Relational Calculus

•Relational calculus query specifies *what* is to be retrieved rather than *how* to retrieve it.
•No description of how to evaluate a query.
•In first-order logic (or predicate calculus), *predicate* is a truth-valued function with arguments.
•When we substitute values for the arguments, function yields an expression, called a *proposition*, which can be either true or false.
•If predicate contains a variable (e.g. '*x* is a member of staff'), there must be a range for *x*.
•When we substitute some values of this range for *x*, proposition may be true; for other values, it may be false.
•
•When applied to databases, relational calculus has forms: *tuple* and *domain.*
•Comes in two flavors: Tuple relational calculus (TRC) and Domain relational calculus (DRC).
•Calculus has variables, constants, comparison ops, logical connectives, and quantifiers.
•*TRC*: Variables range over (i.e., get bound to) *tuples*.
•*DRC*: Variables range over *domain elements* (= field values).
•Both TRC and DRC are simple subsets of first-order logic.
•Expressions in the calculus are called *formulas*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.
•Interested in finding tuples for which a predicate is true. Based on use of tuple variables.
•
•Tuple variable is a variable that 'ranges over' a named relation: i.e., variable whose only permitted values are tuples of the relation.
•Specify range of a tuple variable *S* as the Staff relation as:
Staff(S)
•To find set of all tuples S such that P(S) is true:
{S | P(S)}
Query has the form: { T | p(T)}
Answer includes all tuples T that
make the *formula* p(T) be *true*.
Formula is recursively defined, starting with
simple *atomic formulas* (getting tuples from relations or making comparisons of values), and building bigger and better formulas using the logical connectives.
•To find details of all staff earning more than £10,000:
{S | Staff(S) Ÿ S.salary > 10000}

•To find a particular attribute, such as salary, write:
{S.salary | Staff(S) Ÿ S.salary > 10000}
•Can use two *quantifiers* to tell how many instances the predicate applies to:
•Existential quantifier $ ('there exists')
•Universal quantifier " ('for all')
•
•Tuple variables qualified by " or $ are called *bound* variables, otherwise called *free* variables.
•Universal quantifier is used in statements about every instance, such as:
("B) (B.city π 'Paris')

•Means 'For all Branch tuples, the address is not in Paris'.
•Can also use ~($B) (B.city = 'Paris') which means 'There are no branches with an address in Paris'.

•Formulae should be unambiguous and make sense.
•A (well-formed) formula is made out of <u>atoms</u>:
•*R*(S_i), where S_i is a tuple variable and *R* is a relation
•$S_i.a_1$ q $S_j.a_2$
•$S_i.a_1$ q c
•Can recursively build up formulae from atoms:
•An atom is a formula
•If F_1 and F_2 are formulae, so are their conjunction, F_1 Ù F_2; disjunction, F_1 Ú F_2; and negation, ~F_1
•If *F* is a formula with free variable *X*, then ($*X*)(*F*) and ("*X*)(*F*) are also formulae.

•List the names of all managers who earn more than £25,000.
{S.fName, S.lName | Staff(S) Ÿ
S.position = 'Manager' Ÿ S.salary > 25000}

•List the staff who manage properties for rent in Glasgow.
{S | Staff(S) Ÿ ($P) (PropertyForRent(P) Ÿ (P.staffNo = S.staffNo) Ù P.city = 'Glasgow')}

•List the names of staff who currently do not manage any properties.
{S.fName, S.lName | Staff(S) Ÿ (~($P) (PropertyForRent(P)Ÿ(S.staffNo = P.staffNo)))}
Or
{S.fName, S.lName | Staff(S) Ÿ (("P) (~PropertyForRent(P)⁄
~(S.staffNo = P.staffNo)))}

•Expressions can generate an infinite set. For example:
{S | ~Staff(S)}
•To avoid this, add restriction that all values in result must be values in the domain of the expression.

Domain Relational Calculus

•Uses variables that take values from <u>domains</u> instead of tuples of relations.
•If F(d_1, d_2, . . . , d_n) stands for a formula composed of atoms and d_1, d_2, . . . , d_n represent domain variables, then:
{d_1, d_2, . . . , d_n | F(d_1, d_2, . . . , d_n)}
is a general domain relational calculus expression.
•Find the names of all managers who earn more than £25,000.
•
{fN, lN | ($sN, posn, sex, DOB, sal, bN)
(Staff (sN, fN, lN, posn, sex, DOB, sal, bN) Ÿ
posn = 'Manager' Ÿ sal > 25000)}

•List the names of staff who currently do not manage any properties for rent.
{fN, lN | ($sN)
(Staff(sN,fN,lN,posn,sex,DOB,sal,bN) Ÿ
(~($sN1) (PropertyForRent(pN, st, cty, pc, typ,
rms, rnt, oN, sN1, bN1) Ù (sN=sN1))))}

•List the names of clients who have viewed a property for rent in Glasgow.
{fN, lN | ($cN, cN1, pN, pN1, cty)
(Client(cN, fN, lN,tel, pT, mR) Ÿ
Viewing(cN1, pN1, dt, cmt) Ÿ
PropertyForRent(pN, st, cty, pc, typ,
rms, rnt,oN, sN, bN) Ù
(cN = cN1) Ù (pN = pN1) Ù cty = ‘Glasgow’)}

•When restricted to safe expressions, domain relational calculus is equivalent to tuple relational calculus restricted to safe expressions, which is equivalent to relational algebra.
•Means every relational algebra expression has an equivalent relational calculus expression, and vice versa.

•Encoding Relational Algebra

Let’s consider the first direction of the equivalence: can the relational algebra be coded up in the (domain) relational calculus?
•This translation can be done systematically, we define a translation function [-]
•Simple case:
[R] = {<x_1,…,x_n> | <x_1,…,x_n>ŒR}

•Assume
[e] = {<x_1,…,x_n> | F }
•Then
[s_c(e)] = {<x_1,…,x_n> | F Ÿ C’}
where C’ is obtained from C by replacing each attribute with the corresponding variable

•Can we code up the relational calculus in the relational algebra?
•At the moment, NO!
•Given our syntax we can define ‘problematic’ queries such as
{S | ÿ (SŒSailors)}
•This (presumably) means the set of all tuples that are not sailors, which is an infinite set… ☹

Safe queries

•A query is said to be safe if no matter how we instantiate the relations, it always produces a finite answer
•Unfortunately, safety (a semantic condition) is undecidable
•That is, given a arbitrary query, no program can decide if it is safe
•Fortunately, we *can* define a restricted syntactic class of queries which are guaranteed to be safe
•Safe queries can be encoded in the relational algebra

•Transform-oriented languages are non-procedural languages that use relations to transform input data into required outputs (e.g. SQL).
•Graphical languages provide user with picture of the structure of the relation. User fills in example of what is wanted and system returns required data in that format (e.g. QBE).