Query Language #1/3: Relational Algebra

Pure, Procedural, and Set-oriented

- To express a query, we use a set of operations. Each operation takes one or more relations as input parameter (set-oriented). Since each operation produces a new relation, the operations can be input parameters. → procedural

- Relational-Algebra operations:
  ⇔ Relational algebra consists of a set of operations. Composing relational algebra operations into an expression is just like composing arithmetic operations (e.g., +, -, /, *) into an arithmetic expression.
  ⇔ Fundamental Set of Relational-Algebra Operations:
    \( \text{Selection} \ (\sigma) \), \( \text{Projection} \ (\pi \Pi) \), \( \text{Union} \ (\cup) \), \( \text{Set-difference} \ (-) \), \( \text{Cartesian-product} \ (\times) \), \( \text{Rename} \ (\rho \rho) \)
  ⇔ Some Additional Operations:
    \( \text{Set-intersection} \ (\cap) \), \( \text{Natural-join} \ (\bowtie) \), \( \text{Outer-join} \ (\Rightarrow \bowtie, \Rightarrow \bowtie=, \text{or} \Rightarrow \bowtie=) \), \( \text{Division} \ (\div) \), \( \text{Assignment} \ (\leftarrow) \)
  ⇔ The additional operations can be expressed through some fundamental operations. However, they are included in the relational algebra to facilitate easier construction of more complex queries. In addition, some people include more operations, e.g. complement operation. These additional
operations do not improve the expressive power of relational
algebra.

- **Selection (sigma σ)**
  - Unary operation on a relation \( r(R) \), which produces a
    relation \( r'(R) \subseteq r(R) \).

  - Let \( F \) be a predicate \( A_i = a \), where \( A_i \) is an attribute name
    in \( R \) and \( a \) is a value in the domain of \( A_i \). Then,
    \[ \sigma_F(r) = \{ t | t \in r \land t[A_i] = a \} \].

  - Predicate can involve:
    - Constant and attributes;
    - Comparative operators: =, >, <, ≤, ≥, and ≠;
    - Logical operators: ∧, ∨, and ¬;
    - Parenthesis: ( and ).

  - **Example1**

    \[
    r
    \]
    \[
    \begin{array}{ccc}
    A & B & C \\
    1 & b1 & c1 \\
    2 & b2 & c1 \\
    3 & b2 & c2 \\
    \end{array}
    \]

    \[ \sigma_{A>1 \land A < 3}(r) \]
    \[
    \begin{array}{ccc}
    A & B & C \\
    2 & b2 & c1 \\
    \end{array}
    \]

  - **Example2**

    \[ \sigma_{\text{branch-name}="Laramie" \land \text{balance}>2000} (\text{account}) \]
• Projection (\(\text{pi } \Pi\))
  
  - \(\Pi_X(r)\) is an unary operation on a relation \(r(R)\), which produces a relation \(r'(X)\) in which \(X \subseteq R\). Any duplicate tuple is eliminated.
  
  \[\Pi_X(r) = \{t (X \subseteq R) \mid t \in r\}\]
  
  - Generalized Projection:
    - \(X\) can be \(F_1, F_2, \ldots, F_n\);
    - Each \(F\) can be:
      - an attribute or an arithmetic expression
      - (e.g., balance, balance*0.3, balance-limit)

  ➢ Example1

  \[
  r
  \]

  \[
  \begin{array}{ccc}
  A & B & C \\
  1 & b_1 & c_1 \\
  2 & b_2 & c_2 \\
  3 & b_2 & c_2 \\
  \end{array}
  \]

  \[
  \Pi_{B,C}(r)
  \]

  \[
  \begin{array}{cc}
  B & C \\
  b_1 & c_1 \\
  b_2 & c_2 \\
  \end{array}
  \]

  ➢ Example2

  \[
  \Pi_{\text{account\#, balance}}\ (\text{account})
  \]

  \[
  \Pi_{\text{account\#, balance*0.25}}\ (\text{account})
  \]
• Union (\(\cup\))

- Set \(r(R) \cup s(S)\) is a relation \(q(R)\), which contains tuples from \(r\) or from \(s\). Any duplicate tuple is eliminated. \(R\) and \(S\) must be the same.

- \(r(R) \cup s(S) = \{ t | t \in r \lor t \in s \}\)

- Example

\[
\begin{array}{ccc}
\text{r} & & \\
A & B & C \\
1 & b1 & c1 \\
2 & b2 & c2 \\
3 & b2 & c2 \\
\end{array}
\]

\[
\begin{array}{cc}
\text{s} & \\
B & C \\
b1 & c1 \\
b4 & c5 \\
\end{array}
\]

\[
\Pi_{B, C}(r) \cup s
\]

\[
\begin{array}{cc}
B & C \\
b1 & c1 \\
b2 & c2 \\
b4 & c5 \\
\end{array}
\]
• Set-difference (-)
  
  ➢ Set r(R) - s(S) is a relation q(R), which contains tuples in r that are not in s. R and S must be the same.

