Chapter 4

Other Relational Languages

4.1 Query-by-Example (QBE)

1. QBE is both a query language and the name of a DB system including it. The system is no longer in use, but the language is part of IBM's Query Management Facility (QMF).

4.1.1 Basic Structure

1. QBE has "two-dimensional" syntax.
2. Queries are expressed by example.
3. Close correspondence with domain relational calculus
4. Non-procedural.
5. Queries are expressed using **skeleton tables**.
6. User selects the skeletons needed.
7. User fills in skeletons with example rows.
8. An example row consists of constants and example elements which are really domain variables.
9. Domain variables are preceded by an underscore character.
10. Constants appear without any qualification.

We'll look at examples from the text.

4.1.2 Simple Queries

1. For example, to find all customers having an account at the SFU branch:

deposit	bname	account#	cname	balance
	SFU		P._x	

- A P. before the variable causes printing.
- A P.ALL. prefix suppresses duplicate elimination.
- A P. in front of the row prints all attributes.

- The domain variable may be omitted if it is not used elsewhere.
- Arithmetic expressions are allowed.
- Comparison operators are allowed, space on left hand side is left blank.

2. To find the names of all branches not located in Burnaby:

branch	*bname*	*assets*	*bcity*
	P.		¬ Burnaby

3. To find all customers having an account at both the SFU and the MetroTown branch:

deposit	*bname*	*account#*	*cname*	*balance*
	SFU		P._x	
	MetroTown		_x	

4. To find all customers having an account at either branch or both:

deposit	*bname*	*account#*	*cname*	*balance*
	SFU		P._x	
	MetroTown		P._y	

5. Find all customers having an account at the same branch as Jones:

deposit	*bname*	*account#*	*cname*	*balance*
	_x		Jones	
	_x		P._y	

4.1.3 Queries on Several Relations

1. Queries on several relations require several skeleton tables.

2. To find the name and city of all customers having a loan at the SFU branch:

borrow	*bname*	*loan#*	*cname*	*amount*
	SFU		_x	

customer	*cname*	*street*	*ccity*
	P._x		P._y

3. Find the name of all customers having an account at the SFU branch, but no loan from that branch.

   Queries involving negation can be expressed by putting a ¬ sign under the relation name beside an example row:

deposit	*bname*	*account#*	*cname*	*balance*
	SFU		P._x	

borrow	*bname*	*loan#*	*cname*	*amount*
¬	SFU		_x	

4. To find all customers who have accounts at two different branches:

deposit	*bname*	*account#*	*cname*	*balance*
	_y		P._x	
	¬ _y		_x	

4.1.4 The Condition Box

1. When it is difficult or impossible to express all constraints on the domain variables within the skeleton tables, the **condition box** may be used.

2. To add the constraint that we are only interested in customers other than Jones to the above query, we include the condition box:

conditions
_x ¬= Jones

3. To find all account numbers with balances between \$1,300 and \$1,500:

deposit	*bname*	*account#*	*cname*	*balance*
		P.		_x

conditions
_x $\geq$ 1300   _x $\leq$ 1500

4. Logical expressions **and** and **or** may appear in the condition box.

5. To find all account numbers where the balance is between \$1,300 and \$2,000, but is not \$1,500:

deposit	*bname*	*account#*	*cname*	*balance*
		P.		_x

conditions
_x = ($\geq$ 1300 **and** $\leq$ 2000 **and** $\neg$ 1500)

6. An unconventional use of the **or** construct allows comparison with several constant values:

conditions
_x = (Burnaby **or** Richmond)

4.1.5 The Result Relation

1. If the result of a query includes attributes from several relation schemes, we need a way of displaying the result in a single table.

2. We can declare a temporary *result* relation including the attributes to be displayed. We put the print command only in that table.

3. To find the customer names and cities and account numbers for all customers having an account at the SFU branch:

deposit	*bname*	*account#*	*cname*	*balance*
	SFU	_z	_x	

customer	*cname*	*street*	*ccity*
	_x		_y

result	*cname*	*ccity*	*account#*
P.	_x	_y	_z

4.1.6 Ordering the Display of Tuples

1. The order in which tuples are displayed can be controlled by adding the command AO. (ascending order) or DO. (descending order) to the print command:

deposit	*bname*	*account#*	*cname*	*balance*
	SFU		P.AO.	

2. To sort first by name, and then by balance for those with multiple accounts:

deposit	*bname*	*account#*	*cname*	*balance*
	SFU		P.AO(1).	P.DO(2).

4.1.7 Aggregate Operations

1. QBE includes the aggregate operators AVG, MAX, MIN, SUM and CNT. As QBE eliminates duplicates by default, they must have ALL. appended to them.

2. To find the total balance of all accounts belonging to Jones:

deposit	*bname*	*account#*	*cname*	*balance*
			Jones	P.SUM.ALL.

3. All aggregate operators must have ALL. appended, so to override the ALL. we must add UNQ. (unique). (NOTE: a number of examples in the text incorrectly show UNQ. replacing ALL.)

deposit	*bname*	*account#*	*cname*	*balance*
	Main		P.CNT.UNQ.ALL.	

