p 1

Graphic queries and updates for 2-step

Report ISSI-91-003

Adolfo Guzmán

International Software Systems Inc.[1]

ABSTRACT. We define a graphic manner to represent information, sets, queries, updates, and constraints. We also define a text manner to represent above concepts. Finally, we give a series of examples expressed in:

(a) SQL, a traditional query language for databases;

(b) the "sequence" operation, defined by Yin [ ];

(c) our graphic notation (Chapter I), which is obtained by "filling in the details" of an "empty graph" (such as Figure 5•) representing the schema of the whole database;

(d) our text notation, called "path expressions", which can be mechanically (i. e., automatically) derived from (c), as described in Chapter II.

The empty graph of (c) representing the schema can be obtained and automatically drawn from the database description; this means that the user is not required to *draw* anything, merelly to *fill in* an already correct graph. Mistakes are minimized in this manner.

Our notation can represent in an unified manner relational databases, ERA, object databases, etc.

I Graphs representing permanent information

By "permanent information" we mean that stored in files, databases and other "permanent" media. Our basic idea is to represent with long ellipses **sets** of homogeneous information, while with circles (really, short ellipses) we represent these **individuals**: usually strings or numbers, but ocassionally individual objects. Arrows represent **relations** (properties, slots, relations of the ER model, ...) that go to ellipses or to circles. Example: see Figure 3• We can graphically represent information stored in: *(1) databases* expressed with tables (Section 1•1); *(2) relational databases* (Section 1•2); *(3) Entity relationship attribute* (ERA) model, and its variants or extensions (Section 1•2); *(4) object databases*, where an object can have pointers to other objects (Section 1•3); *(5)* many *conventional file systems* (Section 1•4), such as index sequential, etc; *(6)* information stored in a *hybrid combination* of (1) through (5) above.

Figure 1• shows how most common constructs used for permanent storage of information are represented in our notation; details are given in Sections 1•2 through 1•5 of this Chapter.

On these "schemas", a particular query is expressed by paths that go from a fat arrow (the domain of a query) to every shaded ellipse (the things that we want to know) representing the targets

1. *Address:* ISSI. 9430 Research Blvd., Bldg. IV-250. Austin, TX. 78759 Tel. (512) 338 5700 Telex: 499 1223 ISSIC Fax: (512) 338 5757 email: guzman%issi.uucp@cs.utexas.edu, cs.utexas.edu!issi!guzman

file: /home/guzman/publications/graphic-queries-and-updates-for-2-step.fm
Version 1 (Mar 91); changed Apr 2, 91; *printed Jul 17, 91*; 33 pages.

Relational database	**Table**: Represented by an ellipse, with values as circles, attributes as arrows from ellipse to circle. **Properties, fields** or **attributes**: represented as arrows to circles
ERA model	**Entity**: Represented as Table above. **Relation**: Represented by an ellipse pointing to the two entities it relates. **Attribute**: Arrows going from ellipse to small circles; circles represent the values of a given attribute for a given relation or entity.
Object base (object database)	**Class** and individual **object**: Represented as an ellipse **Slot**: represented as an arrow between ellipses. **Simple slots** (properties, attributes): arrow from ellipse to circle **Value** (simple value, number, string): circle.

Figure 1• Summary of our graphic notations for some types of databases

or ranges of a query. Example: see Figure 35• These paths can be expressed textually (Chapter II) or graphically (Chapters III and IV). It is possible to graphically show relations; functions (Sec. 3•1•3); predicates (Sec.3•1•2); queries (Sec. 3•1); updates and deletion (Sec. 3•2); sets [other than those shown in Figure 1•, which could be considered sets "directly related to the database"] and constraints (Chapter IV).

1•1 Introduction

The information of an application database (Such as Figure 2•) can be presented (after observing the mappings of Figure 1•) in the form shown in Figure 3•, which looks like a graph where the nodes are relations or values, and the arcs represent the attributes.

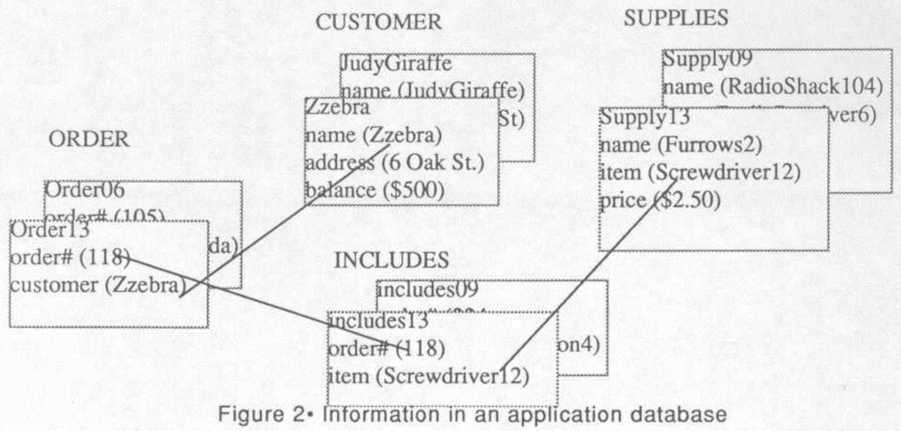


Figure 2• Information in an application database

For a given company, there is just one figure like Figure 3•, which describes how the data is organized in that company (it is called "company information graph", it is nothing else than our well known "database schema"; we use the first name, which sounds friendlier to the final user). Either or these can be derived from the definitions of the database schema (in DDL, for instance).

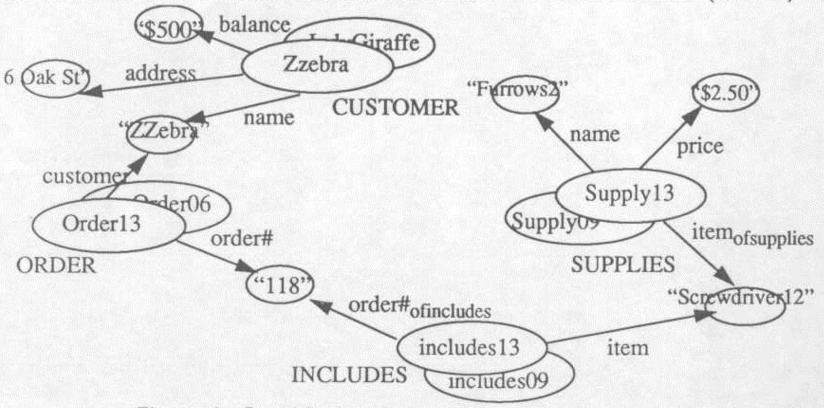


Figure 3• Graphical representation of the info contained in a db
The graph that results after deleting the "contents" (such as Order13) of the bubbles is unique for a given company or organization, and it is thus called "the company information graph".

Large ellipses represent relation or entity elements (such as Order13); the names of the relation appear in capital letter, such as INCLUDES; the names of the entities applear also in capital letters, such as ORDER, CUSTOMER, SUPPLIES. Circles (really, smaller ellipses) represent "data" (strings, numbers), such as "118". Arcs represent properties or fields, such address or balance. When they are non-unique, such as name or order#, we can always make them unique by noticing from which relation they emanate. Thus, we differentiate order# by writing, if needed, order#$_{oforder}$ and order#$_{ofincludes}$.

Figure 2• shows a typical relational database; the entities are Customer, Supplies. The relations are order, includes. A relation can only have two fields, each of which is a key (Screwdriver12 is a unique name for a product; 118 is a unique order# for an order). [A relation in a relational database contains only two items; each is a key. These keys are numbers or strings; they are not pointers to "records" or entities]

A variant is shown in Figure 4•, where the relations point to the records themselves.

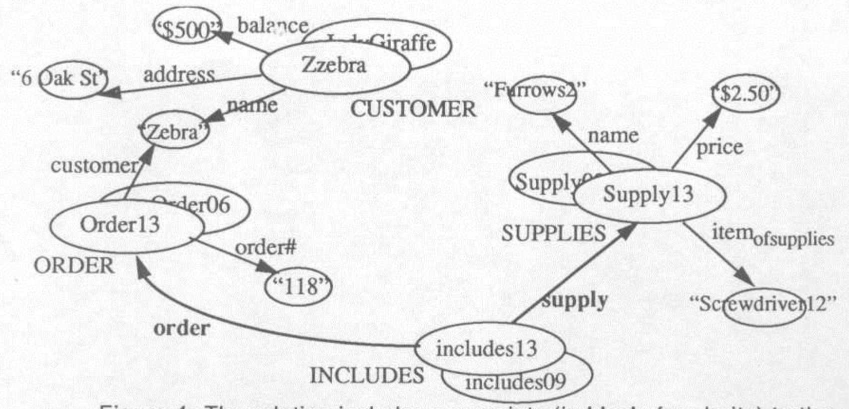


Figure 4• The relation includes now points (in **black**, for clarity) to the "records" such as order13 and Supply13. Compare with Figure 3•

The representation of Figure 3• can become uncluttered if we do not represent the instances (particular orders, customers, ..), as given in Figure 5•

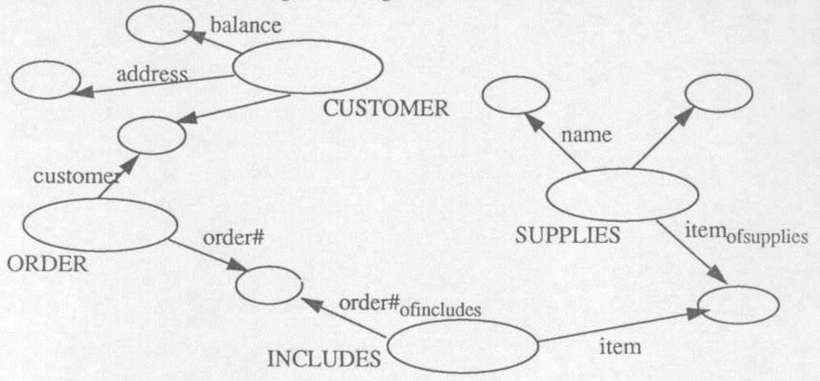


Figure 5• Simplified diagram where no instances are shown. Compare with Figure 3•

1•2 Relational databases and the ERA model

The basic entity-relationship (ER) model uses the concepts of entities, relationships, and attributes to represent data. Entities of the same type are grouped into an entity set. In Figure 6• there are two entity sets, STUDENT and CLASS, and the relationship IS-TAKING. STUDENT and CLASS are called the participating entity sets for the relationship IS-TAKING.

