Relational algebra

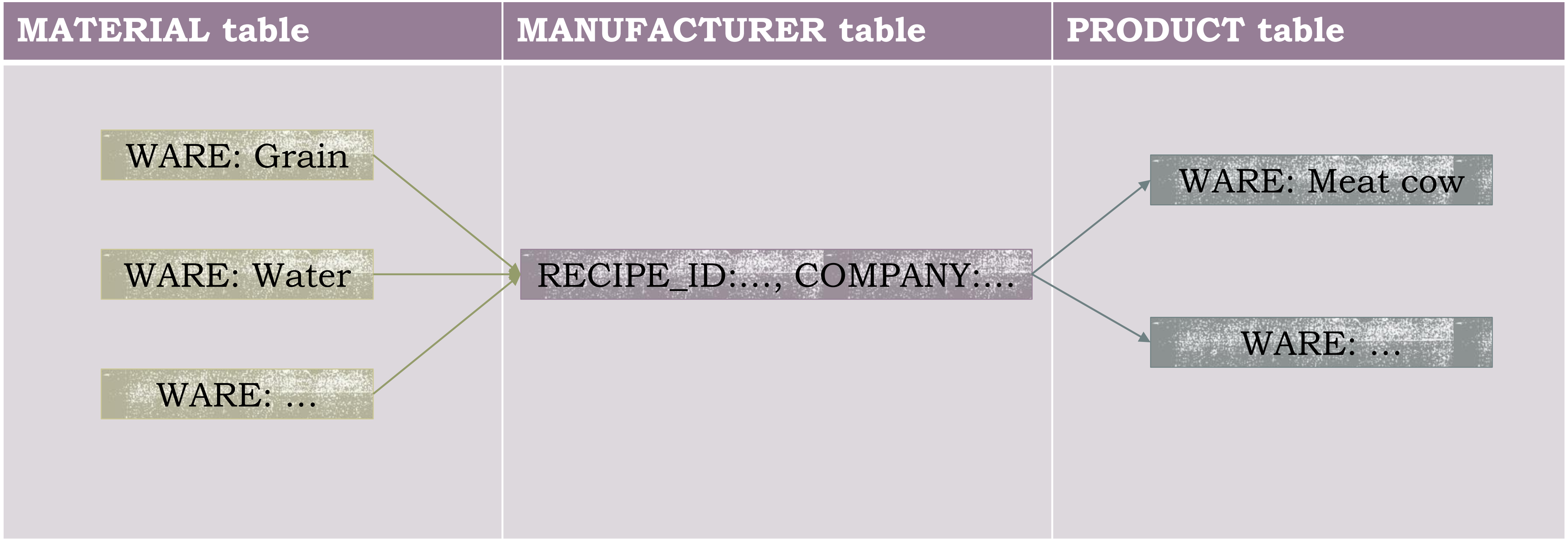
Denis Miginsky

The question of the day

For each ware what materials are required to produce it?

The structure of a recipe in DB

Problem: to reason about recipes, production chains, etc. it is essential to query multiple tables simultaneously.



Basic terms (a bit more formal)

n-tuple (tuple) – sequence of **n** elements: $(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n})$

Type: x is of type **T** iff $\mathbf{x} \in \mathbf{T}$

Type of tuple: if $\mathbf{x_i}$ is of type $\mathbf{T_i}$, then the whole tuple is of type $\mathbf{T_1 \times T_2 \times \ldots \times T_n}$

Record is a **tuple** with named attributes. Will use both terms interchangeably.

n-ary relation over the sets $\mathbf{T_1, T_2, \ldots T_n}$ – a **subset** of $\mathbf{T_1 \times T_2 \times \ldots \times T_n}$, (Cartesian product or cross-product) i.e. a set of tuples of this type

Table (multiset) in RDB and **relation** (set) in math are almost the same.

