specified attributes during a creation of an instance will be given null values, but values for key attributes must be given.
2. In the specification of a relationship instance, an entity instance can be specified by any attributes (that is, not necessarily the key attribute) as long as the system can find a unique instance with the given attribute values. Future work may extend the language such that in place of the attribute values, the user may put in a retrieval query which will select a unique instance of that entity.

   If the system finds two or more instances matching the given attribute values, then there is a question of whether the user has not given a tight

enough "selection" of one instance, or that the user deliberately means to include multiple instances. Probably, the system should confirm with the user that multiple instances of the entity type are to be selected, and consequently multiple instances of the relationship are to be created.

Creation and Update of IS_A Linked Instances

The following shows how the user can create entity instances that are involved in some IS_A relationship. The basic philosophy is to treat instances that are IS_A-linked as one big instance. This imposes the requirement that the system must be able to perform – by itself, without further instruction from the user – all kinds of inheritance. It must do downward inheritance, upward inheritance, and even sideway inheritance. These inheritances apply to relationships as well as attributes.

We assume that no conflict or ambiguity arises in the inheritance of attributes and relationships. Further work may add a module to detect possible ambiguities and obtain clarifications from the user.

To create instances, the user can input the following command, using syntax as for non-IS_A-linked entities:

```
new superentity-name,
attribute1=value1, . . ., attributeN=valueN.
```

The attributes may belong to lower entities; in which case, it is assumed that this instance of the top entity is also an instance of the lower entity whose

attributes have been listed.

The user can also input the following

```
new subentity-name,
attribute1= value1, . . .,
attributeN= valueN.
```

The attributes may belong to the superentity type or to other subentity types. If so, then instances will be created in these other types if the same instance is not yet existent. If the same instance already exists in the other types, then the attribute values will be changed.

As implied by the IS_A relationship, a subentity instance cannot exist without the same instance existing at the superentity types. However, the user is not required to create the superinstances before creating the subinstances. The system will create the superinstances if they are not already there.

The same flexibility is allowed for changes during updates, whereby the user may say `change employee profession = chemical` even though the attribute profession belongs to a subentity type instead of belonging to employee.

Retrievals using IS_A inheritance have been described in the main text.

APPENDIX D - FRAMEWORK COMPARISON OF KQL AND SQL

A language must be evaluated together with its (data) model. Users cannot understand SQL without understanding the relational model, and they cannot understand KQL without understanding the ER model. Hence, criteria about models and languages are important for our comparison.

This section reviews three comparison frameworks by McGee (1976), by Shneiderman (1977) and by Jarke and Vassiliou (1985). In general these frameworks contain many factors that are considered to be relevant or important in choosing a query language. The factors include the model, the language, the user and the computer system.

Out of the massive number of criteria contained in these frameworks, a smaller number will be selected to compare KQL and SQL.

1. Framework by McGee

The criteria proposed by McGee (1976) for evaluating data models include use criteria and implementation criteria. The implementation criteria are related to the physical implementation issues such as efficiency and availability of machines. The use criteria include the following factors:

1. learning the data model
    a. simplicity - a small number of structure types, and minimum number of attributes for the user to know.
    b. elegance - direct modelling capability with the smallest number of structure types.

c. picturability - for easier comprehension

2. using the model to model real world situations. McGee suggests that the real world can be described using these concepts: attribute, attribute value, property, entity, entity type, entity aggregation, entity association, and composite entity.

3. writing data definitions and programs to manipulate the structures of the model. The effort to write these will depend on

   a. data definition facility

   b. level of programming language, the higher the better.

   c. implementation independence

   d. data independence

   e. directness of modelling.

   f. partitionability

2. Framework by Shneiderman

Shneiderman (1978) proposes a framework for discussing database usage. The framework has the following factors:

1. functions — these include deletion, insertion, retrieval, and ancillary functions such as locking/unlocking, data definition, data security, and other utility functions.

2. Tasks — these are components in the user performance of the functions.

   a. learning the syntax and semantics of function specification.

   b. composition of the syntax required to perform a function.

   c. comprehension of function syntax composed by someone else.

   d. debugging of syntax or semantics.

e. modification of a function for a new query.

3. Interaction Modes — these include host language embedding of a query sublanguage, self-contained language, computer-directed language, natural language interface, and human intermediary.

4. retrieval response types — these include simple verification ('yes' or 'no' answer), single record retrieval, record collection retrieval, and total report listing of all information in a file.

5. Query features — these are further decompositions of the retrieval function. The following types of retrievals are identified:

   a. simple mapping which returns data values when a known data value for another field is supplied.

   b. selection of all data values associated with a specified key value.

   c. projection of columns

   d. Boolean queries which allow and/or/not connectives

   e. set operations

   f. built-in functions such as max, min, average, count and sum.

   The above query features are considered easy. The following types are difficult.

   g. combination queries (also known as composition) where the output of a query is the input of another

   h. grouping of items with a common domain value

   i. universal quantification.

6. user types — these include untrained intermittent users, skilled frequent users, and professional database users.

3. Framework by Jarke and Vassiliou

The framework proposed by Jarke and Vassiliou (1985) for comparing query languages include the following factors:

1. usability (query formulation effort)
   a. initial training effort to learn the query language. The effort, they suggest, will depend on the user type. The effort can be divided into two components:
      1) composition (learning to formulate)
      2) comprehension of other's queries
   b. repeat effort to query. This will be composed of
      1) thinking effort - remembering syntactic construct and using some query formulation model
      2) input - clerical effort
      3) error correction
2. functional capabilities of the query language.
   a. power — this criterion is a combination of many factors, including
      1) application dependence
      2) database dependence
      3) amount of functionality (queries, browsing, report generation, updates, deletions)
      4) degree of selectivity (relational completeness) - availability of operators that allow the user to specify as precisely as possible what data he or she wants to retrieve.
   b. output presentation - the amount of user control over the output.
3. interaction methods provided by the query language. These can be classified

into two types:

a. ergonomically oriented

1) function key

2) menu selection

3) line by line prompting

b. programming language oriented

1) record at a time

2) restricted natural language

3) linear keyword

4) mathematical

4. user types - the users are classified based on four factors

a. familiarity with programming concepts

b. frequency of system usage

The above two factors produce three classifications: novice, skilled and professional users

c. range of operations

d. application knowledge

The last two factors produce four classifications: casual, clerical, managerial and application specialist users.

4. Selection of Criteria

We will proceed on the assumption that most of the criteria in the frameworks are in general relevant for comparison of query languages. Even with the frameworks, comparing two query languages is not easy. There is no standard for measuring most of the factors. It is also not clear how factors, if

measurable, can be aggregated. The aggregation may depend heavily on the user. Furthermore, some factors are contradictory. For example, a language with more functionality (which make the language more powerful) may be harder to learn.

Since we are comparing two specific query languages – KQL and SQL – we can eliminate many of the criteria so as to make the comparison easier. The criteria are eliminated either because they are irrelevant or because they have the same values for both languages.

The criteria proposed by McGee appear to be relevant except for the factor of implementation independence. Both KQL and SQL are implementation independent, at least theoretically. In practice, different implementations of SQL have slighlty different versions.

Of the criteria in the framework proposed by Shneiderman, only two are relevant: tasks and query features. On the other criteria there are no differences between KQL and SQL. On the criterion of functions, KQL is intended to have as many functions as SQL. The KQL functions will include deletion, insertion, retrieval, and 'data definition'. The other functions like data security and concurrency control are not investigated in this dissertation. On the criterion of interaction modes, we will compare KQL and SQL both as self-contained languages. Theoretically, KQL can be embedded in other host languages, as in the case of SQL. On the criterion of retrieval response type, both KQL and SQL retrieve collections of instances/tuples meeting the query conditions.

Of the criteria in the framework proposed by Jarke and Vassiliou, only the usability criteria and selectivity are relevant. The others are the same for both KQL and SQL. On the functional capability, both KQL and SQL are application independent and they are both database dependent. The amount of functionality is the same, as discussed above. On the interaction method, both KQL and SQL belong to the linear keyword type of language.

In conclusion, only the following factors extracted from the frameworks are relevant in the comparison of KQL and SQL.

1. effort required for learning the model
2. effort required to model the real world
3. facility for data definition/ knowledge declaration
4. data independence
5. retrieval and update
    a. learning the syntax
    b. thinking effort
    c. composition of the query
    d. completeness
    e. typing effort in entering the query.
6. comprehension of others' queries

The comparison will not focus on any specific type of user. Both novice and specialist users can use SQL, though the specialist is probably better at using the more complicated features of the language. Similarly, KQL is designed so that both novice and specialist users can use it. There are simple features for

the novice, and there are advanced features to suit more complex queries.

5. Comparison Based on the Framework Criteria

This section will further compare KQL and SQL based on the six criteria distilled in the previous section.

The first two criteria are actually very closely related. There is not much point in having a model that is easy to learn but very difficult to use. We therefore consider these two criteria together, and compare the actual usage of the models. The model used in SQL is the relational model and that used in KQL is the ER model. The literature favours the ER model. The many advantages of using the ER model have been stated in Chapter V. Those arguments are not repeated here.

In addition, a common database design approach uses the ER model as the conceptual model and the relational model as the underlying logical model (Chen 1976, Poonen 1979, Braind et al. 1985, Ling 1985, Parent and Spaccapietra 1985, Storey 1986, Teorey, Yang and Fry 1986, Azar and Pichat 1987, Reiner et al. 1987, Schuldt 1987, Brady and Dampney 1987). This approach uses the ER model rather than the relational model for validation with the user. This strongly suggests that users are believed to understand the ER model better than the relational model. The relational model, despite its apparent simplicity, has many technical details, such as normalization (there are more than six levels of this) and the use of keys as logical pointers.

