Data & Knowledge Engineering 11 (1993) 171–202
North-Holland



DATAK 179

# GRAQULA: A graphical query language for entity-relationship or relational databases

Gary H. Sockut[a]*, Luanne M. Burns[b], Ashok Malhotra[b] and Kyu-Young Whang[c]

[a]*IBM Santa Teresa Laboratory, P.O. Box 49023, San Jose, CA 95161-9023, USA*
[b]*IBM Research Division, T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA*
[c]*Computer Science Department and Center for Artificial Intelligence Research, Korea Advanced Institute of Science and Technology, 373-1 Koo-Sung Dong, Yoo-Sung Ku, Daejeon, South Korea*

*Abstract*

GRAQULA is a graphical language for querying and updating a database. One version of GRAQULA provides a user interface for the entity-relationship data model, and another version (with almost identical syntax) provides a user interface for the relational model. Each version is relationally complete, and each depicts relationships (or expected joins) graphically. GRAQULA provides logical operations (e.g. negation) on graphical objects; these operations have user-specified scopes, allow nesting, and can involve existential or universal quantification. Aggregates (e.g. average) also have user-specified scopes. Queries can invoke other queries, and users and queries can pass parameters to queries. The design reflects a specified set of goals, including expressive power, consistency, and limitation of required memorization.

*Keywords.* Database query languages; graphical interfaces; scope of logical operations; universal quantification; existential quantification; human factors; entity-relationship data model; relational data model.

## 1. Introduction

The intended purposes of many textual languages for databases include interactive queries, but end users often find such languages hard to use. Difficulties involve memorization of syntax, memorization of schemas (database definitions), complexity in recognizing relationships, and inconvenience in specifying and modifying queries; these difficulties can lead to users' errors. We conducted an experiment [6] involving 19 undergraduates who were *not* computer scientists or database experts; they had introductory experience with a relational database system on a personal computer. We showed them textual definitions of tables, and we asked them to write relational queries (based on our English queries) in a textual query language of their choice. Nine subjects spontaneously drew diagrams with arcs for relationships during their work, and these subjects scored considerably higher than the other subjects (for correctness and speed of writing), especially for complex queries (e.g. for queries with 3 or more joins, 82% correct vs. 32%). Such observations motivate the use of a graphical interface that shows relationships. We believe that such an interface can increase

* *Corresponding author.* Email: GHS@STLVM22.VNET.IBM.COM, fax: 408-463-3114.

comprehension, reduce required memorization and analysis, and thus improve correctness and speed.

This paper describes GRAQULA (GRAphical QUery LAnguage), a graphical language for querying and updating a database. Most of our discussions about queries also apply to updates. We designed GRAQULA for a workstation with a mouse, but the database need not reside at the workstation. One version of GRAQULA provides a user interface for the entity-relationship (E–R) data model [9]; another version provides a user interface for the relational model [10]. Most of our discussions describe the E–R version, but the same constructs apply to the relational version; we will describe the differences between our E–R and relational diagrams. The intended set of users includes database specialists and nonspecialists who issue unpredictable queries, e.g. accountants and statisticians; they see a database's organization into entity and relationship types. We do not define application-specific interfaces that are more appropriate for database nonspecialists who issue more predictable queries, e.g. bank tellers; they do not see the organization.

We have implemented part of GRAQULA's capabilities in a relational query language (GARP) [8], and we implemented two E–R browsers [5, 7] before designing GRAQULA. These implementations and a related implementation for E–R schema definition and query (RMGraph) [18] have graphical features resembling those of full GRAQULA, e.g. database diagrams that users can tailor and a framework of windows, action bars, and menus. We will describe the implementation status and users' experience. We will also define a mapping from GRAQULA's E–R queries into relational queries, as a way to process E–R queries. GRAQULA is not part of RMGraph or any other IBM product.

GRAQULA's design reflects a set of goals (many of which come from well-known principles in human factors, e.g. Shneiderman's guidelines [27]):

- Sufficient expressive power and functions, including relational completeness; rarity of restrictions
- Ability to learn a database's structure without extensive time, effort, or memorization
- Ease of use, including learning, remembering, writing, and reading the language's constructs
- Consistency, predictability, and naturalness (in both syntax and function); rarity of exceptions
- Simplicity of features and conciseness of work required of the user, especially for frequent or conceptually simple tasks; ability to ignore language features that are irrelevant to a particular query
- A high probability that users will write error-free queries
- Clarity of definition; lack of ambiguity
- Ability to modify queries to form new queries incrementally.

GRAQULA's principal contribution is its *combination* of the features listed below. We sketch the features here and describe them in more detail later.

(1) GRAQULA provides a graphical interface that uses popular, convenient techniques (e.g. windows, action bars, and menus) for manipulating the graphical objects that form a query.

(2) As we will demonstrate, the operations in the relational version and the operations on entities in the E–R version are relationally complete; each version includes ad hoc joins and the other required functions. Of course, the E–R version also supports relationships.

(3) Users can specify the nature and scope of logical operations (conjunction, disjunction, and negation) on sets of graphical objects, including objects involving existential quantification, via *frames* (rectangles that enclose scopes). Nesting of frames graphically

shows nesting of operations and scopes. Expressive power requires specification of scopes and nesting of logical operations. No other graphical language, to our knowledge, provides such thorough capabilities for logical operations on graphical objects.

(4) Frames also support *direct* specification of implication with an arbitrary consequent and universal quantification; users need not specify it indirectly via set operations or nested negation. We allow nesting, and we allow any number of tuple variables per scope of universal quantification. This paper also provides a guideline to ease the writing of universally quantified graphical queries, based on English queries. Thomas [30] describes an experiment (not involving GRAQULA) showing that universal quantification can be a difficult concept for users; we believe that our direct specification and guideline can help. Again, no other graphical language, to our knowledge, provides such thorough capabilities.

(5) Frames also support user-specified scopes for aggregates (e.g. average). Expressive power requires specification of scopes of aggregates.

(6) Queries can involve cyclic patterns of relationships; users need not restrict queries to trees.

(7) GRAQULA includes updates.

(8) In the graphical depictions of schemas and queries, the relational version draws arcs for expected joins (with automatic drawing in schemas when referential integrity constraints exist), and the E–R version draws arcs for relationships. Our experiment showed the value of such arcs in improving users' correctness and speed in writing queries.

(9) A construct (called an *execution regulator*) lets queries invoke other queries, and users and queries can pass parameters to queries.

(10) We define a mapping from E–R queries into relational queries, for clarity of definition and as an implementation technique.

(11) GRAQULA provides conveniences like ordering of results, saving of results, and saving of queries.

Here we compare GRAQULA with other graphical query languages (in the research literature) for E–R or relational databases. For each language, we note which of features 2–8 above are not fully present. Many languages appear in the surveys by Kim [19] and Batini et al. [3].

CUPID [24] is a relational language. The user writes a query by choosing tables and placing icons for operations next to tables' column names. CUPID minimizes typing and uses graphical notations for almost all features, including arithmetic expressions and scopes of aggregates. It prompts the user with ways to correct invalid queries. Missing features include logical operations with scopes, implication with universal quantification, and arcs for expected joins in schemas.

Office-by-Example (OBE) [32] includes a relational language resembling its predecessor, Query-by-Example (QBE) [39]. The user chooses tables and types operators and conditions under column names; example elements (symbols) denote connections (e.g. joins). A directory lists the available tables. Negation (of existential quantification) lacks nesting and always has a scope of one table. Missing features include graphical disjunction, direct specification of implication with universal quantification, scopes of aggregates, and arcs for expected joins, although a version of QBE for a hierarchical data model [40] draws arcs for relationships. Summary-Table-by-Example [25] includes scopes of aggregates.

GUIDE [36] is an E–R language. A user chooses a subset of a schema diagram and specifies conditions, which appear textually at the bottom. GUIDE emphasize exploration of a database; it provides zooming, a directory, and commands to find the relevant parts of a

large schema. Missing features include ad hoc joins, logical operations with scopes, implication with universal quantification, scopes of aggregates, cycles of relationships in queries, and updates.

In gql/ER [38], the user adds conditions in OBE-like forms that appear separately from the E–R diagram. The user can request a default connection between user-chosen entity types. The paper does not mention logical operations with scopes, implication with universal quantification, scopes of aggregates, or updates.

In the language of Elmasri and Larson [14], a query uses a hierarchical view of an E–R schema, with a user-identified root. Attributes can migrate among a query's entity types to reflect the hierarchy. The paper discusses translation from graphics into English. It does not mention ad hoc joins, logical operations with scopes, direct specification of implication with universal quantification, scopes of aggregates, cycles of relationships in queries, or updates.

The relational language of Embley et al. [15] has a corresponding algebraic textual language and a formal description. It allows definition of connectors between tables, and a connector to a temporary constant table provides a selection. The language lacks logical operations with scopes, implication with universal quantification, scopes of aggregates, and updates.

The relational language PICASSO [20] displays tables and their columns, and the user draws conditions next to the columns. Joins are graphical, with arcs that connect a comparison operator to two columns. A multiple-step method specifies scopes of aggregates. PICASSO's universal relation data model automatically exhibits most of the expected joins; PICASSO does not draw arcs for the remaining expected joins between the universal relation and itself. Other missing features include logical operations with scopes, implication with universal quantification, and updates.

Pasta-3 [21] is an E–R language with inheritance and deductive rules. It provides a directory to find a schema's relevant parts, and the user can let the system connect user-chosen entity types. The user specifies conditions under attribute names. Pasta-3 has a multiple-step method for disjunction, and its negation lacks nesting and user-specified scopes. Its implication with universal quantification does not support nesting or more than one tuple variable immediately inside a scope of universal quantification. It uses text for scopes of aggregates, and it uses replication and joins to provide the effect of cycles of relationships in queries. It does not draw arcs for relationships in queries.

The language of Czejdo et al. [12] emphasizes operations to edit an E–R diagram (and its displayed attribute names) to form a query. Conditions appear next to attributes. The language has a formal description, and it supports subtypes of entity types. It lacks logical operations with scopes and implication with universal quantification.

QBD* [2] is an E–R language with transitive closure. It has a corresponding textual language and a formal description. After choosing parts of the schema, the user can request the display of related parts. The user draws conditions on attribute lists. QBD* lacks logical operations with scopes, implication with universal quantification, and updates.

Finally, DFQL [37] is a relational algebra-based language using a data flow paradigm. It is relationally complete and includes aggregates with scopes. We do not explicitly compare it with other specific features of GRAQULA, since its paradigm differs so much from paradigms of GRAQULA and other languages.

Our description of GRAQULA begins with an overview. We then describe schema diagrams, query diagrams, and additional contents that a user builds in queries. We describe the facilities to control queries, and we demonstrate relational completeness. We describe implementations (including some actual screens) and discuss users' experience. Finally, we summarize our work. An introductory knowledge of database management should suffice for

understanding the paper. An earlier version of this paper [28] covers more details of relationships, aggregates, updates, and a few other topics.

## 2. Overview of GRAQULA

We envision GRAQULA as a query facility in a set of graphical facilities for databases. Other facilities can define schemas, restructure schemas, browse, and generate reports. We have implemented browsing [7] and part of GRAQULA's capabilities [8], and we have sketched a design of the other facilities [22].

Information appears in windows on a workstation. For GRAQULA and most of the other facilities above, the user starts by identifying the database (and thus the schema) during creation of a *schema window*, which contains a *schema diagram*. For brevity, we will discuss only schemas, not subschemas, but use of GRAQULA from a subschema is identical. The GRAQULA user views the entity and relationship types in the schema diagram and then creates any number of *query windows*; each can contain a user-specified *query diagram*, which signals the participation of entity and relationship types in a query. In a query window, the user adds details, e.g. *projections* (specifications of data to list) and *conditions*. A condition is anything whose instantiation returns 'TRUE' or 'FALSE', e.g. 'salary > 50000.' We also execute a query in a query window, and GRAQULA displays the result as a table in a *result window* or sends it to a user-specified file or printer. The user can switch between the windows' activities at any time. Some variations on these activities, e.g. retrieval of a saved query, are possible.

For later reference, *Fig. 1* shows the generic hierarchy of types of graphical objects; we describe the objects briefly here and in more detail later. We mentioned most of the windows above; a SQL window displays a generated query in SQL [1]. A schema element (entity or relationship type) is part of the schema diagram in the schema window. A query element appears in a query window and involves a condition. An image (entity or relationship image) is part of a query diagram and signals participation of an entity or relationship type in that query. A comparison (e.g. a join) can be graphical or textual. A condition box specifies a combination of logical operations on textual conditions. A frame (parentheses, negation, implication, or consequent frame) specifies a logical operation and quantification for query elements. A regulator (duplicate, order, destination, comment, set operation, or execution regulator) appears in a query window and controls an aspect of the query's format or execution. Images suffice for most queries that we consider common; only more complex queries need other query elements or regulators.