Attributes describe the properties of entities and relationships. In Figure 6•, the attributes of STUDENT are SS#, Name, and Address.

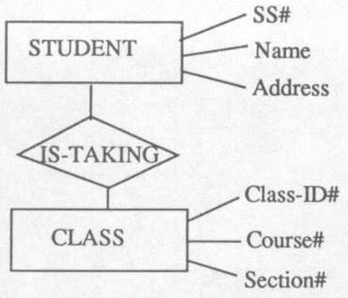


Figure 6• A simple ER schema diagram

Figure 7• is a conceptual representation of some data instances corresponding to the schema in Figure 6•. Each relationship instance relates two entitites, one from each of the participating entity sets.

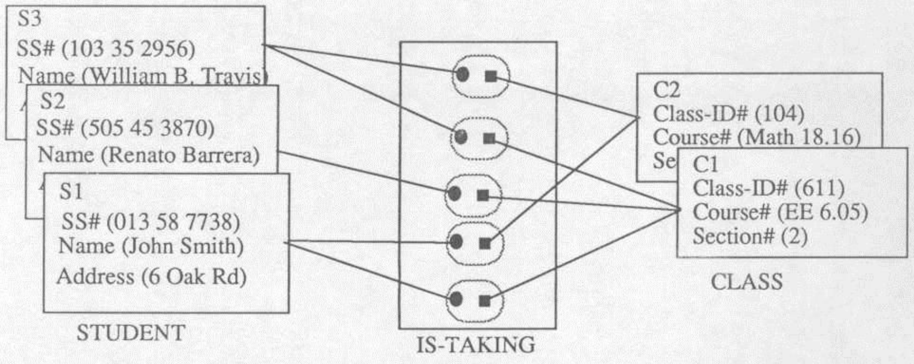


Figure 7• Some data instances for the ER diagram of Figure 6•.
Each element of the relation (table) is-taking is a key (a property that uniquely identifies the item). Example: the SS# of the student and the Class-ID# of the class

In our graphic notation, Figure 7• becomes Figure 8•.

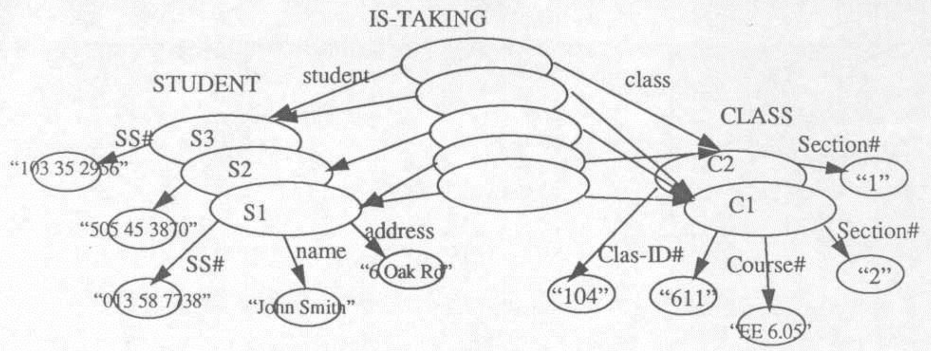


Figure 8• Our graphic representation for the ER diagram of Figure 7•.

When it is not necessary to represent instances, Figure 8• becomes Figure 9•.

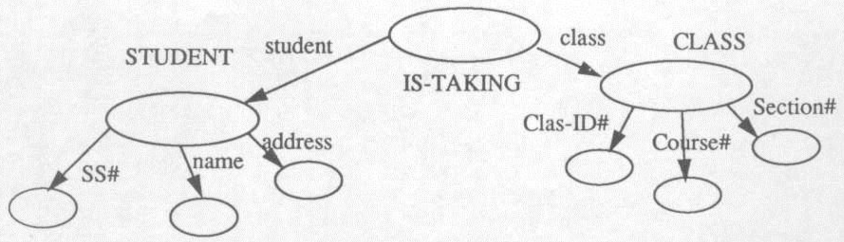


Figure 9• Our graphic representing the ER diagram of Figure 6•, showing no instances.

A manner to simplify Figure 9• is to represent the relation is-taking by a single arrow n (shown **black** in the figure for clarity), so that Figure 9• now becomes Figure 10•

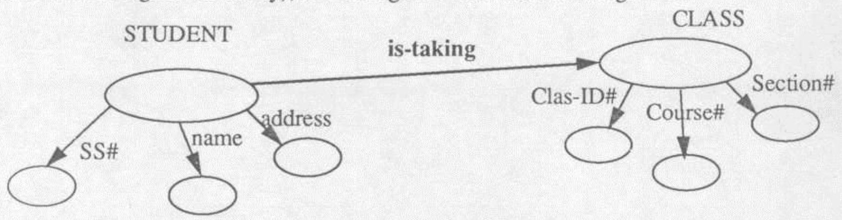


Figure 10• An arrow representing the relation **is-taking** simplifies Figure 9•

As a final example, a more complex ER diagram, extended to contain partitions (+) and unions (∪) is shown in Figure 11•, and its corresponding graphic in our notation is shown in Figure 12•, taken from [ ].

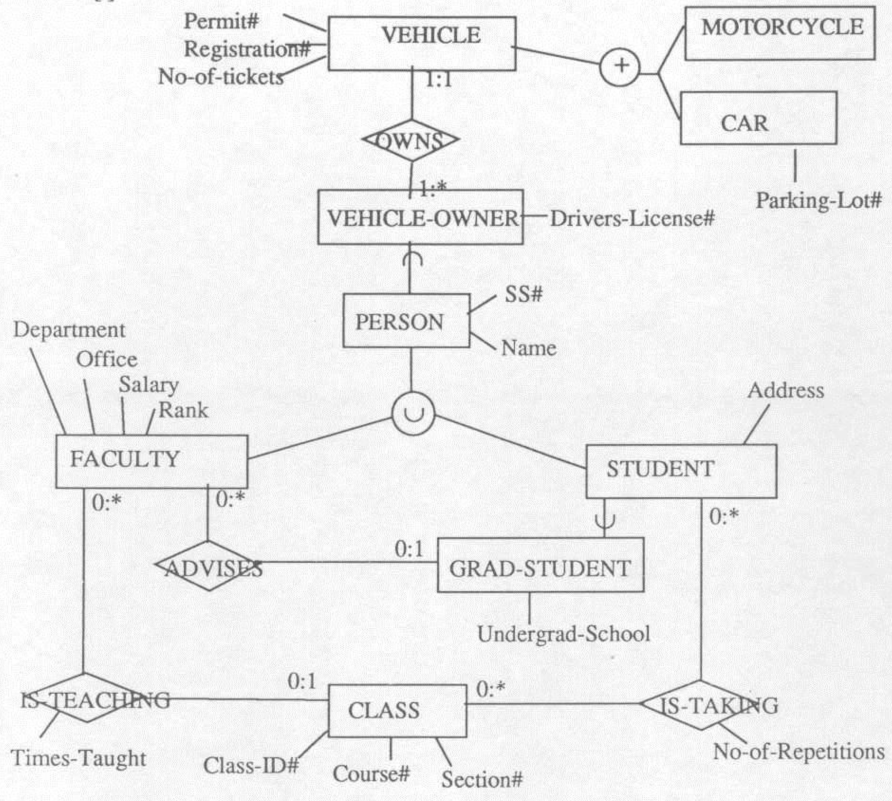


Figure 11• An example of an extended Entity relationship diagram (ECER). An arrow with a ∪ as tip means subset: Vehicle-owner is a subset of Person. Participation constraints of the form (min, max) are represented on each participating entity set. Example: 1:1 in vehicle means that a vehicle participates exactly once in the relationship OWNS, which means that a vehicle must be related exactly to one owner, whereas a VEHICLE-OWNER entity participates one or more times (1:*), which means that a vehicle owner owns at least one vehicle, but possibly more. A circle with an ∪ means union: a person is either a faculty or a student or both; with a + means disjunction: a vehicle is either a motorcycle or a car, but not both.

