Two-Dimensional Specification of Universal Quantification in a Graphical Database Query Language

Kyu-Young Whang, *Senior Member, IEEE*, Ashok Malhotra, Gary H. Sockut, Luanne Burns, and Key-Sun Choi

Abstract — **We propose a technique for specifying universal quantification and existential quantification (combined with negation) in a two-dimensional (graphical) database query language. Unlike other approaches that provide set operators to simulate universal quantification, this technique allows a *direct representation* of universal quantification. We present syntactic constructs for specifying universal and existential quantifications, two-dimensional translation of universal quantification to existential quantification (with negation), and translation of existentially quantified two-dimensional queries to relational queries. The resulting relational queries can be processed directly by many existing database systems. Traditionally, universal quantification has been considered a difficult concept for typical database programmers. We claim that this technique renders universal quantification easy to understand. To substantiate this claim, we provide a simple, easy-to-follow guideline for constructing universally quantified queries. We believe that the direct representation of universal quantification in a two-dimensional language is new and that our technique contributes significantly to the understanding of universal quantification in the context of database query languages.**

Index Terms— **Universal quantification, existential quantification, graphical query languages, databases, relational calculus, entity-relationship model.**

I. Introduction

UNIVERSAL quantification is an important element in relational calculus [2]. Yet it has not been fully integrated in many practical database query languages. There are two possible reasons: 1) in a linear-syntax language, complex syntax is needed to support universal quantification, and 2) universal quantification can be replaced with existential quantification and negation, which many languages provide. Some approaches support universal quantification by using set operators [17], [11]. However, in these approaches the user has to transform a universally quantified query to multiple subqueries connected by set operators. Oftentimes, the transformation is a nontrivial task for average database programmers. SQL [5] supports universal quantification that can be specified in the form, *expression = ALL (subquery)*. However, only very limited cases of universal quantification can be represented in this form.

In this paper we present a simple elegant technique for specifying universal quantification. Our technique employs a two-dimensional representation of queries.[1] Unlike other set-oriented approaches, this technique allows a *direct* representation of universal quantification. We first present syntactic constructs for specifying universal quantification and existential quantification (with negation). We then present an algorithm for transforming automatically a universally quantified query to an existentially quantified query. Next, we present an algorithm to transform an existentially quantified query to a relational calculus query. This sequence of transformations proves that the universally quantified query specified in our two-dimensional language can be easily implemented by using any of many existing relational database systems, provided that it supports negation and existential quantification. (Many database systems support existential quantification implicitly or explicitly. See Section IV for more discussion on this aspect.)

Many two-dimensional query languages have been proposed in the literature [17], [10], [15], [16], [4], [7]. We examine these languages by classifying their features into three categories: the data model, aggregation, and quantification. We pay special attention to aggregation and quantification, because these features require a *scoping operator* to define parts of the query (i.e., *subqueries*) to which they apply. In a linear syntax, the scoping operator is a parenthesis or a keyword. In a two-dimensional syntax, it will be a box or an enclosure. As we discuss in subsequent sections, we use boxes to represent quantifications. Aggregation requires a scoping operator when it appears in certain conditions, as exemplified in [10].

A pioneering work in two-dimensional representation of database queries is Query-by-Example (QBE) [17]. A language based on the relational model, it supports aggregation and existential quantification (with negation). It also supports universal quantification by using a set notation. Due to the lack of the scoping operator (i.e., subqueries cannot be defined), however, ambiguity can arise if aggregation appears in a condition or if quantification involves more than one relation.

Manuscript received November 6, 1989; revised December 6, 1991. Recommended by T. Ichikawa. This work was supported in part by the Korean Ministry of Science and Technology under Contract No. N07450.

K.-Y. Whang and K.-S. Choi are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea.

A. Malhotra, G. H. Sockut, and L. Burns are with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 9106476.

[1] We describe our technique using the entity-relationship model because of its elegance in representing the relationship. Nonetheless, the technique is equally applicable to the relational model where relationships are replaced with join conditions.

CUPID [10] is also based on the relational model and has features similar to QBE's. However, it does not provide the mechanisms for specifying quantification, although it does provide a scoping operator for specifying subqueries involving aggregation. GUIDE [15] is based on the entity-relationship model, but does not support aggregation or quantification. Elmasri and Larson [4] also proposed a language based on the entity-relationship model. It provides set operators and aggregation operators, but does not provide explicit scoping operators. However, it is possible to resolve ambiguity in scoping by rephrasing the queries in English and asking the user to verify them. PICASSO [7] uses the universal relation model [9] as its basis and supports set operators as well as a scoping operator to be used for each maximal object. The scoping operator can be used for aggregation, but not for quantification. Quantification can be handled through set operators, although this aspect was not discussed explicitly in the paper. GQL/ER [16] combines the features of the entity-relationship model and the universal relation model. This language does not support aggregation or quantification. Finally, Ozsoyoglu [11] proposed a linear syntax language called RC/S*. RC/S* is a variation of relational calculus that replaces universal quantification with operations on sets.

Our query language supports aggregation, universal quantification, and existential quantification (with negation). In this paper we concentrate on the facilities for quantifications and do not discuss aggregation in depth. Here, we identify two distinct contributions of this paper. First, we claim that our quantification scheme is easy to use. Traditionally, universal quantification has been considered a difficult concept for typical database programmers. Substantiating this claim, we present a simple and easy guideline for constructing universally quantified queries. This guideline works for most of the commonly encountered queries. Second, we believe that the direct representation (without using set operators) of universal quantification in a two-dimensional language is new and contributes to the understanding of universal quantification in database query languages. The class of universally quantified queries that can be expressed in out language is formally defined in Section V. We believe that it includes most of the queries commonly encountered in practical situations.

