CS 2441: Database Systems

Formal Query Languages:
Relational Algebra and Calculus

Codd’s Relational Algebra

- A set of mathematical operators that compose, modify, and combine tuples within different relations

- Relational algebra operations operate on relations and produce relations (“closure”)
  
f: Relation → Relation
  f: Relation x Relation → Relation
Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The *schema for the result* of a given query is also fixed! Determined by definition of query language constructs.

- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL

Relational Algebra

- Basic operations:
  - *Selection* ($\sigma$) Selects a subset of rows from relation.
  - *Projection* ($\pi$) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* (−) Tuples in reln. 1, but not in reln. 2.
  - *Union* (∪) Tuples in reln. 1 and in reln. 2.

- Additional operations:
  - Intersection, *join*, assignment, division, renaming: Not essential, but (very!) useful.

- Since each operation returns a relation, operations can be composed! (Algebra is “closed”.)
Relational Algebra Expression: Syntax

- RA operators operate on relations and produce relations – closed algebra
  - Defined recursively
- (B) basic expression consists of a relation in the schema or a constant relation
  - What is a constant relation?
- (R) Let $E_1$ and $E_2$ be RA expressions, then

RA expressions..contd..

- $(E_1 \cup E_2)$ is a RA expression
- $(E_1 - E_2)$ is a RA expression
- $(E_1 \times E_2)$ is a RA expression
- $\sigma_P(E_1)$ is a RA expression
- $\pi_S(E_1)$ is a RA expression
- $\rho_R(E_1)$ is a RA expression

- A CFG for the above language?
  - Similar to regular expressions...
Relational Algebra Operators

- Can be composed
  - If R1, R2 are relations (sets), then R1 <op> R2 is also a relation (set) -- <op> is any of the relational algebra operators
    - Closed algebra – how is closure defined ??
- Operations are defined as Set operations
  - Input is a set, output is a set
    - SQL allows duplicated, RA does not

Data Instance for Mini-Banner Example

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>Takes</th>
<th>COURSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>sid</td>
<td>exp-grade</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>Jack</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td>4</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fid</th>
<th>name</th>
<th>PROFESSOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Narahari</td>
<td>1 550-0103</td>
</tr>
<tr>
<td>2</td>
<td>Youssef</td>
<td>2 700-1003</td>
</tr>
<tr>
<td>8</td>
<td>Choi</td>
<td>8 501-0103</td>
</tr>
</tbody>
</table>
**Projection, \( \Pi_\alpha \)**

- Given a list of column names \( \alpha \) and a relation \( R \), \( \pi_\alpha (R) \) extracts the columns in \( \alpha \) from the relation.
- Example: find sid and grade from enrollment table

<table>
<thead>
<tr>
<th>sid</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

**Note:** duplicate elimination. In contrast, SQL returns by default a multiset and duplicates must be explicitly removed.

**Selection, \( \sigma_\theta \)**

- Selection \( \sigma_\theta R \) takes a relation \( R \) and extracts those rows from it that satisfy the condition \( \theta \). Example:

<table>
<thead>
<tr>
<th>sid</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

\[ \sigma_{\exp-grade="A" \land \text{sid}="3"}(\text{Takes}) \]
Complex Predicate Conditions

- Conditions are built up from boolean-valued operations on the field names.
  - exp-grade > “A”, name = “Jill”, STUDENT.sid=Takes.sid
- RA allows comparison predicate on attributes
  - =, not=, >, <, >=, <=
- Larger predicates can be formed using logical connectives – or (\(\lor\)) and and (\(\land\)) and not (\(\lnot\))
- Selection predicate can include comparison between attributes
- We don’t lose any expressive power if we don’t have complex predicates in the language, but they are convenient and useful in practice.

