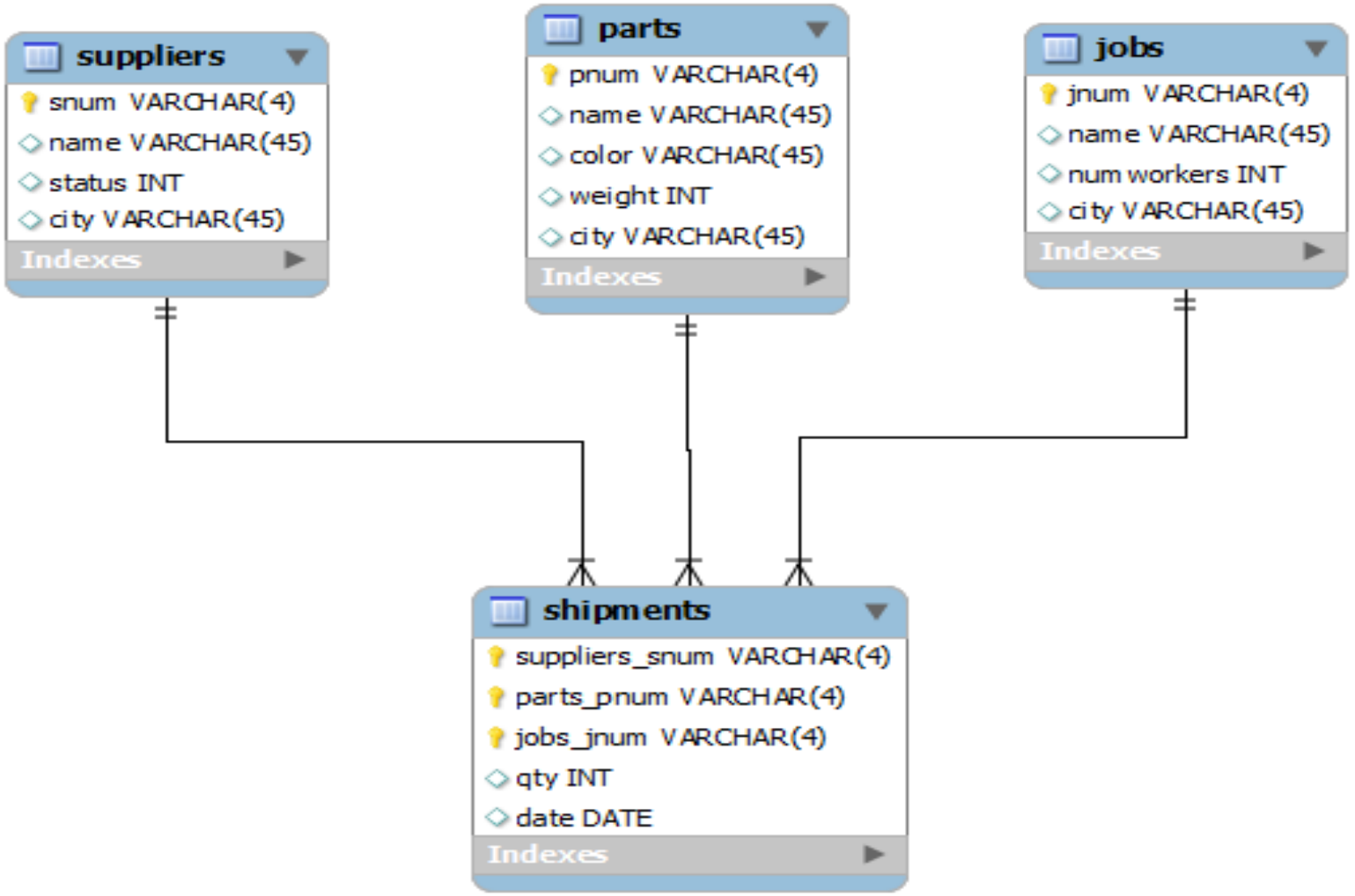
COP 4710: Database Systems
Fall 2013

Chapter 4 – Relational Query Languages – Part 2

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4710/fall2013

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Sample Database Scheme



Some Practice Queries Using Only Five Fundamental Operators

To simplify the query expressions assume that S = suppliers, P = parts, J = jobs, and SPJ = shipments

