Information Systems
(Informationssysteme)

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

Summer 2014
Part V

The Relational Data Model
The relational model was proposed in 1970 by Edgar F. Codd:

“The term relation is used here in its accepted mathematical sense. Given sets $S_1, S_2, \ldots, S_n$ (not necessarily distinct), $R$ is a relation of these $n$ sets if it is a set of $n$-tuples each of which has its first element from $S_1$, its second element from $S_2$, and so on.”

In other words, a relation $R$ is a subset of a Cartesian product

$$R \subseteq S_1 \times S_2 \times \cdots \times S_n.$$

$R$ contains $n$-tuples, where the $i$th field must take values from the set $S_i$ ($S_i$ is the $i$th domain of $R$).

---

Relations are Sets of Tuples

A relation is a **set of n-tuples**, e.g., representing cocktail ingredients:

\[
\text{Ingredients} = \{ \langle \text{“Orange Juice”}, 0.0, 12, 2.99 \rangle, \\
\langle \text{“Campari”}, 25.0, 5, 12.95 \rangle, \\
\langle \text{“Mineral Water”}, 0.0, 10, 1.49 \rangle, \\
\langle \text{“Bacardi”}, 37.5, 3, 16.98 \rangle \}
\]

Relations can be illustrated as **tables**:

<table>
<thead>
<tr>
<th>Name</th>
<th>Alcohol</th>
<th>InStock</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange Juice</td>
<td>0.0</td>
<td>12</td>
<td>2.99</td>
</tr>
<tr>
<td>Campari</td>
<td>25.0</td>
<td>5</td>
<td>12.95</td>
</tr>
<tr>
<td>Mineral Water</td>
<td>0.0</td>
<td>10</td>
<td>1.49</td>
</tr>
<tr>
<td>Bacardi</td>
<td>37.5</td>
<td>3</td>
<td>16.98</td>
</tr>
</tbody>
</table>

→ Each column must have a **unique name** (within one relation).
A relation consists of two parts:

1. **Schema**: The schema of a relation is its list of attributes:

   \[
   \text{sch(Ingredients)} = (\text{Name, Alcohol, InStock, Price}).
   \]

   Each attribute has an associated domain that specifies valid values for that column:

   \[
   \text{dom(Alcohol)} = \text{DECIMAL(3,2)}.\]

   Often, key constraints are considered part of the schema, too.

2. **Value** (or instance): The value/instance \( \text{val}(R) \) of a relation \( R \) is the set of tuples (rows) that \( R \) currently contains.
Sets of Tuples

Relations are **sets of tuples:**

- The **ordering** among tuples/rows is **undefined**.
- A relation **cannot contain duplicate rows**.
  - A consequence is that every relation has a key. Use the set of all attributes if there is no shorter key.

© Jens Teubner · Information Systems · Summer 2014
Atomic Values

Attribute domains must be **atomic**:

- Column entries must not have an internal structure or contain “multiple values”.
- A table like

<table>
<thead>
<tr>
<th>Ingredients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Orange Juice</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Campari</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

is **not** a valid relation.
Since relations are sets in the mathematical sense, we can use mathematical formalisms to reason over relations.

In this course we will use
- **relational algebra** and
- **relational calculus**

to express queries over relational data.

Both are used *internally* by any decent relational DBMS.
- Knowledge of both languages will help in understanding SQL and relational database systems in general.
Relational Algebra

In mathematics, an **algebra** is a system that consists of
- a **set** (the carrier) and
- **operations** that are closed with respect to the set.

In the case of **relational algebra**, 
- the **carrier** is the **set of all finite relations**.
- We’ll get to know its **operations** in a moment.

Algebraic operators are **closed** with respect to their set.
- Every operator takes as input one or more relations
  (The number of input operands to an operator $f$ is called the **arity** of $f$.)
- The output is again a relation.