Product X

- “Join” is a generic term for a variety of operations that connect two relations. The basic operation is the cartesian product, R \(\times\) S, which concatenates every tuple in R with every tuple in S. Example:

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>SCHOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
</tr>
</tbody>
</table>

<p>| STUDENT \times SCHOOL |</p>
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>school</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>UPenn</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>GWU</td>
</tr>
</tbody>
</table>

What if the attribute of SCHOOL was called “name”?
Basic Product/Join Operation

- The product $\times$ operation allows combination of info from two tables – it is the set product.
- $R_1 \times R_2$ is collection of tuples from cross product of the two relations.
- If $R_1$ has $k$ columns and $R_2$ has $n$ columns then $R_1 \times R_2$ has $k.n$ columns.
  - Resulting schema is concatenation of two schemas.
  - Refer to attribute $B_i$ of relation $R_1$ as $R_1.B_i$.
    - Customer.Cust-ID

Product/Join

- Tuple in $R_1 \times R_2$ constructed by associating a tuple from $R_1$ with every tuple in $R_2$.
- If $R_1$ has $n_1$ tuples and $R_2$ has $n_2$ tuples how many does $R_1 \times R_2$ have?
- Same attribute can appear in both tables?
  - “link” between the two tables.
Join, $\bowtie$: A Combination of Product and Selection

- Products are hardly ever used alone; they are typically used in conjunction with a selection.
- Example: Find students (id and name) and courses they took with grades and cid

$$\sigma_{\text{STUDENT}.\text{sid} = \text{Takes}.\text{sid}} (\text{STUDENT} \times \text{Takes}) = \text{STUDENT} \bowtie_{\text{STUDENT}.\text{sid} = \text{Takes}.\text{sid}} \text{Takes}$$

<table>
<thead>
<tr>
<th>sid:1</th>
<th>name</th>
<th>sid:2</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Alex</td>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Alex</td>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

“Natural” Join, $\bowtie$

- The most common join to do is an equality join of two relations on commonly named fields, and to leave one copy of those fields in the resulting relation. Example:

$$\text{STUDENT} \bowtie \text{Takes} = \rho_{\text{sid}:1 \rightarrow \text{sid}} (\Pi_{\text{sid}:1, \text{name}, \text{exp-grade}, \text{cid}} (\text{STUDENT} \bowtie_{\text{STUDENT}.\text{sid} = \text{Takes}.\text{sid}} \text{Takes}))$$

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Nick</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Nick</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>Sina</td>
<td>F</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

What if all the field names are the same in the two relations?
What if the field names are all disjoint?
**Union ∪**

- If two relations have the same structure (Database terminology: are **union-compatible**. Programming language terminology: have the same type) we can perform set operations.

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Darby</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
</tr>
<tr>
<td>3</td>
<td>Dan</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Darby</td>
</tr>
<tr>
<td>12</td>
<td>Youssef</td>
</tr>
<tr>
<td>18</td>
<td>Choi</td>
</tr>
</tbody>
</table>

**Type Matching for Set operations**

- Same number of attributes
- Same type of attributes
  - Each position must match domain
    - Real systems sometimes allow sub-types: CHAR(2) and CHAR(20)
**Difference** –

- Another set operator. Example:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Darby</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
</tr>
<tr>
<td>3</td>
<td>Dan</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Darby</td>
</tr>
<tr>
<td>12</td>
<td>Youssef</td>
</tr>
<tr>
<td>18</td>
<td>Choi</td>
</tr>
</tbody>
</table>

**STUDENT** – **FACULTY** – **STUDENT – FACULTY**

**Intersection??**

- People who are both Students and Faculty ??
- Do we need an Intersection operator?
**Deriving Intersection**

Intersection: as with set operations, derivable from difference

\[ A \cap B \equiv (A - (A-B)) = (B - (B-A)) \]

**Rename, \( \rho_\alpha(R) \)**

- The rename operator can be expressed several ways:
  - The book has a very odd definition that’s not algebraic BUT is more readable!
  - An alternate definition:
    - \( \rho_\alpha(x) \): Takes the relation \( x \) and returns a copy of the relation with the name \( \alpha \).
    - General Def: can rename only attribute list with new names \( \beta \)
  - Rename isn’t all that useful, except if you join a relation with itself
**Rename Operator**