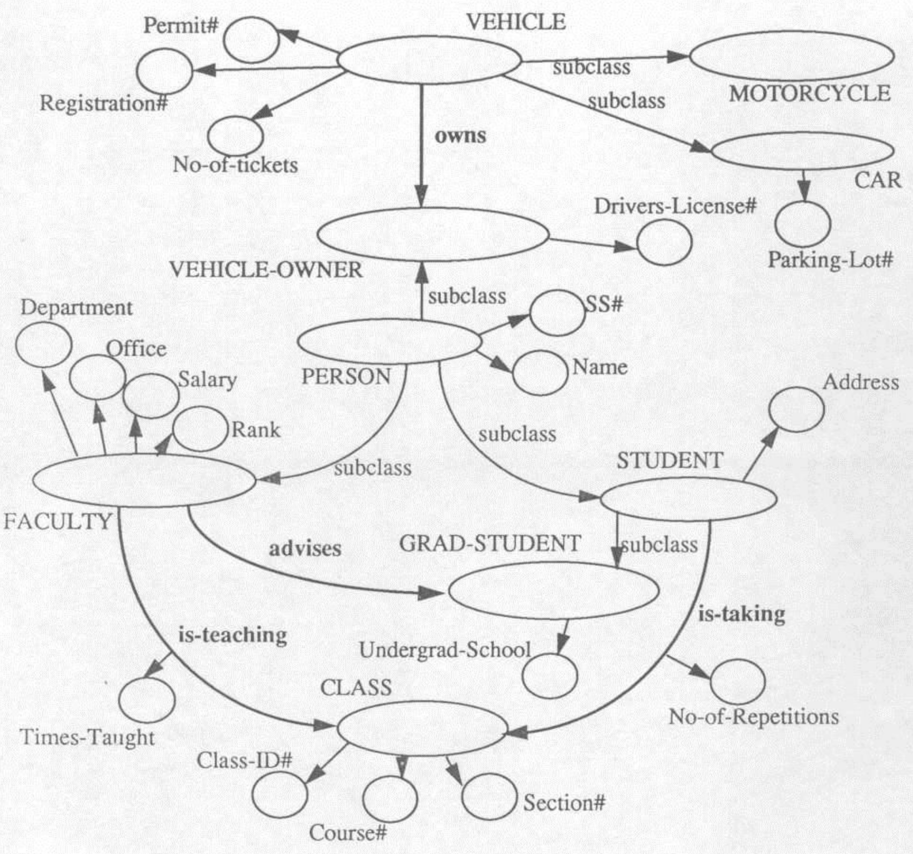


Figure 12• Our graphic representation of the ECER of Figure 11•
We have shown with thick arrows the relations of Figure 11• We prefer to keep the names of the entities (big bubbles) outside, since the bubbles will be used later in the formulation of queries. The relations , + and the participation constraints such as 1:*, are expressed in a form explained in the text.

1•3 Information from Object bases

Class and individual **object**: Represented as an ellipse

Slot: represented as an arrow between ellipses.

Simple slots (properties, attributes): arrow from ellipse to circle

Value (simple value, number, string): circle

Example: Figure 12•

1•4 Information from file systems

Sequential files. Index sequential files. Random access files.

1•5 Information from hybrid storage mechanisms

It is possible to represent with our graphic notation information coming from hybrid sources, such as partly coming from a relational database and part from an object system, say.

II Text representing sets and queries

A query can be associated with a set: the set of elements (of another set) "answering" or "satisfying" the query. Therefore, when we define below (Sec. 2•1•2) how to represent sets, queries will be automatically represented, too.

2•1 Definitions

2•1•1 Dot notation

For e an element (a row of a relation, such as Order13, or an entity, such asSupply13), not a set, and s a property (or slot, such as customer), we define e•s to be the value of that property for that element. Example: Order13•customer is "ZZebra", and Guzmán•age (Figure 13•) is 40. Ocasionally, we overload the meaning of • to represent the singleton set containing that value; we distinguish in context. Example: Guzmán•age is a number in Guzmán•age+4, but is a set in union(Guzmán•age, S2).

2•1•2 Set representation

The following things are considered to be sets, and are represented as such:

2.1.2.1 In the ER model

(a) A relation (a table of a relational database). Example: INCLUDES.

(b) An entity class, such as CUSTOMER.

2.1.2.2 In an object base

(c) A class, such as Person in Figure 13•

2.1.2.3 In our notation

(d) an explicitly enumerated set: {Yin, Guzmán, Jack}, {Blue, Red, Green}, { }, etc. formed by listing (separated by commas or not) the elements inside { and }.

(e) The notation R•s, where R is a set (that is, one of (a), (b), (c) or (d) of Sec. 2•1•2)and s is an attribute or property (such as customer, or name of Figure 3•, for ER models), or a slot (such as wife of Figure 13•, for object bases) denotes the set of r•s with r ε S. Example: If R = {John

Jack}, then R•wife represents the set consisting of {Joe•wife, Jack•wife}, that is, {Mary, Jill}. Ref. to Figure 13•

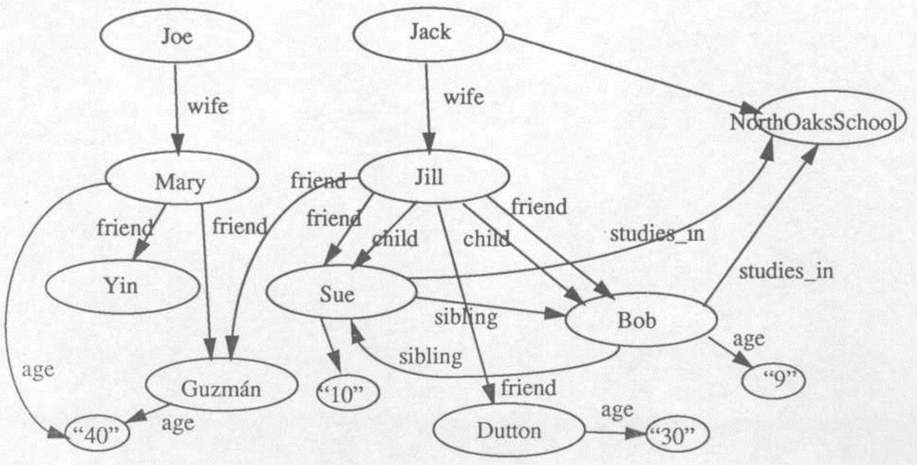


Figure 13• Some relations among certain persons and some objects

Similarly, R•wife•friend = {Mary, Jill}•friend = {Yin, Guzmán, Sue, Dutton, Bob}; R•wife•-child = {Sue, Bob}, etc. Ocassionally, o•s returns a singleton, such as Guzmán•age = {40}, and in this case we also use o•s (overloading its meaning) to denote both that set and its unique element, differentiating using context. Thus, Guzmán•age denotes {40} in union (Guzmán•age, S2),while it denotes 40 in Guzmán•age + 8.

(f) The notation { s s in S | P(s)}, where s is a variable, S is a set and P is a predicate, denotes the subset of elements of S that satisfy P. Example: {s s in {Sue, Bob, Mary} | s•age < 30} is {Sue, Bob}; {s s in Jack•wife•friends | s•studies_in = NorthOaksSchool} = {Sue, Bob}.

(g) We similarly define { f(s) | s in S | P(s)}, a collection of objects of the form f(s).

(h) union (S1, S2 , ...), intersect (S1, S2), set_difference (S1, S2), etc., defined as usual.

2•2 Predicates and functions

2•2•1 Aggregate functions

Arithmetic functions: For e1 and e2 two numbers, we define e1 + e2, *, etc., as usual.

Arithmetic predicates: we also define the predicates e1 < e2, =, ≤, etc.

Arithmetic functions for sets: **avg** (S), **min** (S), **max** (S), **sum** (S), where S is a set of numbers.

Functions on sets: **count** (S) where S is a set. Gives how many elements S has.

groupby (f(e) e **in** S | P(e)) groups set S into several groups (sets, too) g1, g2, ..., each of them having equal values for f(e), which is called the *discriminant* function. Example: **groupby** (e•age e **in union** (Mary, Jill•friend) | e != Yin} groups set {Mary, Guzmán, Sue, Bob} into {{Guzmán, Mary}, {Sue}, {Bob}}, the ages of each group being 40, 10 and 9, respectively.

Predicates from properties: If s holds between o and v, thus: s•o = v, then we use s as a (logical) relation, thus: s(o, v), meaning: property s holds between objects o and v. Example: friend (Mary, Yin). This defines the predicate friend (Mary, x) which is true is x is a friend of Mary.

Membership: For e an object and S a set, we define e ε S in the usual manner.

Universal quantifier: forall (s in S | P(s)) returns True if all elements of S satisfy P.

Existential quantifier: thereis (s in S | P(s)) returns True if at least one element of S satisfies P.

2•2•2 Functions that alter the database

new (Relation) or **new** (Class), returns a new element of that class or relation. Example: new (SUPPLIES).

new_or_exists (Relation, *string*) returns the element having name = *string*, or a new one (to which assigns the name *string*) if needed.

putv (e, property_1: value_1 property_2: value_2 ...) adds value_i under property_i for element (not a set) e. Each property_i is a property, attribute or slot. Often written as {e•property_i, e•property_2} := {value_1, value_2}. Example: **putv** (Guzmán, age: 50, friend: Jack) adds (changes, since age is uni-valued) 50 to Guzmán•age and adds Jack to Guzmán•friend. Example: Guzmán•age = Guzmán•age + 3 increments 3 to Guzmán•age.

delete (o, s, v); **delete** (o, s), **delete**(o), deletes, resp., the value v from o•s, the property s from o, or the object o from its class or table.