The organization of the paper is as follows. Section II briefly introduces our two-dimensional query language. Section III presents the syntactic constructs for composing queries with universal quantification. Similarly, Section IV presents the constructs for existential quantification with negation. Section V formally defines the class of universally quantified queries that we handle and presents the algorithm for transforming a universally quantified query to an existentially quantified query with negation. Section VI presents the algorithm for transforming an existentially quantified query to a relational calculus query. We present a simple guideline for composing universally quantified queries in Section VII and discuss a more complex case in Section VIII. Finally, we conclude the paper in Section IX.

II. A Two-Dimensional Query Language

In this section we briefly introduce our two-dimensional database query language. We present only those features that are relevant for the discussions in this paper. A full description of the language will be presented in a future paper.

A *query* is a specification of conditions according to which entities are selected from among those contained in the database. We define a *schema diagram* as a graph that represents the structure of a database. We use the entity-relationship (ER) model [1] for its basis. A schema diagram consists of three constructs: entity sets, one-to-many (including one-to-one) relationship sets, and many-to-many (including nonbinary) relationship sets. An entity set appears as a rectangular node with the name of the entity set in it. A one-to-many relationship set appears as an arc, with the name of the relationship in the middle. An end of the arc adorned with the symbol "*" represents a cardinality of "many," while an unadorned end represents a cardinality of "one." A many-to-many relationship set or a nonbinary relationship set appears as a rhombus node with the name of the relationship in it. We draw unadorned arcs between the rhombus and the entity sets participating in the relationship.

A *query graph* is a subgraph of the schema diagram, with possibly certain nodes and arcs replicated. In addition, each node of the query graph can have *logical conditions* and projection information associated with it. There is also a global condition box in which complex conditions can be specified. We classify logical conditions into three categories: a *selection condition* that applies to a single node, a *join condition* that applies to a set of nodes, and an *aggregation condition* that involves an aggregation operation. These conditions are specified in an area called a *query box*. For each node, one or more query boxes can be created by clicking the mouse with the cursor positioned on the node. For the purpose of this paper, however, we simply write the condition next to the node without using a query box. Thus, we write a selection condition next to the node representing the entity set to which the condition applies. Similarly, we write a join condition next to any one of the nodes representing the entity sets to which the condition applies. We do not discuss aggregation conditions, because they are beyond the scope of this paper. We select a *projection* attribute by clicking on the attribute name in the query box. Selected projection attributes are shown in reverse video. In this paper, for simplicity and without loss of generality, we assume that all the attributes of the entity set (rather than a subset of the attributes) are projected. We indicate projection by writing the symbol "proj." next to the entity set.

In Fig. 1 we illustrate the use of these constructs by using a simple query. The query states: "List the employees whose salaries are more than one-tenth of the budget of their department and who participate in a project that has more than ten members." The query contains three entity sets; Dept, Emp, and Project; a one-to-many relationship set, employ; and a many-to-many relationship set, Participate. A selection condition is specified for the entity set Project, and a join condition is specified for the entity sets Emp and

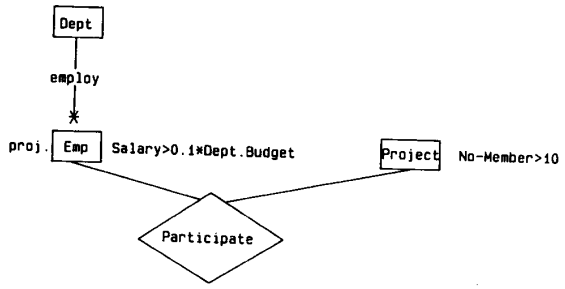


Fig. 1. An example of two-dimensional query.

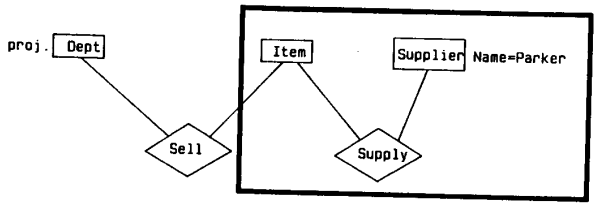


Fig. 2. A query with universal quantification.

Dept. The result of the query is projected from the entity set Emp.

III. Universally Quantified Queries

In this section we present how universally quantified queries are expressed in our two-dimensional query language. Consider the following query: "List the departments that sell all the items supplied by the supplier Parker." In relational calculus, the query is represented in (1).

For convenience, we assume in this section that an entity set or a relationship set is mapped to a relation. In Section VI, we relax this restriction by mapping a one-to-many relationship set to a foreign key without representing it as a separate relation. The query is represented in Fig. 2, where a *universal quantification box (U-box)* drawn with bold lines encloses universally quantified variables: I (for Item), Su (for Supply), and S (for Supplier). Note that the query in this figure is different from the query in Fig. 3, which says, "List the departments such that all the items they sell are supplied by the supplier Parker." In relational calculus this query is represented in (2). In query (1) the phrase "they sell" modifies the noun (items) that is universally quantified, thus composing a noun phrase. In the relational calculus representation, the variables corresponding to this noun phrase are Sl (for Sell) and I (for Item). These variables are universally quantified, because they represent the universally quantified noun and the conditions associated with it. Thus, we enclose Sell and Item in a U-box.