- Variations allow renaming of specific attributes
  - $\rho_{X(C,D)}(R(A,B))$
    - Relation $R$ renamed to $X$
    - Fields $A,B$ in $R$ are now renamed to $C,D$ in $X$

**More examples…**

- This completes the basic operations of the relational algebra.
  - These form a complete set of operations
  - Additional operators provided for better writability
- Try writing queries for these:
  - The IDs of students named “Bob”
  - The names of students expecting an “A”
  - The names of students in Youssef’s class
  - The sids and names of students not enrolled in any class
Examples

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>Takes</th>
<th>COURSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Sina</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sid</td>
<td>exp-grade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PROFESSOR

<table>
<thead>
<tr>
<th>fid</th>
<th>name</th>
<th>Teaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Narahari</td>
<td>1 550-0103</td>
</tr>
<tr>
<td>2</td>
<td>Youssef</td>
<td>2 700-1003</td>
</tr>
<tr>
<td>8</td>
<td>Stanton</td>
<td>8 501-0103</td>
</tr>
</tbody>
</table>

Recap: How to write a RA query?

- Find out which tables you need to access
  - Compute $\times$ of these tables
- What are the conditions/predicates you need to apply?
  - Determines what select $\sigma$ operators you need to apply
- What attributes/columns are needed in result
  - Determines what project $\pi$ operators you need
Modifying the Database

- Need to insert, delete, update tuples in the database
- What is insert?
  - Add a new tuple to existing set = Union
- What is delete?
  - Remove a tuple from existing set = Set difference
- How to update attribute to new value?
  - Need new operator: $\delta$

Another Example: Bank Database

- Simplified Bank database
  - Bank has a number of branches at different locations
  - Bank handles customer accounts and loans
- Four relations/tables
  - Customer: stores info about customer
  - Deposit: stores accounts at the branch/bank
  - Loan: stores loans given by bank/branch
  - Branch: stores info about the branch
**Schema of Bank DB**

- **Customer** (CustID, Name, street, city, zip)
  - Customer ID, Name, and Address info: street, city, zip
- **Deposit** (CustID, Acct-num, balance, Branch-name)
  - Customer ID, Account number, Balance in account, name of branch where account is held
- **Loan** (CustID, Loan-num, Amount, Branch-name)
  - Customer ID, loan number, amount of loan...
- **Branch** (Branch-name, assets, Branch-city)

**Modifying Database**

- Delete all accounts of Customer with CustID=3333
  - Deposit ← Deposit – (tuples of CustID 3333)
- Insert tuple (4444, Downtown, 1000, 1234)
  - Deposit ← Deposit ∪ (4444, Downtown, 1000, 1234)
- Update: \( \delta_{A \rightarrow E}(R) \)
  - Update attribute A to E for tuples in relation R
  - \( \delta_{balance \leftarrow 1.05\times balance}(Deposit) \) : updates balances
  - Can also specify selection condition on Deposit
    - Update balances only for customers with CustID=1234
Views: Important Concept

- Relational Model, using SQL, allows definition of a view
  - View is a virtual relation
  - Executed each time it is referenced
  - More when we get to SQL

Next . . .

- Quick look at Relational Calculus
  - Tuple calculus
  - Domain calculus

- SQL query language
Relational Calculus: An Equivalent, But Very Different, Formalism

- Codd invented a relational calculus that he proved was equivalent in expressiveness
  - Based on a subset of first-order logic – declarative, without an implicit order of evaluation
    - Tuple relational calculus
    - Domain relational calculus
  - More convenient for describing certain things, and for certain kinds of manipulations
- The database uses the relational algebra internally, but query languages (e.g., SQL) use concepts from the relational calculus

Relational Calculus

- 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.
Tuple Relational Calculus (RC)

- A **tuple variable** is a variable whose values can be tuples of a relational schema.