Operations in relational algebra

- **Projection** - takes the subset of attributes from each tuple. Acts like the unique projection in SQL without derivative columns.
- **Extended projection –** like unique projection in SQL. Could include derivative columns.
- **Selection** – subset of tuples that passes the filter provided, written as propositional formula. The same as in SQL.
- **Rename**
- **Set operations:** union, intersection, difference

- **Join** (like product)
  - **Cross-join**
  - **Natural join**
  - **Equijoin**
  - **Outer join**
  - **θ-join**
  - **Semijoin**
  - **Antijoin**

- **Division**

Rename

Relational algebra:
$\boldsymbol{\rho}_{y/x}$**(R)**, result is the same as R except element with name **x** renamed to **y**

SQL:

```
SELECT WARE AS WARE_NAME, CLASS
FROM CATEGORY
```

```
>> WARE_NAME  CLASS
   ----------------
   Charcoal   Fuel
   Water      Mineral
   …
```

Cross join

Relational algebra:
$\mathbf{R_1} \times \mathbf{R_2}$ cartesian product of two relations, i.e. two sets

SQL: get an answer for the question of the day, the first try

```
SELECT *
FROM MATERIAL, PRODUCT
```

```
>> …

   …

   PROFIT?
```

Semantics of cross join

MATERIAL

RECIPE_ID	WARE	…
1	Meat cow	
2	Water	
2	Grain	
3	Gold ore	
…	…	

…

PRODUCT

RECIPE_ID	WARE	…
1	Meat	
1	Leather	
2	Meat cow	
3	Gold	
…	…	

Natural join

Relational algebra:
$\mathbf{R_1} \bowtie \mathbf{R_2}$ all the combinations of tuples from $\mathbf{R_1}$ and $\mathbf{R_2}$ equal on attributes with the same names.

SQL: the second try

```
SELECT *
FROM MATERIAL, PRODUCT
<MAGICAL RITUAL TO MAKE THE DB TO PERFORM ⋈>


>> NULL NULL NULL NULL
```

Disclosure of the black magic (that failed)

```
SELECT MATERIAL.RECIPE_ID,
       MATERIAL.WARE,
       MATERIAL.AMOUNT,
       PRODUCT.PRICE
FROM MATERIAL, PRODUCT
WHERE MATERIAL.RECIPE_ID=PRODUCT.RECIPE_ID
  AND MATERIAL.WARE=PRODUCT.WARE
  AND MATERIAL.AMOUNT=PRODUCT.AMOUNT;
```

```
>> NULL NULL NULL NULL
```

Equijoin

Relational algebra:
$\mathbf{R_1} \bowtie_{a=b} \mathbf{R_2}$ all the combinations of tuple from $\mathbf{R_1}$ and $\mathbf{R_2}$,where **a** is some attribute of $\mathbf{R_1}$, **b** is attribute of $\mathbf{R_2}$ and **a** equals to **b**.
In extended form (implemented by SQL) multiple attributes from both relations can be used.

SQL: the third try

```
SELECT MATERIAL.WARE, PRODUCT.WARE
FROM MATERIAL, PRODUCT
WHERE MATERIAL.RECIPE_ID=PRODUCT.RECIPE_ID;
```

```
>> WARE        WARE
   ----------------
   Charcoal    Drinking water
   Grain       Meat cow
   Charcoal    Drinking water
   …
```

Semantics of equijoin

MATERIAL

…	WARE	RECIPE_ID
	Meat cow	1
	Water	2
	Grain	2
	Gold ore	3
	…	…

…

PRODUCT

RECIPE_ID	WARE	…
1	Meat	
1	Leather	
2	Meat cow	
3	Gold	
…	…	

Final touch on the query

- There are many duplicates, DISTINCT should be applied.
- Rename is useful to distinguish columns in the query result.

```
SELECT DISTINCT MATERIAL.WARE AS MATERIAL,
                PRODUCT.WARE  AS PRODUCT
FROM MATERIAL, PRODUCT
WHERE MATERIAL.RECIPE_ID=PRODUCT.RECIPE_ID;

>> MATERIAL   PRODUCT
   ----------------
   Charcoal   Drinking water
   Grain      Meat cow
   …
```

Alternative SQL syntax for equijoin

```
SELECT DISTINCT MATERIAL.WARE AS MATERIAL,
                PRODUCT.WARE  AS PRODUCT
FROM MATERIAL
INNER JOIN PRODUCT
ON MATERIAL.RECIPE_ID=PRODUCT.RECIPE_ID;
```

- The query is equivalent to the previous
- INNER keyword is optional (in contrast to other types of join)
- For the inner join ON section works the same as WHERE section (wrong for other types of join). However, it is conventional to place the join conditions under ON and other selection conditions under WHERE.