Some types of objects have an action bar (a list of actions, each invoked by mouse-

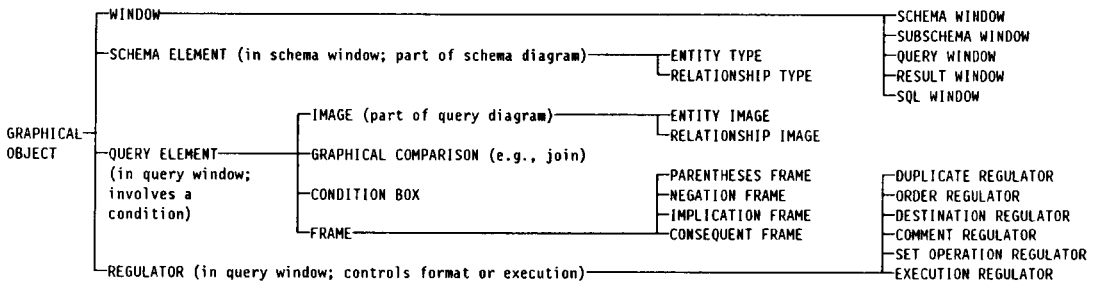



Fig. 1. Generic hierarchy of types of graphical objects.

clicking) and scroll bars. To create most text, both typing and clicking are available; for the user's convenience, we interpret upper and lower case identically in names and keywords. For brevity, we will concentrate on queries' contents, not on the well-known techniques [4] to manipulate those contents (e.g. dragging and zooming).

## 3. Data models and diagrams in GRAQULA

Here we describe GRAQULA's E–R data model, schema diagrams, query diagrams, and the relational version of GRAQULA.

### *3.1. GRAQULA's entity-relationship data model*

The E–R model has become popular for modeling information, and it has many proposed versions. GRAQULA's version, which we describe now, is similar but not identical to the version in Repository Manager [26]. We use *schema element* as a generic term for entity type and relationship type. For brevity, we often omit the words 'type', 'instance', and 'image' (a query's use of a schema element) after the word 'entity' or 'relationship' when the meaning is obvious from the context.

- An *entity type* represents a collection of things, e.g. employees. It has a name and attributes; an attribute (e.g. salary) has a name and a data type. Each entity type has a key, which is one or more attributes.
- A *relationship type* represents an association, e.g. departments' employment of employees. It has a name, e.g. 'EMPLOYS,' and optionally has an inverse name, e.g. 'WORKS_IN.' Relationships are binary; each connects two schema elements (entities and/or relationships), which we call its *ends*. A relationship's definition labels one end as the *source* and the other as the *target*; the labels are necessary to prevent ambiguity when the two ends are the same element. Each relationship type has an *instance control* of one-to-one, one-to-many, or many-to-many. In a one-to-many relationship type, either end (source or target) can be the many-end; for brevity, we do not use the additional term 'many-to-one.'

We define a mapping from E–R into relational for GRAQULA schemas and queries. One purpose is to define our constructs clearly; however, an E–R user who is not trying to map between data models need not be aware of the mapping. Another purpose is to show feasibility of implementation on relational systems; however, we do not require this particular mapping or even a relational system as GRAQULA's implementation. Earlier papers on universal quantification [33, 34] describe the mapping more formally.

We begin by defining three concepts that the mapping uses. A one-to-many or one-to-one relationship's *domestic end* is the many-end for a one-to-many relationship or the target end for a one-to-one relationship; the *foreign end* is the other end. Many-to-many relationships do not use those two concepts. Any schema element's *identifier* is the set of attributes whose values uniquely identify the element's instances:

- An entity's identifier includes its key attributes.
- A many-to-many relationship's identifier includes both ends' identifiers.
- A one-to-many or one-to-one relationship's identifier is the domestic end's identifier.

Now we define the relational mapping of schemas; later we map queries:

- An entity and its attributes (key and nonkey) map into a table and columns (primary key and nonkey).
- A many-to-many relationship and its identifier map into a table and primary key columns.
- For a one-to-many or one-to-one relationship, the foreign end's identifier maps into more

columns (a foreign key). They become part of the table that the domestic end uses; this table-choosing algorithm recurses if the domestic end is a one-to-many or one-to-one relationship.

In an environment where users can issue relational updates directly (i.e. without using E–R updates), the relational schema should include some integrity constraints [23], which we omit here for brevity.

### *3.2. Schema diagrams*

A schema diagram depicts the schema elements. An entity type appears as a node containing the entity name; a relationship type appears as an arc (or, in this paper's figures, line segments) with an adjacent relationship name. Each end has an 'm' (for many) or a '1,' and the target has an arrowhead. Schema and query diagrams need not be planar. Users can tailor the positions of schema elements; query diagrams have the same flexibility. *Figure 2* shows the schema that our example queries use. For example, CONTAINS is a one-to-many relationship between the DIVISION and DEPARTMENT entities.

In the schema window, the user can expand entity types to show scrollable lists of attributes and their data types. In our schema, the DIVISION entity has NAME (key), BUDGET, and YEAR_FORMED attributes; DEPARTMENT has NAME (key) and BUDGET; EMPLOYEE has NAME (key), SALARY, and YEAR_HIRED; TITLE has TITLE (key); and SKILL has SKILL (key). A real database might use some different keys (e.g. employee *number*), but we kept our example simple.

Here we show the relational mapping of our example relationships, including one possible set of names for tables and foreign key columns. The CONTAINS relationship type maps into the DIVISION_NAME column in the DEPARTMENT table. EMPLOYS, PAYS, and HAS_TITLE map into the E_DEPARTMENT_NAME, P_DEPARTMENT_NAME, and TITLE columns in the EMPLOYEE table. HAS_SKILL maps into the HAS_SKILL table with NAME and SKILL columns.

### *3.3. Query diagrams*

Even with a large *schema*, most *queries* use few entity and relationship types. For example, a query to list the Toy department's employees uses only EMPLOYEE, EMPLOYS, and DEPARTMENT. A query diagram contains *images* (pictures) of one or more entity types (as nodes) and zero or more relationship types (as arcs), to signal their participation in the query. Diagram A in *Fig. 3* is a diagram (not a complete query) for listing the Toy department's employees. A query diagram contains several images of a particular schema element if the query uses that element in several ways. For example, to list the employees in Ann Smith's department, we use EMPLOYEE to list employees and to specify Ann Smith, as in diagram B. An image's name is the schema element's name, with

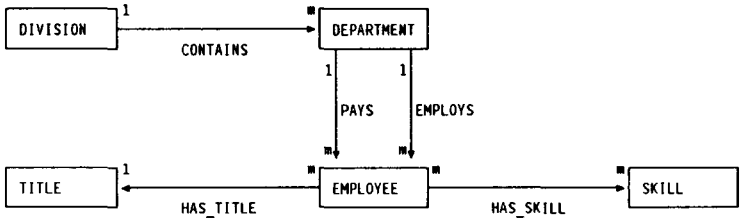



Fig. 2. A schema diagram.

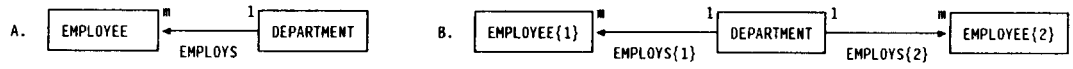



Fig. 3. Query diagrams.

system-generated suffixes like '{1}' and '{2}' if the schema element has several images. We show later how to specify projections, conditions, and updates.

A query window's query diagram is initially empty; dragging a schema element from the schema window to a query window adds an image to the query diagram (without changing the schema diagram). In adding a relationship image, GRAQULA chooses its source image (and then its target) as follows:

- If no images of the source exist yet, GRAQULA adds one. If the missing source is a relationship, the strategy for addition recurses, as if the user had dragged the source relationship from the schema.
- If exactly one image of the source exists, it becomes the relationship image's source.
- If several images of the source exist, GRAQULA highlights them, prompts the user to choose one (or to cancel the addition of the relationship image), and removes the highlighting after the choice.

Dragging an image from a query window deletes the image; deletion recurses if a deleted image is an end of a relationship image. To track progress, GRAQULA highlights the schema window's schema elements that have images in the active query window; GUIDE and GARP also highlight a schema's chosen subset. To help construct the rare queries that use the majority of the schema elements, an action creates a query diagram by copying the entire schema diagram. Another action deletes an entire query diagram.

The user can also add images via a directory; its sections list entity types, relationship types, and attributes. For each attribute, the attribute section lists the entity types containing that attribute. An action (in the schema window or a query window) displays the directory. The directory's first entry is initially blank; the user can type a name there. Choosing an entry (possibly the first) can add or delete an image or can scroll the schema or query diagram to center it on that type or image; this is especially helpful for a large schema. OBE, GUIDE, Pasta-3, and GARP also use directories to find a schema's relevant parts.

Suppose that a query diagram contains two or more sets of entity images, but no joins or relationship images connect those sets. If the user requests execution, GRAQULA requests confirmation and displays the relevant part of the relationship section of the directory; the user optionally adds relationship images to connect the entity images. This feature deters inadvertent specification of a Cartesian product (which we do allow) when the user wants a join [21]. Similarly, in gql/ER, Pasta-3, and QBD*, the user can let the system connect user-chosen entity images. GARP informs the user when no connections exist.

We now introduce the relational mapping of queries; we will expand the mapping as we define more GRAQULA constructs. A query maps into a SELECT statement in SQL [1].

- An entity image in a query maps into use of a table in a FROM clause in SQL. More formally, in relational calculus [11], it maps into an atomic formula of the form '$R(s)$,' where $R$ is a table and $s$ is a tuple variable. For images that are not inside frames (described later), the FROM clause is part of the SELECT statement in SQL, and we bind the images' tuple variables to existential quantifiers (as in '$\exists s \exists t$') whose scope (range of effect) is the entire query in calculus. We later generalize the quantification to allow smaller scopes (subqueries) and universal quantification.

- A one-to-one or one-to-many relationship image maps into a join between the foreign key columns in the domestic end's table and the primary key columns in the foreign end's table.[1]
- A many-to-many relationship image maps into use of a table and two joins.
- Projections and conditions map into relational projections and conditions.

Using separate diagrams for a schema and a query (instead of a single diagram that highlights schema elements that participate in a query) has two advantages. One is reduced cognitive complexity via obvious omission of nonparticipants. The other advantage is ease of expressing queries with several uses for a schema element. If relationships or graphical logical operators involve an element with several uses, then with one diagram, duplication of elements or of OBE-like query rows could produce clutter and confusion.

### *3.4. The relational version of GRAQULA*

Most of our discussions describe GRAQULA in terms of a user interface for the E–R model (with a possible underlying mapping into relational queries), but the same constructs apply to a version of GRAQULA with a user interface for the relational model. A relational schema diagram contains nodes for tables and arcs for *referential integrity* [13] constraints (with an arrowhead at the end containing the foreign key). In addition, or for a relational system with no referential integrity, a user (perhaps the schema designer) *optionally* tells GRAQULA which joins are expected to occur frequently, and GRAQULA draws arcs for them in the schema diagram. A query diagram contains nodes for uses of tables and arcs for expected joins; GRAQULA also allows ad hoc joins. Similarly, Wiederhold and Elmasri [35] use a notation with arcs for relationships. Besides interactive queries, another application for the relational version of GRAQULA is generation of SQL queries for execution outside GRAQULA. A user of the relational version can also display a generated SQL query in a *SQL window*.

## 4. Contents of queries

*Query elements* (e.g. a query diagram's images) are graphical objects that involve conditions. After specifying a query diagram, the user can expand images (e.g. to show attributes); add other query elements; modify the diagram at any time; add projections, conditions, and updates in expanded images; and add expressions (e.g. aggregates), logical operations, and quantification.

### *4.1. Expanded images*

For each image, the user can switch between a compact format (the initial format) and an *expanded* format. For example, if we expand all three images in query diagram A in *Fig. 3*, we produce the diagram in *Fig. 4*. Expanded entity images include rows, a heading for rows, and scroll bars. We use 'row' to mean horizontal text, not a tuple of data in a table. A row in an expanded entity image contains an attribute name (e.g. 'SALARY'), space for a condition's operator (e.g. '>'), and space for a condition's right operand (most commonly a simple value). The '(AND)' at the right shows the image's logical operation, which we

[1] GRAQULA is definable with either two- or three-valued logic. For simplicity, this paper uses two-valued logic, and we treat SQL as if it used two-valued logic. With SQL's three-valued logic, a one-to-one or one-to-many relationship image maps into a join and a test that the foreign key columns are not null; this makes negation intuitive.

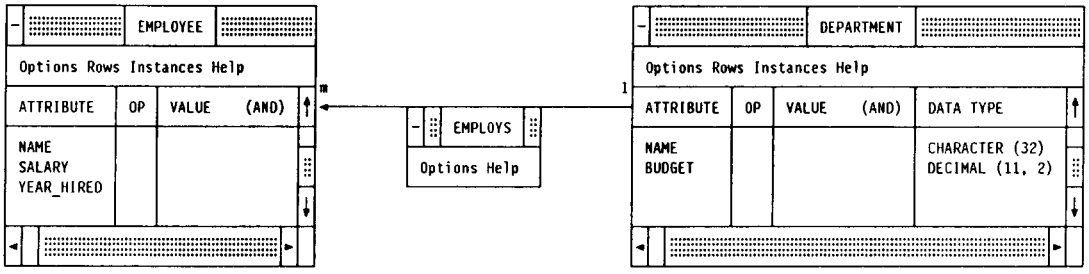



Fig. 4. A query diagram with expanded images.

discuss later. GRAQULA highlights key attributes' names. We discuss the images' action bars and the DEPARTMENT image's DATA TYPE column below.