  ➢ r(R)-s(S) = \{ t | t \in r \land t \notin s \}

  ➢ Example

  \[
  \begin{array}{ccc}
  r & & \\
  A & B & C \\
  1 & b_1 & c_1 \\
  2 & b_2 & c_2 \\
  3 & b_2 & c_2 \\
  \end{array}
  \]

  \[
  \begin{array}{ccc}
  s & & \\
  B & C \\
  b_1 & c_1 \\
  b_4 & c_5 \\
  \end{array}
  \]

  \[
  \Pi_{B,C}(r) - s \\
  \begin{array}{ccc}
  B & C \\
  b_2 & c_2 \\
  \end{array}
  \]
• Cartesian-product (\(\times\))

  ➢ Set \(r(R) \times s(S)\) is a relation \(q(Q)\) whose schema \(Q\) is the concatenation of \(R\) and \(S\).

  ➢ \(r(R) \times s(S) = \{ t \ (Q=\text{R}^+\text{S}) \mid \exists t_1 \in r \ (t[R] = t_1) \land \exists t_2 \in s \ (t[S] = t_2)\}\)

  ➢ Example

  \(r\)
  \[
  \begin{array}{ccc}
  A & B & C \\
  1 & b_1 & c_1 \\
  2 & b_2 & c_2 \\
  3 & b_2 & c_2 \\
  \end{array}
  \]

  \(s\)
  \[
  \begin{array}{cc}
  B & C \\
  b_1 & c_1 \\
  b_4 & c_5 \\
  \end{array}
  \]

  \(r \times s\)
  \[
  \begin{array}{cccccc}
  A & r.B & r.C & s.B & s.C \\
  1 & b_1 & c_1 & b_1 & c_1 \\
  1 & b_1 & c_1 & b_4 & c_5 \\
  2 & b_2 & c_2 & b_1 & c_1 \\
  2 & b_2 & c_2 & b_4 & c_5 \\
  3 & b_2 & c_2 & b_1 & c_1 \\
  3 & b_2 & c_2 & b_4 & c_5 \\
  \end{array}
  \]
• Rename ($\rho$)

  $\rho_x(r)$, whose $r$ is on relation schema $R$, is an unary operation that returns a relation $x(R)$.

  If one use $\rho_{x(A_1, A_2, \ldots, A_n)}(r)$ then the attributes of $R$ are renamed to $A_1, A_2, \ldots, A_n$.

  Example

  $r$

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>b4</td>
<td>c5</td>
</tr>
</tbody>
</table>

  $r \times \rho_{s(D)}(\Pi_C(r))$

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
<td>c1</td>
</tr>
<tr>
<td>b1</td>
<td>c1</td>
<td>c5</td>
</tr>
<tr>
<td>b4</td>
<td>c5</td>
<td>c1</td>
</tr>
<tr>
<td>b4</td>
<td>c5</td>
<td>c5</td>
</tr>
</tbody>
</table>

\footnote{Concatenation}

Dr. Byunggu Yu
- **Set-intersection (\(\cap\))**

  - Set \(r(R) \cap s(S)\) is a relation \(q(R)\) that contains common tuples in \(r\) and \(s\) after removing the duplicates. \(R\) and \(S\) must be the same.

  - \(r(R) \cap s(S) = \{ t | t \in r \land t \in s \}\)

- **Example**

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>b2</td>
<td>c2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
<td></td>
</tr>
<tr>
<td>b4</td>
<td>c5</td>
<td></td>
</tr>
</tbody>
</table>

\[\Pi_{B, C}(r) \cap s\]

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>c1</td>
<td></td>
</tr>
</tbody>
</table>
• Natural-join (▷◁)

➢ Set r(R) ▷◁ s(S) is a relation q(Q) whose schema Q is the union of R and S. Every tuple in relation q satisfies (t ∈ q ∧ t[Q ∩ R] ∈ r ∧ t[Q ∩ S] ∈ s)

➢ r(R) ▷◁ s(S) = \{ t (Q=R ∪ S) | t[Q ∩ R] ∈ r ∧ t[Q ∩ S] ∈ s \} = \Pi_{R ∪ S}(σ_{r.A1=s.A1∧r.A2=s.A2∧…∧r.A_n=s.A_n} r×s) where R ∩ S = \{ A_1, A_2, …, A_n \}.

➢ We can give a simple or composite predicate θ on attributes in R+2S, i.e. r ▷◁θ s (theta join).