2•2•3 Iteration

foreach (x **in** S | P(x)) *function* executes function for each element of S satisfying P. *function* may modify the database.

2•3 Path

(Definition) The path between two bubbles b1 and b2 is the sequence of properties that we find when traveling (in a graph such as Figure 3•) from b1 to b2. We separate each property by the symbol •; moreover, when a property gets crossed along its own direction, it is just written down; if it gets traversed opposite to its own direction, we write it with a $^{-1}$ exponent, meaning "the inverse.". Example: the path from Order13 to Screwdriver12 is order#•$\text{order\#}^{-1}_{\text{ofincludes}}$•item. Example: the path from ZZebra to 118 is customer^{-1}•order#, and the path from 118 to ZZebra is $\text{order\#}^{-1}$•customer.

Given a path from a to b, the *inverse path* is written by reversing the order of the first path.

The paths can be seen as "super-properties" or super-slots, linking any two bubbles no matter how far appart they are, much in the same way that ordinary properties link two immediate bubbles.

2•4 Path Expression

A path expression expresses how to compute a set, using a set (a relation, a table, such as ORDER, or an entity set, or a class) or an element of a relation (a row of such a table) as starting point, and a path to define the new set.

Given a set S and a path p, the path expression S•p denotes the set S•p already defined in 2•1•2 (e), as long as the path p does not contain inverses ($^{-1}$ as exponent). Example: ZZebra•address is "6 Oak St"; Joe•wife•friend = {Yin, Guzmán}.

We need to define now how to compute S•p^{-1}, for instance Joe•wife^{-1} or "Screwdriver12"•item$^{-1}_{ofsupply}$. It is easy to see (refer to Figure 3•and Figure 13•) that S•p^{-1} is obtained by searching the domain set of p and collecting all those elements e where e•p belongs to S. Example: "Screwdriver12"•item$^{-1}_{ofsupply}$ is { s s in supplies | s•item$_{ofsupply}$ = "Screwdriver12"} = {Supply13}.

Example: 40•age^{-1} is {p p in person | p•age = 40} = {Guzmán Mary}, denotes the set of people having 40 years of age. The figures show clearly how to travel graphically these paths.

Example: 40•age^{-1}•friend^{-1}•child = {Guzmán Mary}•friend^{-1}•child = {Mary Jill}•child = {Sue Bob} = "the children of those people who have friends of age 40,"

Example: Order13•order#•order#$^{-1}_{ofincludes}$ = {118}•order#$^{-1}_{ofincludes}$ = {includes13}.

Care must be taken, when using the figures to compute these sets, that they usually (due to space restrictions) do not show *all* the possible properties or values, but only a few representatives or examples. Thus, the above examples are correct assumming that the figures express all the objects in the database.

2•5 Locking

The idea is to have a good set of defaults, so that the normal user does not worry about locking. Each one of these defaults can be over-written and specified by the knowledgeable user.

Every SSDL bubble is a transaction.

2•5•1 Other issues

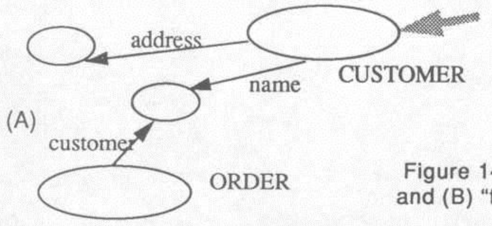
Rollback.

Full objects. ERA explanation. Inheritance in presence of relational databases. Add more flexibility.

III Graphs representing queries

Chapter II explained how to represent sets, including those normally represented by "queries," by using *paths* (Sec. 2•3) or *path expressions* (Sec. 2•4). This chapter describes how to draw or graphically express arbitrary queries. The person formulating a query does so by adding a few signs on top of the company information graph (refer to Figure 3•). A query is always expressed over a "primary" set (a long ellipse): "give me all the employees that ..." has as primary set employees, etc. The user denotes the primary set by placing a fat arrow in its ellipse (in the company information graph of Figure 3•), as shown in Figure 14•. It means: *the answer is among these.*

Sometimes you want to find certain properties (say, names or addresses) which are circles, not big ellipses (i. e., they are not entities in the ER model). You must ask yourself: the name of whom? Or: the address of whom? "Of customers" is the answer. Then, place the arrow on the ellipse (set) customer. Examples: Sec. 5•4•1, Sec. 5•4•2.



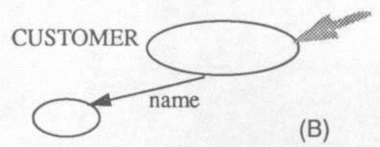


Figure 14• (A) "Find all customers that ...", and (B) "find all the names that ..."

3•1 "Drawing queries on the company information graph

Remember that the "company information map" is just another name for the schema of the database (Ref. Figure 3•).

3•1•1 Representing the primary set (or domain) of a query

The primary set of a query is denoted by a fat arrow, (Figure 14•), and represents the domain of the query: it is the set of which an appropriate subset will be the answer to the query. If you want as anwer some strings (Names, say) or numbers (ages, say), you still want to specify as primary set the set (customer, say) of which you want these names or ages.

3•1•2 Representing predicates

3.1.2.1 Representing the existence or inexistence of an attribute

"a customer having an address" is represented by count>0 inside the circle for address; "a customer having no address" is represented by count=0 inside the circle for address See Figure 15•

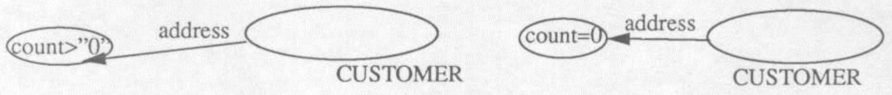


Figure 15• Customers with some address; customers without address.

3.1.2.2 Predicates comparing to a constant

(a) With equality. To represent the predicate attribute = constant, we write the constant inside the bubble receiving the arrow corresponding to the attribute (see figure below). That bubble is unique, if the attribute is unique. If that is not the case (for instance, the property name could represent either $name_{ofcustomer}$ or $name_{ofsupplies}$), we take a look to the set (relation) to which it qualifies, in order to disambiguate the property. We ask ourselves: the *name* of who?.

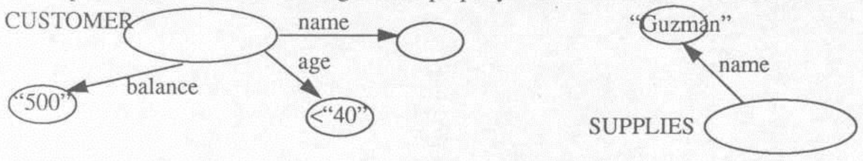


Figure 16• Representing predicates comparing with a constant
We show balance=500, age<40, and $name_{ofsupplies}$=Guzmán.

(b) To represent predicates having a relation other than equality, such as age < 40, write it: "< 40", inside the small bubble receiving the arrow age. Notice that "500" is a shortcut for ="500" .

3.1.2.3 Predicates comparing two attributes

attr1 rel attr2, for instance, balance = age, is represented by drawing a comparison arrow (which looks just like an arrow representing a relation) between attr1 and attr2, and by writing the predicate on such arrow. See Figure 17•.

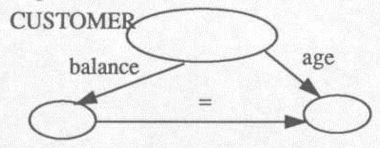


Figure 17• Predicate balance = age

3•1•3 Representing functions

3.1.3.1 Functions of one argument

Functions of one argument are represented by a function arrow (which looks just like the arrow representing attributes) labeled with the function name, going to a new small circle. Example: age+3 in Figure 18• Notice that +3 is a shortcut for the unary function f(x) = x+3. Now, it is easy to see (Figure 18•) how to represent the predicate sqrt (balance) < age + 3.

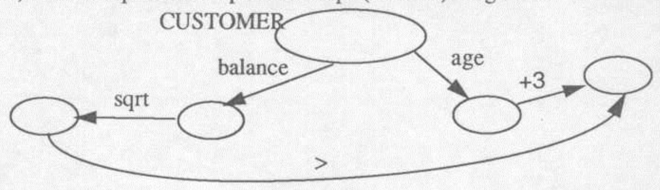


Figure 18• Predicate sqrt (balance) > age + 3

Alternatively, the function could be typed inside an small circle, as in (a) or (b) of Figure 19•.

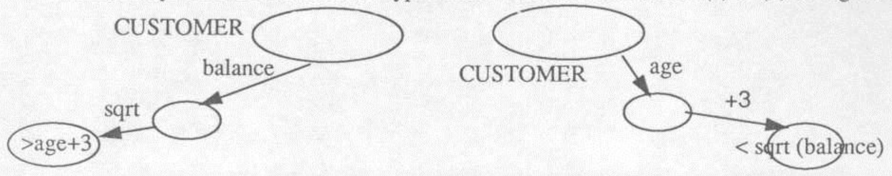


Figure 19• (a) and (b) show two other variations to sqrt (balance) > age+3 of Figure 18•.

