Information Systems (Informationssysteme)

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

Summer 2018

Part V

The Relational Data Model

The Relational Model

The relational model was proposed in 1970 by Edgar F. Codd:[7]

> *"The term* **relation** *is used here in its accepted mathematical sense. Given sets $S_1, S_2, \dots, S_n$ (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 ith field must take values from the set S_i (S_i is the ith **domain** of R).

[7] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, vol. 13(6), June 1970.

Relations are Sets of Tuples

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

$$
\begin{array}{rll}
\mathit{Ingredients} = \{ & \langle \text{“Orange Juice”} , & 0.0, 12, 2.99 \rangle, \\
& \langle \text{“Campari”} , & 25.0, 5, 12.95 \rangle, \\
& \langle \text{“Mineral Water”} , & 0.0, 10, 1.49 \rangle, \\
& \langle \text{“Bacardi”} , & 37.5, 3, 16.98 \rangle \}
\end{array}
$$

Relations can be illustrated as **tables**:

Ingredients			
Name	Alcohol	InStock	Price
Orange Juice	0.0	12	2.99
Campari	25.0	5	12.95
Mineral Water	0.0	10	1.49
Bacardi	37.5	3	16.98

→ Each column must have a **unique name** (within one relation).

Schema vs. Value

A relation consists of **two parts**:

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

   $$\mathrm{sch}(\mathit{Ingredients}) = (\mathit{Name}, \mathit{Alcohol}, \mathit{InStock}, \mathit{Price}) \quad .$$

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

   $$\mathrm{dom}(\mathit{Alcohol}) = \texttt{DECIMAL(3,2)} \quad .$$

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

2. **Value** (or **instance**): The **value**/**instance** $\mathrm{val}(R)$ 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**.
  - $\rightarrow$ A consequence is that every relation has a key. Use the set of all attributes if there is no shorter key.

Atomic Values

Attribute domains must be **atomic**:

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

<table>
<tr><th colspan="3">Ingredients</th></tr>
<tr><th>Name</th><th>Alcohol</th><th>SoldBy</th></tr>
<tr><td>Orange Juice</td><td>0.0</td><td><table><tr><th>Supplier</th><th>Price</th></tr><tr><td>A&P Supermarket</td><td>2.49</td></tr><tr><td>Shop Rite</td><td>2.79</td></tr></table></td></tr>
<tr><td>Campari</td><td>25.0</td><td><table><tr><th>Supplier</th><th>Price</th></tr><tr><td>Joe's Liquor Store</td><td>14.99</td></tr></table></td></tr>
</table>

is **not** a valid relation.

Querying Relational Data

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** σ_p selects a **subset** of the tuples of a relation, namely those which satisfy the **predicate** p.

$$\sigma_{A=1}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 1 & 4 \\ 2 & 5 \\ \hline\end{array}\right) = \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 1 & 4 \\ \hline\end{array}$$

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

$$\mathrm{sch}\big(\sigma_p(R)\big) = \mathrm{sch}(R) \enspace .$$

- This best compares to the `WHERE` clause in SQL.

Relational Algebra: Selection

The **predicate** p is a Boolean expressions composed of

- literal **constants**,
- **attribute names**, and
- **arithmetic** ($+$, $-$, $*$, $\dots$), **comparison** ($=$, $>$, $\leq$, $\dots$), and **Boolean operators** ($\wedge$, $\vee$, $\neg$).

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** π_L eliminates all **attributes** (columns) of the input relation but those listed in the **projection list** L.

$$\pi_{A,C}\left(\begin{array}{ccc} A & B & C \\ \hline 1 & 3 & 2 \\ 1 & 3 & 5 \\ 2 & 5 & 2 \end{array}\right) = \begin{array}{cc} A & C \\ \hline 1 & 2 \\ 1 & 5 \\ 2 & 2 \end{array}$$

- Intuitively: *"σ_p discards rows; π_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**: $\mathrm{sch}\big(\pi_L(R)\big) = L$.
  (All attributes listed in L must exist in $\mathrm{sch}(R)$.)