In Figs. 4 and 5 we present two additional examples of universally quantified queries. The schema diagrams in

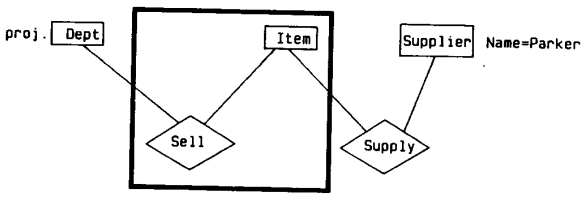


Fig. 3. Another query with universal quantification.

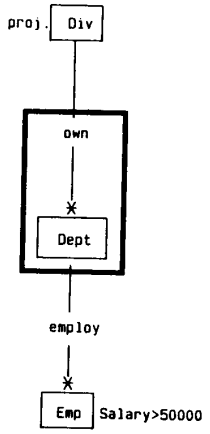


Fig. 4. A universally quantified query with one-to-many relationship sets.

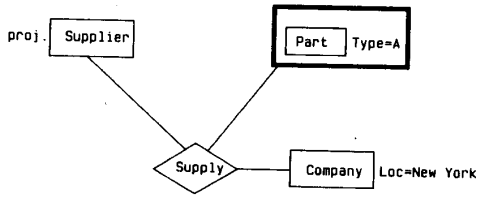


Fig. 5. A universally quantified query with a ternary relationship set.

$$\{T \mid \exists_{DT}(Dept(DT) \wedge (DT = T) \wedge \forall_{I,Su,S}(Item(I) \wedge Supply(Su) \wedge Supplier(S) \wedge I[1] = Su[1] \wedge S[1] = Su[2] \wedge S[2] = \text{Parker} \rightarrow \exists_{Sl}(Sell(Sl) \wedge Sl[1] = DT[1] \wedge Sl[2] = I[1])))\} \quad (1)$$

$$\{T \mid \exists_{DT}(Dept(DT) \wedge (DT = T) \wedge \forall_{I,Sl}(Item(I) \wedge Sell(Sl) \wedge Sl[1] = DT[1] \wedge Sl[2] = I[1] \rightarrow \exists_{S,Su}(Supplier(S) \wedge Supply(Su) \wedge Su[1] = I[1] \wedge Su[2] = S[1] \wedge S[2] = \text{Parker})))\} \quad (2)$$

$$\{T|\exists_{DV}(Div(DV) \wedge (DV = T) \wedge \forall_{DT,O}(Dept(DT) \wedge own(O) \wedge O[1] = DV[1] \wedge O[2] = DT[1] \rightarrow \exists_{E,H}(Emp(E) \wedge employ(H) \wedge H[1] = DT[1] \wedge H[2] = E[1] \wedge E[2] > 50000)))\} \quad (3)$$

these examples contain one-to-many and ternary relationship sets, whereas those in Figs. 2 and 3 contain many-to-many relationship sets. We use these schema diagrams throughout the paper for illustrative purposes.

The query in Fig. 4 states: "List the divisions where all the departments they own have at least one employee whose salary is greater than 50 000 dollars." In relational calculus, it is represented by (3). Note that the universally quantified noun phrase is, "the departments they own." Thus, we enclose the entity set Dept and the relationship own in the U-box.

The query in Fig. 5 states: "List the suppliers that supply all the parts of type A to companies located in New York." The universally quantified noun phrase is, "the parts of type A." Thus, the U-box encloses the entity set Item with the condition Type = A.

IV. Existential Quantification and Negation

We discuss in this section how existential quantification is specified in our two-dimensional query language. Existential quantification is implicitly supported by many relational query languages. For example, consider the SQL query, "SELECT dept.* FROM dept, emp WHERE dept.dno = emp.dno AND emp.salary > 50000." This query can be represented in relational calculus as follows:

$$\{T|\exists_{DT,E}(dept(DT) \wedge (DT = T) \wedge emp(E) \wedge DT[1] = E[3] \wedge E[2] > 50000)\}.$$

Note that the existential quantification on DT and E is implicit in the SQL query. In these query languages, however, existential quantification is made explicit when negation is involved. For example, consider a SQL query, "SELECT * FROM dept X WHERE NOT EXISTS (SELECT * FROM emp WHERE X.dno = emp.dno AND emp.salary > 50000)." This query returns the dept tuples only when there is no employee in the dept who earns more than 50 000 dollars. The query is represented in relational calculus as follows:

$$\{T \mid \exists_{DT}(dept(DT) \wedge (DT = T) \wedge \neg\exists_{E}(emp(E) \wedge DT[1] = E[3] \wedge E[2] > 50000))\}. \quad (4)$$

SQL supports explicit existential quantification even without negation. For example, consider the query,

```
SELECT   supnum, partnum, shiptime, onorder
FROM     parts1
WHERE    EXISTS   (SELECT   *
                   FROM     parts2
                   WHERE    supnum = suppnum AND
                            partnum = parttnum)
```

This type of query has an explicit existential quantifier, but it can be easily translated to a join query without explicit existential quantification [6]. For example, the query can be translated as follows:

```
SELECT   supnum, partnum, shiptime, onorder
FROM     parts1, parts2
WHERE    supnum = suppnum AND
         partnum = parttnum
```

With an existential quantifier, associated is a scope within which the quantification is effective. For example, in the query,

$$\{T|\exists_{V1}[C(V1) \wedge (V1 = T) \wedge \neg\exists_{V2,V3}[A(V2) \wedge B(V3) \wedge V1[1] = V2[3] \wedge V2[1] = V3[2]]]\}$$

the scopes of existential quantification are enclosed by the brackets. In a two-dimensional language we represent a scope by a two-dimensional bracket; i.e., a *box*. In our language, we allow use of explicit existential quantification[2] only when it is used in conjunction with negation. Thus, a box for negated existential quantification (*NE-box*) represents NOT EXISTS (a subquery) in the SQL syntax. The use of this NE-box (drawn with broken lines) is illustrated in examples 1 and 2.