Operators and relations can be **composed** into **expressions** (or **queries**).
Relational Algebra: Selection

The selection \( \sigma_p \) selects a subset of the tuples of a relation, namely those which satisfy the predicate \( p \).

\[
\sigma_{A=1} \begin{pmatrix} A & B \\ 1 & 3 \\ 1 & 4 \\ 2 & 5 \end{pmatrix} = \begin{pmatrix} A & B \\ 1 & 3 \\ 1 & 4 \end{pmatrix}
\]

- Selection acts like a filter on its input relation.
- Selection leaves the schema of the relation unchanged:

\[ \text{sch}(\sigma_p(R)) = \text{sch}(R) \]

- This best compares to the WHERE clause in SQL.
The **predicate** $p$ is a Boolean expressions composed of
- literal **constants**,  
- **attribute names**, and
- arithmetic ($+$, $-$, $\ast$, $\ldots$), **comparison** ($=$, $>$, $\leq$, $\ldots$), and **Boolean operators** ($\land$, $\lor$, $\neg$).

$p$ is evaluated **for each tuple in isolation**.

$\rightarrow$ **Quantifiers** ($\exists$, $\forall$) or **nested relational algebra expressions** are **not** permitted within predicates.
Relational Algebra: Projection

The **projection** $\pi_L$ eliminates all **attributes** (columns) of the input relation but those listed in the **projection list** $L$.

\[
\pi_{A,C} \begin{pmatrix}
1 & 3 & 2 \\
1 & 3 & 5 \\
2 & 5 & 2
\end{pmatrix} = \begin{pmatrix}
1 & 2 \\
1 & 5 \\
2 & 2
\end{pmatrix}
\]

- Intuitively: “$\sigma_p$ discards rows; $\pi_L$ discards columns.”
- Database slang: “All attributes not in $L$ are **projected away**.”
- Projection can also be used to **re-order** columns.
- Projection affects the **schema**: $\text{sch}(\pi_L(R)) = L$.  
  (All attributes listed in $L$ must exist in $\text{sch}(R)$.)
Relational Algebra: Projection

Projection might change the cardinality (i.e., the number of rows) of a relation.

\[ \pi_{A,B} \begin{pmatrix} A & B & C \\ 1 & 3 & 2 \\ 1 & 3 & 5 \\ 2 & 5 & 2 \end{pmatrix} = \begin{pmatrix} A & B \\ 1 & 3 \\ 2 & 5 \end{pmatrix} \]

- Remember that relations are duplicate-free sets!
Relational Algebra: Projection

Often, $\pi_L$ is used also to express **additional functionality** (needed, e.g., to implement SQL):

- **Column renaming:**

  \[ \pi_{B_1 \leftarrow A_{i_1}, \ldots, B_k \leftarrow A_{i_k}}(R) \]

- **Computations:**

  \[ \pi_{Name, Value \leftarrow \text{InStock} \ast Price}(\text{Ingredients}) \]

Alternatively, a separate **re-naming operator** $\varrho_L$ is often seen to express such functionality, e.g.,

\[ \varrho_{B_1 \leftarrow A_{i_1}, \ldots, B_k \leftarrow A_{i_k}}(R) \]

Often, ‘:’ is used instead of ‘$\leftarrow$’ (e.g., $\varrho_{B_1:A_{i_1}, \ldots, B_k:A_{i_k}}(R)$).
Relational Algebra: Projection and SQL

In SQL, duplicate rows are not eliminated automatically.

→ Request duplicate elimination explicitly using keyword DISTINCT.

```
SELECT DISTINCT Alcohol, InStock
FROM Ingredients
WHERE Alcohol = 0
```

In SQL, projection is expressed using the SELECT clause:

```
\[ \pi_{B_1 \leftarrow E_1, \ldots, B_k \leftarrow E_k}(R) \]
```

```
SELECT DISTINCT E_1 AS B_1, \ldots, E_k AS B_k
FROM R
```
Relational Algebra: Cartesian Product

The **Cartesian product** of two relations $R$ and $S$ is computed by concatenating each tuple $r \in R$ with each tuple $s \in S$.

$$
\begin{array}{|c|c|} 
\hline
A & B \\
\hline
1 & 3 \\
2 & 5 \\
\hline
\end{array}
\times
\begin{array}{|c|c|} 
\hline
C & D \\
\hline
7 & 2 \\
3 & 4 \\
\hline
\end{array}
= 
\begin{array}{|c|c|c|c|} 
\hline
A & B & C & D \\
\hline
1 & 3 & 7 & 2 \\
1 & 3 & 3 & 4 \\
2 & 5 & 7 & 2 \\
2 & 5 & 3 & 4 \\
\hline
\end{array}
$$