1. Find all the supplier numbers for suppliers located in Milan or who ship to any job in a quantity greater than 40.

$$[\pi_{(s\#)}(\sigma_{(city = Milan)}(S))] \cup [\pi_{(s\#)}(\sigma_{(qty > 40)}(SPJ))]$$

2. Find all the supplier numbers for suppliers who ship only red parts.

$$[\pi_{(S.name)}(\sigma_{((SPJ.s\#=S.S\#) \text{ AND } (SPJ.p\#=P.p\#) \text{ AND } (color=red))}(SPJ \times S \times P))]$$

$$- [\pi_{(S.name)}(\sigma_{((SPJ.s\#=S.S\#) \text{ AND } (SPJ.p\#=P.p\#) \text{ AND } (color \neq red))}(SPJ \times S \times P))]$$

Some Practice Queries Using Only Five Fundamental Operators (cont.)

3. Find the supplier names for those suppliers who are located in the same city as a job to which they ship parts.

- $T1 = (S \times SPJ \times J)$
- $T2 = \sigma_{(S.s\# = SPJ.s\#)}(T1)$ //select tuples which match on s#
- $T3 = \sigma_{(J.j\# = SPJ.j\#)}(T2)$ //select tuples which match on j#
- $T4 = \sigma_{(J.city = S.city)}(T3)$ //select tuples from the same city
- $T5 = \pi_{(S.name)}(T4)$ //project final attribute set

Some Practice Queries Using Only Five Fundamental Operators (cont.)

4. Find all the part numbers which are shipped by both supplier "S1" and supplier "S2".

NOTE: The following expression in **not** correct! Why not?

$$\pi_{(p\#)}(\sigma_{((s\# = S1) \text{ AND } (s\# = S2))}(SPJ))$$

The following is the correct way of expressing this query in RA.

$$[\pi_{(p\#)}(\sigma_{(s\#=S1)}(SPJ)] - ([\pi_{(p\#)}(\sigma_{(s\#=S1)}(SPJ)] - [\pi_{(p\#)}(\sigma_{(s\#=S2)}(SPJ)])$$

Some Practice Queries Using Only Five Fundamental Operators (cont.)

5. Find the supplier numbers for those suppliers who supply both a red part and a blue part.

NOTE: The following expression in **not** correct! Why not?

$$\pi_{(s\#)}(\sigma_{((color = blue) \text{ AND } (SPJ.p\# = P.p\#) \text{ AND } (color=red))}\ (P \times SPJ))$$

The following is the correct way of expressing this query in RA.

$$T1 = \pi_{(s\#)}(\sigma_{((color = blue) \text{ AND } (SPJ.p\# = P.p\#))}\ (P \times SPJ))$$

$$T2 = \pi_{(s\#)}(\sigma_{((color = red) \text{ AND } (SPJ.p\# = P.p\#))}\ (P \times SPJ))$$

$$T3 = T2 - T1$$

$$T4 = T2 - T3$$

Some Practice Queries Using Only Five Fundamental Operators (cont.)

6. Find all pairs (s#, j#) such that the supplier and the job are located in the same city, yet that supplier does not have a shipment to that job.

$T1 = \pi_{(s\#,\ j\#)}(\sigma_{(S.city = J.city)}(S \times J))$ //all (s#,j#) pairs in same city

$T2 = \pi_{(s\#,\ j\#)}(\sigma_{((S.city = J.city)\ AND\ (SPJ.j\# = J.j\#)\ AND\ (SPJ.s\# = S.s\#))}(S \times SPJ \times J))$

//T2 contains all (s#,j#) pairs representing shipments by suppliers to jobs in the same city.