Another question

Which wares do have two or more categories?

Self-join

```sql
SELECT DISTINCT fst.WARE
FROM CATEGORY fst, CATEGORY snd
WHERE fst.WARE=snd.WARE AND fst.CLASS<>snd.CLASS;
```

Table is joined to itself, so table aliasing is a must in this case.

In other cases aliasing is optional, however it is conventional to use in any join-query.

Q: Identify the join type in this query:

- ❑ Cross join
- ❑ Equijoin
- ❑ Join of unknown type

Equivalent query rewriting

Cross join form:

```
SELECT DISTINCT fst.WARE
FROM CATEGORY fst
INNER JOIN CATEGORY snd
WHERE fst.WARE=snd.WARE
  AND fst.CLASS<>snd.CLASS;
```

Equijoin form:

```
SELECT DISTINCT fst.WARE
FROM CATEGORY fst
INNER JOIN CATEGORY snd
ON fst.WARE=snd.WARE
WHERE fst.CLASS<>snd.CLASS;
```

θ-join (theta-join) **form:**

```
SELECT DISTINCT fst.WARE
FROM CATEGORY fst
INNER JOIN CATEGORY snd
ON fst.WARE=snd.WARE
AND fst.CLASS<>snd.CLASS;
```

θ-join

Relational algebra:

θ-join is generalization of equijoin.

$\mathbf{R_1} \underset{a\theta b}{\bowtie} \mathbf{R_2}$ all the combinations of tuples from $\mathbf{R_1}$ and $\mathbf{R_2}$ where **a** is some attribute of $\mathbf{R_1}$, **b** is attribute of $\mathbf{R_2}$, **θ** is one the predicates (>, ≥, <, ≤, =, ≠) and **a θ b**.

SQL:

Inner join implements extended version of θ-join. The extensions are the following:

- a propositional formula over any number of attributes can be used as a join condition
- additional predicates can be used (LIKE for example)

Formally there are no differences between θ-join and cross join with selection, all these variants are implemented by inner join in SQL.

θ-join: example

Schema

```
--people and their money
TABLE PERSON:
   NAME TEXT
   CASH INTEGER

Ex.: ('John Smith', 1000)
```

```
--manufacturing goods and prices
TABLE GOOD:
   GOOD_NAME TEXT
   PRICE INTEGER

Ex.: ('Soap', 10)
```

Q: What can be bought by each person?

SQL:

```
SELECT p.NAME AS PERSON,
       g.GOOD_NAME AS GOOD
FROM PERSON p, GOOD g
WHERE p.CASH >= g.PRICE; -- θ-join condition
```

θ-join: yet another example

Schema

```
--two-dimensional points
TABLE POINT_2D:
  X REAL
  Y REAL

Ex.: (10.5, 6.3)
```

Q: What points are close to each other (i.e. with distance less that 1.0)?

SQL:

```
SELECT fst.X AS X1, fst.Y AS Y1,
       snd.X AS X2, snd.Y AS Y2
FROM POINT_2D fst, POINT_2D snd
WHERE (fst.X-snd.X)*(fst.X-snd.X)+
      (fst.Y-snd.Y)*(fst.Y-snd.Y)<=1.0 -- θ-join condition
-- there is a bug in this query
```

Notes on usage and performance

- Equijoin is the most used form of join. The generalized θ-join is much more rare.
- There are efficient algorithms to perform equijoin and some other special forms of θ-join (will be discussed in lectures later).
- Despite the fact that θ-join could be implemented as cross join with additional selection this is used as the worst case scenario only due to terrible efficiency.