On the third criterion, SQL provides for data definition and KQL provides for declaration of the ER model at the knowledge level. Thus both have 'data definition' facility. However the knowledge declaration will be easier than the data definition in that the user need not worry about foreign keys, logical pointers and other representational problems.

On the fourth criterion of data independence, it has been shown that the relational model provides physical data independence but not logical data independence (Staley and Anderson 1985, Vossen and Brosda 1985, Date 1982 p139). KQL being on the knowledge level provides both physical and logical data independence. By definition of the knowledge level system, a user using KQL will make absolutely no references to any physical or logical data.

So far, KQL has been shown to be better than SQL on the first four criteria. A very detailed comparison has been done on the fifth criterion of retrievals and updates, where the sub-criteria are very closely related; for example, the syntax will depend on the type of query, and so is the thinking effort. Thus, it is impossible to consider many of these sub-criteria separately. Instead, we compare based on the knowledge needed by the user to compose the query, and also based on the data manipulation operations that the user has to perform.

It has been shown that KQL requires less knowledge of the user and reduces the number of data manipulation operations, which are often difficult, that the user has to perform when using SQL.

In Appendix E, we propose an additional measure of retrieval complexity and use it to evaluate equivalent KQL and SQL queries.

On the sub-criterion of retrieval completeness, it has been shown that KQL is relationally complete. SQL is also relationally complete. Which is more complete? All the SQL conditions are available in KQL. In addition, KQL as defined in general allows much more functions than SQL. A clear example is the inheritance of attributes and relationships. KQL can perform the inheritance — by itself. SQL allows the user to perform inheritance by specifying joins on relations. Since completeness includes coverage, ease and conciseness, we can say that KQL is more complete than SQL. It is easier and more concise to specify the inheritances in KQL than in SQL.

Also, from general considerations, we will not expect SQL to be able to do anything that KQL cannot. The reasoning is as follows. SQL operates on the data which is a representation of the concepts of the real world. KQL operates directly on these concepts. A representation cannot contain more details than the concept itself. Hence, if SQL can produce something that KQL cannot, that something must be faulty and contrary to the real world. Hence KQL is more complete than SQL.

Now we come to the sixth and last criterion of comprehensibility. It is only logical that in order to comprehend a query, one will need to understand the model used in the query. So in order to comprehend an SQL query, one needs to understand the relations and columns used in the query. Similarly, to

comprehend KQL queries, one needs to understand the ER elements used in the queries. It has been shown, in the section *The Knowledge Interface*, that the ER model is much easier to understand than the relational model. This is one factor that will contribute to easier comprehension of KQL queries compared to SQL queries.

There are other favourable qualities. As seen in the previous sections, KQL queries explicitly express the relevant domain knowledge; they are more direct; they are uncluttered with data level details and operations. These qualities contribute to easier comprehension. An example showing these differences is given below:

K-users:
```
select supplier name
where supplier supplies-related all parts.
```

D-users:
```
SELECT S.SNAME
FROM S
WHERE NOT EXISTS
     (SELECT *
     FROM P
     WHERE NOT EXISTS
          (SELECT *
          FROM SP
          WHERE SP.PNO=P.PNO
             AND SP.SNO=S.SNO))
```

In this example, the KQL query expresses domain knowledge; the SQL query describes relations and columns. The KQL query specifies the condition in a

direct way; the SQL query uses the "not exists" twice and, in addition, requires two joins to specify the condition. The data details in SQL make the query considerable more complex.

Incidentally, the easy comprehensibility allows for a better foundation to build intelligent interfaces such as interfaces that accept incomplete queries from the user. Motro (1986) describes a system that accepts tokens which are any pieces of data or fields or relations from the user. The system constructs the relational queries and asks the user for confirmation or clarification. The system is meant for naive users. However, we have to ask if the naive users can understand the relational queries provided to them. If not, then usage of the system is likely to lead to erroneous results. The knowledge level query, being easier to comprehend, will be better suited to be the base for such a system.

In summary, KQL is better than SQL on all the six criteria distilled from the frameworks.

APPENDIX E - MEASURING COMPLEXITIES OF KQL AND SQL RETRIEVALS

Program complexity is often measured by the number of lines of code or the number of function points (Albrecht and Gaffney 1983). A similar measure for query complexity is proposed here. We will count the number of operations and conditions in the query. Admittedly, there are serious drawbacks: one condition may be less complex than another, one combination may be more complex than another combination, and the query complexity may be in the selection and organization of the conditions. With these shortcomings, this measure is proposed only as a further indication of the difference between KQL and SQL retrievals.

In KQL, each of the following counts as one operation/condition:

1. one select clause
2. one instance declaration, whether explicit or implicit
3. one specific condition
4. one combination condition
5. one "NOT" condition
6. one statistical operation

In SQL, each of the following counts as one operation/condition:

1. one select clause
2. each relation in the FROM clause
3. one statistical operation
4. one "NOT" condition
5. one of these: <, >, =, EXISTS, IN, GROUP BY, HAVING.

A subquery in KQL or SQL counts as one operation/condition in addition to the number of operations/conditions within the subquery.

The following examples illustrate this measure of complexity in equivalent SQL and KQL queries. The curly brackets count the conditions/operations.

Example 1:

K-users:
```
select{1} supplier{2} name
where supplier supplies-related all part{3}.
```

D-users:
```
SELECT{1} S.SNAME
FROM S{2}
WHERE NOT{3} EXISTS{4}
     (SELECT{5} *
     FROM P{6}
     WHERE NOT{7} EXISTS{8}
          (SELECT{9} *
          FROM SP{10}
          WHERE SP.PNO=P.PNO{11}
             AND SP.SNO=S.SNO{12}){13}){14}
```

Example 2:

K-users:
```
p is a part{1}
s is a supplier{2}
select{3} s name, p name
where s supplies{4} p{5}
```

D-users:
```
SELECT{1} SNAME, PNAME
FROM S{2}, SP{3}, P{4}
WHERE S.SNO=SP.SNO{5} AND P.PNO=SP.PNO{6}.
```

APPENDIX F - BRIEF COMPARISON WITH OTHER ER LANGUAGES

When we compare KQL with the logical data level interfaces, there is a clear standard model, the relational data model, and a clear standard relational language, SQL, to compare with. There is therefore no need to compare with the many other data models and many other relational languages.

However, when we compare KQL with other ER languages, we are faced with many ER languages without one that is clearly better than the rest. There is therefore a massive task of comparing with about twenty other ER languages (Shoshani 1978 (Sho78), Poonen 1979 (Poo79), Atzeni and Chen 1981 (AC81), Atzeni et al. 1981 (ABLV81), Elmasri and Wiederhold 1981 (EW81), Hwang and Dayal 1981 (HD81), Markowitz and Raz 1983 (MR83), Nakano 1983 (Nak83), Zhang and Mendelzon 1983 (ZM83), Chen 1984 (Che84), Parent and Spaccapietra 1984 (PS84), Campbell et al. 1985 (CEC85), Cardenas and Wang 1985 (CW85), Elmasri and Larson 1985 (EL85), Roesner 1985 (Roe85), Velez 1985 (Vel85), Parent and Spaccapietra 1985 (PS85), Dogac et al. 1987 (DEA87), Flory and March 1987 (FM87), Junet 1987, (Jun87) Subieta and Missala 1987 (SM87)). The comparison is made more difficult by the often incomplete description of the languages.

To simplify and shorten the comparison, we will point out the advantages of KQL, and we will use examples from the other ER languages to illustrate the differences. Some of the comparison criteria such as completeness and number of functions will be objective. Some, such as naturalness and preferred syntax, will be subjective.

Probably the most important advantage for KQL is that it is defined at the knowledge level with the user in mind. There is a clear separation of the knowledge level and the data level. While many of the other languages claim to be of high level, it is not clear what the level is. There are no data structures in KQL; and there are no conditions or operations that are not in the real world. At the same time there are no unnecessary restrictions on the conditions that can be specified.

Illustration of conditions/operations not in the real world:

DEA87: The user defines a relationship by specifying the keys of the participating entities, rather than the names of the entities. This involves decisions on the logical pointers.

DEA87: In order to specify a relationship between two entity instances, the user has to link up the types, with conditions such as `link supplier, link supply, link part`.

EL85: From the ER diagram, the system creates and presents the user with a hierarchical diagram that is substantially different from the ER diagram. As a result, the user has to deal with another representation of his real world.

MR83: This has data structures where the order of the entities listed in a relationship is important; for example, "[e_1,. . .,e_n] represents a relationship . . . their ordering is significant." - (p330). Although it seems that in the actual queries, the orders do not appear anywhere.

HD81: This has data structure where the order of the entities listed in the relationship is important. This order appears in the actual queries.

Nak83: This has data structure where the order of the fields is important.

CEC85: The manipulation is difficult to understand. For example, the definition of add_relationship_union(D1, D2, Z) is

> If D1 and D2 are disjoint diagrams of the current ER model and Z is a bijection between equality-comparable attributes of D1 and D2, then add_relationship_union creates a new relationship R among the entities of D1 and D2 and causes the tuples associated with R to be the union of the tuples associated with D1 and D2. All attributes of both entities and relationships in D1 and D2 are removed, and a new set of attributes, one for each pair in Z, is created and placed in R. - (CEC85, p92)

This manipulation is so complex that it will be very hard to relate it to the real world.

CEC85: After transforming/manipulating the ER model, the user needs to write a relational algebra expression to display the result. This means that users need to know the relational representations for the ER model and also the relational algebra very well.

SM87: The user has to understand the data structure and the implementation details before he can understand the ER language. "The result produced by a query may depend on internal data structures" - (p203). The semantics of an operation is described in data/implementation terms. Two examples are:

> Semantics: The construct 'm is the number of j' augments the table returned by j by a new column containing pairs (m,i) where i is a consecutive number of a tuple within the table. - (212).