$T3 = T1 - T2$

Renaming Operator

- Unlike the base relations in a database, the intermediate relations which are produced as the result of the evaluation of a query, do not have names to which they can be referred. Unless the intermediate relation is explicitly saved, it does not exist after the execution of the query. Many times, however, it is quite useful to save an intermediate relation as it may contain a set of tuples which answer a related query, or it will contain a set of tuples which can be used to evaluate another query and saving the intermediate relation will mean that the same work will not need to be repeated.
- The rename operation is represented by the lowercase Greek letter rho (ρ) and it can be used to rename both relations as well as attributes.
- The first common form of the rename operation applies to relations.
- General form: $\rho_{\text{new relation name}}(\text{relation})$
- Thus, $\rho_x(r)$ renames the relation *r* to relation *x*.

Renaming Operator (cont.)

- The second form of the rename operation applies to the renaming of both the relation as well as the attributes of that relation. Assuming that the operand relation is of degree *n*, then the form of this version of the rename operation is:

- General form: $\rho_{\text{new relation name (A1, A2, …, AN)}}(\text{relation})$

- Thus, $\rho_{\text{x(one, two, …, last)}}(\text{r})$ renames relation *r* to relation *x* and the *n* attributes of relation *x* are names *one*, *two*, ..., *last*.

Redundant Operators in Relational Algebra

- It can be proven (although we aren't going to go through that proof) that the five fundamental relational operations are sufficient to express any relational-algebra query.
- What this proof doesn't state however, is that some complex queries will require extremely lengthy and difficult query expressions.
- There have been several extensions of the set of operations available in the relational algebra that provide no additional expressive power, but do provide a simplification in the expression required for more complex queries.
- We'll look at the most important and common of these redundant operations and also show their definition in terms of the five fundamental operations

Intersection Operator

Type: binary
Symbol: $\cap$
General form: $r \cap s$ where *r* and *s* are union compatible relations
Schema of result relation: schema of operation relation
Size of result relation (tuples): $\leq |r|$
Definition: $r \cap s \equiv r - (r - s)$
Example:

$$(\pi_{(p\#)}(SPJ)) \cap (\pi_{(p\#)}(P))$$

- The intersection operation produces the set of tuples that appear in **both** operand relations.

Intersection Operator Examples

R

A	B	C	D
a	a	yes	1
b	d	no	7
c	f	yes	34
a	d	no	6

S

E	F	G	H
a	a	yes	1
b	r	yes	3
c	f	yes	34
m	n	no	56

$r = R \cap S$

A	B	C	D
a	a	yes	1
c	f	yes	34

$r = R \cap T$

A	B	C	D

T

E	F	G	H
a	r	no	31
b	f	yes	30

Join Operators

- As we saw in some of our earlier query expression which involved the Cartesian product operator, we had to provide additional selection operations to remove those combinations of tuples that resulted from the Cartesian product which weren't related (they didn't make sense like when a shipment of a specific part was combined with part information but the part information didn't belong to the part that was being shipped).
- This occurs so commonly that an operation which is a combination of the Cartesian product and selection operations was developed called a *join operation*.
- There are several different join operations which are called, *theta-join, equijoin, natural join, outer join,* and *semijoin*. We will examine each of these operations and explore the conditions of their use.

Theta-Join and Equijoin Operators

Type: binary

Symbol/general form: $r \rhd\lhd_{(predicate)} s$

Schema of result relation: concatenation of operand relations

Definition: $r \rhd\lhd_{(predicate)} s \equiv \sigma_{(predicate)}(r \times s)$

Examples:

$r \rhd\lhd_{(color='blue' \text{ AND } size=3)} s$ $r \rhd\lhd_{(color='blue' \text{ AND } size>3)} s$

an equijoin

a theta-join

- The theta-join operation is a shorthand for a Cartesian product followed by a selection operation.
- The equijoin operation is a special case of the theta-join operation that occurs when all of the conditions in the predicate are equality conditions.
- Neither a theta-join nor an equijoin operation eliminates extraneous tuples by default. Therefore, the elimination of extraneous tuples must be handled explicitly via the predicate.