Besides help, an expanded entity image's action bar includes these actions:

- The Options action creates a pull-down menu with entries for Rename, Expression, Update, and Data type; each entry switches between absence (the initial setting) and presence of a column. In a RENAME column, the user can type alternative names for any of the attributes. We explain EXPRESSION and UPDATE columns later. A DATA TYPE column shows each attribute's data type, as in the DEPARTMENT image. The menu for a relationship image's Options action contains just Update.
- The Rows action's menu copies, reorders, adds, or deletes rows, e.g. to specify many conditions or expressions. An added row does not represent an attribute, but any row can contain an expression.
- The Instances action's menu shows the current set of values in the database for all attributes, as in GARP, or for a chosen attribute, as in the IBM Data Interpretation System (DIS) [17]; the user can then use those values (or other values) in conditions in a query.

To save space in this paper, later figures omit the action and scroll bars.

We define a related textual construct that we use later: an *attribute specification* is a reference to an attribute. In the most general case, it contains an entity image name, a period, and an attribute name. However, an attribute name alone suffices when it appears in the expanded image of an entity that includes that attribute or when the attribute name is unique among the query's attributes.

### 4.2. Projection

We use *projection* to mean specification of data to include in a query's result. Clicking an attribute name in an expanded entity image switches between projection and no projection (the initial setting) of the attribute. Projected attributes' names appear in reverse video. Clicking the image's name (which can also appear in reverse video) switches projection of *all* attributes. Our first complete query, at the left of *Fig. 5*, lists each employee's name and


Fig. 5. A query and its result.

salary; a corresponding SQL query is 'SELECT DISTINCT NAME, SALARY FROM EMPLOYEE.' By default, the execution creates and displays a result window to show the result, as in the right part of *Fig. 5.* The figure omits the widow's action and scroll bars. If several images have projections, the one result includes all projected attributes; i.e. we calculate the images' Cartesian product (or, if specified, their join).

The user's steps to create this query are

(1) viewing the schema diagram in the schema window,
(2) dragging to create a query diagram (containing only an EMPLOYEE image) in the query window,
(3) clicking to expand the image,
(4) clicking to project NAME and SALARY in the image, and
(5) clicking the query window's execution action to execute the query and create the result window.

For several options, a query window has *regulators*, which users can ignore for most queries. Actions switch between hiding (the initial setting) and displaying regulators. We discuss four of the regulators now:

- The *duplicate regulator* specifies removal (the initial setting) or retention of the result's duplicate rows.
- The *order regulator* specifies the order of appearance of the result's rows. For example, 'SALARY, NAME' means a major sort on SALARY and a minor sort on NAME. Specifications of any number of projected attributes (from any number of entity images) can appear. 'ASCENDING' (or 'ASC') or 'DESCENDING' (or 'DESC') optionally follows each attribute specification; the default is ascending. An empty regulator (the initial setting) implies no user-specified order.
- The *destination regulator* specifies (before query execution) the result window's destination. The choices are 'SCREEN' (the initial setting), 'FILE name,' and 'PRINTER name.' If the choice is 'SCREEN,' an action in the result window (after execution) can copy the result to a file or printer.
- The *comment regulator* contains a comment to document the query; it is initially empty. We will describe how to save queries between GRAQULA sessions.

### *4.3. Conditions and expressions*

The user limits a query's result through conditions. Each type of query element involves a *graphical condition*; for example, images represent joins and/or uses of tables. The user can add any number of *textual conditions* inside expanded entity images and condition boxes. A textual condition is a textual comparison (e.g. a selection) or any of various tests (e.g. for a null value). In an expanded entity image, a textual condition on an attribute involves an operator in the OP entry of the attribute's row and a right operand in the VALUE entry. A right operand with a blank operator implies that the operator is '=,' and blanks in both entries mean lack of a condition. An expanded entity image represents two graphical conditions:

(1) use of a table, and
(2) the conjunction or disjunction of any textual conditions inside.

By default, an expanded entity image's second graphical condition is the conjunction of its textual conditions, and a query's condition is the conjunction of its query elements' graphical conditions; we discuss disjunction later. An expanded relationship image contains no textual conditions.

A *textual comparison* (e.g. a selection) uses a comparison operator (e.g. '>='). A *selection* compares an attribute and a constant; a constant is a number, quoted string, or unquoted

hyphen (null). A *join* compares attributes of different entity images. A *restriction* [11] compares attributes of the same entity image. For example, the query in *Fig. 6* lists the name and salary of each employee whose salary exceeds 50000 (a selection) and whose year of hiring equals the Research division's year of formation (a join).

A condition's right operand can be an expression, which can involve attribute specifications, constants, aggregates, parentheses, arithmetic operators, and string operators. For projecting an expression or using it as a condition's left operand, an expanded entity image can have an EXPRESSION column. For example, the query in *Fig. 7* lists the name and twice the salary of Edward Jones. Any projection and any condition in a row apply to the expression if the row has a nonblank EXPRESSION entry; they apply to the attribute if the row has a blank EXPRESSION entry or if the image has no EXPRESSION column.

The 'IS' operator signals any of several types of *tests* (textual conditions); the right operand determines which type. The possible right operands (optionally preceded by 'NOT') are 'NULL,' 'BETWEEN expression, expression,' 'IN expression, . . . , expression' (a member of the set of values), and 'LIKE expression' (as in SQL's pattern matching).

GRAQULA also supports *graphical comparisons*. For example, the query in *Fig. 8* lists the names of the departments whose budgets exceed the Research division's budget. The 'L' and 'R' indicate which entity images supply the left and right operands. If such an image is expanded, and the operand's row is not scrolled off the image, then the comparison's arc connects to the row in the image, as in the language of Embley et al., PICASSO, GARP, and DIS [17]; otherwise it connects to any part of the image. In the relational version of GRAQULA, this also applies to arcs in schema and query diagrams.

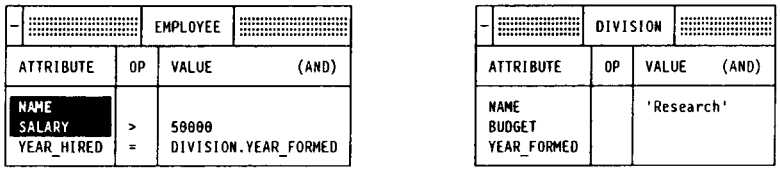



Fig. 6. Selections and a join.

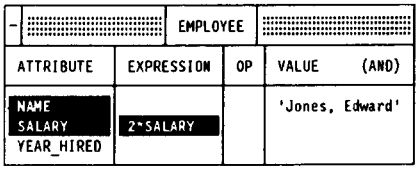



Fig. 7. An expression.

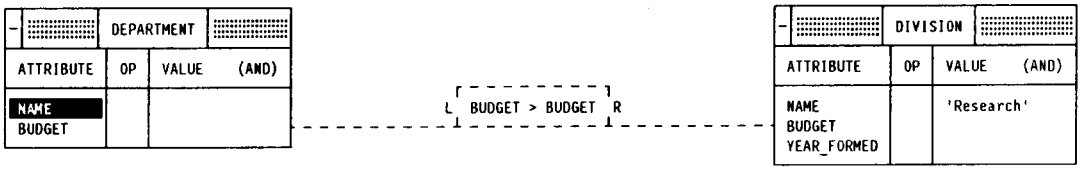



Fig. 8. A graphical comparison.

### 4.4. Relationship images

*Figure 9* shows queries with relationship images. Query A lists the names of the Toy department's employees. Images can form a cycle (nonhierarchical pattern), as in query B, which lists the name of each employee who works in the department that pays that employee. Query C uses two EMPLOYEE images and two EMPLOYS images to list the names of the employees in Ann Smith's department.

### 4.5. Logical operations for textual conditions

The user switches between conjunction and disjunction of an expanded entity image's textual conditions by clicking the displayed '(AND)' (the initial setting) or '(OR)'. For more complex combinations of conditions, the user creates any number of *condition boxes*. A condition box, as in *Fig. 10*, contains a graphical tree; each nonleaf node is a conjunction or disjunction, optionally negated. Each leaf node can be a textual condition (like 'SALARY > 50000'), optionally negated. Trees are horizontal, because a vertical tree with several collinear long textual conditions would be much wider. Our figure omits the action and scroll bars. CUPID and Pasta-3 also use trees. A tree should help the user understand a large or deeply nested combination. To accommodate users who prefer textual logical operators (perhaps to type a small combination quickly), a leaf can contain a combination of ANDs,

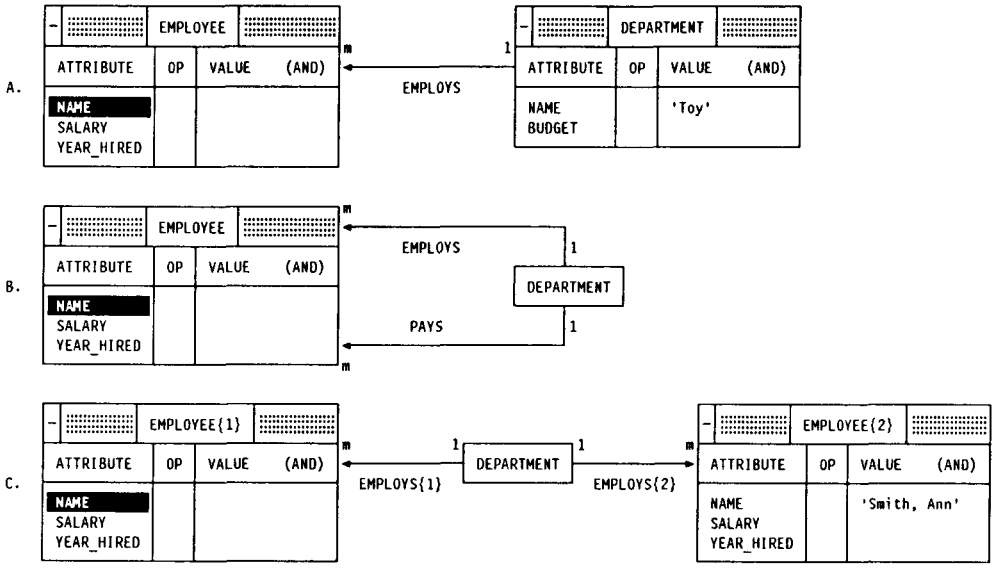



Fig. 9. Queries with relationship images.

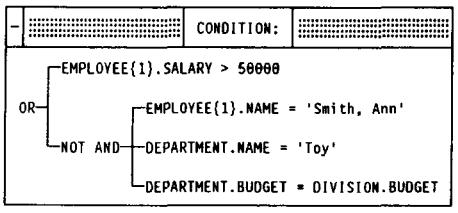



Fig. 10. A condition box.

ORs, and NOTs of textual conditions and parentheses; here a tree with one node suffices, and the condition box resembles OBE's.

### 4.6. *Logical operations and quantification for query elements*

We have shown the constructs for logical operations (conjunction, disjunction, and negation) for *textual* conditions. We now discuss the constructs for logical operations and quantification for *query elements*.

We begin by discussing an operation's *scope* (the set of operands). Many textual languages delimit scopes via parentheses. For examples, 'NOT(A = B) AND C = D' and 'NOT(A = B AND C = D)' have different meanings; the most general case requires parentheses.

To specify the nature and scope of a logical operation for query elements, GRAQULA uses a *frame* (a rectangle enclosing a scope). The operations are identity (with conjunction or disjunction), negation (with conjunction or disjunction), and implication. To nest operations, we nest frames. We say that a query element (possibly a frame) is at the *outermost level* if it is not inside any frame, and it is *immediately* inside a frame if it is inside that frame but not inside any more deeply nested frame. Besides specifying an operation, a frame has other important functions. It determines the type of quantification (existential or universal) for the tuple variables for the entity and many-to-many relationship images (if any) immediately inside the frame, and it limits the scope of that quantification and of any aggregates.

This paper's figures use repetition of a character, e.g. '¬,' for a frame's sides; possible alternative implementations include colors, line styles, and fill patterns. GRAQULA has four types of frames. We list here each frame's full name (and abbreviated name below), logical operation, English word that appears in the upper right corner, character for drawing the sides, and type of quantification:

| *Name* | *Logical operation* | *Word in corner* | *Sides* | *Quantification* |
|---|---|---|---|---|
| ● *Parentheses frame* (*P-frame*) | identity (with conjunction or disjunction | 'AND' or 'OR' | '(' | existential |
| ● *Negation frame* (*N-frame*) | negation (with conjunction or disjunction) | 'AND' or 'OR' | '¬' | existential |
| ● *Implication frame* (*I-frame*) | implication | 'IMPLIES' | '*' | universal |
| ● *Consequent frame* (*C-frame*) | implication's consequent (with conjunction or disjunction) | 'AND' or 'OR' | ':' | existential |

In each P-, N-, or C-frame, the user can click the word to switch between 'AND' (the initial setting) and 'OR.' First we will discuss P- and N-frames and some issues that apply to all four types; then we will discuss I- and C-frames. An appendix discusses considerations that led to our design of frames.