Semantics for the construct "j where p" follows: For each tuple t retrieved by j the following actions are performed:
- sub(con)(t) is put on the top of fovst.
- Predicate p is evaluated for this new state.
- The top of fovst is removed, that is, we return to the previous state.

Thus, each tuple retrieved by j is associated with a Boolean value produced by p. The result table contains tuples for which p returns 'true'. - (207)

Illustration of unnecessary restrictions:

DEA87: This does not have the concept of entity identity. To compare two instances if they are the same, the user has to compare their key values.

AC81: The user cannot compare attributes of entities that are not related.

AC81: The user can retrieve information of only one entity or relationship type in one query.

MR83: The relationships do not have any names. The relationship attributes has to be specified after the keyword AT immediately after a relationship condition.

SM87: This has rolenames, but no names for the relationships.

Che84: This is for a binary ER model with no relationship names or relationship attributes.

HD81: This has no attributes for relationships. Although some ER model has this restriction, this is probably an unnecessary restriction.

CEC85: The user has to manipulate whole sets instead of individual instances. To get two instances of the same type, the user has to duplicate the whole ER model and delete the unwanted duplications. Similarly, it is felt that Vel85 relies on too many set operations.

In general: Some of the languages, such as FM87 and Jun87, cannot specify multiple instances of the same entity/relationship type. This is a serious limitation on the query completeness.

FM87: This has very limited retrieval conditions. The user is presented with a list of all the attributes, and he can only specify printing or equalities (including > and <).

Jun87: The user can retrieve only one relation, and the conditions are also limited to one relation.

Rolenames are important in ER models. Besides conveying knowledge of the real world, these are absolutely essential when defining relationships with one entity having multiple roles. KQL has rolenames. Many of the other languages do not have rolenames (HD81, ABLV81, AC81, EW81, CEC85, PS84/85, ZM83). Some use rolenames differently. In EL85, if E1 and E2 are related with roles Ro1 and Ro2 respectively, the set of e2's that a e1 is related to is referred to as "Ro2 of e1" (the e1 and e2's are the instances). In MR83, non-existent rolenames can be used in the same way as defined rolenames. In Vel85, rolenames are used in place of the entity names.

KQL has functions for knowledge definition, insertions, deletions, changes

and retrievals. The descriptions for many of the other languages are limited to retrievals (AC81, Poo79, Shos78, CEC85, MR83, SM87, Vel85, EL85, ZM83). Some descriptions include "data defintion" but do not include updates (DEA87, FM87).

Even on the retrievals, KQL is more complete than the others. KQL handles IS_A inheritances. None of the other languages do this. Roe85 and ABLV81 allow the representation of IS_A relationships, but they do not perform inheritance for queries. KQL allows easier and more powerful combination conditions. None of the other languages provide this. KQL can be viewed as a query structure that allows more and more conditions to be attached.

Now we come to syntactical preferences. KQL adopts a clear separation of output and conditions. This has worked very well for SQL. The number of keywords are kept to the minimum. Too many keywords tend to confuse the user, as found in the case of SQL's "WHERE . . . GROUP BY . . . HAVING" (Welty 1985).

Illustration of keywords:

KQL: There are only two keywords: SELECT and WHERE; there is another keyphrase FOR EACH for statistical operations. SELECT and WHERE are used only once in a query, except in subqueries where they are repeated.

AC81: There are these keywords: FIND, WITH, THROUGH HAVING, HAVING, HAVING FOUND. A query can use WITH, THROUGH and HAVING many times.

Roes85: There are these keywords: FIND, WITH, THAT, WHICH. A query can use WITH, THAT and WHICH many times.

MR83: This uses GET and TIS. TIS is used in place of the more common WHERE. TIS is used many times in a query, such as GET ITEM TIS 'HAVING COLOR TIS = 'RED' OR = 'BLUE'. - (p338). Notice that TIS is used to qualify instances as well as attributes.

Vel85: This has SELECT, FROM and WHERE. Relationship conditions are specified with FROM while other conditions appear with WHERE.

Some of the other languages are influenced by graph theory terminology. For example, HD81 and CW85 want users to connect and disconnect entities in order to specify relationship instances. DEA87 also wants users to link the types.

Some of the other languages use SELECT in place of the more common WHERE (Shos78, DEA87). This may be a bit confusing. Some languages totally omit SELECT or its equivalent such as GET or FIND (SM87, Poo79). This omission tends to confuse what is exactly being retrieved, for example, SM87: "(supplier with (supplies.part))" will print all details of supplier, supplies and part. Also, SM87 uses the dot "." with two meanings: it can join entity/relationship types as well as specify attributes to be printed.

Because of the unique referencing of instances used in MR83, some of its queries may be a bit confusing as to the specific meanings. For example,

> GET ITEM TIS' REQUESTED "BY, AND 'SUPPLIED "TO, DEPARTMENT HAVING FLOOR=2

is equivalent to

> GET ITEM TIS 'REQUESTED "BY DEPARTMENT 'HAVING FLOOR =2 AND 'SUPPLIED "TO DEPARTMENT 'HAVING FLOOR=2. - (p343)

The meaning of these two queries is to get the items requested and supplied to departments located on the second floor. It will be quite logical to wrongly assume that the first query is asking for items supplied to and requested by the same department. The difficulty in determining the instances is also shown in this example:

> GET EMPLOYEE TIS 'EMPLOYED "BY DEPARTMENT 'MANAGED "BY EMPLOYEE HAVING SALARY < SALARY 'OF EMPLOYEE! - (p336).

Though commonsense probably dictates that the last EMPLOYEE is the same as the first EMPLOYEE, it is risky to rely on commonsense.

Many of them have immediate qualifications, for example, "DEPT THAT HAS NAME='FINANCE'". It becomes difficult or impossible to specify operations or conditions on attribute values of many instances, since the conditions may have to appear very far from the instances.

MR83 allows rolenames that are not defined in the model. Any undefined "rolename" can be used between the instance and the attributes. Also, comments are allowed within the query; for example, "employee 'earning salary", "employee 'with "double salary" and "employee 'spending salary" will all refer to the same thing: the salary of that employee. These extra comments are meant to make the query more comprehensible. But depending on the usage, these can create more confusion.

Some languages use special punctuation marks to denote instances of a type, such as using a space or a "!".

MR83: The instance declaration is by using a special punctuation mark: supplier!s2.

Vel85: The instance declaration is by leaving a space: "supplies s".

DEA87: This uses pointers instead of declaring instances, for example,

SELECT EMP POINTED BY PTR1

SELECT EMP (MGR=PTR1.ENO AND SAL>PTR1.SAL)

OUTPUT EMP - (p329)

KQL avoids complicated concepts like: a list of entities becoming an attribute of another entity, which may in turn be the attribute of another entity (PS85), relationships between relationships (Jun85), and the complex operations of CEC85.

Lastly, we try to answer some queries with the various ER languages.

There are five questions; they demand the use of quite a number of conditions, but they in no way exhaust the query possibilities. These queries are meant more to give a feel of the languages rather than as definitive evaluations. These are formulated by the author after much serious effort in understanding the languages; nevertheless, it might be that some misunderstanding has occurred, and that the original designers would have produced better queries.

Consider a simple ER model with entities SUPPLIER and PART. The attributes of SUPPLIER are NAME and NUMBER. The attributes of PART are NAME and NUMBER. The relationship between the entities is SUPPLY. The roles are respectively SUPPLIES and SUPPLIED_BY. The attributes of the relationship are PRICE and QUANTITY.

Question 1: find the name of suppliers who supply some parts.

KQL: `select supplier name`

`where supplier supplies part.`

SM87: `(supplier with (supplies.part)).sname`

note: some renaming is required.

Jun87 : The query cannot be formulated.

FM87: The user puts "?" against the attribute supplier name presented on the screen by the system. The relationship is assumed by the system.

DEA87: `select supplier`

`link supply`

`link part`

`output supplier.name`

Roe85: `Find supplier that 'supplies' part.`

(It seems that attributes to be printed cannot be specified)

Vel85: `select s.name`

`from supplies s of part p.`

CEC85: The user has to delete all the other unwanted types. Then he has to write a relational agebra to join supplier, supplies, part, and to project the supplier name.

EL85: The procedure is as follows:

The system presents the ER diagram.

The user selects `supplier, supply, part` as of interest.

The system erases all other `types`.

The user identifies `supplier` as of main interest.

The system transforms ER diagram to a hierarchical diagram.

The user specifies the attributes to be printed.

CW85: `GET(supplier. name) WHERE (supply)`

PS84/85: Strangely, it seems that this query and the next four queries cannot be formulated. An attribute produced by the operation "relationship join" is unnamed, and therefore cannot be compared with other values.

Assuming we call this new attribute X, then the query is:

[supplier.name]$\sigma_{\{(\_,\_)\}}$ subset X (supplier $\overset{*}{\text{supply}}$ part)).

where [...] represents a projection, σ represents a selection, and the "_" inside means any value.

The result is an entity that needs another command to display.

MR83: Get name 'given "of supplier; TIS 'supplies part.

Nak83: This is not possible unless the predicates (which are Prolog-like) are known. Assume that the predicates are s(sname, sno), p(pname, pno) and sp(price,qty,sno,pno), then the query is:

```
get(s) where (s(x,_),p(y,_),sp(_,_,x,y))
```

ZM83: The user marks supplier, supply and part in this order on the graph shown by the system, then he marks the attribute supplier name to be printed.

EW81 - This is similar to EL85. The query is based on the hierarchical diagram.