- Formula/Query in RC is expressed as:
  \[ \{ t \mid P(t) \} \]
  - \( t \) is a tuple variable
  - \( P(t) \) is property of tuple \( t \); it is a predicate formula that describes properties of the tuple variable
    - Thereby defining the possible values of \( t \)
  - Result is set of all tuples for which predicate \( P \) is true

- Find all loans at the Downtown branch

\[ \{ t \mid t \in \text{Loan} \land t[\text{branch-name}] = "\text{Downtown}" \} \]
RC Formulas

- **Atomic formula:**
  - $t \in Rname$, or $X op Y$, or $X op$ constant
  - $op$ is one of $<, >, =, \leq, \geq, \neq$

- **Formula:**
  - an atomic formula, or
  - $\neg p, p \land q, p \lor q$, where $p$ and $q$ are formulas, or
  - $\exists X (p(X))$, where variable $X$ is free in $p(X)$, or
  - $\forall X (p(X))$, where variable $X$ is free in $p(X)$

- The use of quantifiers $\exists X$ and $\forall X$ is said to **bind** $X$.
  - A variable that is not bound is free.

Domain Relational Calculus (DRC)

Queries have form:

\[
\{<x_1, x_2, \ldots, x_n> \mid p\}
\]

- **Predicate:** boolean expression over $x_1, x_2, \ldots, x_n$
- **Answer** includes all tuples that make the formula *true*.
- The variables come from the domain of the attributes in the relation schema
  - in contrast to the tuple calculus where variables are tuples
**Domain Relational Calculus**

- Define domain of each attribute in result set and the type
- Find branch-name, loan number, amount, custID for loans over $1200

\[
\{ <b,l,c,a> | <b,l,c,a> \in \text{loan} \land b='Downtown' \}
\]

**More definitions of formulas in RC**

- This is nothing but “cs123 on steroids”!!
More Complex Predicates in Relational Calculus (for RC and DRC)

Starting with these atomic predicates, build up new predicates by the following rules:

- **Logical connectives:** If p and q are predicates, then so are \( p \land q \), \( p \lor q \), \( \neg p \), and \( p \Rightarrow q \)
  - \((x>2) \land (x<4)\)
  - \((x>2) \land \neg(x>0)\)

- **Existential quantification:** If p is a predicate, then so is \( \exists x. p \)
  - \( \exists x. (x>2) \land (x<4) \)

- **Universal quantification:** If p is a predicate, then so is \( \forall x. p \)
  - \( \forall x. x>2 \)
  - \( \forall x. \exists y. y>x \)

Logical Equivalences

- Recall from CS123 😊
- There are two logical equivalences that are heavily used:
  - \( p \Rightarrow q \equiv \neg p \lor q \)
    (Whenever p is true, q must also be true.)
  - \( \forall x. p(x) \equiv \neg \exists x. \neg p(x) \)
    (p is true for all x)

- The second can be a lot easier to check!

- **Example:**
  - The highest course number offered
Free and Bound Variables

- The use of quantifiers $\forall$ or $\exists$ in a formula is said to bind the variables.
- A variable $v$ is bound in a predicate $p$ when $p$ is of the form $\forall v \ldots$ or $\exists v \ldots$
- A variable occurs free in $p$ if it occurs in a position where it is not bound by an enclosing $\forall$ or $\exists$.
- Examples:
  - $x$ is free in $x > 2$.
  - $x$ is bound in $\exists x. x > y$.
- Important restriction: the variables that appear to the left of `$|$` must be the only free variables in the formula $p(\ldots)$.

Can Rename Bound Variables Only

- When a variable is bound one can replace it with some other variable without altering the meaning of the expression, providing there are no name clashes.
- Example: $\exists x. x > 2$ is equivalent to $\exists y. y > 2$.
- Otherwise, the variable is defined outside our “scope”…
Free Variables

- A variable $v$ is bound in a predicate $p$ when $p$ is of the form $\forall v \ldots$ or $\exists v \ldots$
- Important restriction: the variables that appear to the left of `|' must be the only free variables in the formula $p(...)$.  
- Implication: the values that the free variables can legally take on are the results of the query!