➢ Example

<table>
<thead>
<tr>
<th>r</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>b2</td>
<td>c2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>b1</td>
<td>c1</td>
<td></td>
</tr>
<tr>
<td>b4</td>
<td>c5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r ▷◁ s</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>b1</td>
<td>c1</td>
</tr>
</tbody>
</table>

---

2 concatenation
• Outer-join (Left outer-join: \(\Rightarrow<\), Right outer-join: \(\Rightarrow<=\), and full outer-join: \(\Rightarrow <=\))

  ➢ \(r(R) \Rightarrow< s(S)\): takes all tuples in \(r\) that did not match with any tuple in \(s\), pads the tuples with null values for the attributes \(S-R\), and adds them to the result of the natural join.

  ➢ \(r(R) \Rightarrow<= s(S)\): takes all tuples in \(s\) that did not match with any tuple in the \(r\), pads the tuples with null values for the attributes \(R-S\), and adds them to the result of the natural join.

  ➢ \(r(R) \Rightarrow< s(S) = r(R) \Rightarrow< s(S) \cup r(R) \Rightarrow<= s(S)\)

  ➢ Example

\[
\begin{array}{|c|c|}
  \hline
  \text{r} & \text{s} \\
  \hline
  \text{A} & \text{B} & \text{D} \\
  1 & b1 & c1 \\
  2 & b2 \text{null} & \\
  3 & b2 \text{null} & \\
  \hline
\end{array}
\]

\[
\begin{array}{|c|c|}
  \hline
  \text{r} = \Rightarrow< s \\
  \hline
  \text{A} & \text{B} & \text{D} \\
  1 & b1 & c1 \\
  2 & b2 \text{null} & \\
  3 & b2 \text{null} & \\
  \hline
\end{array}
\]
• Division (÷)

- Set \( r(R) ÷ s(S) \) is a relation \( q(Q) \) satisfying the following conditions:
  1. \( S \subseteq R \land Q=R-S \)
  2. \( t \in q \land \forall t_s \in S (\exists t_r \in R (t_r[S]=t_s \land t_r[R-S] = t)) \)

- \( r(R) ÷ s(S) = \{ t(Q=R-S) \mid \forall t_s \in s (\exists t_r \in r (t_r[S]=t_s \land t_r[R-S] = t)) \} \)

- Example

\[
\begin{array}{ccc}
A & B & C \\
1 & b1 & c1 \\
1 & b4 & c5 \\
3 & b2 & c2 \\
\end{array}
\]

\[
\begin{array}{cc}
B & C \\
b1 & c1 \\
b4 & c5 \\
\end{array}
\]

\[
\begin{array}{c}
r ÷ s \\
A \\
1 \\
\end{array}
\]
• Assignment (←)
  
  ➢ Assignment to temporary relation variable: A query can be written as a sequential program.
  e.g.,  
  
  \[ \text{temp1} \leftarrow \Pi_{R-S}(r) \]
  
  \[ \text{temp2} \leftarrow \Pi_{R-S}((\text{temp1} \times s) - r) \]
  
  \[ \text{temp1} - \text{temp2} \]
  
  vs.
  
  \[ \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - r) \]
  
  ➢ Assignment to permanent relation: used to express database modification (insertion, deletion, update).
  e.g., Insertion:  
  
  \[ \text{account} \leftarrow \text{account} \cup \{("Laramie", 28345, 5000)\} \]
  
  Deletion:  
  
  \[ \text{account} \leftarrow \text{account} - \sigma_{\text{account#}=28345}(\text{account}) \]
  
  Update:  
  
  \[ \text{account} \leftarrow \Pi_{\text{branch-name, account#, balance}\times1.05}(\text{account}) \]
  
  \[ \text{account} \leftarrow \sigma_{\text{account#}\neq28345}(\text{account}) \cup \]
  
  \[ \Pi_{\text{branch-name, account#, balance}\times1.05}(\sigma_{\text{account#}=28345}(\text{account})) \]
Query Language #2/3: Tuple Relational Calculus

Pure, Non-procedural, Relational

We describe what we want without how to obtain it. Specifically, \{t | P(t)\}, where "t" means "result tuple", is read as: “(I want) The set of all tuples t such that the query predicate P is true for t.”

\[ e.g. 1 \{t \mid t \in \text{account} \land t[\text{balance}] > 1200\} \]

Note, \( t[A] \) denote the values of tuple t on attribute (or attributes) A

\[ e.g. 2 \{t \mid \exists a \in \text{account} \ (t[\text{account#}] = a[\text{account#}] \land a[\text{balance}] > 1200)) \} \]