AC81: Find supplier through having supply

HD81: (There are no relationship name and no relationship attributes)

```
range of s is supplier
range of p is part
retrieve into result(name(s))
where [s,p] in supplies.
```

ABLV81: find supplier display(name) through supply having found part

Poo79 : The retrieval specifies the datatypes of the attributes rather than their names. If we assume that attributes have datatypes of the same name as the attributes prefixed with D, then the query is:

```
Dname of supplier in supply.
```

Sho78:
```
select supplier.supply
output supplier.name
```

Question 2: This question is the same as the previous question except that attributes are needed from three entity/relationship types instead of one. The question is: Find the suppliers and parts that are related through supply. Print the attributes in this order: supplier name and number, supply price and quantity, part name and number.

KQL:
```
select supplier name, supplier number, supply price,
        supply quantity, part name, part number
where supplier supply part.
```

SM87:
```
(supplier with (supplies.part)).(name, number, price,
quantity, pname, pnumber)
```

note: some renaming is required.

Jun87 : The query cannot be formulated.

FM87: The user puts "?" against the list of all the attributes presented on the screen by the system.

DEA87: select supplier

link supply

link part

output supplier.name, supplier.number, price, quantity, part.name, part.number

Roe85: The query cannot be formulated. It appears that retrievals are limited to a single entity/relationship type.

Vel85: select s.name, s.number, sp.price, sp.quantity, p.name p.number

from (supplies s, supplied_by p) of supply sp.

CEC85: The user has to delete all the other unwanted types, and then he has to write a relational agebra to join supplier, supplies, part and project the attributes.

EL85: The procedure is as follows:

The system presents the ER diagram;

the user selects supplier, supply, part as of interest;

the system erases all other types;

the user identifies supplier as of main interest;

the system transforms the ER diagram to a hierarchical diagram;

the user specifies the attributes to be printed.

CW85: GET(supplier. name, supplier. number, supply. price, supply. quantity, part. name, part. number)

WHERE (supply)

PS84/85: Making the same assumptions as for query 1, the query is:

[supplier.name, supplier.number,

X.supply.price, X.supply.quantity, X.part.name,

X.part.number]($\sigma_{\{(\_,\_)\}}$ subset X(supplier $\overset{*}{\text{supply}}$ part)).

where [...] represents a projection, σ represents a selection, and the "_" inside means any value. This query produces a new entity that needs a further command to display it.

MR83: Get name 'given "of supplier; number 'of supplier, price; quantity; name 'of part; number 'belonging "to part; TIS 'supplied_by supplier! at price and quantity.

(A relationship has no name)

Nak83: Assume the same predicates as in the first question, then the query is:

get(a,b,c,d,e,f) where (s(a,b), sp(c,d,a,e), p(e,f))

ZM83: The user marks supplier, supply and part in this order on the graph shown by the system. Then he marks all the attributes to be printed.

EW81 - This is similar to EL85; the user gets the same graphical display as in El85. The input in non-graphical but still based on the hierarchical diagram.

AC81: The query cannot be formulated.

HD81: There are no relationship name and no relationship attributes.

```
range of s is supplier
range of p is part
retrieve into result(name(s), number(s), name(p), number(p))
where [s,p] in supplies.
```

ABLV81: It appears that retrievals are limited to a single entity or relationship types

Poo79 : The retrieval specifies the datatypes of the attributes rather than their names. Omitting the attributes specification, the query is:

```
supplier, supply ,part in supply.
```

Sho78:

```
select supplier.supply.part
output supplier.name, number, supply.price, quantity,
part.name, number
```

Question 3: find suppliers who do not supply part 'P2'.

KQL:

```
select supplier name
where supplier not supply-related part,
part number='P2'.
```

SM87:

```
(supplier where not for any (supplies.part) holds
pnumber='P2').name
```

note: some renaming is required.

Jun87 : The query cannot be formulated.

FM87: The query cannot be formulated.

DEA87: (select supplier output sname)

difference

(select supply (pnumber='P2') output sname)

Roe85: Find supplier that not 'supplies' part with number='P2' (maybe?)

Vel85: select s.name

from supplies s of part p, part p2

where p2.number='P2'

and {p} by s interset {p2} ={}

CEC85: This is probably possible since the language claims to be relationally complete, but it is too difficult to write.

EL85: The procedure is as follows:

The system presents the ER diagram.

The user selects supplier, supply, part as of interest.

The system erases all other types.

The user identifies supplier as of main interest.

The system transforms the ER diagram to a hierarchical diagram.

The user points to part number and specify NOT INCLUDE 'P2'.

The user specifies the attributes to be printed.

CW85: This is probably impossible.

PS84/85: Making the same assumptions as for query 1, the query is:

[supplier.name]$\sigma_{\{='P2'\}}$ not subset X.part.number (supplier $\overset{*}{\text{supply}}$ part)).

The result is an entity that needs another command to display.

MR83: Get name 'given "of supplier TIS not 'supplies part 'having number='P2'

(maybe?)

Nak83: Assume the same predicates as before, then the query is:

get(x) where not (s(x,_), sp(_,_,x,'P2'))

ZM83: The query cannot be formulated.

EW81: This is similar to EL85 where the query is based on the hierarchical diagram.

AC81: find supplier not (through supply having part with (number='P2'))

HD81: The query cannot be formulated.

ABLV81: This is the same as AC81.

Poo79: The query cannot be formulated.

Sho78: The query cannot be formulated.

Question 4: find suppliers who supply more than 3 parts.

KQL: select supplier name

where supplier supply-related >3 part.

SM87: (supplier where count(supplies) >3).name

Jun87 : The query cannot be formulated.

FM87: The query cannot be formulated.

DEA87: (select supply group by sname

count(pnumber) >3) output sname

Roe85: This may be possible, but it is not described.

Vel85:
```
select s.name
    from supplies s of part p
    where count{p} by s > 3.
```

CEC85: The query cannot be formulated.

EL85: The query cannot be formulated.

CW85: The query cannot be formulated.

PS84/85: It appears that this is not possible.

MR83:
```
Get name 'given "of supplier TIS 'supplies count part >3
```

Nak83: Assume the same predicates as before, then the query is:

```
get(x) where (s(x,_),GT(count(y/sp(_,_,x,y)),3))
```

ZM83: The query cannot be formulated.

EW81: This is similar to EL85 where the query is based on the hierarchical diagram.

AC81: The query cannot be formulated.

HD81: The query cannot be formulated.

ABLV81: This is the same as AC81.

Poo79: This may be possible, but the count function is not described.

Sho78: The query cannot be formulated.

Question 5: find suppliers who supply every part.

KQL:
```
select supplier name
    where supplier supply-related all part.
```

SM87:
```
(supplier where (supplies.part) superset part).name
```

Jun87 : The query cannot be formulated.

FM87: The query cannot be formulated.

DEA87:
```
(select supply group by sname keep pnumber)
contains
(select part keep pnumber)
output sname
```

Roe85: This may be possible since it has a count function, but it is not described.

Vel85:
```
select s.name
from supplies s of part p, part p2
where {p} by s = {p2}
```

CEC85: This is probably possible since the language claims to be relationally complete, but it is too difficult to write.

EL85: This is probably possible but not described.

CW85: The query cannot be formulated.

PS84/85: It appears that this is not possible.

MR83:
```
Get name 'given "of supplier TIS 'supplies set part eq set part.
```

Nak83: Assume the same predicates as before, then the query is:

```
get(x) where (s(x,_), forall y(p(y,_)->sp(_,_,x,y))
```

ZM83: The query cannot be formulated.

EW81: This is similar to EL85.

AC81:
```
find s:supplier with (find part through supply having s)
superset (find part)
```

HD81: The query cannot be formulated.

ABLV81: This is the same as AC81.

Poo79: This is probably not possible.

Sho78: The query cannot be formulated.

APPENDIX G - PROCESSES OF COMPILER-1

This appendix shows the processes done by compiler-1. As described in the main text, compiler-1 accepts the query from the parser and performs the following processes.

a. Processing of Inheritance Conditions

The queries involving the "inheritance" expressions are "filled-out" before further processing by compiler-2. In this way, no further algorithm is needed for compiler-2 or the executor.

The "filling-out" procedure is as follows:

1. Replace each case of "e_i b_j" where b_j is not an attribute of e_i by "IS_A(e_i E_j) b_j" where b_j is an attribute of E_j and E_j is IS_A-linked to the entity type of e_i.
2. For each relationship existence condition involving e_i and R_j where R_j is not a relationship involving the entity type of e_i, replace "e_i" by "IS_A(e_i E_j)" where R_j is a relationship involving E_j and E_j is IS_A-linked to E_i.
3. Note all "IS_A(e_i E_j)", and do the following:
   a. add an instance declaration next to the declaration of e_i:

      e_j is an instance of E_j

      (e_j must not have been used in the query already, and thus it should be a unique system-generated name.)
   b. add one condition to the where-clause immediately following the instance-clause containing e_i:

$e_i\ k_i = e_j\ k_j$

where k_i is the key attribute of e_i, and k_j is the key attribute of e_j.

c. replace each "IS_A(e_i E_j) b_j" in the outermost select-clause and which is not part of any arithmetic operation by "$e_i\ b_j = e_j\ b_j$".

d. replace all other "IS_A(e_i E_j)" in the query by "e_j".

4. Repeat the previous step till there are no more "IS_A(e E)" in the query.

The other IS_A conditions are replaced as follow:

1. e IS_A E_i =

EXISTS

(e_i is a E_i

SELECT e_i

WHERE e k = e_i k_i)

where k is the key attribute of e, and k_i is the key attribute of e_i. The parser will have checked that E_i and the entity type of e are IS_A-linked, else the condition is invalid.

2. $e_i = e_j$ is replaced by

$e_i\ k_i = e_j\ k_j$

where k_i and k_j are the key attributes of e_i and e_j respectively. The parser will have checked that the entity types of e_i and e_j are IS_A-linked, else the condition is false.

b. Processing of Combination Conditions

This section shows how the combination conditions are replaced by the more primitive conditions.