3.1.3.2 Functions of several arguments

These can be expressed with the help of the set notation (Section 2•1), and can be signaled by adding a new arrow to the relation it qualifies, with a temporary or missing name. Example: Suppose we want to compute the function f(balance, age) = 3*balance + 2*age, we draw a new circle next to customer, type the formula inside as 3*customer•balance + 2*customer•age, and add the arrow labeled f, as shown in Figure 20•

These functions are restricted to have as arguments properties of the same set. The function f will not be added (to the database) as a new attribute of the set customer, unless we use a fat + sign (Sec. 5•7).

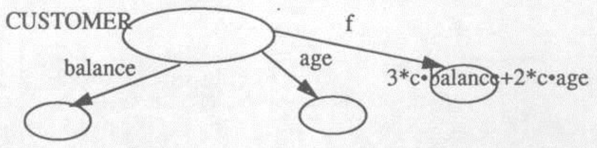


Figure 20• Computing a function with several arguments

3.1.3.3 Aggregate functions

The functions avg, max, min, sum and count are already defined and available for use as labels on a function arrow. Example: Figure 21• represents query "those supplies with price less than half the maximum balance."

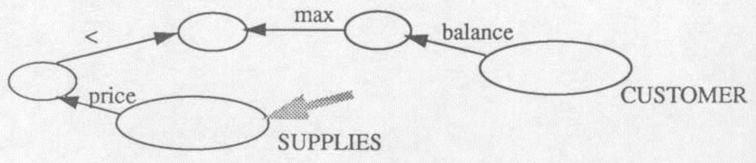


Figure 21• Use of aggregate functions

3•1•4 Representing the targets (or ranges) of a query

The targets of a query are the bubbles (entities, relations) or small circles (constants, values) that we wish to find; they are the range or ranges of a query. They are denoted by shading. For example, Figure 22• represents the query "those supplies with price les than 50", where we shade supplies because we want to find supplies. If we wish to know, in addition to supplies, the item of the supplies, we should shade item, too.

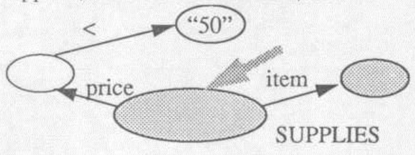


Figure 22• The targets of a query are denoted by a shading

Notice that, when you say "I need those elements of set S having ...", such as "those customers having...", then, the primary set and one of the targets coincide; in the drawing you see the fat arrow pointing to a shaded circle or ellipse. If "you need the age of those customers having ...," you shade the age and place a fat arrow on customer.

3•1•5 Predicates that require duplication of bubbles

Some queries require duplication of relations; for instance, "those customers having balance bigger than the balance of Guzmán," as expressed in Figure 23•.

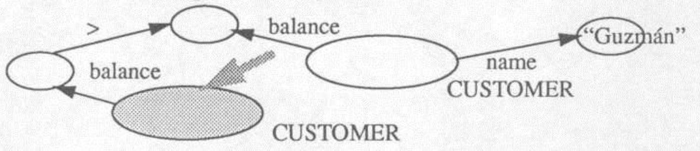


Figure 23• Customers with balance bigger than Guzmán's balance to create that car, then we need to add another fat cross (the middle one) on top of the circle representing the car. Finally, if it was necessary to paint it blue, we add the third fat cross on color, signifying the new *color* of the car is blue. We do not add a cross to "blue", since that would mean that we are creating a new color, and naming it "blue."

IV Graphs representing sets and constraints

We saw in Chapter III how graphs are used to represent queries. Now we will see how to use graphs to represent other useful concepts.

4•1 Graphs representing sets

By considering a query as denoting the set that produces as answer, the graph for that query can also be thought to represent the "answer" set. Example: Figure 24• represents the set "students speaking languages that Guzmán does not speak."

4•2 Graphs representing constraints

A constraint is a relation that holds among certain objects or sets of objects. For instance "Black men should make less money than White men," interpreted to mean "Every Black person should make less money than any White person." The trick is to keep as "constants" all the objects mentioned in the query, except one, which is a set. Translate the above query to "The set of Black men making less money than White men." Transform it in such a way that such set should be always empty if the constraint is satisfied. In our example: "The set of all Black men making *more* money than White men." Now, that is a set, and, following Section 4•1, can be represented by a graph (Figure 26•). We interpret that the constraint is satisfied if the graph always returns an empty set, otherwise the constraint is violated.

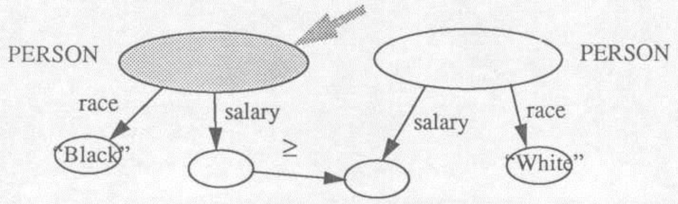


Figure 26• All Black men should make less salary than White men. Notice the ≥ sign.

This manner to validate constraints is "global", and should be compared to an "incremental" way to validate if a constraint holds in a database: each time a person changes (or obtains for fist time) a salary, verify the constraint *only with respect to him*, since it is assumed that the database is already in a consistent state, i.e., the constraint was already holding. This incremental way should be preferred, but we will not elaborate here further.

There may be more than one manner to reduce a constraint to a set. For the last example, another graph could be .Figure 27•, which represents the set of White men making *less* money than Black men.

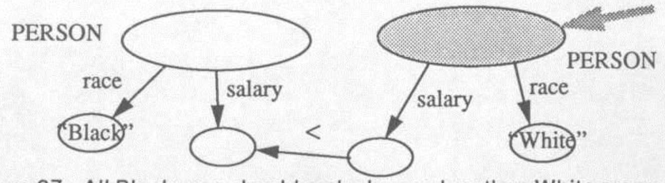


Figure 27• All Black men should make less salary than White men; variant. Notice the relation < going from right to left, saying "salary of White should be less than salary of Black.

4•2•1 More examples of constraints

4.2.1.1 Referring to Figure 12•

(a) A graduate student can not take a course that he has taken before.

(b) No faculty member may have a degree other than a Ph.D.

(c) An undergraduate student can not be registered in a graduate course.

4.2.1.2 Relating several sets, Figure 12•

(a) A graduate student who is a Research Assistant must have an advisor.

(b) A faculty member who is not an advisor of any graduate student must teach at least one section.

(c) Every section must have some courses.

(d) Teaching Assistants whose major is in the Department of Electrical Engineering can not teach courses that belong to other departments.

(e) Teaching Assistants can only teach courses that belong to the Department that they (the TA's) belong.

(f) The GPA of a graduate student majoring in any Deparment of the College of Engineering muyst be > 3.3. Also, if such a graduate student is a Teaching Assisant, he must have an advisor.

V Examples

In these examples, we give the following representations:

(a) SQL, a traditional query language for databases;

(b) sequence operation, defined by Yin [ ];

(b•1) the graph associated or representing that query, as described in Chapter III;

(b•2) the path, obtained by traveling from the start (an arrow) to the primary set (a filled-in ellipse), as described in Sec. 2•3;

(c) the path expression, which can be mechanically (i. e., automatically) derived from (c), as described in Sec. 2•4.

It is useful to remember that a graphic query can be interpreted as: "From the set that has the fat arrow, find those elements satisfying the relations written inside some ellipses. From the answer, give me those values shown shaded." For example, see Figure 28•

5•1 The select statement of SQL

5•1•1 Example 1

Select the name, address and balance of customers with negative balances from the relation customers.

(a) SQL:

select name, addr, balance

from customers

where balance < 0

(b) Sequence operation:

{ {x.name, x.addr, x.balance} | x **in** customers **where** x.balance < 0}

(1) Corresponding graph (See Figure 28•):

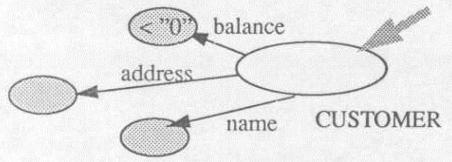


Figure 28• Find addresses, names and balances of customers with negative balance.

How to draw the above graph: You start inside Figure 2•. You start at CUSTOMER, because you ask yourself: from which "primary set" I am going to answer? "From customers" You end in balance, address and name because that's what the query wants you to find.

(2) Path corresponding to graph (1): You start from the start (fat arrow) and walk towards the ends (shaded ovals). The answers are:

c•balance | c•balance < 0

c•address

c•name

where you know that c is in customers.

(c) Path expression corresponding to (2):

{ {c•balance, c•address, c•name} c in Customers | c•balance < 0}

Comparing with (b), they are identical.

5•1•2 Example 2

Select the supplies of the items Zack Zebra ordered.

(a) SQL:

select supplies.name

from orders, includes, supplies

where orders.cust = "Zack Zebra" **and** orders.o# = includes.o#

and includes.item = supplies.item

(b) Sequence Operation

{s.name | s **in** supplies **where**

exist (i | i **in** includes **where** s.item = item

and exist (o | o **in** orders **where** o.o# = i.o# **and** o.cust = "ZackZebra"))}

(1) Corresponding graph. We want to go from ZackZebra to the name of the includes (See Figure 29•):

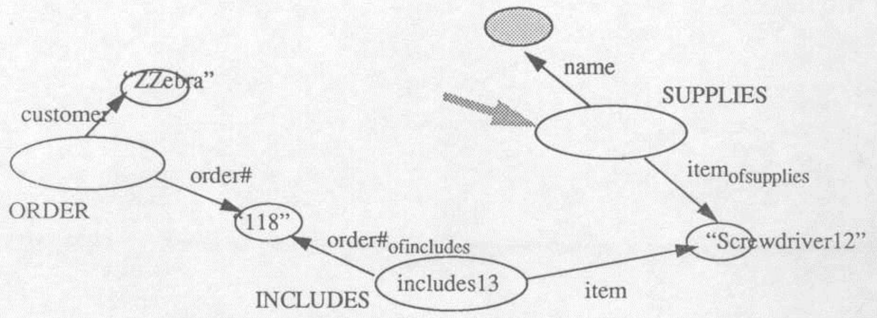


Figure 29• Find the names of the supplies ordered by customer ZZebra.