#### 4.6.1 *Parentheses and negation frames*

A P-frame means '(. . .)' in a textual language and 'it is true that . . .' in English; the ellipsis represents the scope, with 'AND' or 'OR' separating any operands. An N-frame means 'NOT(. . .)' and 'it is false that ....' Before defining the division into operands, we show example queries that use P- and N-frames.

We start with simple examples. The N-frames with 'AND' in *Fig. 11* each have one operand and thus represent simply negation, not negated conjunction:

(A) List the names of the departments that pay an employee with a salary over 50000 and do

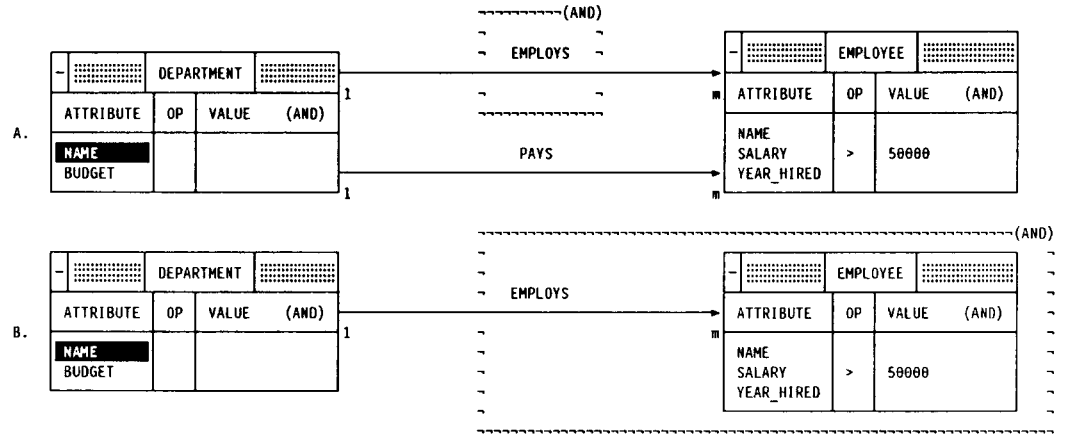



Fig. 11. Negation frames with one operand.

not employ that employee. An alternative wording is '. . . 50000 for whom it is false that the department employs that employee.' A corresponding SQL query is:

```
SELECT DISTINCT D.NAME FROM DEPARTMENT D, EMPLOYEE E
  WHERE E.SALARY > 50000 AND E.P_DEPARTMENT_NAME = D.NAME
    AND NOT(E.E_DEPARTMENT_NAME = D.NAME)
```

Removing the frame would remove 'not' or change 'false' to 'true' in the English.

(B) List the names of the departments that do not employ any employees with a salary over 50000. An alternative wording is '. . . departments for which it is false that there is an employee with a salary over 50000 whom the department employs.' Typically, a query element belongs inside a frame if the element depicts a word that appears *only* inside the corresponding phrase in the English query. For example, 'employ(s)' and 'employee(s)' appear only inside the negated phrase, but 'department(s)' appears outside. However, we believe that the general problem of translation from natural language into a query language (graphical or textual) has not been completely solved.

*Figure 12* shows examples of queries that use frames with two operands:

(A) List the names of the employees whom the Toy department employs or pays. We use a disjunctive P-frame.

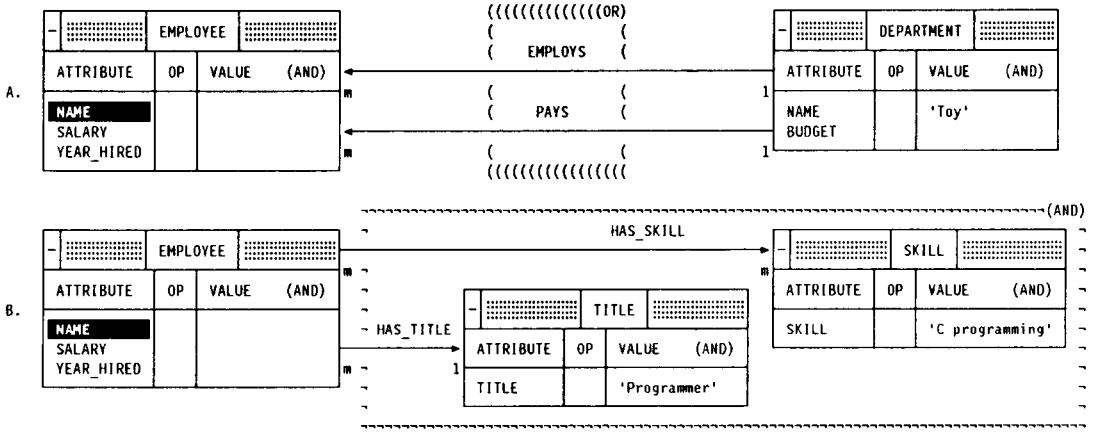



Fig. 12. Frames with two operands.

(B) List the names of the employees for whom it is false that the employee has a title of Programmer and has a skill of C programming. We use a conjunctive N-frame.

To explain the division of a P- or N-frame's scope into operands, we first define the concept of *connection* of query elements. A graphical comparison, relationship image, expanded entity image, or condition box connects with all images it references.[2] A frame connects with an element outside the frame if at least one element inside the frame connects with that outside element. Connection is reflexive, symmetric, and transitive; we can speak of a set of connected elements. We also define another concept; two elements inside a frame *connect inside* the frame if they can meet the definition of connection without using transitivity through any elements outside the frame. For example, in query B in *Fig. 12*, TITLE and HAS_TITLE are a set of elements that connect inside the frame. SKILL and HAS_SKILL are another such set. Those two sets do *not* connect inside the frame, since their connection requires transitivity through EMPLOYEE.

Now we define a P- or N-frame's operands. Each set of elements that are immediately inside the frame and connect inside the frame forms an operand of that frame. Thus each frame in *Fig. 11* has one operand, while each in *Fig. 12* has two. *Within* an operand that contains several elements that are immediately inside the frame, we use the conjunction of those elements' conditions (inside the appropriate scope of quantification, explained shortly), even if the frame is disjunctive. This produces the desired intuitive meanings of the operands, e.g. 'has a title of Programmer' in query B in *Fig. 12.* Adding a nested P-frame around several operands of an outer frame would merge them into one operand of the outer frame. We expect that a typical frame will have one or two operands.

The syntax that signals the introduction of a quantified tuple variable inside a frame is identical to the syntax that signals it outside, i.e. an entity or many-to-many relationship image. If an operand of a P- or N-frame contains any such images that are immediately inside the frame, then the operand immediately contains, and is a scope of, existential quantification. For example, the frame's one operand in query B in *Fig. 11* and each of the two operands in query B in *Fig. 12* are scopes of quantification.

The relational mapping of such an operand uses 'EXISTS subquery' in SQL or existential quantification in calculus. The subquery is '(SELECT 1 FROM table . . .)'; 'table' occurs (with separating commas) for each entity or many-to-many relationship image immediately inside the operand. For example, a SQL query for query B in *Fig. 12* is:

```
SELECT DISTINCT NAME FROM EMPLOYEE E WHERE NOT(
  EXISTS (SELECT 1 FROM TITLE T
    WHERE T.TITLE = 'Programmer' AND T.TITLE = E.TITLE)
  AND EXISTS (SELECT 1 FROM SKILL S, HAS_SKILL H
    WHERE S.SKILL = 'C programming' AND S.SKILL = H.SKILL
    AND E.NAME = H.NAME))
```

If an operand contains no quantification, as in query A in *Fig. 11*, we do not use 'EXISTS' subquery' or quantification. A P- or N-frame's meaning (i.e. '(. . .)' or 'NOT(. . .)') covers frames that contain quantification and frames that do not, with no need for any exceptional cases of a frame's meaning.

A scope can contain another scope. The queries in *Fig. 13* use nested negation frames:

(A) List the names of the departments that employ an employee for whom it is false that there is a skill other than sales that the employee does not have. Each frame has one operand.

(B) List the names of the departments that do not employ any employees who do not have a

[2] We say that a graphical comparison or relationship image *references* the images at its ends; an expanded entity image or condition box references the entity images of the attributes that it mentions.

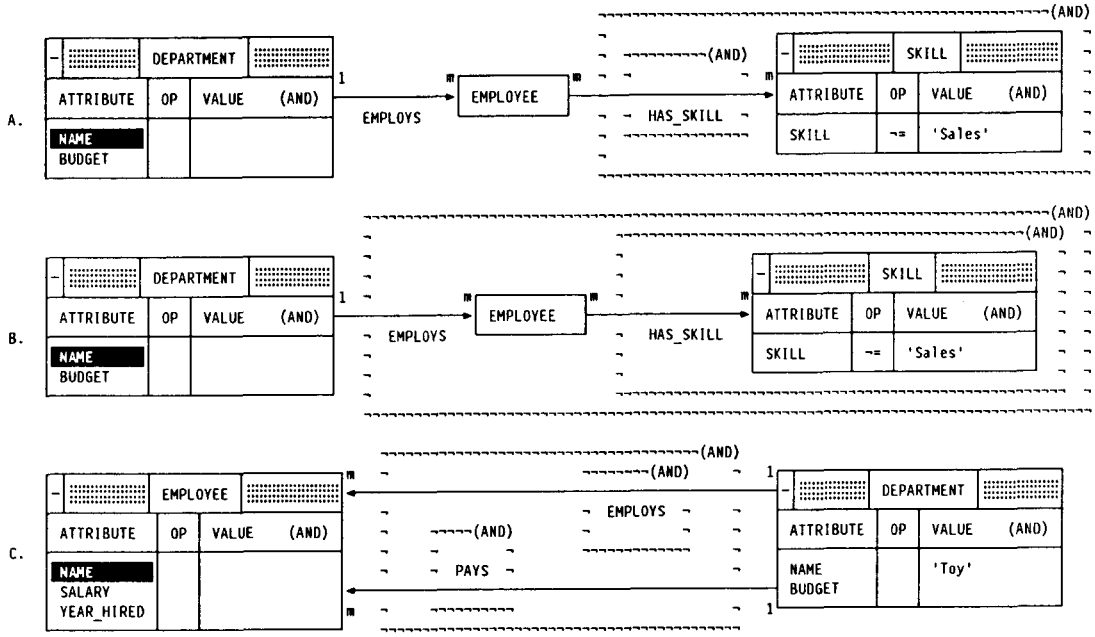



Fig. 13. Nested negation frames.

skill other than sales. An alternative wording is '. . . departments for which it is false that there is an employee whom the department employs who does not have a skill other than sales.' Each frame has one operand.

(C) List the names of the employees where there is a Toy department and it is false that the Toy department does not employ and does not pay the employee. The outer frame has two operands, i.e., the inner frames, each of which has one operand. The equivalence of this query and query A in *Fig. 12* comes from the fact that a conjunctive N-frame represents the NAND of any operands; nested NANDs suffice for the power of all four types of frames. We provide those other types of frames (instead of requiring use of nested NANDs) to meet our goal of ease of use.

Our set operations (union, intersection, and difference) involve frames. A query window's *set operation regulator*, which is initially empty, applies any combination of set operations to sets of projections. For set operations, all projections are immediately inside outermost frames, and no query elements are outside those frames. All those frames contain the same number of projections and are conjunctive P-frames; GRAQULA considers each such frame to have one operand. The set operation regulator, like a condition box, contains a tree. Each nonleaf node is a set operator. For each outermost frame, the set of specifications of all its projected rows appears in at least one leaf. For example, to project the union of departments' names and budgets with divisions' names and budgets, we use a P-frame around DEPARTMENT (with projected NAME and BUDGET), a P-frame around DIVISION (with the same projections), and a regulator with a root of 'UNION,' a leaf of '(DEPARTMENT.NAME, DEPARTMENT.BUDGET),' and a leaf of '(DIVISION.NAME, DIVISION. BUDGET).' A regulator with only unions maps directly into SQL, which does not use 'EXISTS subquery' immediately inside a union's outermost frames.

We specify and justify a few constraints for frames:

- Two frames' sides cannot intersect; this resembles textual languages' standard conventions for nesting.
- A frame (other than an outermost frame of a query with set operations) cannot contain a

projected entity image or an image whose attributes appear in a projected expression. This constraint is for safety; a query is *safe* [31] if it always has a finite result. Projection in a disjunctive P-frame (with two or more operands) or in an N-, I-, or C-frame would unsafely list data that might not exist.

- A frame that contains an image must also contain any condition boxes, expanded entity images, relationship images, and graphical comparisons that reference the image. Informally, this constraint forbids a nonsensical assertion that an instance's properties exist while the instance itself might not exist. In calculus, it forbids use of a quantified tuple variable outside the quantification's scope, as in this illegal expression: $(s.\text{BUDGET} > 500000) \wedge \neg(\exists s(\text{DEPARTMENT}(s)))$

We believe that these constraints do not restrict the class of meaningful queries that we can express.