Example 1

Consider the query, "List the departments where none of the employees in the department has a salary of more than 50 000 dollars." In our two-dimensional query language, the query is expressed as in Fig. 6.

Example 2

Consider the query, "List the divisions that do not own a department where none of the employees has a salary of more than 50 000 dollars." This example shows nested existential quantification with negation. The query is shown in Fig. 7.

V. Translation of a Universally Quantified Two-Dimensional Query to an Existentially Quantified Two-Dimensional Query

In this section we describe how we translate automatically a universally quantified query that the user composes into an existentially quantified query. We present a translation algorithm and show its correctness.

Universally quantified queries in our language are in the following general form:

$$\{T|\exists_{V1}(P(V1,T) \wedge \forall_{V2}(Q(V1,V2) \rightarrow \exists_{V3}(R(V1,V2,V3))))\} \quad (5)$$

where P, Q, and R are formulas, $V1, V2$, and $V3$ are sets of tuple variables, and T is a set of free variables.[3] Free variables represent the tuples that appear in the result of the query; i.e.,

[2] If there is no quantified variable within the NE-box, it represents simply NOT (a condition) in the SQL syntax.

[3] We do not allow free variables inside universal quantification (i.e., projection inside the U-box) for the safety of the query. The safety is briefly discussed in the appendix.

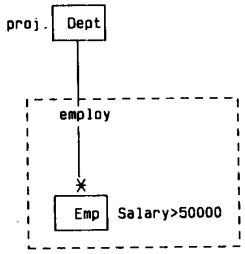


Fig. 6. A query with an NE-box.

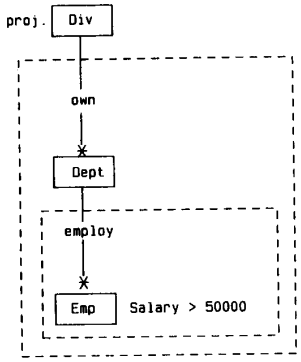


Fig. 7. A query with nested existential quantification with negation.

the tuples that are projected. For example, the query in Fig. 2 was expressed as in (1).

We define a *scope* to be a set of entity sets, relationship sets, and conditions. In (5), a scope corresponds to a set of tuple variables and formulas. We define three different scopes. For convenience, we define an entity set, a relationship set, or a logical condition as an *element*.

1) **Scope 1**: This includes the entity sets that are projected, plus any other elements that are not included in scopes 2 and 3. In (5), scope 1 includes the tuple variables in $V1$ and T, plus the formula $P(V1, T)$.
2) **Scope 2**: This includes the elements enclosed by the U-box (i.e., universally quantified elements). In (5), scope 2 includes the tuple variables in $V2$, plus the formula $Q(V1, V2)$.
3) **Scope 3**: Consider a reduced graph where the projected entity sets are eliminated. Scope 3 includes the elements that are directly or indirectly connected to those in scope 2 in the reduced graph. In (5), scope 3 includes the tuple variables in $V3$, plus the formula $R(V1, V2, V3)$.

Example 3 illustrates how we identify different scopes.

Example 3

In Fig. 4 the entity set Div belongs to scope 1, the entity set

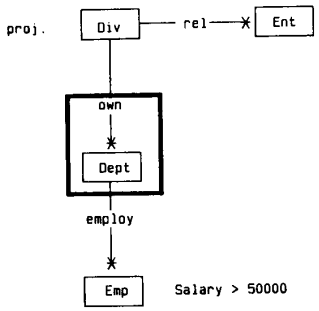


Fig. 8. An example query for identifying scopes.

Dept and the relationship set own belong to scope 2, and the entity set Emp, the relationship set employ, and the condition Salary $>$ 50 000 belong to scope 3. Suppose the query is slightly modified as in Fig. 8. Then the relationship set rel and the entity set Ent also belong to scope 1. Note that they do not belong to scope 3.

We now present the algorithm for translating a universally quantified query to an existentially quantified query.

Algorithm 1 (U-to-E Translation)

1) Put an NE-box around all the elements in scopes 2 and 3.
2) Put an NE-box around all the elements in scope 3. Note that this box is completely enclosed by the NE-box in step 1. If no element exists in scope 3, create an element with the value of "true" and put an NE-box around it.

Correctness of the Translation: Algorithm 1 essentially reflects the following equality:

$$\forall(A \rightarrow \exists B) \equiv \forall(\neg A \vee \exists B) \equiv \neg\exists(A \wedge \neg\exists B). \quad (6)$$

Using this equality, (5) can be transformed as follows:

$$\{T|\exists_{V1}(P(V1,T) \wedge \neg\exists_{V2}(Q(V1,V2) \wedge \neg\exists_{V3} (R(V1,V2,V3))))\} \quad (7)$$

which indicates that an NE-box is applied to all the elements in scopes 2 and 3. In addition, another NE-box is applied to all the elements in scope 3. This proves the correctness of the translation algorithm.