The Cartesian product contains all columns from both inputs:

$$\text{sch}(R \times S) = \text{sch}(R) + \text{sch}(S) .$$

→ $R$ and $S$ must not share any attribute names.
→ If they do, need to **re-name** first (using $\pi/\rho$).
We already learned how a Cartesian product can be expressed in SQL:

\[
\text{SELECT} \ \ast \\
\text{FROM} \ R, S
\]

- SQL systems will not care about the duplicate column names. 
  (In fact, they allow, e.g., computed values with no column name at all.)
- Unique column names will be \textit{generated} by the system if necessary.
The two set operators $\cup$ (union) and $-$ (set difference) complete the set of relational algebra operators:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

$\cup$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

$=\$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

$-$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

$=\$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
Notes:

- In $R \cup S$ and $R - S$, $R$ and $S$ must be **schema compatible**:

  \[ \text{sch}(R \cup S) = \text{sch}(R - S) = \text{sch}(R) = \text{sch}(S) \]

- For $R \cup S$, $R$ and $S$ need not be disjoint.
- For $R - S$, $S$ need not be a subset of $R$.
- In SQL, $\cup$ and $-$ are available as UNION and EXCEPT, e.g.,

```sql
SELECT Name
FROM Cocktails
UNION
SELECT Name
FROM Ingredients
```
The five basic operations of relational algebra are:

1. \( \sigma_p \) Selection
2. \( \pi_L \) Projection
3. \( \times \) Cartesian product
4. \( \cup \) Union
5. \( - \) Difference

- Any other relational algebra operator (we’ll soon see some of them) can be derived from those five.
- A compact set of operators is a good basis for software (e.g., query optimizers) or database theoreticians to reason over a query or over the language.
Monotonicity

Observe that the first four operators, \( \sigma, \pi, \times, \) and \( \cup \), are **monotonic**: 

- New data added to the database might only **increase**, but never **decrease** the size of their output. *E.g.*, 

  \[
  R \subseteq S \Rightarrow \sigma_p(R) \subseteq \sigma_p(S) \ .
  \]

- For queries composed only of these operators, database insertion **never invalidates** a correct answer.

- **Difference** (\( - \)) is the only **non-monotonic** operator among the basic five.
Monotonicity

For queries with a **non-monotonic semantics**, e.g.,

- “Which ingredients cannot be ordered at ‘Liquors & More’?”
- “Which ingredient has the highest percentage of alcohol?”
- “Which supplier offers all ingredients in the database?”

the operators $\sigma$, $\pi$, $\times$, $\cup$ are **not sufficient** to formulate the query. Such queries **require** set difference.

✎ Formulate the first of these queries in relational algebra.
The combination $\sigma \times$ occurs particularly often.

$\rightarrow$ The $\sigma \times$ pair can be used to **combine** data from multiple tables, in particular by following **foreign key relationships**.

Example:

$$\sigma_{\text{ContactPersons}.\text{ContactFor} = \text{Suppliers}.\text{SuppID}}(\text{Suppliers} \times \text{ContactPersons})$$

Because of this, we introduce a **short notation** for the scenario:

$$R \Join_p S := \sigma_p (R \times S)$$

and call operation $\Join_p$ a **join** ("$R$ and $S$ are joined").
With a join operator, the example on the previous slide would read:

$$\text{Suppliers} \bowtie_{\text{ContactPersons.}\text{ContactFor} = \text{Suppliers.}\text{SuppID}} \text{ ContactPersons}$$

or (omitting redundant relation names in the predicate):

$$\text{Suppliers} \bowtie_{\text{ContactFor} = \text{SuppID}} \text{ ContactPersons}$$

The basic join operator exactly expands to a $\sigma - \times$ combination as shown on the previous slide!
The join operator could be used to express any predicate over \( R \) and \( S \) (though this tends to be not so meaningful in practice).

The pattern

\[
R \Join_{A_i \theta B_j} S ,
\]

where \( A_i \) is an attribute from \( R \), \( B_j \) an attribute from \( S \), and \( \theta \in \{\equiv, \neq, <, \leq, >, \geq\} \) is often called a \( \theta \) join (theta join).