*4.6.2 Implication and consequent frames*

We now turn from identity, negation, conjunction, and disjunction (and existential quantification) to implication (and universal quantification). Implication has two operands; the antecedent implies the consequent. Implication uses a pair of frames; a consequent frame encloses the consequent, and an implication frame encloses both operands. Thus the antecedent includes everything inside the I-frame but outside the C-frame. The pair means '$(\ldots \rightarrow \ldots)$' in a textual language and 'if . . . , then . . .' in English. As in P- and N-frames' operands, within an antecedent that contains several query elements immediately inside the I-frame, we use the conjunction of the elements' conditions. P- and C-frames have identical properties inside.

If an I-frame immediately contains any entity or many-to-many relationship images, we bind their tuple variables to *universal* quantifiers (as in '$\forall s \forall t$') in calculus. The scope of quantification is the I-frame, which contains the C-frame. SQL has no general equivalent. The meaning of the pair of frames and their contents together has the form '$\forall s \forall t(\ldots \rightarrow \ldots)$' in calculus; the meaning is 'for all . . . , . . .' in English. A C-frame contains zero or more scopes of existential quantification.

*Fig. 14* shows examples of graphical queries that use implication and consequent frames. Each English query here has a bold **antecedent** and an italicized *consequent*:

(A) List the names of the departments that employ an employee where for all **the skills other than sales,** *the employee has the skill.* Another wording is '. . . an employee who *has* all **the skills other than sales.**'

(B) List the names of the departments for which all **the employees they employ** *have a skill other than sales.* An alternative is '. . . departments **employing** only **employees** *with a skill other than sales.*'

(C) List the skills that only all the employees with a salary over 50000 have. An alternative is '. . . skills for which all **the employees with a salary over 50000** *have the skill* and all **the employees who have the skill** *have a salary over 50000.*' The second wording shows the *two* implications; each has a pair of I- and C-frames. The nesting of each C-frame inside its I-frame matches each consequent with its antecedent.

Properties of implication and universal quantification lead to these properties of I- and C-frames:

- A C-frame, not connection, divides the operands, because the number of sets of connected elements might not be two, and the operands do not commute, unlike operands of conjunction or disjunction.
- Implications and consequents have a one-to-one correspondence; one I-frame immediately contains one C-frame. Within this constraint, GRAQULA can nest any combination of P-, N-, and I-frames inside any frame. The use of two frames (I- and C-) permits the nesting of implication.

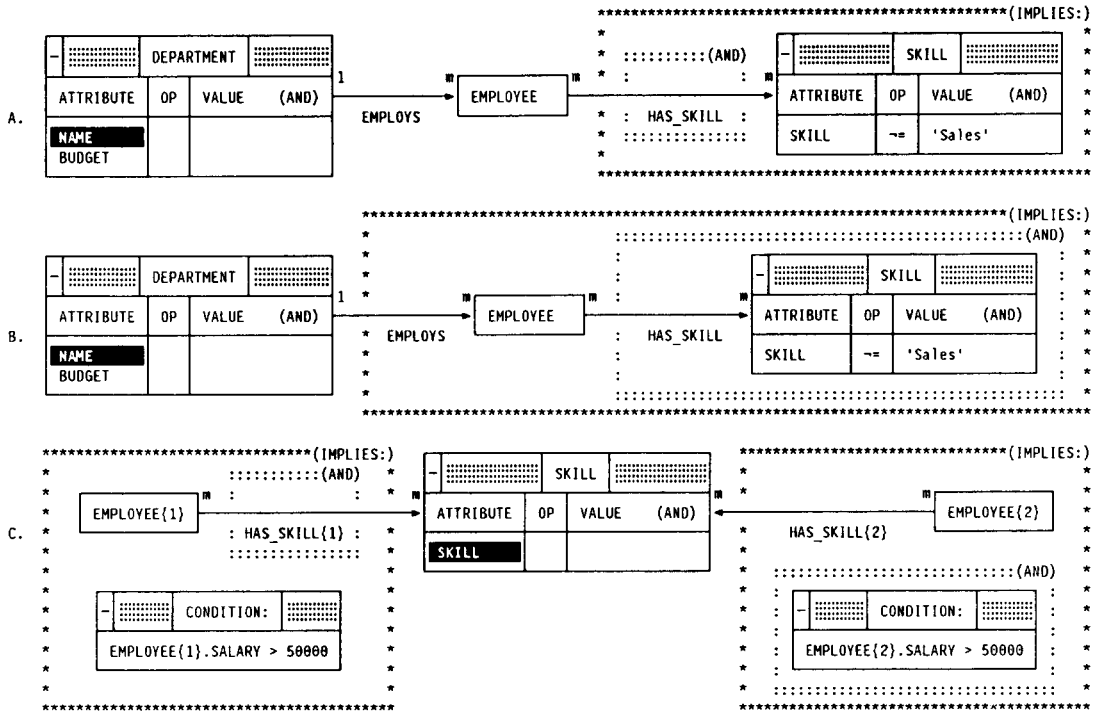



Fig. 14. Implication and consequent frames.

- If an I-frame immediately contains quantification of tuple variables, that quantification's scope must include the consequent, which can mention the tuple variables. Thus the scope is the entire I-frame.

Within the constraints of these properties, I- and C-frames are consistent with P- and N-frames; earlier we described properties that apply to all four types of frames.

We consider implication and universal quantification to be inherently more complex than some other concepts in query languages; Thomas [30] describes an experiment (not involving GRAQULA) showing that universal quantification can be difficult for users. Also, flexibilities of English (e.g. with words like 'all') can give a misleading impression that a query contains universal quantification. Therefore, we provide a guideline to ease the writing of universally quantified graphical queries, based on English queries:

(1) In the English query, eliminate any instances of what we call *inclusive* terms (e.g. 'all,' 'each,' 'every,' 'at least,' and 'whenever') or *exclusive* terms (e.g. 'only,' 'just,' 'solely,' and 'exclusively') where the elimination preserves the query's meaning. For example, eliminate the first 'all' and keep the second in 'List the names of all the employees who have all skills.'

(2) Reword the English query to replace exclusive terms. For example, replace the alternative wording in query B by the first wording. The rewording sometimes leads to two implications, as in query C.

(3) For each inclusive term, identify the noun phrase that the term modifies, and identify the claim that must be true for each instance of the noun phrase. For example, the noun phrase and the claim in query B are the bold and italicized words, respectively.

(4) Construct an *interim* graphical query, based on articles (e.g. 'a') instead of inclusive terms. For example, for query B, construct the figure's query B *without* its frames.

(5) For each inclusive term, add a C-frame (with 'AND') that encloses each query element that depicts the claim but does not also depict anything outside the claim. This produces

the C-frame in query B, since 'have a skill . . .' appears only inside the claim. As an optional simplification, if the C-frame's only operand is a P-frame, copy the P-frame's English word to the C-frame, and remove the P-frame.

(6) For each inclusive term, add an I-frame that encloses the C-frame and encloses each query element that depicts the noun phrase but does not also depict anything outside the phrase and claim. This produces the I-frame in query B, since 'employee(s)' and 'employ(s)' appear only inside the phrase.

Users need not know relational calculus. The guideline handles the patterns of universally quantified English queries that we consider common. Of course, flexibilities of English prevent any claim that the guideline handles all possible queries, and we do not propose an automated mapping from English.

Equalities in logic can map implication (and universal quantification) into negation and conjunction (and existential quantification). '$A \rightarrow C$' maps into '$\neg(A \wedge \neg C)$,' and '$\forall s(A \rightarrow C)$' maps into '$\neg(\exists s(A \wedge \neg C))$.' The mapping has a simple graphical description; an I-frame maps into a conjunctive N-frame, and a conjunctive or disjunctive C-frame maps into a corresponding N-frame. Thus we might process queries A and B in *Fig. 14* as if the user had written those in *Fig. 13*. This mapping allows implementation on current database systems; they typically have negation, conjunction, disjunction, and existential quantification but not implication with both an arbitrary consequent and universal quantification. SQL's consequent is always a comparison, although Fratarcangeli [16] has designed a generalization.

The universally quantified queries that we found in the research literature are expressible directly via I- and C-frames. We believe that I- and C-frames are easier to use than set operations or nested negation, since the system, not the user, maps from the user's perspective into the system's capabilities. Frames concisely specify operations, scopes, and quantification.

### 4.7. *Aggregates*

An expression can include *aggregates*, which calculate a value from a set of values. GRAQULA's aggregate operators are 'MINIMUM' (or, synonymously, 'MIN'), 'MAXIMUM' or 'MAX,' 'COUNT,'[3] 'SUM,' 'AVERAGE' or 'AVG,' 'MEDIAN' or 'MED,' 'VARIANCE' or 'VAR,' and 'STANDARD DEVIATION' or 'STD.' A query can project an aggregate or use it in a condition. An aggregate's syntax includes

(1) the aggregate operator;
(2) 'DISTINCT,' 'ALL,' or neither; and
(3) a parenthesized operand (an expression involving attributes).

'DISTINCT' signals calculation from the set of *distinct* values in the operand; otherwise, an operation calculates *without* merging duplicate values. In an attribute's row in an expanded entity image, the operand is optional; the default is that attribute. Most details of semantics of GRAQULA's aggregates resemble those of SQL; we will not repeat them here.

We define an aggregate's *images* as the entity images whose attributes are in the aggregate's operand. An aggregate's *scope* is the portion of the query used for computing the aggregate. The scope contains the aggregate's images and contains the portion of the query that can mention the aggregate. In SQL, an aggregate's scope is the SELECT statement or subquery whose FROM clause contains the tables whose columns are in the aggregate's operand. Similarly, in GRAQULA, an aggregate's scope is the entire query (if the images are at the outermost level) or an operand of a frame (if the images are immediately inside

[3] We resisted the temptation to call this operator 'COUNT GRAQULA.'

that operand of the frame). The query in *Fig. 15* uses one scope to list the average salary of the Toy department's employees hired in 1980.

Any number of attributes in a scope of aggregates can be *grouping attributes*; their EXPRESSION entries contain 'GROUP BY.' If all the attributes in an entity image's key have grouping, then the image's other attributes automatically have grouping. With grouping, GRAQULA calculates a value for each aggregate for each selected value of the grouping attribute (or each selected value of the grouping attributes' Cartesian product if several attributes have grouping). As in SQL, if a condition involves an aggregate whose scope has no grouping attributes, the scope has one group.

*Fig. 16* shows aggregates with grouping. Query A lists each department's name and the average salary of its employees where the department has employees and their average salary exceeds 50000. Query B uses two scopes of aggregates to list the hiring years for which the average salary of the employees hired in that year exceeds one tenth of the average budget of all the departments. A SQL query for query B is:

```
SELECT DISTINCT YEAR_HIRED FROM EMPLOYEE E
  GROUP BY YEAR_HIRED
  HAVING (EXISTS (SELECT 1 FROM DEPARTMENT D
    HAVING AVG(E.SALARY) > 0.1 * AVG(D.BUDGET)))
```

This is equivalent to:

```
SELECT DISTINCT YEAR_HIRED FROM EMPLOYEE E
  GROUP BY YEAR_HIRED
  HAVING AVG(E.SALARY) >
  (SELECT 0.1 * AVG(D.BUDGET) FROM DEPARTMENT D)
```

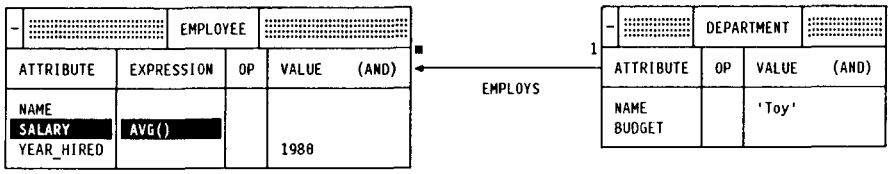



Fig. 15. An aggregate.

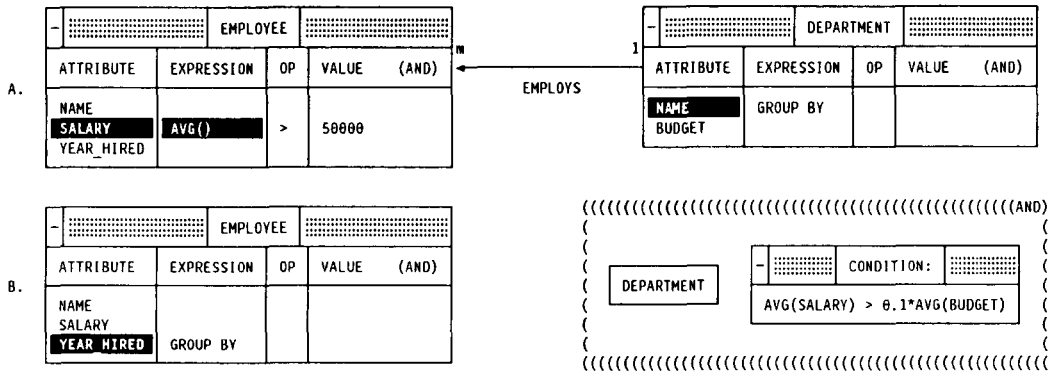



Fig. 16. Aggregates with grouping.