We illustrate this translation in examples 4 and 5.

Example 4

The query in Fig. 4 is transformed from (3) as indicated in (8), which corresponds to the equivalent existential query in Fig. 7.

$$\{T|\exists_{DV}(Div(DV) \wedge (DV = T) \wedge \neg\exists_{DT,O}(Dept(DT) \wedge own(O) \wedge O[1] = DV[1] \wedge O[2] = DT[1] \wedge \neg\exists_{E,H}(Emp(E) \wedge employ(H) \wedge H[1] = DT[1] \wedge H[2] = E[1] \wedge E[2] > 50000))))\} \quad (8)$$

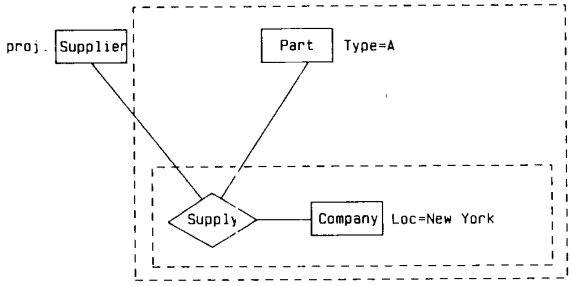


Fig. 9. An existentially quantified query with a ternary relationship set.

Example 5

The query in Fig. 5 is translated into an existentially quantified query in Fig. 9. The query states: "List the suppliers for which there are no parts of type A that they do not supply to companies located in New York."

VI. Translation of an Existentially Quantified Two-Dimensional Query to a Relational Calculus Query

In Section V we discussed how a universally quantified query can be translated to an existentially quantified query with negation. In this section we present an algorithm for translating an existentially quantified query to a tuple relational calculus query. Using this transformation, a universally quantified query can be easily implemented by using existing relational database systems that support only existential quantification with negation.

To translate the query, we first need to translate the schema according to the underlying data model. The translation of an entity-relationship model schema to a relational model schema is well known [13]. Here, we adopt a translation technique using system-generated identifiers; i.e., *surrogates*. We briefly review basic techniques for schema translation, and then present query translation.

A. Schema Translation

For schema translation, we introduce two types of relations: *entity relations* and *relationship relations*. First, for each entity set, we create a relation scheme (entity relation) that consists of all the attributes of the entity set plus a surrogate attribute and foreign key attributes. The surrogate uniquely determines the tuple. A foreign key attribute is added for each one-to-many relationship set in which this entity set is on the many-side of the relationship. The foreign key attribute is the surrogate attribute of the relation on the one-side of the relationship. We treat a one-to-one relationship set like a one-to-many relationship set, adding a foreign key attribute to one of the entity sets. We treat the entity set so chosen as if it were the one on the many-side of the one-to-many relationship. Second, for each many-to-many or nonbinary relationship set, we create a relation scheme (relationship relation) that consists of the surrogate attributes of the entity sets participating in the relationship.

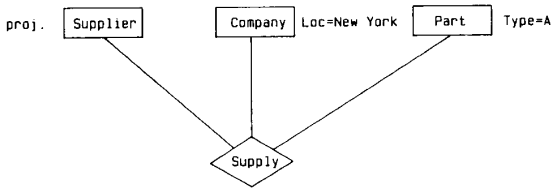


Fig. 10. A query with a ternary relationship set.

B. Query Translation

We now present the query translation algorithm. We describe it in two steps: first we consider queries without NE-boxes, then we consider queries containing NE-boxes.

1) Queries Without NE-Boxes:

Algorithm 2 (Simple-Translation)

Input: A two-dimensional query without NE-boxes

Output: A tuple relational calculus query

Constructing a relational query in this case is straightforward; thus, we only sketch the algorithm. First, we construct an atom of the form $R(V)$ for each entity set, many-to-many relationship set, or nonbinary relationship set, where R is the name of the relation corresponding to the entity set, many-to-many relationship set, or nonbinary relationship set, and V is the tuple variable. We say that the formula $R(V)$ *defines* the tuple variable V. Second, we construct a formula of the form, $V1[A1] = V2[A2]$, for each one-to-many relationship set, where $V1, V2$ are tuple variables for the relations on either side of the relationship, $A1$ is the positional index for the surrogate attribute of the relation on the one-side of the relationship, and $A2$ is the positional index for the foreign key attribute of the relation on the many-side of the relationship. Similarly, we construct two equality formulas for each many-to-many relationship set, equating the tuple variable for each of the two entity relations, and the tuple variable for the relationship relation via the surrogate and foreign key attributes. For a nonbinary relationship set involving n entity sets, we construct n equality formulas. Third, we construct an appropriate formula for each condition specified. Fourth, all these formulas are logically ANDed, and the result is quantified by $\exists_{all\ tuple\ variables\ defined\ in\ the\ formulas}$. Last, we equate each tuple variable to be projected with a free variable, T_i. Example 6 illustrates this algorithm.

