Relational Query Languages

• Query languages: Allow manipulation and retrieval of data from a database.

• Relational model supports simple, powerful QLs:
  • Strong formal foundation based on logic.
  • Allows for optimization.
  • Query Languages != programming languages
  • QLs not expected to be “Turing complete”.
  • QLs not intended to be used for complex calculations.
  • QLs support easy, efficient access to large data sets.
Relational Query Languages

- Two mathematical Query Languages form the basis for “real” query languages (e.g. SQL), and for implementation:
  - Relational Algebra: More operational, very useful for representing execution plans.
    - Basis for SEQUEL
  - Relational Calculus: Let’s users describe WHAT they want, rather than HOW to compute it. (Non-operational, declarative.)
    - Basis for QUEL
Mathematical Foundations: Cartesian Product

• Let:
  • A be the set of values \{ a_1, a_2, \ldots \}
  • B be the set of values \{ b_1, b_2, \ldots \}
  • C be the set of values \{ c_1, c_2, \ldots \}

• The Cartesian product of A and B (written $A \times B$) is the set of all possible ordered pairs $(a_i, b_j)$, where $a_i \in A$ and $b_j \in B$.

• Similarly:
  • $A \times B \times C$ is the set of all possible ordered triples $(a_i, b_j, c_k)$, where $a_i \in A$, $b_j \in B$, and $c_k \in C$.
  • $A_1 \times A_2 \times \ldots \times A_n$ is the set of all possible ordered tuples $(a_{1i}, a_{2j}, \ldots, a_{nk})$, where $a_{de} \in A_d$
Cartesian Product Example

- $A = \{\text{small, medium, large}\}$
- $B = \{\text{shirt, pants}\}$

<table>
<thead>
<tr>
<th>A x B</th>
<th>Shirt</th>
<th>Pants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>(Small, Shirt)</td>
<td>(Small, Pants)</td>
</tr>
<tr>
<td>Medium</td>
<td>(Medium, Shirt)</td>
<td>(Medium, Pants)</td>
</tr>
<tr>
<td>Large</td>
<td>(Large, Shirt)</td>
<td>(Large, Pants)</td>
</tr>
</tbody>
</table>

- $A \times B = \{(\text{small, shirt}), (\text{small, pants}), (\text{medium, shirt}), (\text{medium, pants}), (\text{large, shirt}), (\text{large, pants})\}$
- Set notation
Example: Cartesian Product

• What is the Cartesian Product of AxB?
  • A = \{perl, ruby, java\}
  • B = \{necklace, ring, bracelet\}

• What is BxA?

<table>
<thead>
<tr>
<th>A x B</th>
<th>Necklace</th>
<th>Ring</th>
<th>Bracelet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl</td>
<td>(Perl, Necklace)</td>
<td>(Perl, Ring)</td>
<td>(Perl, Bracelet)</td>
</tr>
<tr>
<td>Ruby</td>
<td>(Ruby, Necklace)</td>
<td>(Ruby, Ring)</td>
<td>(Ruby, Bracelet)</td>
</tr>
<tr>
<td>Java</td>
<td>(Java, Necklace)</td>
<td>(Java, Ring)</td>
<td>(Java, Bracelet)</td>
</tr>
</tbody>
</table>
Mathematical Foundations: Relations

• The domain of a variable is the set of its possible values
• A relation on a set of variables is a subset of the Cartesian product of the domains of the variables.
  • Example: let \( x \) and \( y \) be variables that both have the set of non-negative integers as their domain
  • \( \{(2,5),(3,10),(13,2),(6,10)\} \) is one relation on \( (x, y) \)
• A table is a subset of the Cartesian product of the domains of the attributes. Thus a table is a mathematical relation.
• Synonyms:
  • Table = relation
  • Row (record) = tuple
  • Column (field) = attribute
Mathematical Relations

• In tables, as in mathematical relations, the order of the tuples does not matter but the order of the attributes does.

• The domain of an attribute usually includes NULL, which indicates the value of the attribute is unknown.
What is an Algebra?

- Mathematical system consisting of:
  - Operands --- variables or values from which new values can be constructed.
  - Operators --- symbols denoting procedures that construct new values from given values.
What is Relational Algebra?