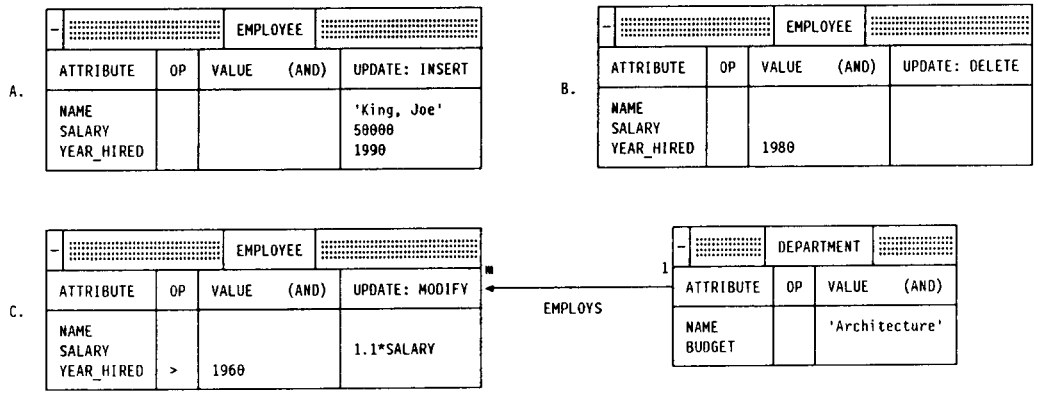



Fig. 17. Updates on entities.

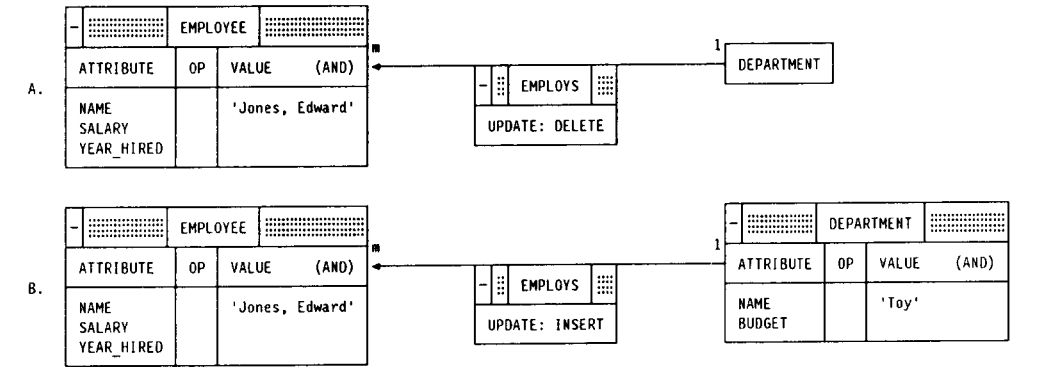



Fig. 18. Updates on relationships.

### 4.8. *Updates*

We now turn from queries to updates, i.e. insertion, deletion, and modification; they use an UPDATE column in an expanded image. *Figure 17* shows updates on entities. Update A inserts an employee named Joe King with a salary of 50000 and a year of hiring of 1990. Update B deletes the employees hired in 1980. Update C is a modification that gives a 10% raise to the Architecture department's employees hired after 1960. In any update, GRAQULA displays the number of updated instances. This is useful feedback because, for example, no change takes place if no instances satisfy an update's condition. By default, GRAQULA does not commit updates immediately; an action for commitment is available.

*Fig. 18* shows updates on relationships. Update A deletes the current employment relationship for Edward Jones; update B inserts one between him and the Toy department. Modification does not apply to relationships. Deleting an end of a relationship instance automatically deletes the relationship instance.

## 5. Control of queries

We have described what queries can contain; we now discuss facilities to control them. A user can name query windows and can save them in files; an action creates a menu to

list, retrieve, delete, save, or replace them. The menu can also save the query window as a view, which will then appear in the schema window as a node (entity type) with 'VIEW:' preceding the name. Another action creates a menu to list the query windows that already exist in the working area and to copy one of them into the window whose action created the menu.

A user might want to cascade a query's result into a later activity (e.g. another query or an update) instead of sending it to a result window. Temporary entity and relationship types, which do not persist in storage, serve this purpose. Typically, the user inserts into temporaries, uses the results, and optionally deletes. For example, a user might create the union of divisions' budgets and departments' budgets by inserting them into the same attribute of a temporary entity. An action in a query window creates a menu to create and delete temporary entity and relationship types and their images; their names begin with an ampersand. Also, an action in a result window can copy a result into a temporary entity.

An action in a query window executes the window; another action commits or undoes all uncommitted updates. The window's *execution regulator* contains lines of text to identify the tasks that execution will perform. 'QUERY' identifies a task as this window's graphical query, 'COMMIT' identifies commitment of updates, and 'CALL' plus a different query window's name identifies that window's execution regulator. For most queries, users can ignore the regulator, since it initially contains one 'QUERY,' i.e. no commitment or invocation of other queries. An action allows switching between hiding (the initial setting) and displaying the regulator and allows modification of the regulator. Use of execution regulators provides the effect of subroutines. It also permits an update and a subsequent query of the result. It also lets a user split a large query into smaller queries that pass data via temporary (or regular) entities and relationships.

Queries (but not users) can pass *sets* of values to other queries in temporary entities and relationships. Queries and users can also pass *individual* values as *parameters*, which are alphanumeric strings with a preceding '#' (e.g. '#TAX'). Query Management Facility [29] has a comparable method. A user who is writing a query can generalize it by replacing a name or constant by a parameter. GRAQULA retains parameter values between query window executions. The execution action's menu has two entries; either entry causes GRAQULA to scan the query to find all parameters before execution. For one entry, GRAQULA prompts the user to supply *only* the parameter values that do not yet exist. For the other entry, GRAQULA displays the current values (if any) of *all* the parameters; the user supplies that ones that do not yet exist and optionally changes the others. GRAQULA stores the values in temporary entities. A query can supply parameter values for a later query by updating the entities or by including a parenthesized sequence of 'parameter = constant' after a query window's name in the execution regulator.

## 6. Relational completeness

Codd introduced *relational completeness* [11] as a relational language's ability to express certain expressions. He also defined a relational algebra and showed its completeness. Its operations include projection, selection (which his paper mentioned only implicitly), Cartesian product, join, difference, union, intersection, division, and restriction. We will not discuss intersection, division, and restriction, since they are definable in terms of the other operations. We show below that the operations on entities in the E–R version of GRAQULA suffice to express all the algebraic operations and thus are relationally complete. Of course, our discussions also apply to the relational version of GRAQULA.

We expect that most queries will not use insertion into temporaries. However, users

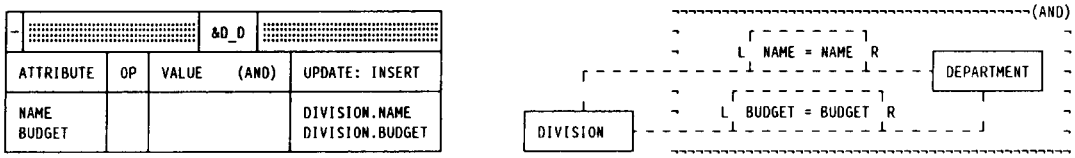



Fig. 19. Difference.

sometimes want to cascade a result into a later activity, so our discussions include temporaries, which allow closure.

Projection and selection are straightforward. We specify Codd's projection by projecting the desired attributes or by inserting them into temporaries. Similarly, we specify selection (which Codd showed as joining with a constant) by using a selection for the desired attributes and then projecting or inserting appropriately. More generally, a condition box can express any combination of conjunction, disjunction, and negation of textual conditions that involve expressions of attributes and/or constants.

Codd defined Cartesian product with projection of all attributes and with no selection. To specify it, we simply project (or insert into temporaries) all the attributes of the two entity images. Join resembles Cartesian product, but it compares attributes to limit the result. We simply add the comparison to the Cartesian product. Of course, relationship images also have the effect of joins.

We can specify difference or union by projecting with a set operation regulator or by using a join and negation. In *Fig. 19*, we insert DIVISION's NAME and BUDGET minus DEPARTMENT's into a &D_D temporary entity image. We can specify a union by writing two queries that insert the second entity type and the difference of the two entity types, respectively, into a temporary entity image. If the two entity types might have instances in common, we need the difference (instead of simply the first entity type) to avoid an attempt to insert the common instances twice.

## 7. Implementations and users' experience

We have implemented part of the capabilities of the relational version of GRAQULA in a relational query language (GARP) [8], and many graphical features that GRAQULA uses also appear in two E–R implementations: a browser (AERIAL) [7] and a facility for schema definition and query (RMGraph) [18]. *Table 1* shows (with 'G,' 'A,' and 'R') whether GARP, AERIAL, and RMGraph implement certain features of full GRAQULA. We show sample screens from the three languages and describe the languages very briefly below; the

Table 1
Implementation of some GRAQULA features in GARP (G), AERIAL (A), and RMGraph (R)

| Languages | | | Feature |
|---|---|---|---|
| G | A | R | Framework of windows, action bars, and menus |
| G | A | R | Database diagrams that user can tailor, with arcs for relationships or expected joins |
| G | A | R | Scrollable list of attributes |
| G | | R | Directory of schema elements |
| | | R | Clicking and highlighting of attributes for projection |
| G | A | R | Result window |
| G | | | User's addition of arcs for ad hoc joins |

references give more details. We also describe users' comments on GARP and RMGraph; the AERIAL prototype has no user community.

GARP is a relational query language that draws arcs for expected joins. In the GARP screen in *Fig. 20,* the user has prepared a query involving students' class grades and has requested a window showing the generated SQL statement. We obtained comments from users of the GARP prototype. They liked GARP, especially the use of diagrams and arcs. One user considered GARP's function to suffice, while others wanted more function. One noted that GARP's extensive use of clicking on positions within objects can be confusing; she preferred action bar entries. One user suggested letting users specify that some tables are closely interrelated and thus should be near each other when GARP chooses a schema layout. One wanted to save the user's tailoring of diagrams. One wanted the ability to display columns' data types.

AERIAL is an E–R browser. Users who are unfamiliar with the database definition or instances can explore to find parts that relate to parts they have already seen. In the AERIAL screen in *Fig. 21*, the user has found all the product groups and is about to make 'cakes' the current product group of interest.

RMGraph, which is part of the Repository Manager [26] product, is a facility for E–R schema definition, schema viewing, and query. It evolved from the AERIAL work. In the RMGraph screen in *Fig. 22*, the user has prepared a query involving students. We obtained comments from users and from people who just saw a demonstration. Most people liked the graphical approach in general, for its ease of use. Several suggested showing instance control on relationships. One person preferred straight lines (with no bends) for relationships. Another preferred grid-based diagrams with relationship lines limited to horizontal, vertical, and 45 degrees, but GARP users whom we asked about this issue preferred a flexible layout.

Our design of GRAQULA also reflects the experience of users of other languages:

- The design of a graphical interface involves trade-offs in the choice between graphical and textual notations and in the amount of prompting and outlines to display near areas for obtaining the user's input. Graphics and displayed information can ease understanding,

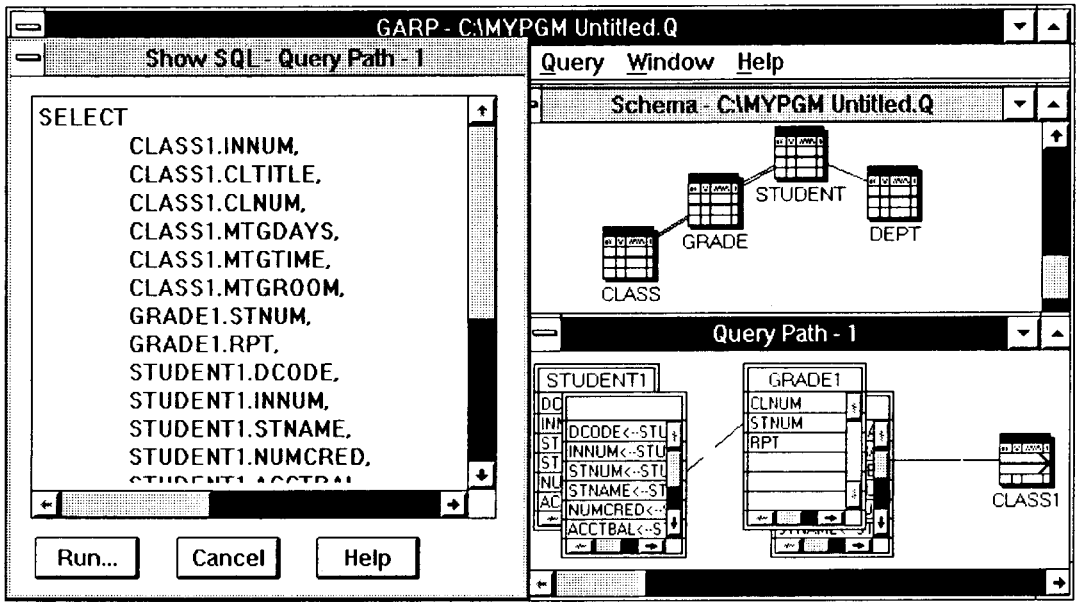



Fig. 20. Part of a Screen from GARP.