Theta-Join Operator Examples

R

A	B	C	D
a	a	yes	1
b	d	no	7
c	f	yes	34
a	d	no	6

S

E	F	G	H
a	a	yes	1
b	r	yes	3
c	f	yes	34
m	n	no	56

$r = R \triangleright\triangleleft_{(R.B < S.F)} S$

A	B	C	D	E	F	G	H
a	a	yes	1	b	r	yes	3
a	a	yes	1	c	f	yes	34
a	a	yes	1	m	n	no	56
b	d	no	7	b	r	yes	3
b	d	no	7	c	f	yes	34
b	d	no	7	m	n	no	56
c	f	yes	34	b	r	yes	3
c	f	yes	34	m	n	no	56
a	d	no	6	b	r	yes	3
a	d	no	6	c	f	yes	34
a	d	no	6	m	n	no	56

Natural Join Operator

Type: binary

Symbol/general form: $r * s$

Schema of result relation: concatenation of operand relations with only one occurrence of commonly named attributes

Definition: $r * s \equiv r \rhd\lhd_{(r.commonattributes = s.commonattributes)} s$

Examples: $s * spj * p$

- The natural-join operation performs an equijoin over all attributes in the two operand relations which have the same attribute name.
- The degree of the result relation of a natural-join is sum of the degrees of the two operand relations less the number of attributes which are common to both operand relations. (In other words, one occurrence of each common attribute is eliminated from the result relation.)
- The natural join is probably the most common of all the forms of the join operation. It is extremely useful in the removal of extraneous tuples. Those attributes which are commonly named between the two operand relations are commonly referred to as the *join attributes*.

Natural Join Operator Examples

R

A	B	C	D
a	a	yes	1
b	r	no	7
c	f	yes	34
a	m	no	6

S

B	M	G	H
a	a	yes	1
b	r	yes	3
a	f	yes	34
m	n	no	56

r = R * S

A	B	C	D	M	G	H
a	a	yes	1	a	yes	1
a	a	yes	1	f	yes	34
a	m	no	6	n	no	56

r = R * T

A	B	C	D	G	H
b	r	no	7	yes	30

T

A	B	G	H
a	f	no	31
b	r	yes	30

Outer Join Operator

Type: binary

Symbol/general form: left-outer-join: $r \supset\!\lhd s$ right-outer-join: $r \rhd\!\subset s$

full outer join: $r \supset\!\lhd\!\rhd\!\subset s$

Schema of result relation: concatenation of operand relations

Definition:

$r \supset\!\lhd s$ ≡ natural join of r and s with tuples from r which do not have a match in s included in the result. Any missing values from s are set to null.

$r \rhd\!\subset s$ ≡ natural join of r and s with tuples from s which do not have a match in r included in the result. Any missing values from r are set to null.

$r \supset\!\lhd\!\rhd\!\subset s$≡ natural join of r and s with tuples from both r and s which do not have a match are included in the result. Any missing values are set to null.

Examples: Let r(A,B) = {(a, b), (c, d), (b,c)} and let

s(A,C) = {(a, d), (s, t), (b, d)}

then $r \supset\!\lhd s$ = (A,B,C) = {(a,b,d), (b,c,d), (c,d,null)},

$r \rhd\!\subset s$ = (A,B,C) = {(a,b,d), (b,c,d), (s,null,t)}, and

$r \supset\!\lhd\!\rhd\!\subset s$ = (A,B,C) = {(a,b,d), (b,c,d), (s,null,t), (c,d,null)},

Outer Join Operator Examples

R

A	B	C
1	2	3
4	5	6
7	8	9

$r = R \supset\rhd\lhd\subset S$

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	null
7	8	9	null
null	6	7	12

S

B	C	D
2	3	10
2	3	11
6	7	12

$r = R \supset\lhd S$

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	null
7	8	9	null

$r = R \rhd\subset S$

B	C	D	A
2	3	10	1
2	3	11	1
6	7	12	null

Semi Join Operator

Type: binary

Symbol/general form: $r \triangleright_{(predicate)} s$

Schema of result relation: schema of r

Definition: $r \triangleright_{(predicate)} s \equiv \pi_{(attributes\ of\ r)} (r \triangleright\triangleleft_{(predicate)} s)$

Examples: see next page

- This is operator is only useful in a distributed environment.
- The idea behind the semi-join operation is to reduce the number of tuples in a relation before transferring it to another side for performing a join operation. Intuitively, the idea is to send the joining column(s) of R to the site where the other relation S is located; this column(s) is then joined with S. Following that, the join attributes, along with any attributes in S required in the result are projected out and shipped back to the original site and joined with R. Hence, only the joining column(s) of R is transferred in one direction, and a subset of S with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in S participate in the join, this can be an extremely efficient operation.
- In its general form, which is shown above, the semi-join is a semi-theta-join. The expected variants of a semi-equijoin and a semi-natural-join are defined in a similar fashion.