• An algebra whose operands are relations or variables that represent relations.
• Operators are designed to do the most common things that we need to do with relations in a database.
• The result is an algebra that can be used as a query language for relations.
Relational Algebra

- A collection of operations that users can perform on relations to obtain a desired result
- This is an introduction and only covers the algebra needed to represent SQL queries
  - Select, project, rename
  - Cartesian product
  - Joins (natural, condition, outer)
  - Set operations (union, intersection, difference)
- Relational Algebra treats relations as sets: duplicates are removed
Relation Instance vs. Schema

• Schema of a relation consists of
  • The name of the relation
  • The fields of the relation
  • The types of the fields
• For the Student table

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
</tr>
<tr>
<td>55517</td>
<td>Ali</td>
<td>ali@math</td>
<td>Sep 22, 1989</td>
<td>3.11</td>
</tr>
<tr>
<td>55518</td>
<td>Smith</td>
<td>smith@math</td>
<td>Nov 30, 1991</td>
<td>3.32</td>
</tr>
</tbody>
</table>

• Schema = Student(SID int, Name char(20), Login char(20), DoB date, GPA real )
• Instance of a relation is an actual collection of tuples
  • Table with rows of values
• Database schema is the schema of the relations in a database
Relational Algebra

One or more relations

Operation

Resulting Relation

For each operation: both the operands and the result are relations
Facts on relational algebra queries

• 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
Basic Relational Algebra Operations

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

- **Additional operations:**
  - Intersection, join, division, renaming: Not essential, but (very) useful.
  - Each operation returns a relation, operations can be composed (Algebra is “closed”)
    - Contains the closure property
  - Since operators’ input is a relation and its output is a relation we can string these operators together to form a more complex operator
Basic Operation: Projection

- Deletes attributes that are not in projection list.
- Schema of result contains exactly the fields in the projection list, with the same names that they had in the input relation.
- Syntax: $\pi_{f_1, f_2} \ldots$ (Relation)
- Projection operator has to eliminate duplicates. (Why?)
  - Note: real systems typically do not eliminate duplicates unless the user explicitly asks for it. (Why not?)

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10,1990</td>
<td>3.82</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
</tr>
<tr>
<td>55517</td>
<td>Ali</td>
<td>ali@math</td>
<td>Sep 22, 1989</td>
<td>3.11</td>
</tr>
<tr>
<td>55518</td>
<td>Smith</td>
<td>smith@math</td>
<td>Nov 30, 1991</td>
<td>3.32</td>
</tr>
</tbody>
</table>

\[
\pi_{\text{Sid}, \text{Name}} (S1) \quad \pi_{\text{Sid}} (S1)
\]
Basic Operations: Select $\sigma$

- Selects rows that satisfy the selection condition.
  - No duplicates in result (Why?)
  - Schema of result is identical to schema of input relation
- Selection predicates can include: <, >, =, !=, and, or, not
  - Examples:
    - $\sigma_{Sid \neq 55516}$ (S1)
    - $\sigma_{Name = 'Smith'}$ (S1)
- Syntax: $\sigma_{Conditional}$ (Relation)
Operator composition example.

Select and Project

\[ \pi_{Sid, Name} \left( \sigma_{Sid > 55516} (S1) \right) \]
Union

- Takes two input relations, which must be union-compatible:
  - Same number of fields.
  - ‘Corresponding’ fields have the same type.

\[ S1 \cup S2 \]
Intersection \( \cap \)

Occurs in S1 and S2

\[ S1 \cap S2 \]

Set difference

Occurs in S1 but not in S2

\[ S1 - S2 \]

### S1

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
</tr>
<tr>
<td>55517</td>
<td>Ali</td>
<td>ali@math</td>
<td>Sep 22, 1989</td>
<td>3.11</td>
</tr>
<tr>
<td>55518</td>
<td>Smith</td>
<td>smith@math</td>
<td>Nov 30, 1991</td>
<td>3.32</td>
</tr>
</tbody>
</table>