(2) Path corresponding to graph (1):

ZZebra•customer^{-1}•order#•order#$^{-1}_{ofincludes}$•item•item$^{-1}_{ofsupplies}$•name, obtained by mechanically walking from source (in-arrow) to sink (shaded ovals) in above figure.

(c) Path expression corresponding to path (2): In what follows we go step by step mechanically deriving the path expression from above path (refer to last figure):

The path expr corresp to Z.Zebra•customer$^{-1}_{oforder}$ is {o o **in** orders | o• customer ="ZZebra"}

The path expr corresp to above•order# is { o•order# o **in** orders | o•customer = ZZebra}, call the set S1 and n an element of it.

The path expr corresp to above•order$^{-1}_{ofincludes}$ is:

(a) the includes having number n = o•order# are { i i **in** includes | i•order#$_{ofincludes}$ = n}.

Thus, the path expr corresp to above•order$^{-1}_{ofincludes}$ is:

{i i **in** includes | i.order#$_{ofincludes}$ ε S1}

The path expr corresp to above•item is {i•item i **in** includes | i•order#$_{ofincludes}$ ε S1}

The path expr corresp to above•item$^{-1}_{ofsupplies}$ is:

(a) the supplies having m = i•item are {s s **in** supplies | s•item = m}, call it S2.

Thus, the path expr corresp to above•item$^{-1}_{ofsupplies}$ is:

{s s **in** supplies | s•item ε S2}

The path expr corresp to above•name is {s•name s in supplies | s•item ε S2}.

Comparing with (b), we see that our expression saves considerable searching over (b).

5•2 Tuple Variables

5•2•1 Example 3

Select the names and addresses of customers whose balance is less than that of Judy Giraffe.

(a) SQL

select c1.name, c1.addr

from customers c1, customers c2

where c1.balance < c2.balance **and** c2.name = "JudyGiraffe";

(b) Sequence Operation

{{c1.name, c1.addr} | c1 **in** customers **where**

exist (c2 | c2 **in** customers **where** c1.balance < c2.balance

and c2.name = "JudyGiraffe"}

(1) The corresponding graph can be seen in Figure 30•

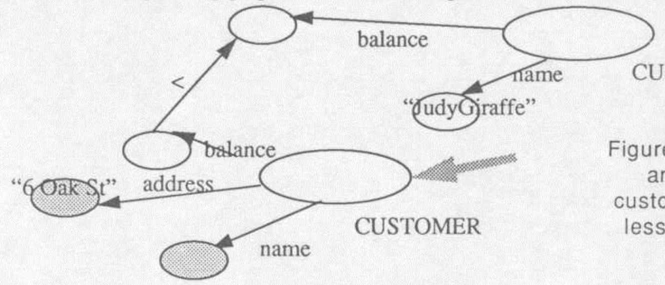


Figure 30• Find the name and addresses of customers with balance less than that of Judy Giraffe.

(2) Paths corresponding to (1): JudyGiraffe•name^{-1}•balance•<$^{-1}$•balance$^{-1}_{\text{ofcustomer}}$•name, and •balance, and •address.

(c) Path expression corresponding to path (2):

The path expr corresp to JudyGiraffe•name^{-1} is {c c **in** Customer | c•name = "JuddyGiraffe"}

The path expr corresp to above•balance is {c•balance c **in** Customer | c•name = "JuddyGiraffe"}; call this set S1. We notice that c•balance gives a unique answer, we will make use of this later.

The path expr corresp to above•<$^{-1}$ is {n n **in** Number | n < b & b ε S1}. But S1 has only one answer, hence the path expr can be written as {n n **in** Number | n < S1}; this is an abuse of the notation for <. Let us also call this set S2. Note: S2 is an infinite set of numbers, all of them less than the balance of Judy Giraffe. We will have to eliminate later that cumbersome set S2.

The path expr corresp to above•balance$^{-1}_{\text{ofcustomer}}$ is {c c **in** Customer | c•balance = m & m ε S2}. Since we know that < only holds among numbers, this expr can be rewritten as {c c **in** Customer | c•balance < S1} and we got rid of N2.

The path exprs corresp to above•name, above•balance, and above•address, are

{c•name c **in** Customer | c•balance < S1}

{c•balance c **in** Customer | c•balance < S1}

{c•address c **in** Customer | c•balance < S1} which could be conveniently grouped as

{{c•name, c•balance, c•address} c **in** Customer | c•balance < S1}.

Comparing with (b), we do not compute the double loop that (b) computes.

5•3 Pattern Matching

5•3•1 Example 4

Select those items from supplies, and the item names begin with "E".

(a) SQL

select item

from supplies

where item **like** "E%".

(b) Sequence Operation

{item | item **in** supplies **where** item.name = "E*"} where the * is a wild character that matches any (zero or more) characters.

(1) Figure 31• shows the corresponding graph

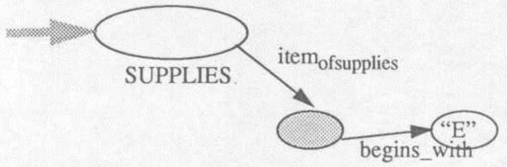


Figure 31• Find those items of supplies that begin with capital E.

(2) Path corresponding to graph (1): s | begins_with (s, "E"). We have added an arrow with the predicate (Sec. 3•1•2) begins_with ("E").

(c) Path expression corresponding to path (2): {s•item s **in** supplies | begins_with (s•item, "E")}

Comparing with (b), we notice an error in (b), since items is not a relation, and it does not contain a "field" called name.

5•3•2 Example 5

Select those orders from Order, the order numbers are in the range 1000 to 1999, assume that order numbers are stored as character strings rather than integers.

(a) SQL

select *

from order

where o# **like** "1_ _ _"

(b) Sequence Operation

{o | o **in** order **where** o.o# = "1_ _ _"}

(1) Corresponding graph (we give two variants, see Figure 32•):

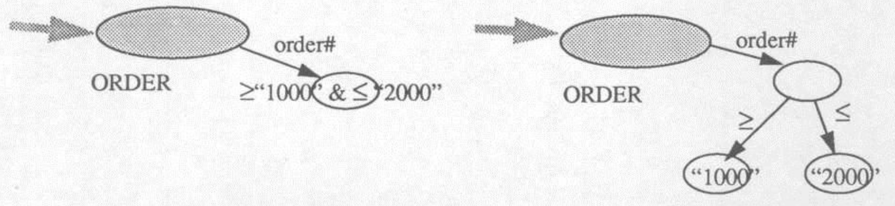


Figure 32• Give me those orders with numbers between 1000 and 2000, two variants

(2) Paths corresponding to graph (1): o | o.order# ≤ "2000"; o.order# ≥ "1000"

(c) Path expression corresponding to path (2): {o o **in** Order | o•order# < "2000" & o•order# > "1000"

Comparing with (b), (c) is equivalent to it.

5•4 Set Operations

5•4•1 Example 6

Select items from supplies when the item price is as large as any appearing in supplies.

(a) SQL:

select item

from supplies

where price >= **all** (**select** price **from** supplies);

(b) Sequence Operation

{s.item | s in supplies **where all** (o | o **in** supplies **where** s.price ≥ o.price)}

(1) The corresponding graph can be seen in Figure 33•

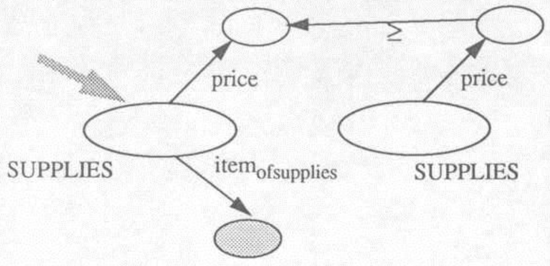


Figure 33• Those items of supplies having the largest price appearing in supplies.

(2) Path corresponding to graph (1): s•item$_{ofsupplies}$ | s•price ≥ o•price

(c) Path expression corresponding to path (2): {s•item s **in** supplies, o **in** supplies | s•price ≥ o•price}

Comparing with (b), both make a cartesian product search through supplies. An smarter solution would be p = **max** {s•price s **in** supplies}; answer is {s•item s **in** supplies | s•price ≥ p}, which only makes two single searches through supplies.