Returning to the question of the day

The solution was:

```
SELECT DISTINCT MATERIAL.WARE AS MATERIAL,
                PRODUCT.WARE  AS PRODUCT
FROM MATERIAL, PRODUCT
WHERE MATERIAL.RECIPE_ID=PRODUCT.RECIPE_ID;
```

Q: Have we taken everything into the account?
What about the **water**? Do we have recipes for it?

Left outer join

Relational algebra:
$\mathbf{R_1} ⟕ \mathbf{R_2}$ is $\mathbf{R_1} \bowtie \mathbf{R_2}$ (θ variant in general case) with additional tuples from $\mathbf{R_1}$ that have no matching tuples (with NULLs for the missing attributes from $\mathbf{R_2}$)

SQL: the fourth try

```
SELECT DISTINCT m.WARE, p.WARE
FROM PRODUCT p
LEFT OUTER JOIN MATERIAL m
ON m.RECIPE_ID=p.RECIPE_ID;
```

```
>> WARE        WARE
   ----------------
   Water       Drinking water
   NULL        Water
   …
```

Warning: unlike the inner join, moving the join condition from ON to WHERE section will break the join itself.

Right and full outer join

SQL: RIGHT OUTER JOIN

Relational algebra:
$\mathbf{R_1} ⟖ \mathbf{R_2}$ is $\mathbf{R_1} \bowtie \mathbf{R_2}$ (θ variant in general case) with additional tuples from $\mathbf{R_2}$ that have no matching tuples (with NULLs for the missing attributes)

SQL: FULL OUTER JOIN

Relational algebra:
$\mathbf{R_1} ⟗ \mathbf{R_2} = \mathbf{R_1} ⟖ \mathbf{R_2} \cup \mathbf{R_1} ⟕ \mathbf{R_2}$

Note: in SQLite only the left join is implemented.
However, the right join can be replaced with the left one and the appropriate projection.
The full join can be implemented directly by its definition.

Outer join: alternative syntax

```
--left join in ANSI SQL
--this syntax is not supported by SQLite

SELECT DISTINCT m.WARE, p.WARE
FROM PRODUCT p, MATERIAL m
WHERE p.RECIPE_ID=m.RECIPE_ID(+);
```

NULL value

- **NULL**s can be explicitly put into DB or produced by some statements (OUTER JOIN for example)
- **NULL** is considered as value of any primitive type (TEXT, INTEGER, etc.)
- Causes any regular predicate or function/operation to return **NULL** (that is logically false)
- Ignored by any aggregation function

NULL statements and functions

- X IS NULL – explicitly checks if the value is NULL
- X IS NOT NULL – explicitly checks if the value is not NULL
- COALESCE(X, Y) – returns X if it is not NULL and Y otherwise

Programming style examples

```
SELECT tab1_lft.ATTR1, tab2_rght.ATTR1
FROM TABLE1 tab1_lft
INNER JOIN TABLE1 tab1_rght
ON tab1_lft.KEY1=tab1_rght.KEY2
WHERE tab1_lft.ATTR2>0 AND
      tab1_rght.ATTR2>0
```

```
SELECT tab1_lft.ATTR1 AS LEFT,
       tab2_rght.ATTR1 AS RIGHT
FROM TABLE1 tab1_lft
JOIN TABLE1 tab1_rght
  ON tab1_lft.KEY1=tab1_rght.KEY2
WHERE tab1_lft.ATTR2>0
  AND tab1_rght.ATTR2>0
```

```
SELECT tab1_lft.ATTR1 AS LEFT,
       tab2_rght.ATTR1 AS RIGHT
FROM TABLE1 tab1_lft
JOIN TABLE1 tab1_rght
     ON tab1_lft.KEY1=tab1_rght.KEY2
WHERE tab1_lft.ATTR2>0
      AND tab1_rght.ATTR2>0
```