The case \( \theta \equiv \equiv \) is also called an equi join.
The most frequent join operation is an (equi) join that follows a **foreign key constraint**.

It is good practice to use the **same attribute name** for a **primary key** and for **foreign keys** that reference it.

*E.g.*,

<table>
<thead>
<tr>
<th>Cocktails</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CockID</td>
<td>CName</td>
<td>Alcohol</td>
<td>GlassID</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Glasses</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GlassID</td>
<td>GlassName</td>
<td>Volume</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>

(where *GlassID* in *Cocktails* references the *GlassID* in *Glasses*).
The Natural Join

To simplify notation for that common case, we introduce the following convention:

If no explicit predicate is given in the join operator, we interpret this as:

- an equi join over all pairs of columns that have the same name

and

- the column used for joining is only reported once in the join result.

We call this situation a natural join.
The Natural Join

Based on the example schema on slide 109, the natural join

\[
\text{Cocktails} \Join \text{Glasses}
\]

would perform the (intuitively expected) join over \textit{GlassID} columns (\textit{Cocktails.GlassID} = \textit{Glasses.GlassID}) and have the return schema

<table>
<thead>
<tr>
<th>CockID</th>
<th>CName</th>
<th>Alcohol</th>
<th>GlassID</th>
<th>GlassName</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

The example worked out, because I used \textbf{different column names} for all non-join attributes. Otherwise, \Join would have implicitly joined over, e.g., \textit{Name}, too.
Join as a Filter

Consider the join expression

\[ \text{Suppliers} \times \text{ContactPersons} \],

where we assume that \textit{ContactPerson} has a foreign key \textit{SupplID} (and no other column pairs with same name exist).

The query will report all suppliers with their contact person.

But:

- Suppliers where no contact person is stored in \textit{ContactPersons} will not appear in the result. The join effectively implies a filtering behavior.
Sometimes, this **filtering behavior** is **everything we really need** from the join operation.

*E.g.*, "All suppliers where we know a contact person."

\[
\pi_{\text{Suppliers}.*}(\text{Suppliers} \bowtie \text{ContactPersons}),
\]

For this situation, database people introduced another explicit notation:

\[
R \bowtie S := \pi_{\text{sch}(R)}(R \bowtie S) \quad R \bowtie_p S := \pi_{\text{sch}(R)}(R \bowtie_p S),
\]

i.e., compute the join \(R \bowtie S\), but keep only columns that come from \(R\). This operation is also called a **semi join**.
What if I want the opposite, all suppliers where we do not know a contact person?

Possible solution:
Suppliers \( \setminus (\text{Suppliers} \bowtie \text{ContactPersons}) \).
In other cases, the filtering effect is **not** desired.

To obtain all suppliers with their contact person **without** discarding *Supplier* tuples, use the **outer join** (here: **left outer join**): 

\[
\text{Suppliers \Join ContactPersons}.
\]

**Assuming the input**

<table>
<thead>
<tr>
<th>Suppliers</th>
<th>ContactPersons</th>
</tr>
</thead>
<tbody>
<tr>
<td>SupplID</td>
<td>SuppName</td>
</tr>
<tr>
<td>1</td>
<td>Shop Rite</td>
</tr>
<tr>
<td>2</td>
<td>Liquors &amp; More</td>
</tr>
<tr>
<td>3</td>
<td>Joe’s Liquor Store</td>
</tr>
<tr>
<td>SupplID</td>
<td>ContactName</td>
</tr>
<tr>
<td>1</td>
<td>Mary Shoppins</td>
</tr>
<tr>
<td>3</td>
<td>Joe Drinkmore</td>
</tr>
</tbody>
</table>

**what is the result of the above left outer join?**
For certain kinds of queries, the **division** operator is useful.

Given two relations

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th></th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

the division

\[ R \div S \]

returns those \( A \) values \( a_i \), such that for every \( B \) value \( b_j \) in \( S \) there is a tuple \( \langle a_i, b_j \rangle \) in \( R \).
The division would be useful to, e.g., ask for suppliers that offer all ingredients:

\[ Suppliers \Join (Supplies \div \pi_{\text{IngrID}}(Ingredients)) \]
Relational algebra operators may have interesting properties, e.g.,

- The join satisfies the **associativity condition**:

\[(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)\,.