Example 6

Consider the query in Fig. 10: "List the suppliers who supply a part of type A to a company located in New York." The corresponding relational query is as follows:

$$\{T|\exists_{S,C,P,Su}(Suppliers(S) \wedge Company(C) \wedge Part(P) \wedge Supply(Su) \wedge (S = T) \wedge S[1] = Su[1] \wedge C[1] = Su[2] \wedge P[1] = Su[3] \wedge C[2] = \text{New York} \wedge P[2] = A)\}.$$

Here, S, C, P, and Su are tuple variables, $S[1]$, $C[1]$, $P[1]$ represent surrogate attributes of relations Supplier, Company,

and Part, and $Su[1]$, $Su[2]$, $Su[3]$ represent foreign key attributes of the relation Supply. The first three equality formulas come from the ternary relationship set Supply, and the last two come from the conditions for the entity sets Company and Part.

2) Queries with NE-Boxes: We use the notation Q_{outer} to represent the part of the query Q that is outside the outermost NE-boxes within Q. We call the part of the query within an NE-box as Q_{inner}. If a relationship name appears within the NE-box, the relationship set is part of Q_{inner}, even if it may be connected to an entity set outside the NE-box. The parameter n is the number of outermost NE-boxes in Q.

Algorithm 3 (Translation)

Input: A two-dimensional query Q with zero or more NE-boxes

Output: A tuple relational calculus query

Begin

G = Simple-Translation(Q_{outer})

For each outermost $NE\text{-}box_i$ of Q

$F_i = \neg\exists_{\text{all tuple variables defined in } (Q_{inner,i})_{outer}}$ Translation $(Q_{inner,i})$

Output $= G \wedge F_1 \wedge F_2 \wedge \cdots \wedge F_n$

End

In a formula F_i we do not generate existential quantification if $(Q_{inner,i})_{outer}$ has no tuple variables defined in it. In this case, the subquery simply becomes a condition. Note that algorithm 3 is called recursively for the subqueries within NE-boxes. In translating a subquery, all the tuple variables defined outside its scope can be referenced. For example, in example 7 the tuple variable DT is referenced within the innermost subquery.

Example 7

Consider the query in Fig. 7. The translated tuple relational calculus query is as follows:

$$\{T|\exists_{DV}(Div(DV) \wedge (DV = T) \wedge \neg\exists_{DT}(Dept(DT) \wedge DV[1] = DT[2] \wedge \neg\exists_{E}(Emp(E) \wedge DT[1] = E[3] \wedge E[2] > 50000)))\}$$

where $DV[1]$ is the surrogate of Div, $DT[1]$ and $DT[2]$ are the surrogate and the foreign key of Dept, $E[3]$ is the foreign key of Emp, and $E[2]$ is the Salary attribute.

VII. A Guideline for Composing a Universally Quantified Query

Writing a universally quantified query is often not intuitively obvious. Thus we present a simple guideline for composing a universally quantified query. We present this guideline for the following reasons:

1) The concept of universal quantification is more complex than most other concepts in a query language. We believe that this complexity is inherent and is not specific to a query language.
2) Even when a query does not involve universal quantification, the flexibility of a natural language (e.g., words like "all") can give the impression that the query does involve universal quantifications, as we explain in the guideline below.

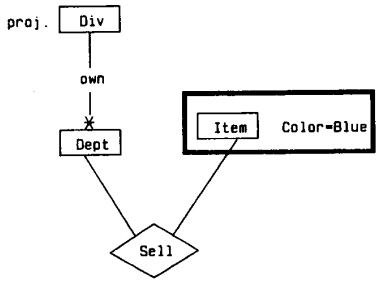


Fig. 11. A query becomes ambiguous if implicit projection is allowed.

Guideline (U-query)

1) Rewrite the query in English by eliminating any occurrences of "all" (or "each," "every," "at least") where the elimination does not alter the meaning of the query. For example, consider the query, "List all the divisions where all departments they own have an employee whose salary is greater than 50 000 dollars." In this query, the first "all" can be removed without altering the meaning of the query, but the second "all" cannot. Thus the reduced query is, "List the divisions where all the departments they own have an employee whose salary is greater than 50 000 dollars."
2) Identify the noun phrase that is quantified by the word "all." A noun phrase includes the noun quantified by the word "all" and the phrase (if any) that modifies the quantified noun. In the example above, the noun phrase is "the departments they own."
3) Compose a query as if the word "all" were replaced by an indefinite article. In the example above we construct the query, "List the divisions where a department they own has an employee whose salary is greater than 50 000 dollars." Construct the corresponding query graph.
4) Put a U-box around the entity sets, relationship sets, and logical conditions that correspond to the noun phrase identified in step 2. Thus the example query is represented as in Fig. 4.

VIII. Queries with Implicit Projection

In Section VII we discussed a basic guideline for composing a universally quantified query. In this section we discuss a case that needs special attention.

A query may have different meanings depending on whether certain elements belong to scope 1 or scope 3. For example, consider the query in Fig. 11. The query states: "List the divisions such that for each blue item there is a department in the division that sells the item." Note that the query $(Query1)$ is different from the following query $(Query2)$: "List the divisions owning a department that sells all the blue items." Query1 qualifies a division if the departments it owns collectively cover all the blue items, while Query2 requires that a single department cover all the blue items.

$$\{T|\exists_{V1}(P(V1) \wedge \forall_{V2}(Q(V1,V2,T) \rightarrow \exists_{V3}(R(V1,V2,V3))))\} = \{T|\exists_{V1}(P(V1) \wedge \forall_{V2}(\neg Q(V1,V2,T) \vee \exists_{V3}(R(V1,V2,V3))))\} \quad \text{(A1)}$$

$$\{T|\exists_{V1}(P(V1) \wedge \forall_{V2}(Q(V1,V2) \rightarrow \exists_{V3}(R(V1,V2,V3,T))))\} = \{T|\exists_{V1}(P(V1) \wedge \forall_{V2}(\neg Q(V1,V2) \vee \exists_{V3}(R(V1,V2,V3,T))))\} \quad \text{(A2)}$$

The scoping rules in Section V interpret the query as Query1, since the entity set Dept and the relationship set own are contained in scope 3. To interpret the query as Query2 we have to assume that there is an *implicit projection* on the entity set Dept, which will put the entity set Dept and the relationship set own in scope 1.

