

Querying in the Relational Model

- Relational model is not all about storage
- Also allows data manipulation (queries, updates)
- Queries in RM: idea is to mathematically specify a new relation that holds only the answer tuples
- Two ways to specify: *Relational Calculus* and *Relational Algebra*

Ways To Specify Queries

- Relational Calculus
 - Related to predicate calculus / first order logic
 - Discrete math strikes again!
 - Idea: specify what the answer tuples look like
 - Advantage: relatively easy to specify the most complex queries
 - Disadvantage: far removed from implementation
- Other way: Relational Algebra

Ways To Specify Queries (Cont'd)

- Relational Algebra
 - Like writing a program
 - Specify a list of operations that are performed on data, step-by-step
 - It's an algebra over relations!
 - Advantage: closely resembles how a database actually works
 - Advantage: simple queries easier to specify in RA compared to RC
 - Disadvantage: more difficult queries are often very difficult to specify in RA
- *But*, RA and RC have same power in the end

Why Not Go Straight to SQL?

- SQL (or “structured query language”) is worldwide standard RDB query language
- Why not skip this RA/RC stuff?
 - Hard to *really* understand SQL w/o background in RA/RC (esp. RC!)
 - Modern DB systems implement RA under the hood
 - So, hard to understand tuning, DBA w/o grounding in RA
- We’ll do RA first

Relational Algebra

- Any algebra needs a domain and a set of operations
- The domain for RA is all valid relations
- The RA consists of a set of unary/binary operations
- Like all algebras, RA is *closed*:
 - Apply an op to a relation or a pair of valid relations
 - Then you get back a valid relation!

Relational Selection

- Most fundamental operation is *relational selection*
- Is a simple filter over a relation's tuples
- Given a relation R , selection is written as $\sigma_B(R)$
 - In databases, the Greek letter σ is the selection operator
 - Different from the SQL `SELECT` clause!
 - B is some boolean function accepting/rejecting single tuples from R
 - Key rule: can only look at *single* tuples
 - Key rule: no other outside information allowed
 - Typical operations in B are $=$, \neq , and, or, not, etc.

Relational Selection

- Example: $LIKES(\underline{drinker}, beer)$
- If ("Joe" , "Milwaukee's Best") is in $LIKES$, it means that "Joe" likes "Milwaukee's Best"
- Say we want all of the people who like the beer "Bud"
 - Simply use $\sigma_{beer = Bud}(LIKES)$

Relational Selection (cont'd)

- If the state of $LIKES(\underline{drinker}, \underline{beer})$ is
 $\{ (Joe, Bud), (Sam, Bud), (Sue, Coors), (John, Beast) \}$
- Then $\sigma_{beer = Bud}(LIKES)$ is
 $\{ (Joe, Bud), (Sam, Bud) \}$
- Say we want all of the people who like either "Bud" or "The Beast" (aka "Milwaukee's Best")
 - Simply use $\sigma_{beer = Bud \vee beer = Beast}(LIKES)$

Relational Projection

- However, if we want the people who drink Budweiser, will $\sigma_{beer = Bud}(LIKES)$ really do it?
- Problem: gives us the beer "Bud" in every tuple
- Projection kills unwanted attributes
- Given a relation R , projection is written as $\pi_P(R)$
 - In databases, the Greek letter π is the projection operator
 - P lists the attributes that you wish to retain
 - Note: changes the schema of the output relation!

Relational Projection (cont'd)

- So $\pi_{drinker}(\sigma_{beer = Bud}(LIKES))$ is what we really want
- If the state of $LIKES(\underline{drinker, beer})$ is $\{(Joe, Bud), (Sam, Bud), (Sue, Coors), (John, Beast)\}$
- Then $\pi_{drinker}(\sigma_{beer = Bud}(LIKES))$ gives us a one attribute relation (attribute name `drinker`) and the tuples $\{(Joe), (Sam)\}$

The Various Join Operators

- In relational model, the answer to a query is often spread across multiple relations
- Can be put together via the various *join* operators
- All of these are based upon the *cross product* operation, denoted by a \times
- Given R and S , then $R \times S$ returns the following:

$result = \{ \}$

For r in R

For s in S

$result = result \cup (r \bullet s)$

Cross Product

- Say we have LIKES and SERVES (bar, beer)
- If (Moe's, Bud) is in SERVES, it means "Moe's" serves "Bud".
- The state of LIKES (drinker, beer) is { (Joe, Bud), (Sam, Bud), (Sue, Coors), (John, Beast) }
- The state of SERVES (bar, beer) is { (Moe's, Bud), (Moe's, Beast) }
- Then $LIKES \times SERVES$ is...