1. e_i R-RELATED e_j =

EXISTS

(r is a R

SELECT r *

WHERE e_i r e_j)

2. e_i R-RELATED ALL E_j =

NOT EXISTS

(e_j is a E_j

SELECT e_j *

WHERE NOT EXISTS

(r is a R

SELECT r *

WHERE e_i r e_j))

3. e_i R-RELATED NO E_j =

NOT EXISTS

(r is a R

e_j is a E_j

SELECT r *

WHERE e_i r e_j)

4. e_i R-RELATED θ N E_j =

EXISTS (r is a R,

e_j is a E_j

SELECT r *

WHERE e_i r e_j, count(e_j) θ N)

5. Similar expansions can be made to the Role-RELATED conditions.

c. Processing of Defined Subset Entities

Recall that a subset entity can be defined as follows:

```
e is an E
[, <instance-clause>s ]
select E' = e
<where-clause>s
```

E' is the name for the subset of E that fits the conditions in the where-clause.

In subsequent queries, wherever E' appears with, say, e' as its instance, it can be replaced by E and the instance e' checked to see that it is included in E'. Thus a general query,

```
[ <instance-clause>, ]
e' is an E'
[, <instance-clause> ]
<select-clause>
[ <where-clause> ]
```

will be converted to

```
[ <instance-clause>, ]
e' is an E
[, <instance-clause> ]
<select-clause>
```

```
[ <where-clause> ] [ WHERE ] EXISTS
    ( e is an E
    [, <instance-clause>s ]
    select e
    <where-clause>s, e = e').
```

d. Processing of Defined Subset Relationships

Recall that a subset relationship can be defined as follows:

```
r is an R
[, <instance-clause>s ]
select R' = r
<where-clause>s
```

R' is the name for the subset of R that fits the conditions in the where-clause.

In subsequent queries, wherever R' appears with, say, r' as its instance, it can be replaced by R and the instance r' checked to see that it is included in R'. Thus a general query,

```
[ <instance-clause>, ]
r' is an R'
[, <instance-clause> ]
<select-clause>
[ <where-clause> ]
```

will be converted to

```
[ <instance-clause>, ]
r' is an R
```

```
[, <instance-clause> ]
<select-clause>
[ <where-clause> ] [ WHERE ] EXISTS
    ( f(( r is an R
        [, <instance-clause>_s ]
        select r
        <where-clause>_s ), r , r')).
```

f(Query, x, y) produces a query where all the "x"s in Query are replaced by "y"s.

An example of pre-processings is the expansion of this condition:

e_i R-RELATED ALL E_j' , where E' denotes a subset of E.

The first step is to expand the combination condition. we get the following:

```
NOT EXISTS
(e_j' is a E_j'
SELECT e_j' *
WHERE NOT EXISTS
    (r is a R
    SELECT r *
    WHERE e_i r e_j' ))
```

The next step is to replace the subset entity. We get the following condition:

```
NOT EXISTS
(e_j' is a E_j
SELECT e_j' *
WHERE NOT EXISTS
```

(r is a R

SELECT r *

WHERE e_i r e_j'),

EXISTS

(e_j is an E_j

[, <instance-clause>$_s$]

select e_j

<where-clause>$_s$, e_j = e_j')).

APPENDIX H - KQL->SQL TRANSLATION ALGORITHM

In order to derive the algorithms for the translation of KQL queries to SQL queries, we first have to describe the relational representation of an ER model. An entity E with key attribute k and other attributes a_1, a_2, . . ., a_n is represented by the relation:

$E(k, a_1, a_2, . . ., a_n)$.

For example, the entity SUPPLIER with attributes NUMBER, NAME and CITY will be represented by the relation:

SUPPLIER(NUMBER, NAME, CITY).

A relationship R with attributes a_1, a_2, . . ., a_n, and involving entities E_1, E_2, . . ., E_m with roles $role_1$, $role_2$, . . ., $role_m$ respectively and keys k_1, k_2, . . ., k_m respectively, is represented by the relation:

$R(E_1\_role_1\_k_1, E_2\_role_2\_k_2, . . ., E_m\_role_m\_k_m,$
$a_1, a_2, . . ., a_n)$.

For example, the relationship SUPPLY with attributes PRICE and QUANTITY involving the entity SUPPLIER with role SUPPLIES and the entity PART with role SUPPLIED_BY will be represented by the relation:

SUPPLY(SUPPLIER_SUPPLIES_NUMBER,
PART_SUPPLIED_BY_NUMBER,
PRICE, QUANTITY).

We have to describe the KQL query that is being translated. Since KQL has been defined with BNF grammar, we will use the BNF description here.

The translator basically goes through each BNF term of the query, translating the terms, sometimes recursively, and reorganizing the translations into an SQL query. The translator will have to know the relations and the ER model.

To simplify the algorithm, we will assume that the user's KQL query has already been processed to remove all ambiguities. Also, the more complex conditions, such as those involving inheritance, subset and combination conditions, have been replaced by more "primitive" conditions. This replacement reduces the number of algorithms needed.

A general KQL query has the following form:

Q_1 **U** Q_2 **U** . . . **U** Q_n

where each Q_i has the following form:

instance_clause(i)

SELECT list(i)

WHERE condition(i)

The following notation is used:

1. e_i refers to an instance of an entity. It is used in place of <entity-instance-identifier>.
2. r_i refers to an instance of a relationship. It is used in place of <relationship-instance-identifier>.
3. ro_i refers to an instance of a role. It is used in place of <role-instance-identifier>.
4. x_i refers to an instance that can be of an entity, a relationship or a role.

It is used in place of <instance-identifier>.

5. a_i refers to the name of an attribute. It is used in place of <attribute-name>.

6. the subscripts are dropped if there is no ambiguity.

The translation algorithms are as follow:

Rule 1: The SQL equivalent of Q is denoted by SQL(Q).

$$SQL(Q_1 \ U \ Q_2 \ U \ . \ . \ . \ U \ Q_n) =$$
$$SQL(Q_1) \ U \ SQL(Q_2) \ U \ . \ . \ . \ U \ SQL(Q_n).$$

Rule 2: Translation of a single query.

SQL(instance_clause, SELECT list, WHERE conditions) =

SELECT SS(list)

FROM SF(instance_clause)

WHERE SW(conditions) SSW(list)

SSW(list) produces conditions only if "list" contains statistical operations.

Rule 3: Translation of the instance clauses. An instance clause may have many instance-of clauses.

$$SF(instance\_of_1, \ instance\_of_2, \ . \ . \ ., \ instance\_of_n) =$$
$$SF(instance\_of_1), \ SF(instance\_of_2), \ . \ . \ ., \ SF(instance\_of_n)$$

Each instance-of clause is translated by these sub-rules.

Rule 3.1:

SF(e is an instance of E) = E e

Rule 3.2:

SF(r is an instance of R) = R r

Rule 3.3:

SF(ro is an instance of Role) = R ro

Since the role names need not be unique, the translator has to check the rest of the query to deduce R. The deduction is as follows: check query for "e_j ro e_k", check the query for E_j and E_k that e_j and e_k are instances of, and then check the ER model for R that connects E_j and E_k and that the role of E_j is Role.

Rule 4: Translation of the select-clause list.

SS(list) =

$SAO(l_1)$, $SAO(l_2)$, . . ., $SAO(l_n)$

where list=[l_1, l_2, . . ., l_n] and each l_i is an <arithmetic-operation>, with or without renaming, as defined in the KQL syntax. Note that a simple "e a" is included as an <arithmetic-operation>.

Rule 4.1: Translation of arithmetic operation with renaming.

SAO(XYZ = <arithmetic-operation>) =

XYZ = SAO(<arithmetic-operation>)

Rule 4.2: Translation of arithmetic operation.

SAO(<arithmetic-operation>) =

SAO(<arithmetic-term>

[<add-minus> <arithmetic-term>)])=

SAT(<arithmetic-term>)

[<add-minus> SAT(<arithmetic-term>)]

Rule 4.3: Translation of an <arithmetic-term>.

SAT(<arithmetic-term>) =

SAT(<arithmetic-subterm> [<time-divide> <arithmetic-subterm>]) =

SAT2(<arithmetic-subterm>)

[<time-divide> SAT2(<arithmetic-subterm>)]

Rule 4.4: Translation of an <arithmetic-subterm>.

SAT2(<arithmetic-subterm>) =

SAO(<arithmetic-operation>) | SV(<simple-value>)

Rule 4.5 Translation of <simple-value>

SV(<simple-value>) =

SV1(<attribute-value>)

| SV2(<user-given-value>)

| SSO(<statistical-operation>)

Rule 4.6 Translation of <attribute-value>

SV1(<attribute-value>) = SV1(x a) = x.a

Rule 4.7 Translation of <user-given-value>

SV2(<user-given-value> = <user-given-value>

Rule 4.8 Translation of statistical operations.

Rule 4.8.1 Counting of entity instances.

SSO(<instance-statistical-operation>)=

SSO(COUNT UNIQUE e [FOR EACH <grouping-values>]) =

COUNT (DISTINCT e.k)

k is the key of E that e is an instance of.

Rule 4.8.2 Counting of entity instances.

SSO(<instance-statistical-operation>)=

SSO(COUNT e [FOR EACH <grouping-values>]) =

COUNT (*)

Note: SQL requires "DISTINCT" for all counts except count(*). This restriction applies to rule 4.8.3 as well. If the key consists of two or more attributes, rule 4.8.1 needs to be modified.

Rule 4.8.3 Translation of <attribute-statistical-operation>

SSO(<attribute-statistical-operation>) =

SSO(<attrribute-statistical-operator>

[UNIQUE]

(x a [FOR EACH <grouping-values>]))=

<attribute-statistical-operator> ([DISTINCT] x.a)

Rule 5 Translation of statistical "FOR EACH" conditions.