4. To compute functions on groups, we use the G. operator. To find the average balance at each branch:

deposit	*bname*	*account#*	*cname*	*balance*
	P.G.			P.AVG.ALL._x

5. To find the average balances at only branches where the average is more than \$1,200, we add the condition box:

conditions
AVG.ALL._x > 1200

6. To find all customers who have an account at all branches located in Burnaby, we can do:

deposit	*bname*	*account#*	*cname*	*balance*
	_y		P.G._x	

branch	*bname*	*assets*	*bcity*
	_y		Burnaby
	_z		Burnaby

conditions
CNT.UNQ.ALL._y = CNT.UNQ.ALL._z

4.1.8 Modifying the Database

1. QBE has facilities for modifying the database.

Deletion

1. We simply use D. instead of the P. operator. Whole tuples may be deleted, or only some columns.

2. Delete all of Smith's account records:

deposit	*bname*	*account#*	*cname*	*balance*
D.			Smith	

3. Delete the branch-city value for the SFU branch:

branch	*bname*	*assets*	*bcity*
	SFU		D.

4. Delete all loans with loan numbers between 1300 and 1500:

borrow	*bname*	*loan#*	*cname*	*amount*
D.		_x		

conditions
_x = ($\geq$ 1300 **and** $\leq$ 1500)

5. Delete all accounts at branches located in Burnaby:

deposit	*bname*	*account#*	*cname*	*balance*
D.	_x			

branch	*bname*	*assets*	*bcity*
	_x		Burnaby

Insertion

1. Insertion uses the I. operator.

2. To insert an account tuple for Smith:

deposit	*bname*	*account#*	*cname*	*balance*
I.	SFU	9372	Smith	1200

3. If values are missing, **nulls** are inserted.

4. To provide all loan customers in the SFU branch with a $200 savings account:

deposit	*bname*	*account#*	*cname*	*balance*
I.	SFU	_x	_y	200

borrow	*bname*	*loan#*	*cname*	*amount*
	SFU	_x	_y	

Updates

1. We can update individual attributes with the U. operator. Fields left blank are not changed.

2. To update the assets of the SFU branch to $10,000,000:

branch	*bname*	*assets*	*bcity*
	SFU	U.10000000	

3. To make interest payments of 5% on all balances:

deposit	*bname*	*account#*	*cname*	*balance*
U.				_x * 1.05
				_x

4.2 Quel

1. We will not cover this section, aside from making a few remarks about this language.

   - Quel was the original query language for the **Ingres** dbms. Ingres is now available with SQL.
   - Quel closely resembles the tuple relational calculus.
   - Queries use the **range of, retrieve** and **where** clauses.

- A typical query:

  range of t **is** *borrow*
  range of s **is** *deposit*
  retrieve unique ($s.cname$)
  where $t.bname$="SFU" **and**
  $s.bname$="SFU" **and**
  $t.cname$=$s.cname$

  This finds the names of all customers who have both a loan and an account at the SFU branch.
- There is no representation for $\forall$ or $\exists$ in Quel.
- Quel has the power of the relational algebra by means of the **any** aggregate function and the use of insertion and deletion into temporary relations.

Converting Queries Easily Into Any Language

1. I've found that students (and myself) have trouble getting from a database query expressed in English to a query expressed in one of the languages we have covered.
2. I've found a method that seems to help, involving an intermediate step.
   (a) Decide on the relations required to answer the query.
   - You'll need relations containing attributes explicitly mentioned, plus relations needed to "traverse" between needed relations.
   - In some cases you will need more than one copy of a relation.
   - Don't include unneeded relations.

   (b) Draw them on a piece of paper.
   - It helps to draw them in a sensible order.
   - Draw them in the order you would "traverse" them. This will simplify the drawing of links.

   (c) Draw in links and constant values.
   - Put links between attributes in different relations wherever the attributes are required to satisfy some comparison operator (*equals, less than*, etc.).
   - I use ordinary lines for *equals*, and write any other comparison operator on the line at some convenient spot.
   - Write in constant values, where some attribute must have a specific value.

   (d) Now take the diagram, and convert it into the language required. We'll discuss this stage in more detail.
3. **Converting the Diagram to a Specific Language Query**

   For simpler queries, the following advice works. Where you need set operations or division, a little more thought is needed.

   (a) **Relational Algebra:** we'll do a correct but not necessarily optimal query.
   - Do an appropriate combination of **Cartesian products** and **natural joins** of the relations required.
   - Do a **select** where the predicate demands that all the links and constants in your diagram be true.
   - Don't forget that natural joins will take care of some of your diagram's links.
   - Finally, do a **project** of the attributes to be printed out.

   (b) **Tuple Relational Calculus:**
   - Create a tuple variable for each of the relations in your diagram.
   - Make sure the parentheses give you the required scope.
   - Ensure each link and constant in your diagram corresponds to some part of your predicate.
   - Make sure t gets the attributes that should be printed out.

(c) **Domain Relational Calculus:**

- Create domain variables. Name them sensibly.
- Remember that equality is forced by using the same domain variable in several places.
- Other comparison operators may be explicitly stated, e.g. $s \leq u$.
- Remember to use the existential qualifier for domain variables, and to make sure your scoping is correct.

(d) **SQL:** similar to relational algebra.

- Put all the relations needed in the **from** clause.
- Remember to use **tuple variables** when you have more than one copy of a relation, or for general convenience.
- Express each of the links and constants in your diagram as part of the predicate in the **where** clause.
- State the attributes to be printed out in the **select** clause.

(e) **QBE:** your diagram is almost QBE to start with.

- Select the skeleton tables needed.
- Remember that you only need **one** skeleton table per relation. You can put more than one line in a skeleton table.
- Force equality on links by using the same domain variables in different places (see the connection to domain relational calculus?).
- Use the **condition box** where necessary.
- Use **P.** to print out the attributes. Remember to use a **result relation** if attributes are printed out from more than one skeleton table.

More complicated queries will take more thought, but I believe this intermediate step of making a diagram is always helpful.