Semi Join Operator Examples

R

A	B	C	D
a	a	yes	1
b	r	no	7
c	f	yes	34
a	m	no	6

$r = R \triangleright_{(R.B > S.M)} S$

A	B	C	D
b	r	no	7
c	f	yes	34
a	m	no	6

$r = R \triangleright S$

A	B	C	D
a	a	yes	1
b	r	no	7

S

B	M	C
a	e	yes
b	r	yes
a	f	no
r	n	no

T

B	G	D
a	4	d
b	7	e
a	4	f
m	2	g

$r = S \triangleright_{(T.G = 4)} T$

B	M	C
a	e	yes
a	f	no
b	r	yes
r	n	no

$r = S \triangleright_{(T.G = 4) AND (S.M > g)} T$

B	M	C
b	r	yes
r	n	no

Division Operator

Type: binary
Symbol/general form: $r \div s$ where r({A}) and s({B})
Schema of result relation: C where C = A – B
Definition: $r \div s \equiv \pi_{(A-B)}(r) - (\pi_{(A-B)}((\pi_{(A-B)}(r) \times s) - r))$
Examples:
Let r(A,B,C) = {(a,b,c), (a,d,d), (a,b,d), (a,c,c), (a,d,d)}
and s(C) = {(c), (d)}
then: $r \div s$ = t(A,B) = {(a,b)}

Requirements for the division operation:

1. Relation *r* is defined over the attribute set *A* and relation *s* is defined over the attribute set *B* such that $B \subseteq A$.

2. Let *C* be the set of attributes in $A - B$.

Given these constraints the division operation is defined as: a tuple *t* is in $r \div s$ if for every tuple t_s in *s* there is a tuple t_r in *r* which satisfies both:

$t_r[C] = t_s[C]$ and $t_r[A-B] = t[A-B]$

Division Operator Examples

R

A	B	C	D
a	f	yes	1
b	r	no	1
a	f	yes	34
e	g	yes	34
a	m	no	6
b	r	no	34

$r = R \div S$

A	B	C
a	f	yes
b	r	no

$r = R \div T$

A	B
a	f

$r = R \div U$

A	B
b	r

$r = R \div V$

A
a

$r = R \div W$

A

S

D
1
34

T

C	D
yes	1
yes	34

U

C	D
no	1
no	34

V

B	C	D
f	yes	1
f	yes	34
m	no	6

W

B	C	D
f	yes	1
g	yes	69

Usefulness of the Redundant Operators

- The redundant relational algebra operators are redundant because they are all defined in terms of the five fundamental operators.
- Their usefulness however, is best illustrated by the division operator.
- Consider the following query based on the suppliers-parts-jobs-shipment database given earlier:

  Query: Find the supplier numbers for those suppliers who ship every part.
- A solution to this query is given on the next page using only the five fundamental operators and then again using the redundant operators.

Usefulness of the Redundant Operators (cont.)

Using only the five fundamental operators

- T1 = $\pi_{(s\#,\ p\#)}(spj)$ //all (s#,p#) pairs for actual shipments
- T2 = $\pi_{(p\#)}(p)$ //all (p#) {all parts that exist, whether shipped or not}
- T3 = $\pi_{(s\#)}(T1) \times T2$ //all s# in T1 paired with every tuple in T2 {spj.s#, p.p#}
- T4 = T3 – T1 //all tuples in T3 which are not also in T1
- T5 = T1 – T4 //all tuples in T1 which are not also in T4.
- T6 = $\pi_{(s\#)}(T5)$ //solution

Final solution is: $\pi_{(s\#)}(spj) - (\pi_{(s\#)}((\pi_{(s\#)}(spj) \times p) - spj))$

Using the redundant operators

Solution is: $(\pi_{(s\#,\ p\#)}(spj)) \div (\pi_{(p\#)}(p))$

Practice Queries Using All Relational Algebra Operators (Answers on Next Page)

1. List all pairs of supplier numbers for suppliers who are located in the same city.

2. List every shipment involving a green part.

3. List all the supplier numbers for suppliers who ship a part that is manufactured in the same city in which the supplier is located.