Rule 5a: SSW(list) = nothing (that is, SSW(list) is ignored)

if "list" does not contain any statistical operation.

Rule 5b: SSW(list) =

$SSW(SO_1)$ AND $SSW(SO_2)$ AND . . . AND $SSW(SO_n)$†

where the SO_is are all the statistical operations which are contained in "list" and which contains the "FOR EACH" condition.

Rule 5.1

SSW(<instance-statistical-operation>) =

SSW(COUNT e [FOR EACH <grouping-values>]) =

GROUP BY SSW(<grouping-values>)

Rule 5.2

SSW(<attribute-statistical-operation>) =

SSW(<attribute-statistical-operator> [UNIQUE] x a

[FOR EACH <grouping-values>]) =

GROUP BY SSW(<grouping-values>)

Rule 5.3 Translation of <grouping-values>

SSW(<grouping-value> [, <grouping-values>]) =

SSW(<grouping-value>) [,SSW(<grouping-values>)]

Rule 5.3b

SSW(<grouping-value>) = SSW(e) = e.k

where k is the key of e.

Rule 5.3c

SSW(<grouping-value>) = SSW(x a) = x.a

Rule 6.1:

SW(<condition-list>) =

SW(<condition-andlist> [OR <condition-list>])=

†A limitation is discussed together with rule 7.2.

SW(<condition-andlist>) [OR SW(<condition-list>)]

Rule 6.2:

SW(<condition-andlist>) =

SW(<condition> [, <condition-andlist>])=

SC(<condition>) [AND SW(<condition-andlist>)]

Rule 6.3:

SC(<condition>) =

SC(not(<condition-list>)) =

NOT (SW(<condition-list>)).

A <condition> is either "not(<condition-list>)" or some actual condition such as "EXISTS (Q)". Rule 6.3 deals with the first case. The actual conditions are dealt with under rule 7.

Rule 7.1: Translation of "EXISTS"

SC(exists (Q)) = exists (SQL(Q))

where Q is a subquery.

Rule 7.2: Translation of θ conditions where θ is one of the following comparisons: =, <, >, <= or >=.

SC(<value> θ <value>) =

SSW(<value>), SSW(<value>)

HAVING SAO(<value>) θ SAO(<value>)

If the SSW()s return nothing (when there are no statistical operations with "FOR EACH" conditions in the values), then omit the word "HAVING" and the result of SSW()s.

If the SSW()s return some "group by" clause, then some restrictions apply. One restriction for the KQL query is that the whole query Q_i

should contain only statistical operations with the same "FOR EACH <grouping-values>". SQL cannot handle two independent GROUP BY...HAVING clauses. Another restriction, by the translator, is that this condition be given at the end of the query even though KQL allows it to be given anywhere in the where-clause. Some tidying up of the result is necessary to suit SQL: the usual AND between conditions must be removed if it is followed by "group by"; if there are no other conditions, then the "WHERE" before the "group by" must be removed; and if there is a "NOT" in front of the "group by", it must be shifted to after the following "having".

Rule 7.3: Translation of IN condition

Rule 7.3a

SC(x a IN value_list) = x.a IN value_list

where value_list is a list of values such as (2,3,8).

Rule 7.3b:

SC(x a IN (Q)) = x.a IN (SQL(Q))

where Q is a subquery.

Note: Rule 7.3a and 7.3b deals with a restricted version of the IN condition. The flexibility described for KQL is not directly translatable to SQL; for example, "3 * x_i a_i IN (2 + x_j a_j, 5 * x_k a_k)" cannot be directly translated.

Rule 7.4: Translation of relationship existence condition

Rule 7.4a

$SC(e_i\ r\ e_j)$ =

$e_i.k_i = r.E_i\_Role_i\_k_i$

AND

$e_j.k_j = r.E_j\_Role_j\_k_j$

where e_i, e_j are instances of E_i and E_j which have keys k_i and k_j. E_i and E_j can be found in the query, but $Role_i$, $Role_j$, k_i and k_j have to be found in the ER model.

Rule 7.4b: (This is similar to rule 7.4a)

$SC(e_i\ ro\ e_j)$ =

$e_i.k_i = ro.E_i\_Role_i\_k_i$

AND

$e_j.k_j = ro.E_j\_Role_j\_k_j$

where e_i, e_j are instances of E_i and E_j which have keys k_i and k_j. E_i, E_j and $Role_i$ can be found in the query, but $Role_j$, k_i and k_j have to be found in the ER model.

APPENDIX I - EXAMPLES OF KQL QUERIES AND THE SQL TRANSLATIONS

In this appendix, we will show quite a number of KQL queries and the equivalent SQL queries produced by the KQL->SQL translator. These queries will also serve as further illustrations of the KQL language.

The KQL queries are based on the domain knowledge shown by the ER model in Figure 3. In this figure, attribute names are italicized . All the key attributes are "number"s. The SQL translations are based on a set of relations that represents the knowledge in the ER model. The relations are:

```
EMPLOYEE(NUMBER, NAME, BIRTHDATE, SALARY)
ENGINEER(NUMBER, PROFESSION)
MANAGER(NUMBER, RANK)
DEPARTMENT(NUMBER, NAME, CITY)
PROJECT(NUMBER, NAME)
WORK(EMPLOYEE_EMPLOYED_IN_NUMBER,
      DEPARTMENT_EMPLOYS_NUMBER, DATE)
MANAGEMENT(MANAGER_MANAGES_NUMBER,
            DEPARTMENT_MANAGED_BY_NUMBER, DATE)
HEAD(ENGINEER_HEADS_NUMBER,
       PROJECT_HEADED_BY_NUMBER, DATE).
```

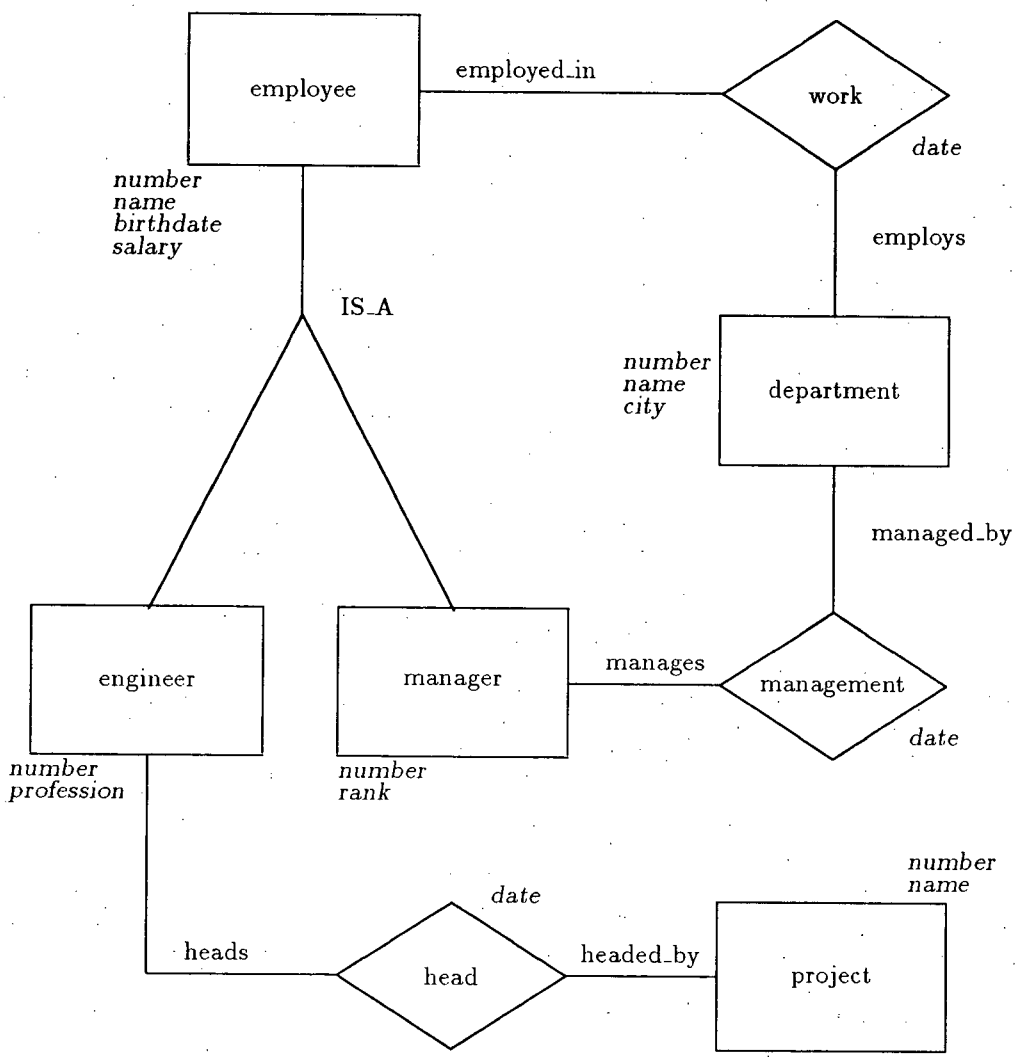

Figure 3: Example of an ER Model

In the examples below, each KQL query is followed by its SQL translation. The SQL queries are shown in block letters. In the belief that the KQL query is easily comprehensible, no natural language description of the query is provided. For some queries, the SQL version is executed and the results are displayed. Unfortunately, because of some bugs in the SQL software used for this research, not every SQL query can be executed properly. In the SQL queries, there is often something like `FROM X X, X Y`. The first `X` is the relation name; the second `X` is a variable to refer to this relation; the third `X` is again the relation name, and the `Y` is another variable to refer to this relation. In this example, relation `X` has been used twice.