Relational Algebra: Projection

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

$$\pi_{A,B}\left(\begin{array}{ccc} A & B & C \\ \hline 1 & 3 & 2 \\ 1 & 3 & 5 \\ 2 & 5 & 2 \end{array}\right) = \begin{array}{cc} A & B \\ \hline 1 & 3 \\ 2 & 5 \end{array}$$

- Remember that relations are **duplicate-free sets**!

Relational Algebra: Projection

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

- **Column renaming:**

$$\pi_{B_1 \leftarrow A_{i_1}, \dots, B_k \leftarrow A_{i_k}}(R) \quad .$$

- **Computations:**

$$\pi_{\mathit{Name}, \mathit{Value} \leftarrow \mathit{InStock} * \mathit{Price}}(\mathit{Ingredients}) \quad .$$

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

$$\varrho_{B_1 \leftarrow A_{i_1}, \dots, B_k \leftarrow A_{i_k}}(R) \quad .$$

Often, ':' is used instead of '$\leftarrow$' (*e.g.*, $\varrho_{B_1 : A_{i_1}, \dots, B_k : A_{i_k}}(R)$).

Relational Algebra: Projection and SQL

In SQL, duplicate rows are **not** eliminated automatically.

$\rightarrow$ 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:

$$\pi_{B_1 \leftarrow E_1, \ldots, B_k \leftarrow E_k}(R)$$

$$\downarrow$$

SELECT DISTINCT E_1 AS B_1, ..., 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 $s \in S$.

A	*B*
1	3
2	5

$\times$

C	*D*
7	2
3	4

$=$

A	*B*	*C*	*D*
1	3	7	2
1	3	3	4
2	5	7	2
2	5	3	4

The Cartesian product contains all columns from both inputs:

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

→ R and S must not share any attribute names.

→ If they do, need to **re-name** first (using π/ϱ).

Cartesian Product and SQL

We already learned how a Cartesian product can be expressed in SQL:

```
SELECT *
  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 **generated** by the system if necessary.

Relational Algebra: Set Operations

The two **set operators** ∪ (**union**) and − (**set difference**) complete the set of relational algebra operators:

A	B
1	3
1	4
2	5

$\cup$

A	B
1	4
3	2

$=$

A	B
1	3
1	4
2	5
3	2

A	B
1	3
1	4
2	5

$-$

A	B
1	4
3	2

$=$

A	B
1	3
2	5

Relational Algebra: Set Operations

Notes:

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

$$\text{sch}(R \cup S) = \text{sch}(R - S) = \text{sch}(R) = \text{sch}(S) \quad .$$

- 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.*,