4. List the names of those suppliers who ship all the blue parts.

5. List the supplier numbers for those suppliers who ship only blue parts.

Practice Queries Using All Relational Algebra Operators (Answers)

1. $\pi_{(s.s\#,x.s\#)}(s \rhd\lhd_{(s.city=x.city)} (\rho_x(s)))$

2. $spj * (\sigma_{(color=green)}(p))$

3. $\pi_{(s.s\#)}(s * p * spj)$

4. $\pi_{(s.name)}(s * (\pi_{(s\#,p\#)}(spj) \div \pi_{(p\#)}(\sigma_{(color=blue)}(p))))$

5. $(\pi_{(s\#)}(spj * (\sigma_{(color=blue)}(p)))) - (\pi_{(s\#)}(spj * (\sigma_{(color\neq blue)}(p))))$

Tuple Relational Calculus

- Relational algebra was a procedural query language. You specified in the query expression *what* data you wanted (this was usually given by the final projection) **and** you specified *how* the DBMS was to go about getting this data. The *how* was specified in the sequence of operations that you put together in order to answer your query.
- Relational calculus is a non-procedural query language that has two basic forms: *tuple relational calculus* and *domain relational calculus*. While they are similar in nature there are fundamental differences in the two forms.
- For now, we will focus on the tuple relational calculus. We won't look at either of the calculus languages in quite the same detail that we did with the relational algebra, we're more interested here in giving you an idea of what these pure languages look like since they form the foundation of the implemented languages such as SQL that we'll see later.

Tuple Relational Calculus (cont.)

- Tuple relational calculus was used as the basis for the query language of the INGRES database system developed at Bell Labs in the late 1970s and domain relational calculus is the basis for the query language QBE (Query-By-Example) developed by IBM as part of the system R project also in the early 1970's.
- In terms of completeness, tuple relational calculus and relational algebra are equivalent. By completeness we mean that any query which can be expressed in relational algebra can also be expressed in tuple calculus and vice versa.
- Tuple calculus is based upon first-order predicate calculus, which is the calculus of logic. The basic tuple calculus expression looks like the following:

$\{ t \mid P(t)\}$ read as: the set of tuples *t* such that the predicate *P* is true.

t is a tuple variable.

Tuple Relational Calculus (cont.)

- A tuple variable is simply a variable which at any time can assume the value of one of the tuples in a relation instance.
- A tuple variable "ranges over" or assumes values from only a single relation instance at a time.
- The typical notation for indicating the range of a tuple variable is: *tuple-variable(relation).*
  - An example might be: t(S) which would indicate that tuple variable *t* assumes values which are tuples from the relation named *S*.
- Since a tuple variable takes on values which are entire tuples from a given relation and we often need only a subset of the attributes contained in a given tuple, the notation for this is *tuple-variable.attribute-name*. Notice that this is basically the same notation as we used for the qualified attribute name in the relational algebra.

Tuple Relational Calculus (cont.)

- In general, the predicate consists of any number of tuple variables occurring in what are known as well-formed formulas (WFFs in predicate calculus parlance).
- A tuple variable exists in one of two states, either *free* or *bound*. A tuple variable is bound to a WFF through a quantifier.
- There are essentially two quantifiers of concern: the *existence quantifier* (denoted by the symbol ∃) and the *universal quantifier* (denoted by ∀).
- The only free tuple variables that can exist in a tuple calculus expression are those which appear to the left of the "such that" bar.
- If *f* is a WFF and *t* is a tuple variable, then if *t* is free in *f* it is bound in both ∃*t*(f) and ∀*t*(f). In other words, the quantification of *t* causes its binding to a WFF (predicate).