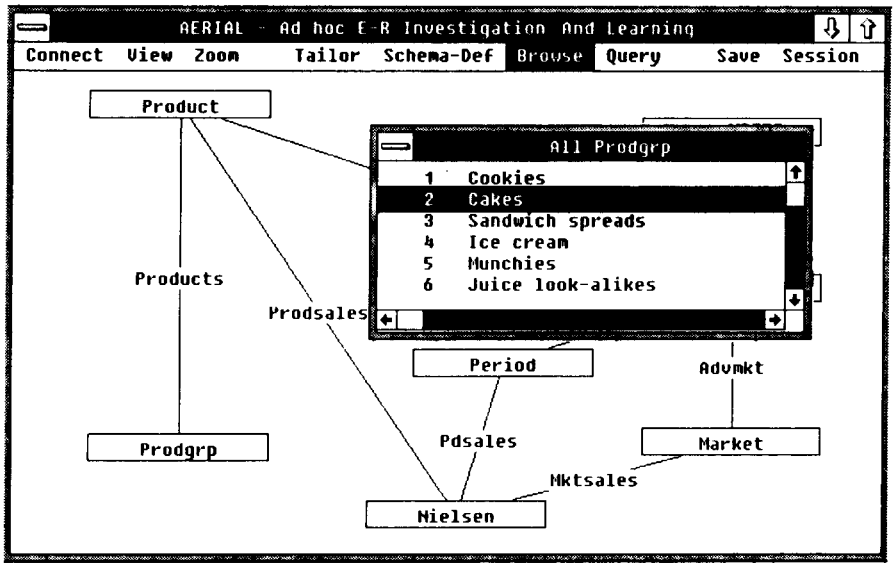



Fig. 21. Part of a screen from AERIAL.

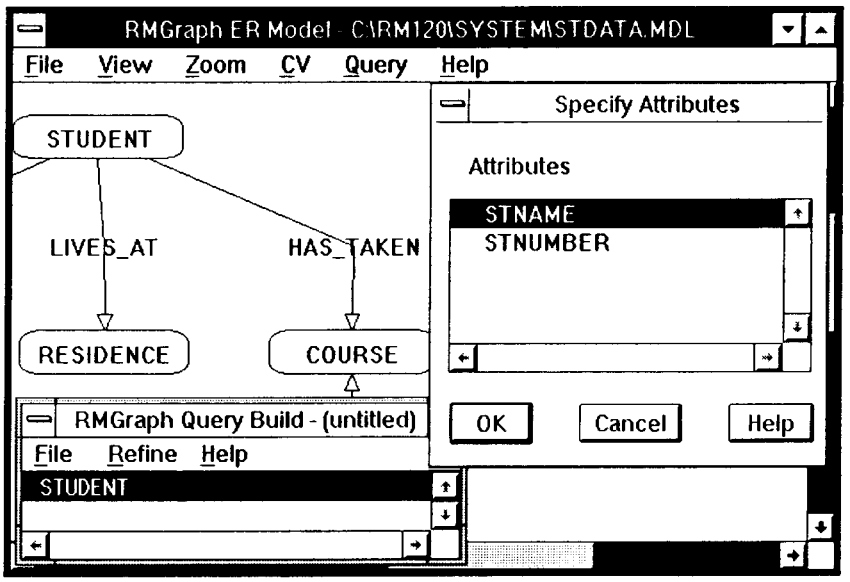



Fig. 22. Part of a screen from RMGraph.

and they reduce the opportunities to make errors, but they occupy space, of course, and construction of graphics can also require more work by the user. For example, in one language that used almost exclusively graphical notations, the necessary activity with the mouse sometimes annoyed users. Therefore, GRAQULA offers a choice between graphical and textual notations for some operations, e.g. logical operations in a condition box. One GARP user wanted a choice between graphics and text for entering queries.

- The developers of Pasta-3 [21] found that users sometimes inadvertently specify a Cartesian product (two entity images with no relationship image) when they want a join

(inclusion of a relationship image). Therefore, Pasta-3 and GRAQULA warn the user and offer to connect the entity images; GARP warns the user about unconnected tables. A few GARP users noted that its red message seems to imply an error, so we learned that we should clearly identify it as just a warning, since a Cartesian product is legal. One GARP user also wanted feedback to suggest connections.

## 8. Summary

We have described GRAQULA, a graphical language for querying and updating a database. One version of GRAQULA provides a user interface for the entity-relationship data model, and another version (with almost identical syntax) provides a user interface for the relational model. The operations in the relational version and the operations on entities in the E–R version are relationally complete. The relational version draws arcs for expected joins (with automatic drawing in schemas when referential integrity constraints exist), and the E–R version draws arcs for relationships. Queries can involve cyclic patterns of relationships. Frames provide logical operations that have user-specified scopes, allow nesting, and can involve existential or universal quantification. We supply a guideline for writing universally quantified queries, and users do not need set operations or nested negation for such queries. Aggregates also have user-specified scopes. Execution regulators let queries invoke other queries, and users and queries can pass parameters to queries. E–R queries can map into relational queries, for clarity of definition and for implementation. GRAQULA uses popular, convenient techniques (e.g. windows, action bars, and menus) for manipulating graphical objects that form a query. The design reflects a specified set of goals, including expressive power, consistency, and limitation of required memorization.

## Acknowledgements

Hugh Darwen, Michel Kuntz, Guy Lohman, Peter Pistor, and Yaron Wolfsthal reviewed a draft of this paper. Michele Angelaccio, David Embley, Hank Korth, and Tom Wu reviewed our descriptions of their work. Peter Chen suggested several improvements. Several people, especially Matt Anderson, provided feedback and discussions on our techniques.

## Appendix: Considerations for the design of frames

We explain below some considerations that led to our design of GRAQULA's frames. We discuss negation, N-frames, and P-frames first; then we discuss I- and C-frames.

Negation (of existential quantification) is an important ability in a query language. For example, we use it to list the divisions that do not contain any departments. QBE (the predecessor of OBE) was perhaps the first graphical query language to include such negation; we will show how GRAQULA improves upon OBE's negation. Query A in *Fig. 23* (involving the DIVISION and DEPARTMENT tables) shows the above query in OBE. The 'P.' means projection of all columns of DIVISION, the '¬' means OBE's *row negation* (negation of existential quantification) applied to DEPARTMENT, and the two uses of the *example element* '_T' below column names mean a join of those columns. Here is that query in SQL:

- SELECT * FROM DIVISION V WHERE

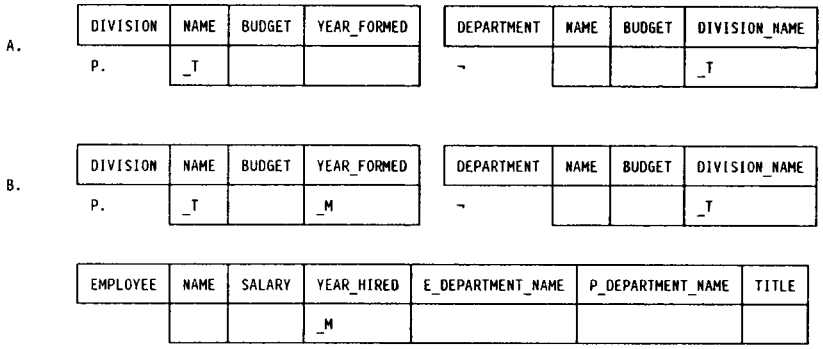



Fig. 23. Two queries in OBE.

NOT ( EXISTS (SELECT 1 FROM DEPARTMENT T
WHERE T.DIVISION_NAME = V.NAME))

The following SQL expression (a subset of the complete SQL query above) corresponds to the negated row in DEPARTMENT (a subset of the OBE query):

- NOT ( EXISTS (SELECT 1 FROM DEPARTMENT T
  WHERE T.DIVISION_NAME = V.NAME))

In SQL and in relational calculus, every negation has a scope. usually delimited by parentheses. For example, 'NOT(A = B) and C = D' and 'NOT(A = B AND C = D)' have different meanings. In some cases (e.g. negation of one 'EXISTS subquery'), the syntax allows omission of the parentheses for negation. However, the most general case requires parentheses for negation, so we included them above.

Query B (involving DIVISION, DEPARTMENT, and EMPLOYEE) shows a problem that arises when an OBE query has *two* or more uses of row negation. In our discussion, we will assume that the database contains the following instances of divisions, departments, and employees:

```
DIVISION:                DEPARTMENT:                EMPLOYEE:
NAME:  YEAR_FORMED:      NAME:  DIVISION_NAME:      NAME:        YEAR_HIRED:
D      1990              X      D                  Adams, Al    1991
B      1991              Y      B                  King, Ken    1992
E      1992
N      1993
```

Without a specification of OBE that states what such a query means, there are *three* possible SQL expressions (which we label 1A, 1B, and 2 below) for the combination of the two negated rows in query B. For each such SQL expression, we show here our label, the expression (where 'V' represents 'DIVISION'), the number of scopes of negation, the number of scopes of existential quantification in the expression, and the divisions that the query's result includes:

1A: NOT ( EXISTS (SELECT 1 FROM DEPARTMENT T, EMPLOYEE M WHERE
T.DIVISION_NAME = V.NAME
AND M. YEAR_HIRED = V.YEAR_FORMED) )
1 scope of negation; 1 scope of existential quantification; result includes D, E, and N.

1B: NOT ( EXISTS (SELECT 1 FROM DEPARTMENT T
WHERE T. DIVISION_NAME = V.NAME) AND
EXISTS (SELECT 1 FROM EMPLOYEE M
WHERE M.YEAR_HIRED = V.YEAR_FORMED) )

1 scope of negation; 2 scopes of existential quantification; result includes D, E, and N.

2. NOT ( EXISTS (SELECT 1 FROM DEPARTMENT T
WHERE T.DIVISION_NAME = V.NAME) ) AND
NOT ( EXISTS(SELECT 1 FROM EMPLOYEE M
WHERE M.YEAR_HIRED = V.YEAR_FORMED) )

2 scopes of negation; 2 scopes of existential quantification; result includes just N.

The result includes D and E if and only if there is 1 scope of negation. Specification of the number of scopes of negation is necessary and sufficient to evaluate the query; specification of the number of scopes of existential quantification is not necessary and not sufficient. Expressions 1A and 1B are equivalent.

If a specification of a language like OBE does not state what such a query means, it is easy to eliminate the ambiguity (without adding a graphical construct) by just adding a sentence to the specification. For example, the OBE implementation used expression 2; negation's scope is always just one table. In contrast, GRAQULA gives users *more power*: Two N-frames specify expression 2, while one N-frame specifies 1B, which is equivalent to 1A. If we wish to provide the ability to specify 1A directly, a possible extension to GRAQULA (not discussed in the main body of this paper) is to allow an English word of 'ONE' in an N-frame; this merges all contained query elements into one operand (as in 1A). In the graphical description of mapping from implication into negation and conjunction, the outer N-frame can use 'ONE.'

Of course, an N-frame need not involve existential quantification (since it might contain just a one-to-many relationship image, for example), but it always involves negation; an N-frame consistently means 'NOT(. . .),' with no exceptions. GRAQULA can also nest negation, as we showed earlier, and the use of P-frames and the use of 'OR' in a frame increase GRAQULA's convenience by obviating the use of nested negation in many cases. In summary, QBE introduced the important concept of graphical negation (of existential quantification); GRAQULA improves upon QBE's negation by allowing user-specified scopes (and thus more power), nesting, and a larger set of logical operations (conjunction and disjunction with or without negation and with or without existential quantification).

We now turn from P- and N-frames to I- and C-frames. Specifically, we sketch a possible alternative syntax for implication with universal quantification, using a single frame instead of a pair of I- and C-frames, and we will explain the advantages of using a pair of frames. We call the alternative's single frame an *antecedent frame* (*A-frame*); it encloses the antecedent. The consequent consists of all the unprojected query elements that connect with the A-frame without using transitivity through any projected query element; here we treat an updated image (or an image whose attributes appear in an UPDATE column) as a projected query element. Earlier papers [33, 34] called an A-frame a *universal quantification box* (*U-box*). Here an A-frame uses an English word of 'IMPLIES:'; it uses '%' for its sides. For example, the query in *Fig. 24* uses an A-frame; it is equivalent to query B in *Fig. 14*.

I- and C-frames have several advantages over A-frames, in expressive power, consistency, and ease of use (besides consistency). These are the advantages in expressive power:

(1) An A-frame cannot directly express some simple queries like query A in *Fig. 14*. Placing an A-frame around SKILL would imply that the consequent includes not only HAS_

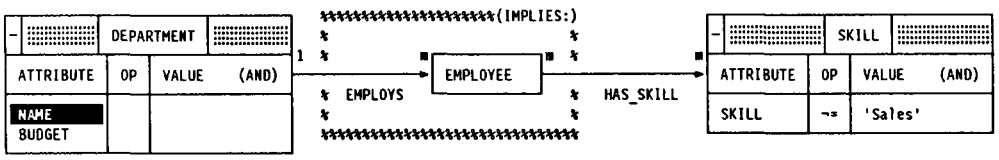



Fig. 24. An antecedent frame.