(We can thus often omit parentheses in “join chains”: \(R \bowtie S \bowtie T\).)

- Join is **not commutative**, however, **unless** it is followed by a projection (to re-order columns):

\[\pi_L(R \bowtie S) \equiv \pi_L(S \bowtie R)\,.

- If \(p\) only refers to attributes in \(S\), then

\[\sigma_p(R \bowtie S) \equiv R \bowtie \sigma_p(S)\]

(this is also known as **selection pushdown**).
Algebraic Expressions

Relational Algebra is an *expression-oriented language*. 

→ Expressions consume and produce relations.
→ Results of expressions can be input to other expressions.

*E.g.*, 

\[
\left( \left( \pi_{\text{IngrID}} \left( \sigma_{\text{Name}=\text{‘Campari’}} \text{Ingredients} \right) \right) \bowtie \text{Supplies} \right) \bowtie \text{Suppliers}
\]

Another way of looking at this is an *operator tree*:
Such operator trees imply an **evaluation order**.

- Computation proceeds **bottom-up** (the evaluation order of sibling branches is not defined).
- Operator trees are thus a useful tool to describe **evaluation strategy and order**.
Most relational **query optimizers** use operator trees internally.

→ The operator tree leads to a **query plan** or **execution plan**.

→ The **execution engine** is defined by operator implementations for all of the algebraic operators.

*E.g.*, IBM DB2 execution plan:
Plan trees can be **re-written** using **algebraic laws**: 

*E.g.*, 

- **selection pushdown**: rewrite expressions to apply **selection predicates** early: 
  
  \[ \sigma_p(R \Join S) \rightarrow R \Join \sigma_p(S) \]

  (we saw this algebraic law before). 

- **decide join order**: 
  
  \[ \pi_L(R \Join S \Join T) \rightarrow \pi_L(T \Join (S \Join R)) \]

The **rewrite direction** is often guided by **heuristics** and/or **cost estimations** (~ Course ‘Architecture of Database Systems’).
The execution order implied by algebraic expressions gives relational algebra a **procedural nature**.

- This is **good** for query optimization.
- It is **not so good** for query formulation (e.g., by users).
  - Want to leave execution strategies up to the database.

For query formulation, we’d much rather like to have a **fully declarative way** to describe queries.

- Specify **what** you want as a result, **not how** it can be computed.
- “I want all tuples that look like . . .” or “I want all tuples that satisfy the predicate . . .”
 Tuple Relational Calculus: Idea

In mathematics, a common way to describe sets is

$$\{x \mid p(x)\}$$,

meaning that the set contains all $x$ that satisfy a predicate $p$.

This inspires the **tuple relational calculus (TRC)**:

In a **tuple relational calculus query**

$$\{t \mid F(t)\},$$

$t$ is a **tuple variable**, $F$ is a **formula** that describes how tuples $t$ must look like to qualify for the result.
Formulas form the heart of the TRC. The **language** for formulas is a subset of **first-order logic**:

An **atomic formula** is one of the following:

- $t \in \text{RelationName}$
- $t \leftarrow \langle X_1, \ldots, X_k \rangle$ (tuple constructor)
- $r.a \theta s.b$ ($r$, $s$ tuple variables; $a$, $b$ attributes in $r$, $s$; $\theta \in \{=, <, \ldots \}$)
- $r.a \theta \text{Constant}$ or $\text{Constant} \theta r.a$
A formula is then recursively defined to be one of the following:

- any atomic formula
- \( \neg F \), \( F_1 \land F_2 \), \( F_1 \lor F_2 \)
- \( \exists t : F(t, \ldots) \)
- \( \forall t : F(t, \ldots) \)

where \( F \) and \( F_i \) are formulas and \( t \) a tuple variable.

Quantifiers \( \exists \) and \( \forall \) bind the variable \( t \); \( t \) may occur free in \( F \).