### S2

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Chen</td>
<td>chen@ccs</td>
<td>Jan 10, 1990</td>
<td>3.01</td>
</tr>
<tr>
<td>55519</td>
<td>Alton</td>
<td>alton@hist</td>
<td>Jun 11, 1992</td>
<td>2.07</td>
</tr>
<tr>
<td>55517</td>
<td>Ali</td>
<td>ali@math</td>
<td>Sep 22, 1989</td>
<td>3.11</td>
</tr>
<tr>
<td>55518</td>
<td>Smith</td>
<td>smith@math</td>
<td>Nov 30, 1991</td>
<td>3.32</td>
</tr>
</tbody>
</table>
Cross-Product x

Each row within S1 is paired with each row of C1

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
<th>CID</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
<td>History 101</td>
<td>C</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
<td>History 101</td>
<td>C</td>
</tr>
</tbody>
</table>

Result schema has one field per field of S1 and C1, with field names `inherited` if possible.
Rename \( \rho \)

- Reassign the field names
  \[ \rho(\mathbf{C}(1\rightarrow S1.sid, 6\rightarrow C1.sid), S1 \times C1) \]

### S1

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
<th>C1.Sid</th>
<th>CID</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
<td>55515</td>
<td>History 101</td>
<td>C</td>
</tr>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
<td>55515</td>
<td>Biology 220</td>
<td>A</td>
</tr>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
<td>55515</td>
<td>Anthro 320</td>
<td>B</td>
</tr>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
<td>55515</td>
<td>Music 101</td>
<td>A</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
<td>55516</td>
<td>History 101</td>
<td>C</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
<td>55516</td>
<td>Biology 220</td>
<td>A</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
<td>55516</td>
<td>Anthro 320</td>
<td>B</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
<td>55516</td>
<td>Music 101</td>
<td>A</td>
</tr>
</tbody>
</table>

### C1

<table>
<thead>
<tr>
<th>Sid</th>
<th>Cid</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>History 101</td>
<td>C</td>
</tr>
<tr>
<td>55516</td>
<td>Biology 220</td>
<td>A</td>
</tr>
<tr>
<td>55517</td>
<td>Anthro 320</td>
<td>B</td>
</tr>
<tr>
<td>55518</td>
<td>Music 101</td>
<td>A</td>
</tr>
</tbody>
</table>

Prepend the name of the original relation to the fields having a collision

Naming columns and result set to C
Conditional Join

- Accepts a conditional
- Operation equivalent to:
  - \( S_1 \bowtie_c C_1 = \sigma_c (S_1 \times C_1) \)
- Filters out tuples according to the conditional expression

\[ S_1 \bowtie_{gpa > 3.0} C_1 \]
Equijoin \( \bowtie_c \)

- What does it do: performs a filtered Cartesian product
- Filters out tuples where the attribute that have the same name have a different value
- \( S \bowtie_c S_1\cdot sid = C_1\cdot sid \)

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
<th>CID</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10,1990</td>
<td>3.82</td>
<td>History 101</td>
<td>C</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
<td>Biology 220</td>
<td>A</td>
</tr>
</tbody>
</table>

Only one copy of Sid is in the resultant relation
Natural Join

• What does it do: performs a filtered Cartesian product
• Filters out tuples where the attribute that have the same name have a different value

\[ S1 \bowtie C1 \]

No need to specify field list

Natural join is equivalent to:
Cartesian project (x)
Selection (\(\sigma\))
Projection (\(\pi\))
Precedence of Relational Operators

1. \([ \sigma, \pi, \rho]\) (highest).
2. \([ \times, \bowtie]\)
3. \(\cap\)
4. \([ \cup, \_\_]\)
Schema of the Resulting Table

- Union, intersection, and difference operators
  - the schemas of the two operands must be the same, so use that schema for the result.
- Selection operator
  - schema of the result is the same as the schema of the operand.
- Projection operator
  - list of attributes determines the schema.
Relational Algebra: Summary

• The relational model has rigorously defined query languages that are simple and powerful.
  • Relational algebra is more operational
  • Useful as internal representation for query evaluation plans
• Several ways of expressing a given query
• A query optimizer should choose the most efficient version
Relational Calculus

- Relational Calculus is an alternative way for expressing queries
  - Main feature: specify what you want, not how to get it
  - Many equivalent algebra “implementations” possible for a given calculus expression
- In short: SQL query without aggregation = relational calculus expression
- Relational algebra expression is similar to program, describing what operations to perform in what order
What is Relational Calculus?