The queries are:

This example shows a simple query on the attributes of one entity.

```
select employee name, number, salary , birthdate.
SELECT EMPLOYEE.NAME, EMPLOYEE.NUMBER,
          EMPLOYEE.SALARY, EMPLOYEE.BIRTHDATE
FROM EMPLOYEE EMPLOYEE   ;
```

This example shows a simple condition on an attribute value.

```
select employee number
where employee salary < 1500.
SELECT EMPLOYEE.NUMBER
FROM EMPLOYEE EMPLOYEE
```

```
WHERE EMPLOYEE.SALARY<1500;
```

This example shows the use of two instances of the same type. The instances have to be explicitly declared.

The previous examples show implicit declaration of instances by using the type names.

```
e1 is employee, e2 is employee
select e1 name, e2 name
where e1 salary > 2 * e2 salary.
SELECT E1.NAME, E2.NAME
FROM EMPLOYEE E1, EMPLOYEE E2
WHERE E1.SALARY>(2*E2.SALARY);
```

This example shows a simple relationship condition.

```
select employee number, department number, work date
where employee work department.
SELECT EMPLOYEE.NUMBER, DEPARTMENT.NUMBER, WORK.DATE
FROM WORK WORK, DEPARTMENT DEPARTMENT, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
  AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER;
```

This example shows a relationship condition specified with the role name.

```
select employee number, department number, employed_in date
where employee employed_in department.
SELECT EMPLOYEE.NUMBER, DEPARTMENT.NUMBER, EMPLOYED_IN.DATE
FROM WORK EMPLOYED_IN, DEPARTMENT DEPARTMENT, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=EMPLOYED_IN.EMPLOYEE_EMPLOYED_IN_NUMBER
  AND DEPARTMENT.NUMBER=EMPLOYED_IN.DEPARTMENT_EMPLOYS_NUMBER;
```

In the SQL queries, a `X_Y = KQLn.B` in the SELECT clause represents a renaming of `KQLN.B` TO `X_Y`. This happens only when an attribute is "inherited". The symbol `KQLn` is a variable name generated by the KQL−>SQL translator. This is shown in the next example.

This example shows inheritance of an attribute.

```
e is employee
select e number, name, profession.
SELECT E.NUMBER, E.NAME, E_PROFESSION = KQL0.PROFESSION
FROM ENGINEER KQL0, EMPLOYEE E
WHERE E.NUMBER=KQL0.NUMBER;
```

This example shows multiple inheritances of attributes.

```
select employee number, profession, rank.
```

```
SELECT EMPLOYEE.NUMBER, EMPLOYEE_PROFESSION = KQL1.PROFESSION,
          EMPLOYEE_RANK = KQL2.RANK
FROM MANAGER KQL2, ENGINEER KQL1, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=KQL2.NUMBER
  AND EMPLOYEE.NUMBER=KQL1.NUMBER;
```

This example shows inheritance of a relationship.

```
select employee name
where employee manages department.
SELECT EMPLOYEE.NAME
FROM MANAGER KQL6, MANAGEMENT MANAGES,
      DEPARTMENT DEPARTMENT, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=KQL6.NUMBER
  AND KQL6.NUMBER=MANAGES.MANAGER_MANAGES_NUMBER
  AND DEPARTMENT.NUMBER=MANAGES.DEPARTMENT_MANAGED_BY_NUMBER;
```

This example shows multiple inheritances of relationships.

```
select employee name
where employee heads project , employee manages department.
SELECT EMPLOYEE.NAME
FROM MANAGER KQL8, ENGINEER KQL7, MANAGEMENT MANAGES,
      DEPARTMENT DEPARTMENT,
      HEAD HEADS, PROJECT PROJECT, EMPLOYEE EMPLOYEE
```

```
WHERE EMPLOYEE.NUMBER=KQL8.NUMBER
  AND EMPLOYEE.NUMBER=KQL7.NUMBER
  AND KQL7.NUMBER=HEADS.ENGINEER_HEADS_NUMBER
  AND PROJECT.NUMBER=HEADS.PROJECT_HEADED_BY_NUMBER
  AND KQL8.NUMBER=MANAGES.MANAGER_MANAGES_NUMBER
  AND DEPARTMENT.NUMBER=MANAGES.DEPARTMENT_MANAGED_BY_NUMBER;
```

This example shows relationship inheritance from a different direction.

```
select engineer name
where engineer work department, department name = 'finance'.
SELECT ENGINEER_NAME = KQL9.NAME
FROM EMPLOYEE KQL9, WORK WORK, DEPARTMENT DEPARTMENT,
        ENGINEER ENGINEER
WHERE ENGINEER.NUMBER=KQL9.NUMBER
  AND KQL9.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
  AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER
  AND DEPARTMENT.NAME='FINANCE';
```

This example shows the use of "NOT" condition.

```
select engineer name
where engineer work department, not ( department name = 'research').
SELECT ENGINEER_NAME = KQL10.NAME
FROM EMPLOYEE KQL10, WORK WORK, DEPARTMENT DEPARTMENT,
```

```
        ENGINEER ENGINEER
WHERE ENGINEER.NUMBER=KQL10.NUMBER
  AND KQL10.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
  AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER
  AND  NOT (DEPARTMENT.NAME='RESEARCH');
```

This example shows the use of the combination relationship condition. This condition also includes a relationship inheritance.

```
select employee name
where employee head-related all project.
SELECT EMPLOYEE.NAME
FROM ENGINEER KQL11, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=KQL11.NUMBER
  AND  NOT (
       EXISTS (
           SELECT KQL12.NUMBER
           FROM PROJECT KQL12
           WHERE  NOT (
                  EXISTS (
                      SELECT KQL13.ENGINEER_HEADS_NUMBER
                      FROM HEAD KQL13
                      WHERE KQL11.NUMBER=HEAD.ENGINEER_HEADS_NUMBER
          AND KQL12.NUMBER=HEAD.PROJECT_HEADED_BY_NUMBER))));
```

This example shows another use of the combination relationship condition.

```
select engineer name
where engineer heads-related no project.
SELECT ENGINEER_NAME = KQL17.NAME
FROM EMPLOYEE KQL17, ENGINEER ENGINEER
WHERE ENGINEER.NUMBER=KQL17.NUMBER
  AND  NOT (
       EXISTS (
         SELECT KQL19.ENGINEER_HEADS_NUMBER
         FROM HEAD KQL19, PROJECT KQL18
         WHERE ENGINEER.NUMBER=HEAD.ENGINEER_HEADS_NUMBER
           AND KQL18.NUMBER=HEAD.PROJECT_HEADED_BY_NUMBER));
```

This example shows the third use of the combination relationship condition. It also shows an attribute inheritance.

```
m is manager
select m name
where m manages-related >1 department.
SELECT M_NAME = KQL5.NAME
FROM EMPLOYEE KQL5, MANAGER M
WHERE M.NUMBER=KQL5.NUMBER
  AND  EXISTS (
       SELECT KQL6.NUMBER
       FROM MANAGEMENT KQL7, DEPARTMENT KQL6
       WHERE M.NUMBER=KQL7.MANAGER_MANAGES_NUMBER
```

```
    AND KQL6.NUMBER=KQL7.DEPARTMENT_MANAGED_BY_NUMBER
  GROUP BY KQL7.MANAGER_MANAGES_NUMBER
  HAVING COUNT( DISTINCT KQL6.NUMBER)>1);
```

This example shows a simple counting operation.

```
select count(employee number).
SELECT COUNT( DISTINCT EMPLOYEE.NUMBER)
FROM EMPLOYEE EMPLOYEE  ;
```

This example shows a simple "MAX" operation.

```
select max(employee salary).
SELECT MAX( EMPLOYEE.SALARY)
FROM EMPLOYEE EMPLOYEE  ;
```

This example shows a simple "MIN" operation.

```
select min(employee salary).
SELECT MIN( EMPLOYEE.SALARY)
FROM EMPLOYEE EMPLOYEE  ;
```

This example shows a simple "SUM" operation.

```
select sum(employee salary).
SELECT SUM( EMPLOYEE.SALARY)
FROM EMPLOYEE EMPLOYEE   ;
```

This example shows an "AVG" operation that needs the "FOR EACH" condition.

```
select avg(employee salary for each department number)
where employee work department.
SELECT AVG( EMPLOYEE.SALARY)
FROM WORK WORK, DEPARTMENT DEPARTMENT, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
  AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER
GROUP BY DEPARTMENT.NUMBER;
```