A TRC query is an expression of the form

\[
\{ t \mid F(t) \}
\]

where \( F \) is a formula and \( t \) is the only free variable in \( F \).
Examples

All tuples in $Ingredients$ where $Alcohol = 0$:

$$\{ t | t \in Ingredients \land t.Alcohol = 0 \}$$

Names and prices of all non-alcoholic ingredients:

$$\{ t | \exists v : v \in Ingredients \land v.Alcohol = 0 \land t \leftarrow \langle v.Name, v.Price \rangle \}$$

Name all ingredients that can be ordered at ‘Shop Rite’:

$$\{ t | \exists u : u \in Suppliers \land \exists v : v \in Supplies \land \exists w : w \in Ingredients$$
$$\land u.Name = ‘Shop Rite’ \land u.SupplID = v.SupplID$$
$$\land v.IngrID = w.IngrID \land t \leftarrow \langle w.Name \rangle \}$$
Observe how Tuple Relational Calculus and SQL are related:

\[
\{ t \mid \exists u : u \in Suppliers \land \exists v : v \in Supplies \land \exists w : w \in Ingredients \\
\land u.\text{Name} = 'Shop Rite' \land u.\text{SupplID} = v.\text{SupplID} \\
\land v.\text{IngrID} = w.\text{IngrID} \land t \leftarrow \langle w.\text{Name}\rangle \}
\]

In SQL:

```
SELECT w.\text{Name}
FROM Suppliers AS u, Supplies AS v, Ingredients AS w
WHERE u.\text{Name} = 'Shop Rite' AND u.\text{SupplID} = v.\text{SupplID}
AND v.\text{IngrID} = w.\text{IngrID}
```
Expressive Power

Idea:
- Use tuple relational calculus (∼ SQL) as a declarative front-end language for relational databases.

Questions:
- Can all relational algebra expressions also expressed using TRC?
- Can all TRC queries expressed using relational algebra? (That is, can all TRC queries be answered with an execution engine that implements the algebraic operators?)

Answer?
- No!
Consider the TRC query

\[ \{ t \mid \neg (t \in \text{Ingredients}) \} \]

(return all tuples that are not in the \text{Ingredients} table).

- The set of tuples described by this query is \text{infinite}.\(^8\)

- Relational algebra expressions operate over (and produce) only relations of \text{finite size}.

→ The above TRC query is \text{not} expressible in relational algebra.

\(^8\)Or bound only by the (very large) domains for the attributes in \text{Ingredients}.
Safe Tuple Relational Calculus

The query on the previous slide was an example of an unsafe TRC query. In practice, queries with an infinite result are rarely meaningful.

Thus:
- **Restrict** TRC to allow only queries with a finite result. (We will refer to the set of allowed queries as the safe TRC.)

“Trick:”
- Define safe TRC based on **syntactic** restrictions on the formula language.
  - ➔ Why “syntactic”?
Safe Tuple Relational Calculus

A formula $F$ in the tuple relational calculus is called **safe** iff

1. it contains no universal quantifiers ($\forall$),
2. in each $F_1 \lor F_2$, $F_1$ and $F_2$ have only one free variable and this is the same variable in $F_1$ and $F_2$,
3. in all maximal conjunctive sub-formulae $F_1 \land F_2 \land \cdots \land F_k$, a variable $t$ may be used in a formula $F_i$ only after it has been limited ("bound") in a formula $F_j, j < i$.

A formula $F_j$ limits $t$ iff

- $F_j \equiv t \in R$ or
- $F_j \equiv t \leftarrow [X_1, \ldots, X_k]$
- $t$ appears free in $F_j$ and $F_j$ itself is a safe TRC formula.

All free variables of a maximal conjunctive sub-formula must be limited.

4. negation only occurs in a conjunction as in 3.
SQL is also “safe” in that sense.

→ All tuple variables must be bound (“limited”) in the FROM part.

SQL is not purely based on safe TRC, but includes a combination of

- **Safe TRC**,  
- **Relational Algebra**,  
  (Which example did we already see?)  
- Additional constructs, such as **aggregation**.
Theorem

Relational algebra and safe tuple relational calculus are equivalent.

This equivalence

- guarantees **expressiveness**, *e.g.*, for SQL,
- yet allows **query compilation** into relational algebra (for query optimization and execution).

The theorem can be proven in a **constructive** way:

- Give **translation rules** that compile any safe TRC query into relational algebra and vice versa.
  - The TRC \(\rightarrow\) algebra direction already instructs us how to build a **query compiler**.
Goal: A function $\text{TRC}$ that translates any algebra expression into a Safe TRC formula.