Tuple Relational Calculus (cont.)

- **ALL** WFFs evaluate to either true or false. In other words, the predicate is either satisfied by the tuple variables or it isn't.
- Thus the WFF, $\exists t(f)$ evaluates to true if there exists some tuple *t* which makes the predicate *f* true. If there does not exist a tuple (that can be assigned to *t*) which makes the predicate *f* true, then the value of this WFF must be false.
- Similarly, the WFF, $\forall t(f)$ evaluates to true only if every tuple which can be assigned to *t* makes the predicate true. If there exists even a single tuple for which the predicate is false, then the WFF will evaluate to false.

Equivalence of Tuple Relational Calculus and Relational Algebra

- To see that the tuple calculus is equivalent to relational algebra (and vice versa), I've included the definitions of several of the more common relational algebra operators as they would appear in the tuple calculus. You don't need to remember these equivalences, just look at them and convince yourself that they are the same.
- Union: $R \cup S \equiv \{t \mid t(R) \text{ or } t(S)\}$ //set of tuples | $t \in R$ or $t \in S$
- Intersection: $R \cap S \equiv \{t \mid t(R) \text{ and } t(S)\}$ //set of tuples | $t \in R$ and $t \in S$
- Difference: $R - S \equiv \{t \mid t(R) \text{ and not}(t(S))\}$ //set of tuples | $t \in R$ and $t \notin S$
- Selection: $\sigma_{(p)}(R) \equiv \{t \mid t(R) \text{ and } P(t)\}$ // tuples t | $t \in R$ and p is true

Equivalence of Tuple Relational Calculus and Relational Algebra (cont.)

- The four relational algebra operations above are fairly simple to express in tuple calculus, however projection and Cartesian product are not quite as simple as you can see below and the join operations get quite nasty, so we'll avoid them altogether.

- Cartesian product:

$$R \times S \equiv \{t^{(r+s)} \mid \exists u(R)\ (\exists v(S)(\ t[1]=u[1] \text{ and } \ldots \text{and } t[r]=u[r] \text{ and}$$

- $$t[r+1]=v[1] \text{ and} \ldots \text{and } t[r+s]=v[s]))\}$$

The notation $t^{(r+s)}$ indicates the degree of the tuple variable which in the case of the Cartesian product is the sum of the degrees of the two operand relations.

- Projection: $\pi_{(i_1, i_2, \ldots, i_k)}(R) = \left\{ t^k \mid \exists u(R)\ (t_1 = u[i_1] \text{ and } \ldots \text{ and } t_k = u[i_k]) \right\}$

Example Tuple Relational Calculus Queries

Query #1

English: List the names of the suppliers who are located in Orlando.

tuple calculus: {t.name | t(suppliers) and t.city = "Orlando"}

- This query sets up a tuple variable named *t* that ranges over the suppliers relation and "selects" those tuples which make the predicate "city = Orlando" true.

relational algebra: $\pi_{(name)}(\sigma_{(city = Orlando)}(p))$

Example Tuple Relational Calculus Queries (cont.)

Query #2

English: For every part list the name of the part and its weight.

tuple calculus: {t.name, t.weight | t(parts)}

- This query is more simple than the first in that for every tuple in the parts relation we are simply listing the two attributes of name and weight.

relational algebra: $\pi_{(name,\ weight)}(p)$

Example Tuple Relational Calculus Queries (cont.)

Query #3

English: List the part numbers for parts shipped to jobs located in Madrid.

tuple calculus: {x.p# | x(spj) and (∃y(j) and y.city=”Madrid” and y.j# = x.j#)}

- This query is a little bit more complicated since two relations are involved. Tuple variable *x* is the only free variable (as it must be) and tuple variable *y* is bound to the WFF which includes the predicates y.city=Madrid and y.j# =x.j#. The way this basically works is this: x assumes the value of one of the tuples from the spj relation (the relation it ranges over) and for each value of x the we are “searching” for value of y, which ranges over the jobs relation, that will make the predicate true. If such a tuple *y* exists, then the part number from the x tuple variable is “returned” to the resulting relation.

