

## EXAMPLES

Time to play. Perhaps postpone examples after comparison with SQL (next subsections)

### Aspects

- join as “extending” operation (cartesian product – “all pairs of X and Y such that ...”)
- equijoin as “restricting” operation
- natural join/equijoin in many cases along key/foreign key relationships
- relational division (in case of queries of the style “return all X that are in a given relation with all Y such that ...”)

105

## 3.2 SQL

SQL: Structured (Standard) Query Language

**Literature:** A Guide to the SQL Standard, 3rd Edition, C.J. Date and H. Darwen, Addison-Wesley 1993

**History:** about 1974 as SEQUEL (IBM System R, INGRES@Univ. Berkeley, first product: Oracle in 1978)

**Standardization:**

**SQL-86** and **SQL-89:** core language, based on existing implementations, including procedural extensions

**SQL-92 (SQL2):** some additions

**SQL-99 (SQL3):**

- active rules (triggers)
- recursion
- object-relational and object-oriented concepts

106

## Underlying Data Model

SQL uses the relational model:

- SQL relations are **multisets (bags)** of tuples (i.e., they can contain duplicates)
- Notions: Relation  $\rightsquigarrow$  Table  
Tuple  $\rightsquigarrow$  Row  
Attribute  $\rightsquigarrow$  Column

The relational algebra serves as theoretical base for SQL as a query language.

- comprehensive treatment in the “Practical Training SQL”  
(<http://dbis.informatik.uni-goettingen.de/Teaching/DBP/>)

## BASIC STRUCTURE OF SQL QUERIES

```
SELECT  $A_1, \dots, A_n$       (... corresponds to  $\pi$  in the algebra)
FROM  $R_1, \dots, R_m$     (... specifies the contributing relations)
WHERE  $F$                 (... corresponds to  $\sigma$  in the algebra)
```

corresponds to the algebra expression  $\pi[A_1, \dots, A_n](\sigma[F](r_1 \times \dots \times r_m))$

- Note: cartesian product  $\rightarrow$  prefixing (optional)

### Example

```
SELECT code, capital, country.population, city.population
FROM country, city
WHERE country.code = city.country
      AND city.name = country.capital
      AND city.province = country.province;
```

## PREFIXING, ALIASING AND RENAMING

- Prefixing: *tablename.attr*
- Aliasing of relations in the FROM clause:

```
SELECT alias1.attr1, alias2.attr2
FROM table1 alias1, table2 alias2
WHERE ...
```

- Renaming of result columns of queries:

```
SELECT attr1 AS name1, attr2 AS name2
FROM ... WHERE ...
```

(formal algebra equivalent: renaming)

109

## SUBQUERIES

Subqueries of the form (SELECT ... FROM ... WHERE ...) can be used anywhere where a relation is required:

Subqueries in the FROM clause allow for selection/projection/computation of intermediate results/subtrees before the join:

```
SELECT ...
FROM (SELECT ...FROM ...WHERE ...),
     (SELECT ...FROM ...WHERE ...)
WHERE ...
```

(interestingly, although “basic relational algebra”, this has been introduced e.g. in Oracle only in the early 90s)

Subqueries in other places allow to express other intermediate results:

```
SELECT ... (SELECT ...FROM ...WHERE ...) FROM ...
WHERE [NOT] value1 IN (SELECT ...FROM ...WHERE)
AND [NOT] value2 comparison-op [ALL|ANY] (SELECT ...FROM ...WHERE)
AND [NOT] EXISTS (SELECT ...FROM ...WHERE);
```

110

## SUBQUERIES IN THE FROM CLAUSE

- often in combination with aliasing and renaming of the results of the subqueries.

```
SELECT alias1.name1,alias2.name2
FROM (SELECT attr1 AS name1 FROM ...WHERE ...) alias1,
     (SELECT attr2 AS name2 FROM ...WHERE ...) alias2 WHERE ...
```

... all big cities that belong to large countries:

```
SELECT city, country
FROM (SELECT name AS city, country AS code2
      FROM city
      WHERE population > 1000000
     ),
     (SELECT name AS country, code
      FROM country
      WHERE area > 1000000
     )
WHERE code = code2;
```

111

## SUBQUERIES

- Subqueries of the form (SELECT ... FROM ... WHERE ...) that result in a **single value** can be used anywhere where a value is required

```
SELECT function(..., (SELECT ... FROM ... WHERE ...))
FROM ... ;

SELECT ...
FROM ...
WHERE value1 = (SELECT ... FROM ... WHERE ...)
      AND value2 < (SELECT ... FROM ... WHERE ...);
```

112

## Subqueries in the WHERE clause

### Non-Correlated subqueries

... the simple ones. Inner SFW independent from outer SFW

```
SELECT name
FROM country
WHERE area >
  (SELECT area
   FROM country
   WHERE code='D');

SELECT name
FROM country
WHERE code IN
  (SELECT country
   FROM encompasses
   WHERE continent='Europe');
```

### Correlated subqueries

Inner SELECT ... FROM ... WHERE references value of outer SFW in its WHERE clause:

```
SELECT name
FROM city
WHERE population > 0.25 *
  (SELECT population
   FROM country
   WHERE country.code = city.country);

SELECT name, continent
FROM country, encompasses enc
WHERE country.code=enc.country
  AND area > 0.25 *
  (SELECT area
   FROM continent
   WHERE name = enc.continent);
```

113

## Subqueries: EXISTS

- EXISTS makes only sense with a correlated subquery:

```
SELECT name
FROM country
WHERE EXISTS (SELECT *
              FROM city
              WHERE country.code = city.country
                 AND population > 1000000);
```

algebra equivalent: semijoin.