• It is a formal language based upon predicate calculus

• It allows you to express the set of tuples you want from a database using a formula
Relational Calculus

• Comes in two flavors: Tuple relational calculus (TRC) and Domain relational calculus (DRC)
• TRC: Variables range over (i.e., get bound to) tuples.
• DRC: Variables range over domain elements (= attribute values)
• Both TRC and DRC are subsets of first-order logic
  • We use some short-hand notation to simplify formulas
• Expressions in the calculus are called *formulas*
• Answer tuple = assignment of constants to variables that make the formula evaluate to true
Relational Calculus Formula

- Formula is recursively defined
  - Start with simple atomic formulas (getting tuples (or defining domain variables) from relations or making comparisons of values)
  - And build bigger and more complex formulas using the logical connectives.
Domain Relational Calculus

• Query has the form:
  • \{<x_1, x_2, \ldots, x_n> \mid p(<x_1, x_2, \ldots, x_n>)\}
  • Domain Variable – ranges over the values in the domain of some attribute or is a constant
  • Example: If the domain variable \(x_1\) maps to attribute - Name char(20) then \(x_1\) ranges over all strings that are 20 characters long
    • Not just the strings values in the relation’s attribute
  • Answer includes all tuples \(<x_1, x_2, \ldots, x_n>\) that make the formula \(p(<x_1, x_2, \ldots, x_n>)\) true.
DRC Formulas

• Atomic Formulas
  1. \(<x_1, x_2, \ldots, x_n> \in \text{Relation}\)
     • where Relation is a relation with n attributes
  2. \(X \text{ operation } Y\)
  3. \(X \text{ operation } constant\)
     • Where operation is an operator in the set \{<, >, =, \leq, \geq, \neq\}

• Recursive definition of a Formula:
  1. An atomic formula
  2. \(\neg p, p \land q, p \lor q\), where \(p\) and \(q\) are formulas
  3. \(\exists X(p(X))\), where variable \(X\) is free in \(p(X)\)
  4. \(\forall X(p(X))\), 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.
Free and bound variables

• Let us revisit the definition of a query:
  • \{<x_1, x_2,\ldots, x_n> | p <x_1, x_2,\ldots, x_n>\}
  • Determine what assignment(s) to <x_1, x_2,\ldots, x_n> make | p <x_1, x_2,\ldots, x_n> true
  • There is an important restriction:
    • The variables x_1,\ldots, x_n that appear to the left of `|’ must be the only free variables in the formula p<...>
    • All other variables in the query are bound
Quantifiers: \( \forall x \) and \( \exists x \)

- Variable \( x \) is said to be *subordinate* to the quantifier
  - You are restricting (or binding) the value of the variable \( x \) to set \( S \)
- The set \( S \) is known as the *range* of the quantifier
- \( \forall x \) predicate true for all elements in set \( S \)
- \( \exists x \) predicate true for at least 1 element in set \( S \)
Formula quantifier: $\forall x$

- $\forall$ is called the universal or “for all” quantifier because every tuple in “the universe of” tuples must make $P(x)$ true to make the quantified formula true.
- $\forall x \ (P(x))$ - is only true if $P(x)$ is true for every $x$ in the universe.
- In our example we wrote:
  - $\forall x \in \text{Boats}(\text{color} = \text{"Red"})$
  - It really means:
    - $\forall x \ ((x \in \text{Boats}) \Rightarrow (\text{color} = \text{"Red"}))$
  - $\Rightarrow$ logical implication What does it mean?
    - $A \Rightarrow B$ means that if $A$ is true, $B$ must be true.
    - Since $A \Rightarrow B$ is the same as $\neg A \lor B$
\( A \Rightarrow B \) is the same as \( \neg A \lor B \)

If \( A \) is true and \( B \) is false, the implication evaluates to false.

If \( A \) is not true, \( B \) has no effect on the answer.