```
SELECT Name
  FROM Cocktails
UNION
SELECT Name
  FROM Ingredients
```

Five Basic Algebra Operators

The **five basic operations of relational algebra** are:

1. σ_p **Selection**
2. π_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, σ, π, $\times$, 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 σ, π, $\times$, $\cup$ are **not sufficient** to formulate the query. Such queries **require** set difference.

✎ **Formulate the first of these queries in relational algebra.**

The Join Operator $\bowtie_p$

The combination σ-$\times$ occurs particularly often.

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

Example:

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

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

$$R \bowtie_p S \ := \ \sigma_p\big(R \times S\big)$$

and call operation $\bowtie_p$ a **join** ("R and S are joined").

The Join Operator $\bowtie_p$

With a join operator, the example on the previous slide would read:

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

or (omitting redundant relation names in the predicate):

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

The basic join operator exactly expands to a σ-$\times$ combination as shown on the previous slide!

The Join Operator $\bowtie_p$ / Theta Join

The join operator could be used to express **any** predicate over R and S (though this tends to be not so meaningful in practice).

$$Ingredients \bowtie_{Flavor \leq Email \wedge Alcohol < 10} ContactPersons$$

The pattern

$$R \bowtie_{A_i \theta B_j} S \quad ,$$

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

The case $\theta \equiv =$ is also called an **equi join**.

The Natural 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.,

Cocktails			
CockID	CName	Alcohol	GlassID
⋮	⋮	⋮	⋮

Glasses		
GlassID	GlassName	Volume
⋮	⋮	⋮

(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

$$\mathit{Cocktails} \bowtie \mathit{Glasses}$$

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

Cocktails					
<u>CockID</u>	CName	Alcohol	GlassID	GlassName	Volume
⋮	⋮	⋮	⋮	⋮	⋮

The example worked out, because I used **different column names** for all non-join attributes. Otherwise, $\bowtie$ would have implicitly joined over, *e.g.*, *Name*, too.

Join as a Filter

Consider the join expression

$$\mathit{Suppliers} \bowtie \mathit{ContactPersons} \quad ,$$

where we assume that *ContactPerson* has a foreign key *SuppID* (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 *ContactPersons* will **not** appear in the result. The join effectively implies a **filtering behavior**.

Join as a Filter—Semi Join

Sometimes, this **filtering behavior** is **everything we really need** from the join operation.

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

$$\pi_{Suppliers.*}\big(Suppliers \bowtie ContactPersons\big) \enspace ,$$

For this situation, database people introduced another explicit notation:

$$R \ltimes S \enspace := \enspace \pi_{\mathrm{sch}(R)}\big(R \bowtie S\big) \qquad R \ltimes_p S \enspace := \enspace \pi_{\mathrm{sch}(R)}\big(R \bowtie_p S\big) \enspace ,$$

i.e., compute the join $R \bowtie S$, but keep only colums that come from R.

This operation is also called a **semi join**.

Quiz

✎ **What if I want the opposite, all suppliers where we do not know a contact person?**

Outer Joins

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**):

$$Suppliers ⟕ ContactPersons \quad .$$

✎ **Assuming the input**

Suppliers	
SuppID	SuppName
1	Shop Rite
2	Liquors & More
3	Joe's Liquor Store

ContactPersons	
SuppID	ContactName
1	Mary Shoppins
3	Joe Drinkmore

,

what is the result of the above left outer join?

Division