Cross Product (cont'd)

$$\{ (\text{Joe}, \text{Bud}), (\text{Sam}, \text{Bud}), (\text{Sue}, \text{Coors}), (\text{John}, \text{Beast}) \} \times \\ \{ (\text{Moe's}, \text{Bud}), (\text{Moe's}, \text{Beast}) \} =$$

$$\{ (\text{Joe}, \text{Bud}, \text{Moe's}, \text{Bud}), (\text{Sam}, \text{Bud}, \text{Moe's}, \text{Bud}), (\text{Sue}, \text{Coors}, \text{Moe's}, \text{Bud}), (\text{John}, \text{Beast}, \text{Moe's}, \text{Bud}), (\text{Joe}, \text{Bud}, \text{Moe's}, \text{Beast}), (\text{Sam}, \text{Bud}, \text{Moe's}, \text{Beast}), (\text{Sue}, \text{Coors}, \text{Moe's}, \text{Beast}), (\text{John}, \text{Beast}, \text{Moe's}, \text{Beast}) \}$$

Cross Product (con't)

- So now, what if we want all people who can get a beer that they like at "Moe's"
- Can use

$TEMP(drinkr, b1, bar, b2) \leftarrow LIKES \times (\sigma_{bar = Moes}(SERVES))$

- For convenience, this specifies a temporary relation to hold the result of the cross product
- Followed by:

$\pi_{drinkr}(\sigma_{b1 = b2}(TEMP))$

The Join Operator

- All of the time, we end up doing cross product followed by selection to “link” across a foreign key
- Can use the *join operator* as shorthand $R \bowtie_B S$
- This does a cross product over R and S , then applies a relational selection using the predicate B
- Ignoring the projection, the last query could be:
 - $LIKES \bowtie_{(L.BAR = S.BAR)} (\sigma_{bar = Moes}(SERVES))$
- Note: convention is to use the “dot” notation to disambiguate attribute names in join predicate

The Natural Join Operator

- Most of the time, the join is a so-called “equi-join” over all attributes having the same name, followed by deletion of duplicate attributes
- Can use the *natural join operator* as shorthand for this: $R * S$

The Natural Join Operator (cont'd)

- Example: $LIKES * (\sigma_{bar = Moe's}(SERVES))$
- This finds all attributes in LIKES and SERVES that have the same name (LIKES.beer and SERVES.beer)
- Then does an equi-join on them
- Then removes the redundant attributes, resulting in the output schema (drinker, beer, bar)
- So all people who can get a beer they like at Moe's is:

$$\pi_{drinker}(LIKES * (\sigma_{bar = Moe's}(SERVES)))$$

The Natural Join Operator (cont'd)

- A bigger example. Say we have

$R(a, b, c, d, e, f)$

$S(d, e, f, g, h, i)$

- Then $R * S$ does a join with the predicate

$$R.d = S.d \wedge R.e = S.e \wedge R.f = S.f$$

- And gives the output schema

$(a, b, c, d, e, f, g, h, i)$

Set Operations

- RA also includes the standard set operations
- *Subtraction*: $R - S$ is all tuples in R but not in S
- *Union*: $R \cup S$ is all tuples in R or in S
- *Intersection*: $R \cap S$ is all tuples in R and in S
- Note: to be applicable, R and S must have the same schema
- Sometimes, the *rename* operation is useful: $\varphi_P(R)$
 - This takes R and changes the names to those in P
- So $LIKES - \varphi_{drinker, bar}(SERVES)$ is a valid (but really strange!) RA expression

Set Operations (cont'd)

- Ex: say we want people who like "Bud" or "The Beast" (aka "Milwaukee's Best") but not both

$$BUD \leftarrow \pi_{drinker}(\sigma_{beer = Bud}(DRINKER))$$

$$BEAST \leftarrow \pi_{drinker}(\sigma_{beer = Beast}(DRINKER))$$

$$ANSWER \leftarrow BUD \cup BEAST - (BUD \cap BEAST)$$

Next Week

- We will do a set of in-class problems that will give examples of how to write more complex RA queries