Since you have already satisfied the clause with \((\neg A)\),
The expression is always true.
Formula Quantifiers: $\exists X$

- $\exists$ is called the existential or “there exists” quantifier because any tuple that exists in “the universe of” tuples may make $p(x)$ true to make the quantified formula true.
- $\exists X(p(X))$
  - Means $p(X)$ is true for some $X$
  - There exists an $X$ for which $p(X)$ is true
- Shorthand notation
  - $\exists X \in \text{Students}(\text{GPA} = 3.2)$
- Real representation of the formula
  - $\exists X ((X \in \text{Students}) \land (\text{GPA} = 3.2))$
  - And logical operation as opposed to implication
Example : Domain RC

• Find all students with the name ‘Smith’
\{ <I, N,L,D,G > |   <I, N,L,D,G > ∈ Students  ∧  N = ‘Smith’ \}

  ▪ Can reference the relation in the expression as opposed to the attribute name
Anatomy of a DRC query

\{<I, N,L,D,G> | <I, N,L,D,G> ∈ Students ∧ N = ‘Smith’\}

• The condition \(<I, N,L,D,G> ∈ Students\)
  • ensures that the domain variables I, N, L,D, and G are bound to fields of
    the same Students tuple.
    • Maps it to an instance in the Students table
• The symbol ‘|’ from predicate calculus means ‘such that’
• The \(<I, N,L,D,G>\) to the left of `|’ (such that ) says that every tuple
  \(<I, N,L,D,G>\) that satisfies N = ‘Smith’ is in the answer set.
• Calculus expresses answers to the query not operations like in
  relational algebra
DRC example with multiple predicates

- Find all students with the name ‘Smith’ and with a GPA > 3.5
- Just add another predicate \( \{ p \land q \} \)
- \( \{ <I, N, L, D, G> | <I, N, L, D, G> \in \text{Students} \land N = \text{‘Smith’} \land G > 3.5 \} \)

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10, 1990</td>
<td>3.82</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
</tr>
<tr>
<td>55517</td>
<td>Ali</td>
<td>ali@math</td>
<td>Sep 22, 1989</td>
<td>3.11</td>
</tr>
<tr>
<td>55518</td>
<td>Smith</td>
<td>smith@math</td>
<td>Nov 30, 1991</td>
<td>3.32</td>
</tr>
</tbody>
</table>
DRC: returning a field satisfying restrictions on multiple tables

- Find the name of all students that received a ‘C’ in a course
  - Retrieving data across 2 tables
  - \( \{<N> | \exists I, L, D, G(<I, N, L, D, G > \in Students \land \exists Ir, CN, CG(<Ir, CN, CG> \in Courses \land Ir = I \land CG = 'C') )\} \)

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Login</th>
<th>DoB</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>Smith</td>
<td>smith@ccs</td>
<td>Jan 10,1990</td>
<td>3.82</td>
</tr>
<tr>
<td>55516</td>
<td>Jones</td>
<td>jones@hist</td>
<td>Feb 11, 1992</td>
<td>2.98</td>
</tr>
<tr>
<td>55517</td>
<td>Ali</td>
<td>ali@math</td>
<td>Sep 22, 1989</td>
<td>3.11</td>
</tr>
<tr>
<td>55518</td>
<td>Smith</td>
<td>smith@math</td>
<td>Nov 30, 1991</td>
<td>3.32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sid</th>
<th>ClId</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>55515</td>
<td>History 101</td>
<td>C</td>
</tr>
<tr>
<td>55516</td>
<td>Biology 220</td>
<td>A</td>
</tr>
<tr>
<td>55517</td>
<td>Anthro 320</td>
<td>B</td>
</tr>
<tr>
<td>55518</td>
<td>Music 101</td>
<td>A</td>
</tr>
</tbody>
</table>

Resulting Table

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
</tr>
</tbody>
</table>
Parsing a DRC query

Find the name of all students that received a ‘C’ in course

\{<N> \mid \\
    \exists I, L, D, G(<I, N, L, D, G> \in Students \\
    \land \exists I_r, CN, CG_r(<I_r, CN, CG_r> \in Courses \land I_r = I \land CG_r = 'C' )\}\}

• Note the use of \(\exists\) to find a tuple in Courses that `joins with’ the Students tuple under consideration
  • Student Id is the same value in both tables
  • Bound value represented via the variable I_r
Unsafe Queries, Expressive Queries

• It is possible to write syntactically correct calculus queries that have an *infinite* number of answers.
  • Such queries are called unsafe.