5•4•2 Example 7

Select all items ordered by Ruth Rhino

(a) SQL

select item

from includes

where o# = (**select** o# **from** orders **where** cust = "Ruth Rhino")

(b) Sequence Operation

{i.item | i in includes where exist (o |o **in** order where o.o# = i.o# and o.customer = "Ruth Rhino")}

(1) The graph corresponding to the above sequence operation appears in Figure 34•

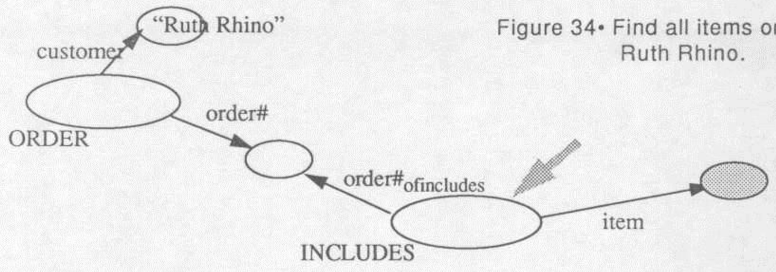


Figure 34• Find all items ordered by Ruth Rhino.

(2) Path corresponding to graph (1): RuthRhino•customer^{-1}•order#•order#$^{-1}_{\text{ofincludes}}$•item

(c) Path expression corresponding to path (2):

The path expr for RuthRhino•customer^{-1} is {c c **in** customer | c•customer = "RuthRhino"}; this is the set of customers having RuthRhino as their name.

The path expr for above•order# is {c•order# c **in** customer | c•customer = "RuthRhino"} = S1.

The path expr for above•order#$^{-1}_{\text{ofincludes}}$ is: {i i **in** includes | i•order#$_{\text{ofincludes}}$ ε S1}.

The path expr for above•item is {i•item i **in** includes | i•order#$_{\text{ofincludes}}$ ε S1}.

Compare with (b).

5•5 Aggregate Operators

5•5•1 Example 8

Compute the average balance of customers and the total number of distinct suppliers.

(a) SQL

select avg (balance) **from** customers;

select count (**distinct** name) **from** supplies.

(b) Sequence Operation

avg (c.balance | c **in** customers);

count (distinct (s.name) | s **in** suppliers)

(1) Corresponding graph: see Figure 35•; we add (draw) the functions **avg** and **distinct**.

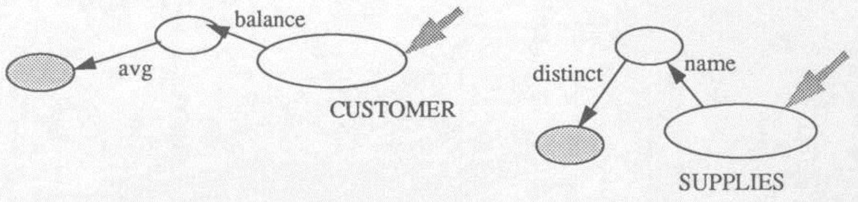


Figure 35• Graphs corresponding to two queries.

(2) Path corresponding to graph (1): c•balance; the other is s•name

(c) Path expression corresponding to path (2): **avg** { c•balance c **in** customer}; the other is

count {**distinct** {s•name s **in** supplies}} or **count_distinct** {s•name s **in** supplies}. **count_distinct** is not needed since the arguments are sets; but we will generalize this later to sequences (which can have repeated elems), and then it will be needed.

Comparing with (b), they are identical.

5•5•2 Example 9

Count how many suppliers sell Brie.

(a) SQL **select count** (name) **from** suppliers **where** item = "Brie"

(b) Sequence Operation

count (s.name | s **in** suppliers **where** s.item = "Brie")

(1) Corresponding graph: refer to Figure 36•; we drew the function **count** to supplies.

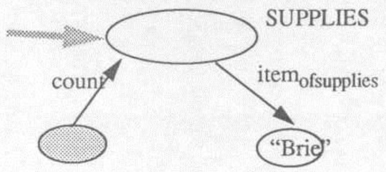


Figure 36• How many suppliers sell "Brie"?

(2) Path corresponding to graph (1): Brie•item-$1_{ofsupplies}$

(c) Path expression corresponding to path (2): {s **in** suplies | s•item = "Brie"}, and we have to count them, thus: **count** {s **in** suplies | s•item = "Brie"}.

Comparing with (b): the expression s.name is counted in (b), while (c) counts s.

5•6 Aggregation by groups

5•6•1 Example 10

Select all the items and their average prices

(a) SQL

select item, avg(price)

from supplies

group by item;

(b) Sequence Operation

{ {s.item, avg (s.price)} | s **in** group (s.item | s **in** supplies)}

(1) Figure 37• shows the corresponding graph; function **average** is drawn pointing to price.

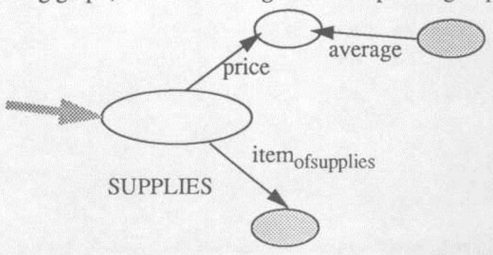


Figure 37• Find all the items and their average prices.

(2) Path corresponding to graph (1): s•price•average-1, grouped by item.

(c) Path expression corresponding to path (2): To group supplies by item, we have:

G1 = **groupby** (s.item s in supplies).

If g ε G1, then g is a group of supplies; the thing being equal is g•item. We want to compute average (s•price), where s ε g. The expression to find the set of average prices is

{**average** (s•price s **in** g) | g **in** G1}, which has a price for every group g. Since we also want the items, the final answer is { {g•item, **average** (s•price s **in** g)} | g **in** G1} which gives a set of tuples, each of the form {"Screwdriver12", $2.50}.

Compare with (b).

5•6•2 Example 11

Select those items that were sold by more than one supplier

(a) SQL

select item, avg (price)

from supplies

group by item

having count(*) > 1;

(b) Sequence Operation

{ {s.item, avg(s.price)} | s **in** group (s.item | s **in** supplies) **where count** (s) > 1}

(1) Corresponding graph: refer to Figure 38•

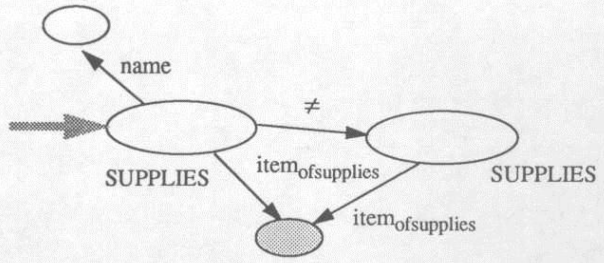

name
≠
SUPPLIES
$\text{item}_{\text{ofsupplies}}$
SUPPLIES
$\text{item}_{\text{ofsupplies}}$

Figure 38• Find those items sold by more than one supplier.

(2) Paths corresponding to graph (1): s•name; s•item; s2•item; constraints: s•item = s2•item, s ≠ s2.

(c) Path expression corresponding to path (2):

{s•name s, s2 **in** supplies | s•item = s2•item & s ≠ s2}

Comparing with (b), we see that (b) avoids the cartesian product search by use of the group operation.

5•7 Insertion

5•7•1 Example 12

Insert into supplies values ("Ajax", "Escargot", 0.24)

(a) SQL **insert into** supplies **values** ("Ajax", "Escargot", 0.24)

(b) Sequence Operation

next (supplies) := {"Ajax", "Escargot", 0.24}

(1) Corresponding graph (Figure 39•). Add word **new** inside bubble supplies, to indicate that you want to create one element, and add three things to it. Otherwise, you would add to every element of supplies the name "Ajax", the price "0.24" and item "Scargot", which is not what you want.

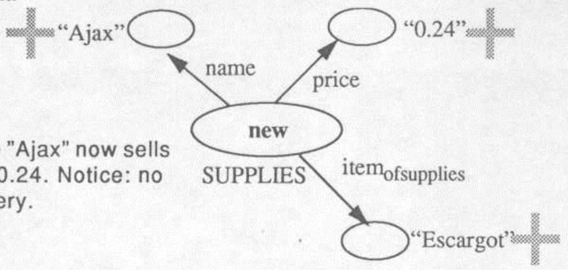


Figure 39• Store "Ajax" now sells "Escargot" at $0.24. Notice: no query.

(2) Path corresponding to graph (1): s•name = "Ajax"; s•item = "Escargot"; s•price = 0.24 & s = **new** (supplies).

(c) Path expression corresponding to path (2): s := **new**(supplies); s•name := "Ajax"; s•item := "Escargot"; s•price := 0.24. The above creates always a new supplier. If that is not desirable, say s := **new_or_exists** (supplies, name: "Ajax"); **putv** (s, item: "Escargot", price: 0.24).

Comparing with (b), (b) has the different values identified by position.

5•7•2 Example 13

insert into supplies (name, price) values ("Ajax", 0.24)

(a) SQL **insert into** supplies (name, price) **values** ("Ajax", 0.24);

(b) Sequence Operation

next (supplies.name, supplies.price):= {"Ajax", 0.24}

(1) Corresponding graph (Figure 40•). We write **new** inside bubble supplies

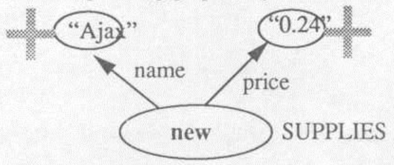


Figure 40• There is a new item (product) that store Ajax sells at $0.24 (we don't know the name of this item)

(2) Paths corresponding to graph (1): s•name; s•price

(c) Path expression corresponding to path (2): s :=**new_or_exists** (supplies, name: "Ajax"); **putv** (s, price: 0.24).