SKILL (the desired consequent) but also EMPLOYEE and EMPLOYS (which we desire to be outside the implication).

(2) I- and C-frames (but not A-frames) can nest an implication inside an antecedent.

(3) I- and C-frames (but not A-frames) can nest an implication inside a consequent.

(4) When a query contains two or more implications that are peers (i.e. they are not nested), I- and C-frames (but not A-frames) let the user specify which consequent matches which antecedent.

These are the advantages in consistency:

(5) Consistently, a P-, N-, I-, or C-frame limits the scope of quantification of the tuple variables for the entity or many-to-many relationship images immediately inside the frame. This is not true for an A-frame, since the scope of universal quantification includes the consequent, not just the antecedent. An antecedent frame does not enclose the entire scope of universal quantification.

(6) To forbid the use of a quantified tuple variable outside the quantification's scope, earlier we specified this constraint:

- A frame that contains an image must also contain any condition boxes, expanded entity images, relationship images, and graphical comparisons that reference the image.

This one constraint applies consistently to P-, N-, I-, and C-frames. However, the scope constraint for A-frames is not consistent with the constraint for P- and N-frames; here is the constraint for A-frames:

- If an A-frame contains any condition boxes, expanded entity images, relationship images, or graphical comparisons, it must also contain any unprojected images that those contained elements reference.

For a tuple variable whose image is in the consequent, this constraint forbids the antecedent from using the tuple variable, since the variable's scope of existential quantification is just the consequent.

These are the advantages in ease of use (besides consistency):

(7) With A-frames, a user who is constructing a query with implication and universal quantification must understand the pattern of query A in *Fig. 14* (see advantage 1 above) and must use either nested negation (instead of an A-frame) or a two-step procedure of insertion from several images (e.g. DEPARTMENT and EMPLOYEE) followed by projection of a subset (e.g. DEPARTMENT's NAME). Besides that simple pattern, queries with two or more implications (whether or not nested) also require use of nested negation instead of A-frames.

(8) After a user constructs a query, that user or another user might examine the query (perhaps on a later day) and try to understand the query, perhaps by translating it to English. With an A-frame, the user must follow a chain of connection to identify the consequent; a C-frame identifies it immediately.

An advantage of A-frames (in ease of use) is that constructing an A-frame query requires identification of just the antecedent, not the consequent, since identification of the consequent is automatic. In summary, A-frames have 1 advantage (in ease of use), and I- and C-frames have 8 advantages (in expressive power, consistency, and ease of use besides consistency).

## References

[1] Amer. Natl. Standards Institute, *Database Language SQL*, New York, NY, X3.135-1992 (1992).

[2] M. Angelaccio, T. Catarci and G. Santucci, QBD*: A graphical query language with recursion, *IEEE Trans. Soft. Engin.* 16 (10) (Oct. 1990) 1150–1163.

[3] C. Batini, T. Catarci, M.F. Costabile and S. Levialdi, Visual strategies for querying databases, *Proc. IEEE Workshop on Visual Languages* (Oct. 1991) 183–189.

[4] R.E. Berry, Common user access – A Consistent and usable human-computer interface for the SAA environments, *IBM Syst. J.* 27 (3) (1988) 281–300.

[5] L.M. Burns, J.L. Archibald and A. Malhotra, A graphical entity-relationship database browser, *Proc. 21st Ann. Hawaii Internat. Conf. Syst. Sciences*, vol. 2 IEEE-CS (Jan. 1988) 694–704.

[6] L.M. Burns, A. Malhotra and J.B. Black, Is a picture worth a thousand queries?, Yorktown Heights, NY, IBM T.J. Watson Research Center, Research Report RC 16172, Oct. 1990.

[7] L.M. Burns, A. Malhotra, G.H. Sockut and K.-Y. Whang, AERIAL: Ad hoc Entity-Relationship Investigation And Learning, *Proc. 1991 IEEE Internat. Conf. Syst. Man and Cybernetics*, vol. 2 (Oct. 1991) 1151–1159. (also *Internat. J. Man-Machine Studies* 38 (4) (Apr. 1993) 607–623.

[8] L.M. Burns, S.P. Perkins, E.A. Shimmin and G.H. Sockut, GARP: Graphical Access to Relational Products, Yorktown Heights, NY, IBM T.J. Watson Research Center, research report, in preparation.

[9] P.P.-S. Chen, The Entity-Relationship Model – Toward a unified view of data, *ACM Trans. Database Syst.* 1 (1) (March 1976) 9–36.

[10] E.F. Codd, A relational model of data for large shared data banks, *Comm. ACM* 13 (6) (June 1970) 377–387.

[11] E.F. Codd, Relational completeness of data base sublanguages, in R. Rustin, ed. *Data Base Systems* (Prentice-Hall, 1972, Englewood Cliffs, NJ) 65–98. (Proc. Courant Computer Sci Symp. 6, May 1971.)

[12] B.D. Czejdo, R.A. Elmasri, M. Rusinkiewicz and D.W. Embley, A graphical data manipulation language for an extended entity-relationship model, *Computer* 23 (3) (March 1990) 26–36.

[13] C.J. Date, Referential Integrity, *Proc. 7th International Conf. Very Large Data Bases*, ACM (Sept. 1981) 2–12.

[14] R.A. Elmasri and J.A. Larson, A graphical query facility for ER Databases, *Proc. 4th Internat. Conf. Entity-Relationship Approach*, IEEE-CS (Oct. 1985) 236–245.

[15] D.W. Embley, B.D. Czejdo and M. Rusinkiewicz, Graphical query formulation by manipulating relational schema diagrams, *Systems Sci.* 12 (3) (1986) 69–87.

[16] C. Fratarcangeli, Technique for universal quantification in SQL, *SIGMOD Rec.* 20 (3) ACM (Sept. 1991) 16–24.

[17] IBM Corp., IBM data interpretation system data access tool set, form SH21-0497-2, Nov. 1991.

[18] IBM Corp., Repository manager/MVS repository modeling guide, form SC26-4619-01, March 1991.

[19] H.-J. Kim, Graphical interfaces for database systems: A survey *Proc. 1986 Mountain Regional ACM Conf.* (April 1986).

[20] H.-J. Kim, H.F. Korth and A. Silberschatz, PICASSO: A graphical query language, *Software Practice and Experience* 18 (3) (March 1988) 169–203.

[21] M. Kuntz and R. Melchert, Pasta-3's graphical query language: Direct manipulation, cooperative queries, full expressive power, *Proc. 15th Internat. Conf. Very Large Data Bases* (Morgan Kaufmann, San Mateo, 1989) 97–105.

[22] A. Malhotra, L.M. Burns, G.H. Sockut and K.-Y. Whang, IRIS: Interactive Repository Interface Services, Yorktown Heights, NY, IBM T.J. Watson Research Center, Research Report RC 16943, June 1991.

[23] V.M. Markowitz and A. Shoshani, On the correctness of representing extended entity-relationship structures in the relational model, *Proc. ACM SIGMOD Internat. Conf. Management of Data, SIGMOD Rec.* 18 (2) (June 1989) 430–439.

[24] N. McDonald and M. Stonebraker, CUPID – The friendly query language, *Proc. ACM Pacific 75 Conf.* (April 1975) 127–131.

[25] G. Özsoyoğlu, V. Matos and Z.M. Özsoyoğlu, Query processing techniques in the summary-table-by-example database query language, *ACM Trans. Database Syst.* 14 (4) (Dec. 1989) 526–573.

[26] J.M. Sagawa, Repository manager technology, *IBM Syst. J.* 29 (2) (1990) 209–227.

[27] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (Addison-Wesley, Reading, MA, 1987).

[28] G.H. Sockut, L.M. Burns, A. Malhotra and K.-Y. Whang, GRAQULA: A graphical query language for entity-relationship or relational databases, Yorktown Heights, NY, IBM T.J. Watson Research Center, Research Report RC 16877, March 1991.

[29] J.J. Sordi, The query management facility, *IBM Syst. J.* 23 (2) (1984) 126–150.

[30] J.C. Thomas, Quantifiers and question-asking, Yorktown Heights, NY, IBM T.J. Watson Research Center, Research Report RC 5866, Feb. 1976.

[31] J.D. Ullman, *Principles of Database Systems, 2nd ed.* (Computer Science Press, Rockville, MD, 1982).

[32] K.-Y. Whang, A.C. Ammann, A.S. Bolmarcich, M.B. Hanrahan, G.T. Hochgesang, K.-T. Huang, A. Khorasani, R. Krishnamurthy, G.H. Sockut, P.K. Sweeney, V.E. Waddle and M.M. Zloof, Office-by-example: An integrated office system and database manager, *ACM Trans. Office Info. Syst.* 5 (4) (Oct. 1987) 393–427.

[33] K.-Y. Whang, A. Malhotra, G.H. Sockut and L.M. Burns, Supporting universal quantification in a two-dimensional database query language, *Proc. 6th Internat. Conf. Data Engin.*, IEEE-CS (Feb. 1990) 68–75.

[34] K.-Y. Whang, A. Malhotra, G.H. Sockut, L.M. Burns and K.-S. Choi, Two-dimensional specification of universal quantification in a graphical database query language, *IEEE Trans. Soft. Engin.* 18 (3) (March 1992) 216–224.

[35] G. Wiederhold and R.A. Elmasri, The structural model for database design, in P.P.-S. Chen, ed., *Entity-Relationship Approach to Systems Analysis and Design* (North-Holland, Amsterdam, 1980) 237–257 (*Proc. Internat. Conf. Entity-Relationship Approach to Systems Analysis and Design*, Dec. 1979).

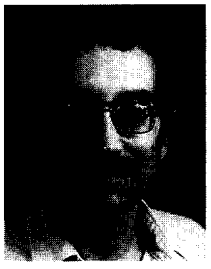
[36] H.K.T. Wong and I. Kuo, GUIDE: Graphical User Interface for Database Exploration, *Proc. 8th Internat. Conf. Very Large Data Bases*, Saratoga, CA: VLDB Endowment (Sept. 1982) 22–32.

[37] C.T. Wu, Visual query language for relational databases, in D.K. Hsiao, E.J. Neuhold, and R. Sacks-Davis, eds. *DS-5 Semantics of Interoperable Database Systems (conf. proc.), vol. 2*, (IFIP, Nov. 1992) 46–59. (longer version avail. from C.T. Wu, Dept. of Computer Science, Naval Postgraduate School, Monterey, CA).

[38] Z.-Q. Zhang and A.O. Mendelzon, A graphical query language for entity-relationship databases, in C.G. Davis, S. Jajodia, P.A. Ng, and R.T. Yeh, eds., *Entity-Relationship Approach to Software Engineering* (Elsevier Science, New York, NY, 1983) 441–448 (Proc. 3rd Internat. Conf. Entity-Relationship Approach, Oct. 1983).

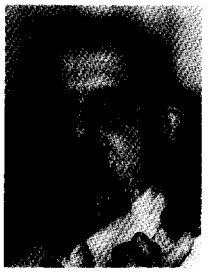
[39] M.M. Zloof, Query-by-example: A data base language, *IBM Syst. J.* 16 (4) (1977) 324–343.

[40] M.M. Zloof, Query-by-example – Operations on hierarchical data bases, *Proc. Nat. Computer Conf.* 45 (AFIPS Press, Reston, VA, June 1976) 845–853.



**Gary Sockut** is an advisory programmer at the IBM Santa Teresa Laboratory. Previously he was with BGS Systems, the National Institute of Standards and Technology, and the IBM T.J. Watson Research Center. He received an Sc.B. (magna cum laude) in Applied Mathematics from Brown University, an S.M. in Electrical Engineering from the Massachusetts Institute of Technology, and a Ph.D. in Applied Mathematics from Harvard University. His main areas of interest are database management (especially reorganization and graphical languages), office systems, and operating systems. He is a member of ACM and the IEEE Computer Society, and he has been a member of Sigma Xi.

**Luanne Burns** is a research staff member at the IBM T.J. Watson Research Center. Previously she was with Kodak. Her primary research interests have been in educational software systems, user interfaces, multi-media, artificial intelligence, expert systems, handwriting recognition, and entity-relationship database systems. She received a B.S. in Computer Science from the Rochester Institute of Technology. She received an M.S. in Computer Science from Columbia University, where her main concentration was in artificial intelligence and expert database systems. Her Ph.D., also from Columbia University, is in Developmental and Educational Psychology.



**Ashok Malhotra** has been a research staff member in the computer science department at the IBM T.J. Watson Research Center since 1975, when he received his Ph.D. from M.I.T. 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.

**Kyu-Young Whang** is an associate professor of computer science, and director of the Database and Knowledge Engineering Laboratory of the Center for Artificial Research, at the Korea Advanced Institute of Science and Technology (KAIST). His research interests include multimedia databases, object-oriented databases, engineering databases, expert systems, and office systems. Previously he was with the IBM T.J. Watson Research Center, where he performed research in databases, office systems, and expert systems. He graduated (summa cum laude) from Seoul National University; he received an M.S. from KAIST and an M.S. and Ph.D. from Stanford University. He was an IEEE Distinguished Visitor from 1989 to 1990, and he twice received the External Honor Recognition from IBM. He is a senior member of IEEE and a member of ACM.