• Theorem: Every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC
  • The converse is also true.
  • Relational Completeness: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.
Relational Calculus: Summary

- Relational calculus is non-operational
  - Users define queries in terms of what they want, not in terms of how to compute it. (Declarativeness.)
- Algebra and safe calculus have the same expressive power, leading to the notion of relational completeness.
- Relational calculus had big influence on the design of SQL and Query-by-Example
Practice with relational algebra

Building up a Relational Algebra Function
Division Operation in RA A/B

• Given 2 relations A (courses) and B (students); A/B = let x, yA be two attributes in A and yB is an attribute in B with the same domain as the domain of yB

• A/B = {<x> such that for all <y> in B there exists <x ,y> an element of A = {< x > | ∀< y > ∈ B ∃ < x, y > ∈ A}

• A/B contains all x tuples (courses) such that for every y tuple (students) in B, there is an xy tuple in A.

• Or: If the set of y values (courses) associated with an x value (students) in A contains all y values in B, the x value is in A/B.

  • In general, x and y can be any lists of attributes
  • y is the list of fields in B, and x U y is the list of fields of A.

• Assume x = course id and y = student id - What is the query asking for?

The MEGA-STUDENT(s) someone who has taken all courses that are in the course table
Example of division

Table A

<table>
<thead>
<tr>
<th>Student Id (x)</th>
<th>Course Id (y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>cs200</td>
</tr>
<tr>
<td>10</td>
<td>cs100</td>
</tr>
<tr>
<td>10</td>
<td>cs300</td>
</tr>
<tr>
<td>10</td>
<td>cs400</td>
</tr>
<tr>
<td>20</td>
<td>cs300</td>
</tr>
<tr>
<td>30</td>
<td>cs200</td>
</tr>
<tr>
<td>15</td>
<td>cs400</td>
</tr>
<tr>
<td>15</td>
<td>cs100</td>
</tr>
<tr>
<td>25</td>
<td>cs100</td>
</tr>
<tr>
<td>25</td>
<td>cs200</td>
</tr>
</tbody>
</table>

Instances of B

<table>
<thead>
<tr>
<th>Course Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs200</td>
</tr>
<tr>
<td>cs100</td>
</tr>
<tr>
<td>Cs 200</td>
</tr>
<tr>
<td>cs300</td>
</tr>
</tbody>
</table>

Corresponding Instances of A/B

<table>
<thead>
<tr>
<th>Student Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>
Basic operations for Division

• Compute all x values in A that are not disqualified
  • How is a value disqualified?
  • If by attaching a y value from B, we obtain a tuple NOT in A
    • $\pi_x((\pi_x(A) \times B) - A)$

• $\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$
Step by step process of Division

<table>
<thead>
<tr>
<th>Student Id (x)</th>
<th>Course Id (y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>cs200</td>
</tr>
<tr>
<td>10</td>
<td>cs100</td>
</tr>
<tr>
<td>10</td>
<td>cs300</td>
</tr>
<tr>
<td>10</td>
<td>cs400</td>
</tr>
<tr>
<td>20</td>
<td>cs300</td>
</tr>
<tr>
<td>30</td>
<td>cs200</td>
</tr>
<tr>
<td>15</td>
<td>cs400</td>
</tr>
<tr>
<td>15</td>
<td>cs100</td>
</tr>
<tr>
<td>25</td>
<td>cs100</td>
</tr>
<tr>
<td>25</td>
<td>cs200</td>
</tr>
</tbody>
</table>

\[ (\pi_x(A) \times B) \]

| 10, cs200 |
| 20, cs200 |
| 15, cs200 |
| 25, cs200 |

\[ (\pi_x(A) \times B) - A \]

\[ \pi_x((\pi_x(A) \times B) - A) \]

\[ \pi_x(A) - \pi_x((\pi_x(A) \times B) - A) \]

<table>
<thead>
<tr>
<th>Student Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>
Relational Algebra

• Like ERM modeling there are many ways to solve the problem at hand
• Given the theory behind RA, a sophisticated query optimization engineer can write algorithms that optimize a query
  • Theory in practice