Example Tuple Relational Calculus Queries (cont.)

Query #4

English: List the supplier numbers for suppliers who do not ship part P2.

tuple calculus: {y.s# | y(s) and not(∃x(spj) and x.p#=”P2” and y.s#=x.s#)}

- For this query we are looking for the existence of a tuple variable y (which ranges over the suppliers relation) for which we cannot find the existence of a tuple variable *y* (which ranges over spj) that makes the predicate true. In other words, if there does exist a tuple in spj with the same supplier number as in the *y* tuple variable and for which the part number is P2, then this is a supplier who ships part P2 and they should not be included in the result relation.

relational algebra: $(\pi_{(s\#)}(s)) - (\pi_{(s\#)}(\sigma_{(p\#=P2)}(spj)))$

Quantifier Implications and Equivalences

- In general, the quantifiers can be transformed into the other quantifier with negation, and/or replace one another, a negated formula becomes un-negated and an un-negated formula becomes negated.
- There are however, some special cases which arise in these equivalences of which you need to be aware.
- A few of the more important ones are shown below along with one which is commonly used by many people, but is incorrect!

Quantifier Implications and Equivalences (cont.)

1. $\forall x(P(x)) \equiv \text{not } \exists x(\text{not } P(x))$
2. $\text{not}(\exists x)(P(x)) \Rightarrow \text{not}(\forall x)(P(x))$
3. $(\exists x)(P(x)) \equiv \text{not } (\forall x)(\text{not } (P(x)))$
4. $(\forall x)(P(x) \text{ and } Q(x)) \equiv \text{not } (\exists x)(\text{not } (P(x)) \text{ or not } (Q(x)))$
5. $(\forall x)(P(x) \text{ or } Q(x)) \equiv \text{not } (\exists x)(\text{not } (P(x)) \text{ and not } (Q(x)))$
6. $(\exists x)(P(x)) \text{ or } Q(x)) \equiv \text{not } (\forall x)(\text{not } (P(x)) \text{ and not } (Q(x)))$
7. $(\exists x)(P(x) \text{ and } Q(x)) \equiv \text{not } (\forall x)(\text{not } (P(x)) \text{ or not } (Q(x)))$
8. $(\forall x)(P(x)) \Rightarrow (\exists x)(P(x))$
9. $\text{not}(\forall x)(P(x)) \not\Rightarrow \text{not}(\exists x)(P(x))$ this implication is not true!

Quantifier Implications and Equivalences (cont.)

- Equivalence 1 can be interpreted in the following manner when considering the right hand side: "there does not exist a value for x for which the predicate is not true". The flip side of this is, of course, for every value of x the predicate must be true.

- Implication 2 can be interpreted in the following manner: (left hand side)"there does not exist a value for x for which the predicate is true" implies that (right hand side) the predicate is not true for every value of x". This one you might have to think about for a minute. Looking at it from another direction consider this: if there exists a value of x that makes the predicate true, then the not in front of the expression would result in a value of false. So, the only way that the left hand side could be true would be the situation when all possible values of x cause the predicate to be false, in which case the not will evaluate to true. From the right hand side think of it this way: if there is one single value that x can assume which makes the predicate false, then the universal quantifier will return false, yet the negation in front of it would return true for this case.

Quantifier Implications and Equivalences (cont.)

- Implication 9 is simply wrong, but it is often mistakenly used, especially by beginning database students … so don't you become one of these who do it wrong!
- The implication is that if not every value of x makes the predicate true then this implies that there does not exist a value of x which does make the predicate true. Clearly, this is not necessarily the case.
  - As an illustration, suppose that there are 12 tuples in our universal space, so x can assume any of these 12 tuples. Now let's suppose that 10 of these tuples satisfy the predicate P and 2 of these tuples do not satisfy P. Then clearly not every value of x satisfies P, but at the same time we cannot say that there does not exist any values of x which satisfy P, since in this case there were 10 of them that did. So clearly, this implication is false, so don't be tempted to use it.