The interesting part is to derive the formula $F$ to construct $\{ t \mid F(t) \}$.

Thus:

- Find $\mathbb{T}(v, \text{Exp})$. Given the name of a variable $v$ and an algebraic (sub)expression $\text{Exp}$, $\mathbb{T}(v, \text{Exp})$ constructs a formula, such that

$$\text{TRC}(\text{Exp}) := \{ t \mid \mathbb{T}(t, \text{Exp}) \}$$

is the TRC equivalent for $\text{Exp}$ and $\mathbb{T}(t, \text{Exp})$ is safe.
Relational Algebra → Safe TRC

Example:

\[ T(v, R) := v \in R \ . \]

Then,

\[ \mathcal{TRC}(R) := \{ t \mid T(t, R) \} = \{ t \mid t \in R \} . \]

Strategy: Syntax-Driven Translation:

\[ T(v, R) := v \in R \quad \text{(see above)} \]
\[ T(v, \sigma_p(Exp)) := ? \]
\[ T(v, \pi_L(Exp)) := ? \]
\[ T(v, Exp_1 \times Exp_2) := ? \]
\[ T(v, Exp_1 \cup Exp_2) := ? \]
\[ T(v, Exp_1 - Exp_2) := ? \]

(Next: Find a translation for each of the five basic algebra operators.)
\[ \sigma_p(\text{Exp}) \rightarrow \text{Safe TRC} \]

Algebra **selection** operator \( \sigma_p \): 

\[
\mathbb{T}(v, \sigma_p(\text{Exp})) := \mathbb{T}(v, \text{Exp}) \land p(v),
\]

where \( p(v) \) is the predicate \( p \) in \( \sigma_p \) and all attribute names in \( p \) are qualified using the variable name \( v \).

\[ \rightarrow \] The resulting formula is **safe** if the result of the recursive construction \( \mathbb{T}(v, \text{Exp}) \) is safe.

Remaining rules for \( \mathbb{T}(v, \text{Exp}) \rightarrow \) exercises.
Safe TRC $\rightarrow$ Relational Algebra

**Goal:** A function $\mathcal{ Alg}$ that translates any safe TRC query into a valid algebra expression.

Safe TRC cannot simply be translated bottom-up, because some of its sub-formulas might be un-safe if considered in isolation.

**Example:** $\{ t \mid t \in R \land t \not\in S \}$ is legal, but the sub-formula $t \not\in S$ would violate rule 3 for safe TRC on slide 132 (and $\{ t \mid \neg (t \in S) \}$ is not expressible in relational algebra).
Safe TRC → Relational Algebra

Thus:
Carry **context information** through the translation process with help of an auxiliary function \( \text{Alg} \):

\[
\text{Alg}(\{ t \mid F(t) \}) := \pi_{t.*}(\text{Alg}(\{\}, F \land \text{true}))
\]

Idea:
- As input, \( \text{Alg} \) receives a **partial algebra plan** (initialized with \( \{\} \)) and a **TRC formula**.
- \( \text{Alg} \) “consumes” a conjunctive formula \( F_1 \land \cdots \land F_k \) piece-by-piece.
- The partial algebra plan is used to provide context and accumulate the overall compilation result.
- We use \( \{\} \times E := E \) and \( F \equiv F \land \text{true} \) to simplify compilation rules.
Let us look at simple formulas first:

\[ \forall(E, t \in R \land F) := \forall \left( E \times_{\pi t.A_1:A_1,\ldots,t.A_k:A_k} R, F \right) \]  \hspace{1cm} (1) 

\[ \forall(E, t \leftarrow [X_1,\ldots,X_k] \land F) := \forall \left( \pi_{\text{sch}(E)}, t.A_1:X_1,\ldots,t.A_k:X_k \right) 
E, F \right) \]  \hspace{1cm} (2) 

\[ \forall(E, X \theta Y \land F) := \forall(\sigma_{X \theta Y} E, F) \]  \hspace{1cm} (3) 

\[ \forall(E, \text{true}) := E \]  \hspace{1cm} (4)
Translation of

\[ \{ r \mid r \in R \land s \in S \land r.A = s.A \land s.B = 42 \} \]

The above TRC expression is not quite correct. Why?
Looks familiar?