The next few examples show some KQL queries, their SQL translations, and the results returned after execution of the SQL queries.

```
select employee name, number, salary, birthdate.
SELECT EMPLOYEE.NAME, EMPLOYEE.NUMBER,
          EMPLOYEE.SALARY, EMPLOYEE.BIRTHDATE
FROM EMPLOYEE EMPLOYEE   ;
result(bill,e1,1000,010151)
```

result(harry,e2,2000,020252)

result(jack,e3,3000,030353)

result(john,e4,4000,040454)

result(tom,e5,5000,050555)

```
select department number, name, city.
SELECT DEPARTMENT.NUMBER, DEPARTMENT.NAME, DEPARTMENT.CITY
FROM DEPARTMENT DEPARTMENT   ;
```

result(d1,finance,london)

result(d2,accounting,paris)

result(d3,research,vancouver)

```
select manager number, rank.
SELECT MANAGER.NUMBER, MANAGER.RANK
FROM MANAGER MANAGER   ;
```

result(e1,senior)

result(e2,junior)

result(e3,junior)

```
select project number, name.
SELECT PROJECT.NUMBER, PROJECT.NAME
FROM PROJECT PROJECT   ;
```

result(p1,granville)

result(p2,fraser)

```
select work date , employee number, department number
where employee work department.
SELECT WORK.DATE, EMPLOYEE.NUMBER, DEPARTMENT.NUMBER
FROM DEPARTMENT DEPARTMENT, EMPLOYEE EMPLOYEE, WORK WORK
WHERE EMPLOYEE.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
  AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER;
```

result(010188,e1,d1)

result(020288,e2,d2)

result(030388,e3,d3)

result(040488,e4,d3)

result(050588,e5,d3)

```
select department number
where department employs employee.
SELECT DEPARTMENT.NUMBER
FROM WORK EMPLOYS, EMPLOYEE EMPLOYEE, DEPARTMENT DEPARTMENT
WHERE DEPARTMENT.NUMBER=EMPLOYS.DEPARTMENT_EMPLOYS_NUMBER
  AND EMPLOYEE.NUMBER=EMPLOYS.EMPLOYEE_EMPLOYED_IN_NUMBER;
```

result(d1)

result(d2)

result(d3)

result(d3)

result(d3)

```
select employee name, rank.
SELECT EMPLOYEE.NAME, KQL0.RANK
FROM MANAGER KQL0, EMPLOYEE EMPLOYEE
WHERE EMPLOYEE.NUMBER=KQL0.NUMBER;)
```

result(bill,senior)

result(harry,junior)

result(jack,junior)

```
select manager name, number , rank
where manager work department.
SELECT KQL1.NAME, MANAGER.NUMBER, MANAGER.RANK
FROM EMPLOYEE KQL1, WORK WORK, DEPARTMENT DEPARTMENT, MANAGER MANAGER
WHERE MANAGER.NUMBER=KQL1.NUMBER
   AND KQL1.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
   AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER;)
```

result(bill,e1,senior)

result(harry,e2,junior)

result(jack,e3,junior)

```
select manager name, department name
where manager work department.
SELECT KQL2.NAME, DEPARTMENT.NAME
FROM EMPLOYEE KQL2, WORK WORK, DEPARTMENT DEPARTMENT, MANAGER MANAGER
WHERE MANAGER.NUMBER=KQL2.NUMBER
   AND KQL2.NUMBER=WORK.EMPLOYEE_EMPLOYED_IN_NUMBER
   AND DEPARTMENT.NUMBER=WORK.DEPARTMENT_EMPLOYS_NUMBER;
```

result(bill,finance)

result(harry,accounting)

result(jack,research)

APPENDIX J - SYNTAX OF KQL

This appendix provides a summary of the syntax of KQL. The BNF descriptions in the previous chapters and appendices are reorganized here.

The input to the KQL system is a <command>.

<command> ::=

<query>

| <union-retrieval-query>

| <new-type>

| <new-isa>

| <type-update>

| <new-instance>

<query> ::=

[<instance-clause>]

<action-clause>

[<report-formatting-clause>]

[<where-clause>].

<instance-clause> ::=

<instance-identifier> <instance-type-connection> <type>

{,<instance-identifier> <instance-type-connection> <type>}

<instance-identifier> ::=

<entity-instance-identifier>

| <relationship-instance-identifier>

| <role-instance-identifier>

These identifiers are terminal.

<instance-type-connection> ::=

IS | IS A | IS AN

| IS AN INSTANCE OF

<type> ::=

<entity-type-name>

| <relationship-type-name>

| <role-name>

These names are terminal.

<action-clause> ::=

<select-clause> [<delete-clause>] [<change-clause>]

| <delete-clause> [<change-clause>]

| <change-clause>

<select-clause> ::=

<entity-relationship-subset-select-clause>

| <value-select-clause>

<entity-relationship-subset-select-clause> ::=

SELECT [<type-name> =] <instance-identifier>

{, [<type-name> =] <instance-identifier>}

Type-name is user-given, and it is terminal.

<value-select-clause> ::=

SELECT <select-item> {,<select-item>}

<select-item> ::=

<instance-identifier> <attribute-name>

| <instance-identifier> *

| <arithematic-operation>

Attribute-name is terminal.

<delete-clause> ::=

DELETE <instance-identifier>

{, <instance-identifier> }

<change-clause> ::=

CHANGE <attribute-value> TO <value>

{, <attribute-value> TO <value>}

<report-formatting-clause> ::=

REPORT <formatting-instructions>

<where-clause> ::=

WHERE <condition-list>

<condition-list> ::=

<condition-andlist> [OR <condition-list>]

<condition-andlist> ::=

<condition> [, <condition-andlist>]

<condition> ::=

NOT (<condition-list>)

| <relationship-existence-condition>

| <equality-attribute-value-condition>

| <comparison-condition>

| <membership-condition>

| <exists-condition>

| <isa-relationship-condition>

| <entity-equality-condition>

| <combination-relationship-condition-1>

| <combination-relationship-condition-2>

<relationship-existence-condition> ::=

<entity-instance-identifier>

[(<role-name>)]

<related-to-instance>

[(<role-name>)]

<entity-instance-identifier>

<related-to-instance> ::=

<relationship-instance-identifier>

| <role-instance-identifier>

<equality-attribute-value-condition> ::=

<attribute-value> = <attribute-value>

| <attribute-value> = <user-given-value>

| <user-given-value> = <attribute-value>

User-given-value is terminal.

<attribute-value> ::=

<instance-identifier> <attribute-name>

<comparison-condition> ::=

<value> <comparison-condition-name> <value>

<comparison-condition-name> ::=

= | < | > | <= | =>

<value> ::= <arithmetic-operation> | <simple-value>

<arithmetic-operation> ::=

[<sign>] <arithmetic-term>

{<add-minus> <arithmetic-term>}

<arithmetic-term> ::=

<arithmetic-subterm>

{<time-divide> <arithmetic-subterm>}

<arithmetic-subterm> ::=

<simple-value> | (<arithmetic-operation>)

<sign> ::= + | -

<add-minus> ::= + | -

<time-divide> ::= * | /

<simple-value> ::=

<attribute-value>

| <user-given-value>

| <statistical-operation>

<statistical-operation> ::=

<instance-statistical-operation>

| <attribute-statistical-operation>

<instance-statistical-operation> ::=

COUNT

[UNIQUE]

(<entity-instance-identifier>

[FOR EACH <grouping-values>])

<attribute-statistical-operation> ::=

<attribute-statistical-operator>

[UNIQUE]

(<instance-identifier> <attribute-name>

[FOR EACH <grouping-values>])

<attribute-statistical-operator> ::=

COUNT | MAX | MIN | AVG | SUM

<grouping-values> ::=

<grouping-value> {, <grouping-values>}

<grouping-value> ::=

<entity-instance-identifier>

| <instance-identifier> <attribute-name>

<membership-condition> ::=

<value>

IN

(<list-of-values>)

<list-of-values> ::=

<value> {, <value>}

<list-of-values> ::=

[<instance-clause>]

SELECT <value>

[<where-clause>]

<exists-condition> ::=

EXISTS

(<retrieval-query>)

<retrieval-query> ::=

[<instance-clause>]

<select-clause>

[<report-formatting-clause>]

[<where-clause>]

<union-retrieval-query> ::=

<retrieval-query>

[UNION <union-retrieval-query>]

<isa-relationship-condition> ::=

<entity-instance-identifier> IS_A <entity-type-name>

<entity-equality-condition> ::=

<entity-instance-identifier> = <entity-instance-identifier>

<combination-relationship-condition-1> ::=

<entity-instance-identifier>

<relationship-related>

<entity-instance-identifier>

<relationship-related> ::=

<relationship-type-name>-RELATED

| <role-name>-RELATED

<combination-relationship-condition-2> ::=

<entity-instance-identifier>

<relationship-related>

<number-specification>

<entity-type-name>

<number-specification> ::=

ALL | NO

| [<comparsion-condition-name>] <integer>

<new-type> ::=

NEW ENTITY <entity-type-name>,

KEY ATTRIBUTE <attribute-name>

WITH DATATYPE <datatype-name>

{, KEY ATTRIBUTE <attribute-name>

WITH DATATYPE <datatype-name>}

{, ATTRIBUTE <attribute-name>

WITH DATATYPE <datatype-name>}.

<new-type> ::=

NEW RELATIONSHIP <relationship-type-name>

{, ATTRIBUTE <attribute-name>

WITH DATATYPE <datatype-name>},

ENTITY <entity-type-name> WITH ROLE <role-name>

AND MAPPING <mapping-degrees>,

ENTITY <entity-type-name> WITH ROLE <role-name>

AND MAPPING <mapping-degrees>

{, ENTITY <entity-type-name> WITH ROLE <role-name>

AND MAPPING <mapping-degrees>}.

<mapping-degrees> ::= (<degree>, <degree>)

<degree> ::= integer | *

<new-isa> ::=

NEW IS_A SUPERENTITY = <entity-type-name>,

SUBENTITY = <entity-type-name>

{, <entity-type-name> }.

<new-instance> ::=

NEW <entity-type-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>}.

<type-update> ::=

DELETE ENTITY <entity-type-name>

| DELETE RELATIONSHIP <relationship-type-name>

| DELETE ATTRIBUTE <attribute-name>

OF ENTITY <entity-type-name>

| DELETE ATTRIBUTE <attribute-name>

OF RELATIONSHIP <relationship-type-name>

| ADD ATTRIBUTE <attribute-name> WITH DATATYPE

<datatype-name> TO ENTITY <entity-type-name>

| ADD ATTRIBUTE <attribute-name> WITH DATATYPE

<datatype-name> TO RELATIONSHIP

<relationship-type-name>

<new-instance> ::=

NEW <relationship-type-name>

{, <attribute-name> = <value>},

ENTITY <entity-type-name> WITH ROLE <role-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>},

ENTITY <entity-type-name> WITH ROLE <role-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>}

{, ENTITY <entity-type-name> WITH ROLE <role-name>,

<attribute-name> = <value>

{, <attribute-name> = <value>} }.