Implicit projection makes the query ambiguous. To disambiguate it we have two alternatives: 1) to provide a syntactic construct to distinguish scope 1 from scope 3 explicitly, or 2) to disallow implicit projection by requiring that the entity set Dept be projected as well. We chose the latter option for the simplicity of the scoping rules and for ease of use. We believe that this requirement is reasonable, since in Query2 the user would quite likely be interested in having in the query result the specific department that covers all the blue items.

IX. Summary

We have presented a technique for specifying universal quantification and existential quantification (with negation) in a two-dimensional database query language. Our technique allows a direct representation of universal quantification in a two-dimensional manner without using set operators. We have also presented a two-dimensional algorithm to transform a universally quantified query to a query with existential quantification and negation, and showed its correctness. Finally, we have presented an algorithm to transform an existentially quantified query to a relational calculus query. This transformation allows the universally quantified queries to be easily processed by many existing database management systems that support existential quantification with negation.

Universal quantification has been considered a difficult concept in database query languages. We claim that our technique renders the concept easy to understand. Substantiating this claim, we have presented a simple, easy-to-follow guideline for constructing queries with universal quantification.

We believe that the technique of directly representing universal quantification without using sets in a two-dimensional query language is new and that its ease of use will contribute to bringing the concept of universal quantification more into the world of database query languages.

Appendix
Safety of Queries

In this appendix we briefly discuss the safety issue, which has been discussed extensively in [3], [14], [8], [12]. We omit detailed proofs and discussions on safety, since they are beyond the scope of the paper.

A query (or a formula) is *safe* if it has a finite result. A class of formulas called *evaluable formulas* defined by Demolombe [3] and refined by Van Gelder and Topor [14] is by far the largest known decidable subset of safe formulas. A class of *allowed* formulas is a subset of evaluable formulas whose intermediate results are finite as well. Thus allowed queries ensure safe execution to produce the results.

It can be shown that any relational calculus formula in the form of (5) is not evaluable if a free variable appears in scopes 2 or 3; i.e., in formulas Q and R. For example, queries (9) and (10) are not evaluable (and in this case unsafe). In (A1) the universally quantified subformula is satisfied regardless of T values if the second disjunct is satisfied. It is also satisfied for all (and possibly an infinite number of) T values that do not satisfy the formula $Q(V1,V2,T)$. (According to the formalism in [14], $gen(T, \neg Q(V1,V2,T))$ and $gen(T, R(V1,V2,V3))$ fail.) Similarly, in (A2) the universally quantified subformula is satisfied regardless of T values if the first disjunct is satisfied. (According to [14], $gen(T, \neg Q(V1,V2))$ fails.) Thus both queries can produce an infinite number of values for T and therefore are unsafe.

It can also be shown that a formula in the form of (5) is allowed (and therefore safe) if it satisfies the following conditions:

1) For every quantified subformula, each quantified variable appears in a base formula that is not contained in a nested quantification. A *base formula* is an atomic formula whose predicate symbol represents a database relation.
2) The subformulas P, Q, and R are conjuncts of base formulas, atomic formulas that are conditions, and universally quantified formulas of the form in (5) that satisfy condition 2 recursively.

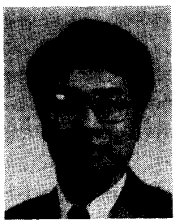
We note that the definition of the allowed class of formulas presented in [14] does not have to be extended for equality, since all the variables appear in base formulas that form conjunctions in the formulas P, Q, and R.

The two-dimensional query language we present in this paper, when mapped to the relational model, satisfies the two properties with one exception. This exception is the treatment of the set of free variables T; i.e., the variables in T do not appear in base formulas. Nevertheless, since they are always equated to variables in the base formulas, they do not affect the safety of the query. Hence, we can treat the queries as if they did not include these free variables. Therefore, the queries in our query language are allowed, and safe.

References

[1] P. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, Mar. 1976.