This is (almost) exactly how your database system compiles SQL!

```sql
SELECT p.*
FROM Professors AS p, Courses AS c
WHERE p.ID = c.heldBy
  AND c.courseID = 42
↓
{ p | p ∈ Professors ∧ c ∈ Courses
  ∧ p.ID = c.heldBy ∧ C.courseID = 42 }
↓
π_{p.*}(\sigma_{p.courseID=42}(Professors \bowtie_{p.ID=c.heldBy} Courses))
```
Safe TRC → Relational Algebra

Time to complete our rule set...

\[
\mathcal{A}(E, (\exists v : G) \land F) := \mathcal{A} \left( \pi_{\text{sch}(E)}, F \right) \mathcal{A}(E, G \land \text{true}) \quad (5)
\]

\[
\mathcal{A}(E, (G_1 \lor G_2) \land F) := \mathcal{A} \left( \mathcal{U} \mathcal{A}(E, G_1 \land \text{true}), \mathcal{A}(E, G_2 \land \text{true}) \right) \quad (6)
\]

\[
\mathcal{A}(E, \neg G \land F) := \mathcal{A} \left( E \pi_{\text{sch}(E)}, F \right) \mathcal{A}(E, G \land \text{true}) \quad (7)
\]
Notes:

- In Rule (5), the $\exists$ quantifier introduces a new variable, which appears free in $G$. After compiling $G$, we “project away” the additional column(s).

- In Rule (6), both parts of the $\cup$ must be schema-compatible, because (by rule 2 for safe TRC on slide 132) $G_1$ and $G_2$ must have the same free variable.

- Observe, in Rule (7), how we can make use of the difference operator, because we made sure that all free variables in $G$ were bound previously (and are thus part of $E$).
Translation of

\[ \{ r \mid r \in R \land (\exists s : s \in S \land r.A = s.A \land s.B = 42)\} \]

This is the correct substitute TRC expression (and its translation) for the one shown earlier on slide 141.
Suppose a database contains a *Flights* relation:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>FlightNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZRH</td>
<td>DRS</td>
<td>OL 277</td>
</tr>
<tr>
<td>DRS</td>
<td>MUC</td>
<td>LH 2127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where a tuple \(\langle f, t, n \rangle\) indicates that there is a flight from \(f\) to \(n\) with flight number \(n\).

The algebra expression

\[
\pi_{To} \left( \pi_{From \leftarrow To} \left( \sigma_{From='ZRH'}(Flights) \right) \right) \bowtie Flights
\]

then returns airport codes for all destinations that can be reached with one stop from Zurich.
More generally, we can use an \textit{n-fold self join} to find destinations reachable with \textit{n} stops.

→ We can write down that self join for every known value of \textit{n}.

→ But it is \textbf{impossible} to express the \textit{transitive closure} in relational algebra.
   (\textit{i.e.}, we cannot write a query that returns reachable destinations with a trip of \textbf{any} length.)

This implies that relational algebra is \textbf{not computationally complete}.

→ This might seem unfortunate. But it is a consequence of the desirable guarantee that \textbf{query evaluation always terminates} in relational algebra.
Expressiveness of SQL

SQL is slightly more powerful than relational algebra (≡ Safe TRC), e.g.,

- **aggregation** (e.g., the SQL `COUNT` operation)
- (very limited) support for **recursion**
  Reachability queries as shown before can actually be expressed in recent versions of SQL.
- explicit support for special use cases (e.g., windowing)

These extensions have been carefully designed to keep the **termination guarantees**, however.
Wrap-Up

Relations:
- finite sets of tuples

Relational Algebra:
- expression-based query language
  - operators $\sigma_p$, $\pi_L$, $\times$, $\cup$, $-$, $\Join_p$, $\ldots$
  - used internally by DBMSs for optimization and evaluation

(Safe) Tuple Relational Calculus:
- declarative query language
  - $\{ t \mid F(t) \}$
  - TRC inspired the design of the SQL language

Expressiveness:
- relational algebra $\equiv$ safe TRC $\subseteq$ SQL