Safety of Operators

- Query of the form $\exists t \in R ( Q(t))$
  - There exists tuple $t$ in set/relation $R$ such that predicate $Q$ is true
- Safety of Expressions
  - What about $\{ t | \sim( t \in \text{loan}) \}$
    - Infinitely many tuples outside loan relation
**Safety of Expressions**

- A query is **safe** if no matter how we instantiate the relations, it always produces a finite answer
  - **Domain independent**: answer is the same regardless of the domain in which it is evaluated
  - Unfortunately, both this definition of safety and domain independence are **semantic conditions**, and are **undecidable**
- There are **syntactic conditions** that are used to guarantee “safe” formulas
  - One solution: For each tuple relational formula $P$, define domain $\text{Dom}(P)$ which is set of all values referenced by $P$
  - The formulas that are expressible in real query languages based on relational calculus are all “safe”
    - Many DB languages include additional features, like recursion, that must be restricted in certain ways to guarantee termination and consistent answers

**Safety and Termination Guarantees**

- There are **syntactic conditions** that are used to guarantee “safe” formulas
  - The definition is complicated, and we won’t discuss it; you can find it in Ullman’s *Principles of Database and Knowledge-Base Systems*
  - The formulas that are expressible in real query languages based on relational calculus are all “safe”
Examples: Relational Calculus

- Find branch name, ID, loan number and amount for loans over $1200
  - What is the “type” of the elements in the result, i.e., where do they come from?
  - What is the property of the elements?

\{ t \mid (t \in Loan) \land (t[amount] > 1200) \}
Domain Relational Calculus

- Define domain of each attribute in result set and the type
- Find branch-name, loan number, amount, custID for loans over $1200

\[ \{ <b,l,c,a> \mid <b,l,c,a> \in \text{loan} \ a > 1200 \} \]

Relational calculus

- What if the type has to be inferred?
- Find only Customer ID attribute in the previous example
  - This type has to be inferred by the query
  - Tuples on ID, for which there is a tuple in Loan with same CustID and amount > 1200.
Cross Products in TRC

- Find Customer names of customers who have loans greater than 1200
  - for Cust ID with loans we had
    - \{t \mid \exists s \in Loan ( s[custID]=t[CustID] \\
    \land s[amount]>1200) \}\n  
- How about name? It exists in Customer relation?
  - For tuple c \in Customer what property does c have?
Cross products in TRC

\{t \mid \exists s \in Loan, \exists c \in Customer ( t[name]=c[name] \land s[amount]>1200 \\
\land c[custID]=s[CustID] ) \}\}

Summary: Relational Model, Formal Query languages

- **Relational Completeness:** Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.
- Relational calculus is non-operational
  - users define queries in terms of what they want, not in terms of how to compute it.
- Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.
Recap: How to write a RA query?

- Find out which tables you need to access
  - Compute $\times$ of these tables
- What are the conditions/predicates you need to apply?
  - Determines what select $\sigma$ operators you need to apply
- What attributes/columns are needed in result
  - Determines what project $\pi$ operators you need

Limitations of the Relational Algebra / Calculus

Can’t do:
- Aggregate operations
- Recursive queries
- Complex (non-tabular) structures

- Most of these are expressible in SQL, XQuery
  - using other special operators
- Sometimes we even need the power of a Turing-complete programming language
Why Formal languages? Example: Optimization is Based on Algebraic Equivalences

- Relational algebra has laws of commutativity, associativity, etc. that imply certain expressions are equivalent in semantics.
- They may be different in cost of evaluation!
  - \( \sigma_{(P_1 \land P_2)} (R) = \sigma_{P_1} (\sigma_{P_2} (R)) \)
  - \( (R_1 \bowtie R_2) = (R_2 \bowtie R_1) \)
  - \( (R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3) \)

- Query optimization finds the most efficient representation to evaluate (or one that’s not bad)

The Big Picture: SQL to Algebra to Query Plan to Web Page

SELECT * FROM STUDENT, Takes, COURSE WHERE STUDENT.sid = Takes.sid AND Takes.clD = cid