For certain kinds of queries, the **division** operator is useful.

Given two relations

R	
A	B
$\vdots$	$\vdots$

and

S
B
$\vdots$

,

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.

Example

A	B
1	a
1	c
2	b
2	a
2	c
3	b
3	c
3	a
3	d

$\div$

B
a
c

$=$

A
1
2
3

A	B
1	a
1	c
2	b
2	a
2	c
3	b
3	c
3	a
3	d

$\div$

B
a
b
c

$=$

A
2
3

The division would be useful to, *e.g.*, ask for suppliers that offer **all** ingredients:

$$Suppliers \bowtie \left(Supplies \div \pi_{IngrID}(Ingredients)\right)$$

Algebraic Laws

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**).

Relational Algebra is an **expression-oriented language**.

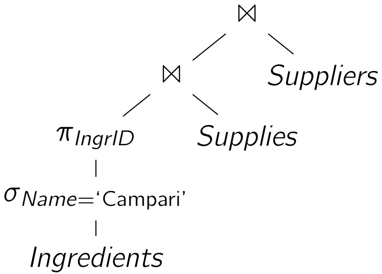
→ Expressions consume and produce relations.

→ Results of expressions can be input to other expressions.

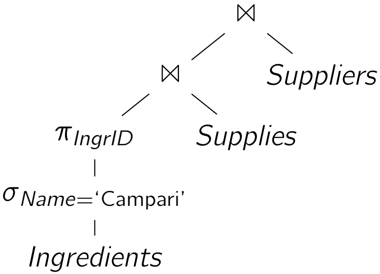
E.g.,

$$\Big(\big(\pi_{\mathit{IngrID}}\,(\sigma_{\mathit{Name}=\text{`Campari'}}\,\mathit{Ingredients})\big) \bowtie \mathit{Supplies}\Big) \bowtie \mathit{Suppliers}$$

Another way of looking at this is an **operator tree**:



Operator Trees



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**.

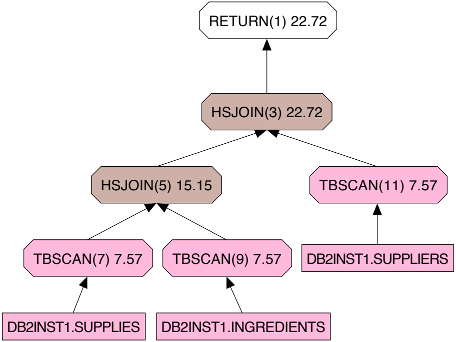
Query Plans

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:



Query Optimization

Plan trees can be **re-written** using **algebraic laws**:

E.g.,

- **selection pushdown**: rewrite expressions to apply **selection predicates** early:

$$\sigma_p(R \bowtie S) \;\rightarrow\; R \bowtie \sigma_p(S)$$

  (we saw this algebraic law before).

- decide **join order**:

$$\pi_L(R \bowtie S \bowtie T) \;\rightarrow\; \pi_L\big(T \bowtie (S \bowtie R)\big)$$

The **rewrite direction** is often guided by **heuristics** and/or **cost estimations** ($\rightsquigarrow$ Course 'Architecture of Database Systems').

Procedural $\leftrightarrow$ Declarative

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.

TRC Formulas

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:

- $t \in \mathit{RelationName}$
- $t \leftarrow \langle X_1, \dots, X_k \rangle$ (tuple constructor)
- $r.a \,\theta\, s.b$ (r, s tuple variables; a, b attributes in r, s; $\theta \in \{=, <, \dots\}$)
- $r.a \,\theta\, \mathit{Constant}$ or $\mathit{Constant} \,\theta\, r.a$

TRC Formulas

A **formula** is then recursively defined to be one of the following:

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

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

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

where F is a formula and t is the only free variable in F.

Examples

All tuples in *Ingredients* where $\mathit{Alcohol} = 0$:

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

Names and prices of all non-alcoholic ingredients:

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

Name all ingredients that can be ordered at 'Shop Rite':

$$\begin{aligned} \{ t \mid\ & \exists u : u \in \mathit{Suppliers} \land \exists v : v \in \mathit{Supplies} \land \exists w : w \in \mathit{Ingredients} \\ & \land u.\mathit{Name} = \text{'Shop Rite'} \land u.\mathit{SupplID} = v.\mathit{SupplID} \\ & \land v.\mathit{IngrID} = w.\mathit{IngrID} \land t \leftarrow \langle w.\mathit{Name} \rangle \} \end{aligned}$$

Tuple Relational Calculus ↔ SQL

Observe how Tuple Relational Calculus and SQL are related:

$$\bigl\{ t \mid \exists u : u \in \mathit{Suppliers} \land \exists v : v \in \mathit{Supplies} \land \exists w : w \in \mathit{Ingredients} \\ \quad \land u.\mathit{Name} = \text{`Shop Rite'} \land u.\mathit{SupplID} = v.\mathit{SupplID} \\ \quad \land v.\mathit{IngrID} = w.\mathit{IngrID} \land t \leftarrow \langle w.\mathit{Name} \rangle \bigr\}$$

In SQL:

```
SELECT w.Name
  FROM Suppliers AS u, Supplies AS v, Ingredients AS w
 WHERE u.Name = 'Shop Rite' AND u.SupplID = v.SupplID
   AND v.IngrID = w.IngrID
```

Expressive Power

Idea:

- Use tuple relational calculus ($\rightsquigarrow$ 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

$$\{t \mid \neg(t \in \mathit{Ingredients})\}$$

(return all tuples that are **not** in the *Ingredients* table).

- The set of tuples described by this query is **infinite**.[8]
- Relational algebra expressions operate over (and produce) only relations of **finite size**.

→ The above TRC query is **not** expressible in relational algebra.

[8] Or bound only by the (very large) domains for the attributes in *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 $F_1 \vee F_2$, 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 $F_1 \wedge F_2 \wedge \cdots \wedge F_k$, 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
   - $F_j \equiv t \leftarrow \langle X_1, \ldots, X_k \rangle$
   - 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.

Safe TRC $\leftrightarrow$ SQL

SQL is also "safe" in that sense.

$\rightarrow$ 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**.

Equivalence of Relational Algebra and Safe TRC

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 → algebra direction already instructs us how to build a **query compiler**.

Relational Algebra → Safe TRC

Goal: A function $\mathbb{TRC}$ 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 $\mathbb{T}(v, Exp)$. Given the name of a variable v and an algebraic (sub)expression Exp, $\mathbb{T}(v, Exp)$ constructs a formula, such that

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

  is the TRC equivalent for Exp and $\mathbb{T}(t, Exp)$ is safe.

Relational Algebra → Safe TRC

Example:

$$\mathbb{T}(v, R) := v \in R \ .$$

Then,

$$\mathbb{TRC}\big(R\big) := \big\{t \mid \mathbb{T}(t, R)\big\} = \big\{t \mid t \in R\big\} \ .$$

Strategy: Syntax-Driven Translation:

$$\begin{array}{rcl}
\mathbb{T}\big(v, R\big) & := & v \in R \quad \text{(see above)} \\
\mathbb{T}\big(v, \sigma_p\,(Exp)\,\big) & := & ? \\
\mathbb{T}\big(v, \pi_L\,(Exp)\,\big) & := & ? \\
\mathbb{T}\big(v, Exp_1 \times Exp_2\big) & := & ? \\
\mathbb{T}\big(v, Exp_1 \cup Exp_2\big) & := & ? \\
\mathbb{T}\big(v, Exp_1 - Exp_2\big) & := & ?
\end{array}$$

(Next: Find a translation for each of the five basic algebra operators.)

$\sigma_p(Exp) \rightarrow$ Safe TRC

Algebra **selection** operator σ_p:

$$\mathbb{T}\big(v, \sigma_p\,(Exp)\,\big) \quad := \quad \mathbb{T}\big(v, Exp\big) \wedge p(v) \ \ ,$$

where $p(v)$ is the predicate p in σ_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 $\mathbb{T}(v, Exp)$ is safe.

Remaining rules for $\mathbb{T}(v, Exp) \rightarrow$ exercises.

Safe TRC → Relational Algebra

Goal: A function $\mathbb{Alg}$ 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: $\{t \mid t \in R \wedge t \notin S\}$ is legal, but the sub-formula $t \notin S$ 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 $\mathbb{A}$:

$$\mathbb{Alg}\left(\{t \mid F(t)\}\right) := \pi_{t.*}\left(\mathbb{A}\left(\{\}, F \land \text{true}\right)\right) \quad .$$

Idea:

- As input, $\mathbb{A}$ receives a **partial algebra plan** (initialized with $\{\}$) and **a TRC formula**.
- $\mathbb{A}$ "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.

Safe TRC $\rightarrow$ Relational Algebra

Let us look at simple formulas first:

$$\mathbb{A}(E, t \in R \wedge F) \quad := \quad \mathbb{A}\left(E \times \pi_{t.A_1:A_1,\dots,t.A_k:A_k}(R), F \right) \tag{1}$$

$$\mathbb{A}(E, t \leftarrow \langle X_1, \dots, X_k \rangle \wedge F) \quad := \quad \mathbb{A}\left(\pi_{\mathrm{sch}(E),t.A_1:X_1,\dots,t.A_k:X_k}(E), F \right) \tag{2}$$

$$\mathbb{A}(E, X\,\theta\,Y \wedge F) \quad := \quad \mathbb{A}\left(\sigma_{X\theta Y} E, F\right) \tag{3}$$

$$\mathbb{A}(E, \mathrm{true}) \quad := \quad E \tag{4}$$

Safe TRC → Relational Algebra

✎ 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?

Safe TRC → Relational Algebra — Detour

Looks familiar?

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