Compare with (b).

5•7•3 Example 14

Create a new relation acme_sells(item, price) containing the item and price components of the supplies with name equal to "Acme."

(a) SQL

```
insert into acme_sells
select item, price
from supplies
where name = "Acme";
```

(b) Sequence Operation

```
foreach (x in suppliers where x.name = "Acme")
    next (acme_sells.item, acme_sells.price) := {x.item, x.price};
```

(1) We can see in Figure 41• the corresponding graph

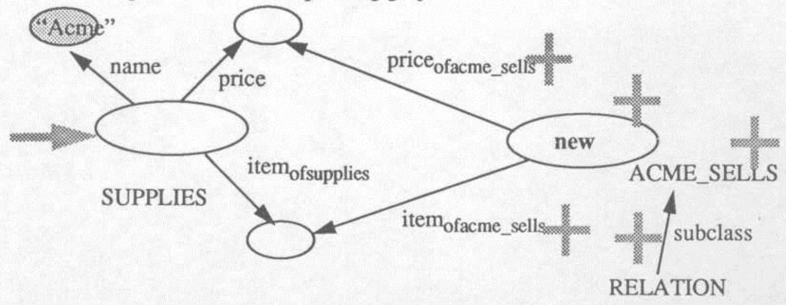


Figure 41• A new relation, acme_sells, is created, containing the item and price of the supplies with name equal to "Acme".

(2) Path corresponding to graph (1): s•name = "Acme"; s'•price := s•price;

s'•item$_{ofacme\_sells}$:= s•item$_{ofsupplies}$.

(c) Path expression corresponding to path (2):

```
r := **new_or_exists** (relation, name: "Acme_Sells");
**foreach** (s in supplies | s•name = "Acme")
    s' := **new_or_exists** (r, item: s•item);
    **putv** (s', price: s•price)
```

Comparing with (b), they are similar.

5•8 Deletion

5•8•1 Example 15

Delete all orders that include Brie

(a) SQL

```
delete from orders
where o# in (select o# from includes where item = "Brie")
```

(b) Sequence Operation

```
delete (o in orders where
    exist (i | i in includes where i.o# = o.# and i.item = "Brie"))
```

(1) The graph is shown in Figure 42•

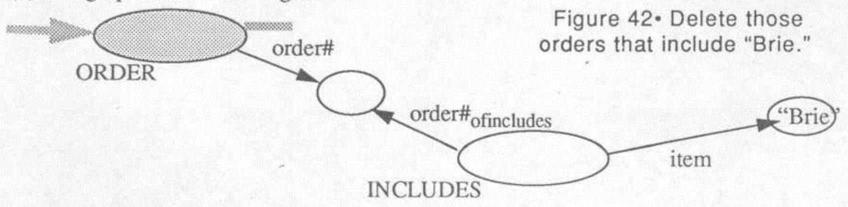


Figure 42• Delete those orders that include "Brie."

(2) Path corresponding to graph (1): **o•order#•order#**$^{-1}$$_{\textbf{ofincludes}}$•item = "Brie".

(c) Path expression corresponding to path **(2)**: We will compute the path expression correspoinding to path Brie•item^{-1}$_{\textbf{ofincludes}}$•order#$_{\textbf{ofincludes}}$, and to o•order#.

(i) The path expr for "Brie"•item^{-1}$_{\textbf{ofincludes}}$ is {i i **in** includes | i•item = "Brie"}.

(ii) The path expr for above•order#$_{\textbf{ofincludes}}$ is {i•order#$_{\text{ofincludes}}$ i **in** includes | i•item = "Brie"}. Call this set S1; the set of order numbers whose item is Brie.

(iii) For the second path, the path expr is {o•order# o **in** order}. Since it must be equal to (ii), we have as final answer foreach (o |o•order ε S1) **delete** o.

Comparing with (b), (c) avoids recomputation of the set S1.

5•8•2 Example 16

Supplies, Acme, no longer sells Perrier. Delete Acme from supplies.

(a) SQL

delete from supplies

where name = "Acme" **and** item = "Perrier";

(b) Sequence Operation

delete (s **in** supplies **where** s.name = "Acme" **and** s.item = "Perrier")

(1) Corresponding graph in Figure 43•

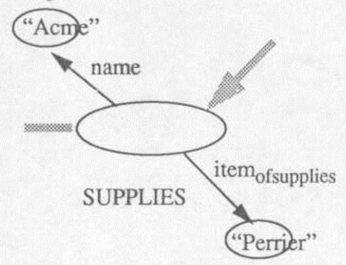


Figure 43• Supplier "Acme" no longer sells "Perrier." Delete Acme as a supplier of Perrier.

(2) Paths corresponding to graph (1): s•name = "Acme"; s•item = "Perrier"

(c) Path expression corresponding to path **(2)**:

foreach (s s **in** supplies | s•name = "Acme" **&** s•item = "Perrier")

delete (s)

5•9 Update

5•9•1 Example 17

Change the price Acme charges for Perrier to $1.00 in supplies.

(a) SQL

update supplies

set price = 1.00
where name = "Acme"

(b) Sequence Operation

foreach (s **in** supplies **where** s.name = "Acme")
s.price = 1.00;

(1) Corresponding graph (Figure 44•)

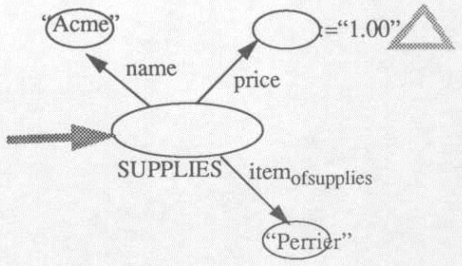


Figure 44• The new price of "Perrier" in "Acme" is $1.00

(2) Paths corresponding to graph (1): s•name = "Acme" & s•item$_{ofsupplies}$ = "Perrier";

s•price.

(c) Path expression corresponding to path (2):

foreach (s s **in** supplies | s•name = "Acme" & s•item$_{ofsupplies}$ = "Perrier")

s•price := 1.0

Comparing with (b), (b) lacks reference to "Perrier."

5•9•2 Example 18

Lower all of Acme's prices by 10% in supplies.

(a) SQL

update supplies
set price = price * 0.9
where name = "Acme"

(b) Sequence Operation

foreach (s **in** supplies **where** s.name = "Acme")
s.price = s.price * 0.9

(1) Corresponding graph (Figure 45•)

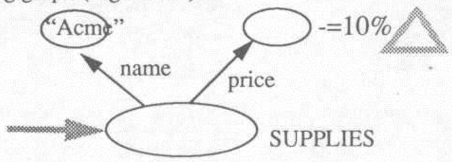


Figure 45• "Acme" has reduced its prices (in supplies) by 10 %.

(2) Paths corresponding to graph (1): s•name = "Acme"; s•price

(c) Path expression corresponding to path (2):

```
        foreach (s s in supplies | s•name = "Acme")
            s•price := s•price * 0.9
```

Compare with (b).

5•10 Other questions and their answers

5•10•1 Possible to pose and answer now

(a) Collect persons who are teachers but not Faculty members

(b) Retrieve all persons. In addition, if a person is a graduate student who is also a Teaching Assistant with a gpa > 3.5, retrieve his Major and Gpa also.

(c) Retrieve the ss# and name for all faculty members. And, if the faculty member is advising a graduate student who is a TA and a RA, retrieve the advising and student information.

5•10•2 Impossible or difficult to pose or answer now

Questions like these can not be answered with the proposed graphic facility. A truly intelligent query facility should provide, like the examples below shown, in addition to the answer to the question, some example supporting the answer.

(a) Who protects plants and animals?

The U.S. Fish and Wildlife Service and the National Marine Fisheries Service of the U.S. Government have the job of seeing that endangered species law is carried out.

There are also people who help carry out state laws that protect fish and wildlife in their states.

There are many groups and private citizens who are intrested in protecting endangered species, too.

(b) Why do plants and animals become endangered or extinct?

(1) **Loss of habitat:** the loss of habitat, or a place to live, is the main reason some species are in trouble.

(2) **Pollution:** Harmful chemicals kill plants and animals. Example: The bald eagle is still on the endangered list, but it is making a comeback. One of the reasons it was in trouble was because of the widespread use of a chemical called DDT. The banning of the chemical has helped to make the bald eagle's habitat safer.

(3) **Can't adjust or adapt:** some animals eat only certain foods. If something happens to this food source, they are in trouble

(4) **Predators:** When animals from other areas are brought into a new area, they sometimes do a lot of harm.

(5) **Diseases:** Diseases often kill native plants and animals.

(c) Why save endangered species?

(1) **Special and different:** every type of animal or plant is different. If we lose one species, we have lost a model that can never be replaced.

(2) **Balance of nature:** Many plants and animals depend on each other to survive. If we lose one thpe of animal or plant, then the whole system might be upset ot thrown off balance.

(3) **Science:** By destroying a type of plant or animal, we give up forever the chance to learn how valuable it might be.

(4) **Respect:** Many people think that we should respect all forms of life. They think that these living things have a right to live.

(5) **Agriculture:** We depend on plants and animals for our food. Our diet is made up of about 20 plants. However, there are 80,000 plants that are edible. If one vanishes, we might have lost a valuable food source for the future.

(d) Which is the best? (Of two insurance plans, say).