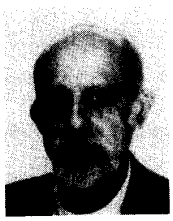
[2] E. F. Codd, "Relational completeness of data base sublanguages," in *Data Base Systems* (*Proc. 6th Courant Computer Science Symp.*, May 1971), R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 65–98.
[3] R. Demolombe, "Syntactical characterization of a subset of domain-independent formulas," *ONERACERT, Tech. Rep.*, 1982.
[4] R. A. Elmarsi and J. A. Larson, "A graphical query facility for ER databases," in *Proc. 4th IEEE Int. Conf. on Entity-Relationship Approach,* 1985, pp. 236–245.
[5] "IBM database 2 reference," 3rd ed., IBM, Mar. 1986.
[6] W. Kim, "On optimizing an SQL-like nested query," *ACM Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, Sept. 1982.
[7] H. J. Kim *et al.*, "PICASSO: a graphical query language," *Software Pract. Experience,* vol. 18, no. 3, pp. 169–203, Mar. 1988.
[8] R. Krishnamurthy and C. Zaniolo, "Safety and optimization of horn clause queries," in *Proc. Foundations of Deductive Databases and Logic Program.* (Washington, DC), Aug. 1986, J. Minker, Ed.
[9] D. Maier and J. D. Ullman, "Maximal objects and the semantics of universal relation databases," *ACM Trans. Database Syst.*, vol. 8, no. 1, pp. 1–14, Mar. 1983.
[10] N. McDonald and M. Stonebraker, "CUPID—the friendly query language," in *Proc. ACM Pacific Conf.* (San Francisco), 1975, pp. 127–131.
[11] G. Ozsoyoglu and H. Wong, "A relational calculus with set operators: its safety, and equivalent graphical languages," *Dept. Comp. Eng. and Sci.,* Case Western Reserve Univ., Cleveland, OH, Tech. Rep., 1987.
[12] J. C. Shepherdson, "Negation in logic programming," in *Proc. Foundations of Deductive Databases and Logic Program.* (Washington, DC), 1986, J. Minker, Ed.
[13] J. D. Ullman, *Principles of Database Systems,* 2nd ed. Rockville, MD: Comput. Sci. Press, 1982.
[14] A. Van Gelder and R. W. Topor, "Safety and correct translation of relational calculus formulas," in *Proc. ACM Symp. on Principles of Database Syst.,* 1987, pp. 313–326.
[15] H. K. T. Wong and I. Kuo, "GUIDE: graphical user interface for database exploration," in *Proc. 8th Int. Conf. on Very Large Data Bases* (Mexico City), Sept. 1982, pp. 22–32.
[16] Z.-Q. Zhang and A. O. Mendelzon, "A graphical query language for entity-relationship databases," in *Entity Relationship Approach to Software Engineering*, C. G. Davis *et al.*, Eds. New York: Elsevier Science, 1983, pp. 441–448.
[17] M. M. Zloof, "Query by example," in *Proc. Nat. Comput. Conf.*, 1975, pp. 431–438.



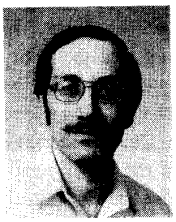
Kyu-Young Whang (S'73–M'75–SM'88) graduated (Summa Cum Laude) from Seoul National University in 1973, and received the M.S. degrees from the Korea Advanced Institute of Science and Technology (KAIST) in 1975, and Stanford University in 1982. He earned the Ph.D. degree from Stanford University in 1984.

From 1975 to 1978, he was a Senior Research Engineer at the Agency for Defense Development, Korea. From 1983 to 1991, he was a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he performed various research projects in databases, office systems (including Office-by-Example), and expert systems. He is now an Associate Professor in the Computer Science Department of KAIST, and the Director of the Database and Knowledge Engineering Laboratory of the Center for Artificial Intelligence Research. His research interests encompass multimedia databases, object-oriented databases, engineering databases, expert systems, and office systems.

Dr. Whang served as an IEEE Distinguished Visitor from 1989 to 1990, received the Best Paper Award from the 6th IEEE International Conference on Data Engineering, served the 5th IEEE International Conference on Data Engineering as a Program Co-Chair, and has served on program committees of numerous international conferences, including ACM SIGMOD and VLDB. He twice received the External Honor Recognition from IBM. He is on the editorial boards of *VLDB Journal, Distributed and Parallel Databases: An International Journal,* and the *IEEE Data Engineering Bulletin*. He is a member of the ACM.



Ashok Malhotra received the Ph.D. degree from the Massachusetts Institute of Technology. He has been a Research Staff Member in the Computer Science Department at the IBM Thomas J. Watson Research Center since 1975. He has managed several projects related to entity-relationship systems: an entity-relationship database, an entity-relationship language and application development system, and a visual interface to entity-relationship databases. His current research interests include application development technology, graphical interfaces for application development, and object-oriented systems and databases. Prior to joining IBM he worked for several years as a Management Consultant, and has made contributions to a methodology for designing application systems and databases based on the business needs of the company.



Gary H. Sockut received the B.S. degree in applied mathematics from Brown University, the M.S. degree in electrical engineering from the Massachusetts Institute of Technology, and the Ph.D. degree in applied mathematics from Harvard University.

He is an Advisory Programmer at the IBM Santa Teresa Laboratory, and earlier worked at BGS Systems, the National Institute of Standards and Technology, and the IBM T. J. Watson Research Center. His main areas of research interest are in database management, office systems, and operating systems.

Dr. Sockut is a member of the IEEE Computer Society and ACM.

Luanne Burns received the M.S. degree in computer science from Columbia University, where her main concentration was in artificial intelligence and expert database systems. She is currently pursuing the Ph.D. degree in cognitive science, also at Columbia University.

She has been with IBM at the Thomas J. Watson Research Center since 1984. Her primary research interests have been entity-relationship database systems, user interfaces, and most recently, graphical interface design and development for relational database systems.

Key-Sun Choi received the B.S. degree in mathematics from Seoul National University, and the M.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology.

He is an Assistant Professor at the Korea Advanced Institute of Science and Technology. Previously, he worked at the Hankuk University of Foreign Studies, and the C&C Information Research Laboratory of NEC in Japan. His main areas of research interest are in natural language processing, information retrieval, office systems, and computational logic.

Dr. Choi is a member of the IEEE Computer Society, ACM, ACL, AAAI, KISS, and IPSJ.