```
SELECT p.*
  FROM Professors AS p, Courses AS c
 WHERE p.ID = c.heldBy
   AND c.courseID = 42
```

$$\downarrow$$

$$\{p \mid p \in \mathit{Professors} \land \exists c : c \in \mathit{Courses} \land p.\mathit{ID} = c.\mathit{heldBy} \land c.\mathit{courseID} = 42\}$$

$$\downarrow$$

$$\pi_{p.*}\big(\sigma_{p.\mathit{courseID}=42}\big(\mathit{Professors} \bowtie_{p.\mathit{ID}=c.\mathit{heldBy}} \mathit{Courses}\big)\big)$$

Safe TRC $\rightarrow$ Relational Algebra

Time to complete our rule set...

$$\mathbb{A}\left(E, (\exists v : G) \wedge F\right) := \mathbb{A}\left(\begin{array}{c}\pi_{\mathrm{sch}(E)}\\ | \\ \mathbb{A}(E, G \wedge \mathrm{true})\end{array}, F\right) \tag{5}$$

$$\mathbb{A}\left(E, (G_1 \vee G_2) \wedge F\right) := \mathbb{A}\left(\begin{array}{ccc} & \cup & \\ \diagup & & \diagdown \\ \mathbb{A}(E, G_1 \wedge \mathrm{true}) & & \mathbb{A}(E, G_2 \wedge \mathrm{true})\end{array}, F\right) \tag{6}$$

$$\mathbb{A}\left(E, \neg G \wedge F\right) := \mathbb{A}\left(\begin{array}{ccc} & - & \\ \diagup & & \diagdown \\ E & & \pi_{\mathrm{sch}(E)} \\ & & | \\ & & \mathbb{A}(E, G \wedge \mathrm{true})\end{array}, F\right) \tag{7}$$

$\square$

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).

Safe TRC $\rightarrow$ Relational Algebra (Example)

✎ Translation of

$$\{r \mid r \in R \land (\exists s : s \in S \land r.A = s.A \land s.B = 42)\} \;\;?$$

Limitations of Relational Algebra / Safe TRC

Suppose a database contains a *Flights* relation

Flights		
From	To	FlightNo
ZRH	DRS	OL 277
DRS	MUC	LH 2127
⋮	⋮	⋮

,

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

The algebra expression

$$\pi_{To}\big(\pi_{From \leftarrow To}(\sigma_{From = \text{'ZRH'}}(Flights)) \bowtie Flights\big)$$

then returns airport codes for all destinations that can be reached with one stop from Zurich.

Limitations of Relational Algebra / Safe TRC

More generally, we can use an *n*-**fold self join** to find destinations reachable with *n* stops.

→ We can write down that self join for every known value of *n*.

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

This implies that relational algebra is **not computationally complete**.

→ This might seem unfortunate. But it is a consequence of the desirable guarantee that **query evaluation always terminates** in relational algebra.

Expressiveness of SQL

SQL is slightly more powerful than relational algebra ($\equiv$ 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
  - $\rightarrow$ operators σ_p, π_L, $\times$, $\cup$, $-$, $\bowtie_p$, ...
  - $\rightarrow$ used internally by DBMSs for optimization and evaluation

(Safe) Tuple Relational Calculus:

- declarative query language
  - $\rightarrow$ $\{t \mid F(t)\}$
  - $\rightarrow$ TRC inspired the design of the SQL language

Expressiveness:

- relational algebra $=$ safe TRC $\subseteq$ SQL