Note, the difference between e.g.1 and e.g.2 is that the result of e.g.2 consist of only one attribute "account#"

\[ e.g. 3 \{t \mid \exists c \in \text{checking} \ (t[\text{customer#}] = c[\text{customer#}] \land \exists s \in \text{saving} \ (t[\text{customer#}] = s[\text{customer#}]) \} \]

\[ e.g. 4 \{t \mid \exists c \in \text{checking} \ (t[\text{customer#}] = c[\text{customer#}] \land \exists s \in \text{saving} \ (t[\text{customer#}] = s[\text{customer#}] \land \neg \exists l \in \text{loan} \ (t[\text{customer#}] = l[\text{customer#}]) \} \]
e.g. 5\{t \mid \forall b \in \text{branch} \ (b[\text{branch-city}] = "Laramie") \Rightarrow
\exists a \in \text{acc-cust} \ (t[\text{customer#}] = a[\text{customer#}]) \land
a[\text{branch#}] = b[\text{branch#}]
\}

Note, P1 \Rightarrow P2 means "P1 implies P2" or "if P1 is true then P2 must be true" (P2 is the superset of P1). (P1 \Rightarrow P2) \equiv \neg(P1 \land \neg P2) \equiv (\neg P1 \lor P2)
Query Language #3/3: Domain Relational Calculus

Pure, Non-procedural, Relational

Domain calculus is the same as tuple calculus except that the expression is always "domain-oriented" (attribute-oriented).
Specifically, \( \{<x_1, x_2, \ldots, x_n> | P(x_1, x_2, \ldots, x_n)\} \) where \( x_1, x_2, \ldots, \) and \( x_n \) are the 1\(^{\text{st}}\), 2\(^{\text{nd}}\), \ldots, and 3\(^{\text{rd}}\) attribute values of a result tuple, respectively.

E.g.1 find every tuple, whose balance is greater than 1200, from "account" relation: \( \{<a, b, c> | <a, b, c> \in \text{account} \wedge b > 1200\} \)

E.g.2 \( \{<a> | \exists b, c (<a, b, c> \in \text{account} \wedge b > 1200)\} \) Same as e.g.1 except that we want only "account#".

E.g.3 \( \{<c> | \exists a, b, o (<a, b, c, o> \in \text{checking}) \wedge \\
\exists a, b, i (<a, b, c, i> \in \text{saving}) \} \)

Note, 'a' stands for account#, 'b' for balance, 'c' for customer#, 'o' for overdraft-amount, and 'i' for interest-rate
e.g. 4 \{<c> | \exists a, b, o (<a, b, c, o> \in \text{checking}) \land \\
\exists a, b, i (<a, b, c, i> \in \text{saving}) \land \\
\neg \exists a, l (<l, a, c> \in \text{loan})
\}

Note, 'l' stands for loan# and a for loan-amount

e.g. 5\{<c> | \forall b, c1 (<b, c1> \in \text{branch} \land c1 = "Laramie" \Rightarrow \\
\exists a (<a, c, b> \in \text{acc-cust})
\)
\}

Note, 'b' stands for branch#, 'c1' stands for branch-city, and 'a' stands for account#.

A dot (.) in a predicate is used to indicate that the scope of the preceding quantifier extends to the end of the entire predicate. This is a convenient way to eliminate deep nesting of parenthesis.

Note, people generally agree with that the expressive powers of all three relational languages are the same.
1. View

A view is a *virtual relation* that is made visible to a user (or a group of users) but does not exist in logical level database (recall the three level schema). We define a view with a query expression.

*E.g.*, `create view Checking_Customers as`

\[ \Pi_{\text{c-name, customer#, account#, telephone#, address}} (\text{customer} \bowtie \text{checking-account}) \]

Since a view is virtually a relation, it changes dynamically. Therefore, in the preceding example, if we modify customer or checking-account relations, view must be also modified. If views exist physically, data redundancy and inconsistency problems occur. Usually, DBMSs don’t store views as actual relations but store their definitions.

However, if the underlying database has only the definitions, whenever a view is used as a relation in a query, each view name must be replaced by the query expression in the view definition. That is, the view must be recomputed every time. With this, the performance will
deteriorate. Therefore, some people have proposed materialized views. In this approach, views are actually stored as physical relations. However, there are added complexity and overhead for updates.

Since each view is a relation to the users, the users may update their views. However, because of its complexity, view modifications are generally not allowed.

Note, to users, views are relations.
Structured Query Language (SQL)

- Commercial, Procedural + Non-procedural, Not always relational
- Easy to use
- Based on both relational algebra and relational calculus. (e.g., You don't explicitly express projection operation…rather the operation is expressed in select statement as what you want. In contrast, you express the procedure in some complex queries, such as union of two sub select statements, embedded select, …).

Details will be explained and discussed in LectureNote#4.