- NOT EXISTS can be used to express things that otherwise cannot be expressed by SFW:

```
SELECT name
FROM country
WHERE NOT EXISTS (SELECT *
                  FROM city
                  WHERE country.code = city.country
                     AND population > 1000000);
```

Alternative: use (SFW) MINUS (SFW)

114

## SET OPERATIONS: UNION, INTERSECT, MINUS/EXCEPT

```
(SELECT name FROM city) INTERSECT (SELECT name FROM country)
```

Often applied with renaming:

```
SELECT *  
FROM (SELECT river AS name, country, province FROM geo_river)  
     UNION (SELECT lake AS name, country, province FROM geo_lake)  
     UNION (SELECT sea AS name, country, province FROM geo_sea)  
WHERE country = 'D'
```

115

## GROUPING AND AGGREGATION

### General Structure of SQL Queries

SELECT $A_1, \dots, A_n$	list of attributes
FROM $R_1, \dots, R_m$	list of relations
WHERE $F$	condition(s)
GROUP BY $B_1, \dots, B_k$	list of grouping attributes
HAVING $G$	condition on groups, same syntax as WHERE clause
ORDER BY $H$	sort order

Aggregation: SUM, AVG, MIN, MAX

Applied to a whole relation or to each group (GROUP BY):

```
SELECT MAX(population) FROM country  
  
SELECT country, SUM(population), MAX(population)  
FROM City  
GROUP BY Country  
HAVING SUM(population) > 10000000;
```

SELECT contains only aggregates, and attributes that are the same inside each group.

116

## CONSTRUCTING QUERIES

For each problem there are multiple possible equivalent queries in SQL (cf. Example 3.15). The choice is mainly a matter of personal taste.

- analyze the problem “systematically”:
  - collect all relations (in the FROM clause) that are needed
  - generate a suitable conjunctive WHERE clause⇒ leads to a single “broad” SFW query  
(cf. conjunctive queries, relational calculus)
- analyze the problem “top-down”:
  - take the relations that directly contribute to the result in the (outer) FROM clause
  - do all further work in correlated subquery/-queries in the WHERE clause⇒ leads to a “main” part and nested subproblems
- decomposition of the problem into subproblems:
  - subproblems are solved by nested SFW queries that are combined in the FROM clause of a surrounding query

117

### Comparison

SQL:

```
SELECT  $A_1, \dots, A_n$  FROM  $R_1, \dots, R_m$  WHERE  $F$ 
```

- **equivalent expression in the relational algebra:**

$$\pi[A_1, \dots, A_n](\sigma[F](r_1 \times \dots \times r_m))$$

- **Algorithm (nested-loop):**

FOR each tuple  $t_1$  in relation  $R_1$  DO

    FOR each tuple  $t_2$  in relation  $R_2$  DO

        :

            FOR each tuple  $t_n$  in relation  $R_n$  DO

                IF tuples  $t_1, \dots, t_n$  satisfy the WHERE-clause THEN

                    evaluate the SELECT clause and generate the result tuple (projection).

Note: the tuple variables can also be introduced in SQL explicitly as alias variables:

```
SELECT  $A_1, \dots, A_n$  FROM  $R_1$   $t_1, \dots, R_m$   $t_m$  WHERE  $F$ 
```

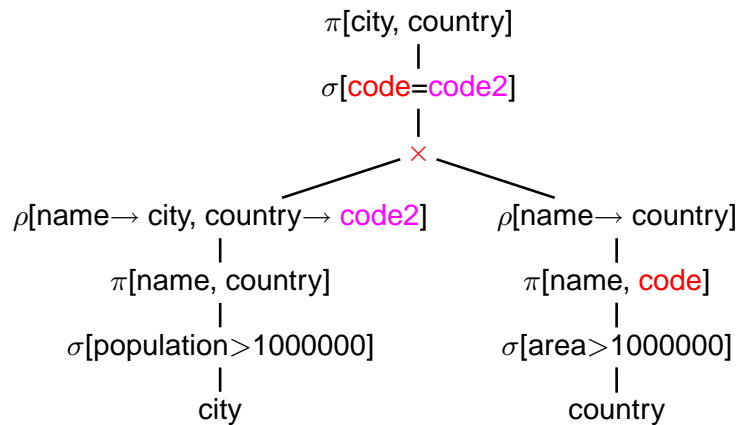
(then optionally using  $t_i.attr$  in SELECT and WHERE)

118

## Comparison: Subqueries

- Subqueries in the FROM-clause (cf. Slide 111): **joined subtrees** in the algebra

```
SELECT city, country.name
FROM (SELECT name AS city,
           country AS code2
       FROM city
       WHERE population > 1000000
    ),
     (SELECT name AS country, code
       FROM country
       WHERE area > 1000000
    )
WHERE code = code2;
```

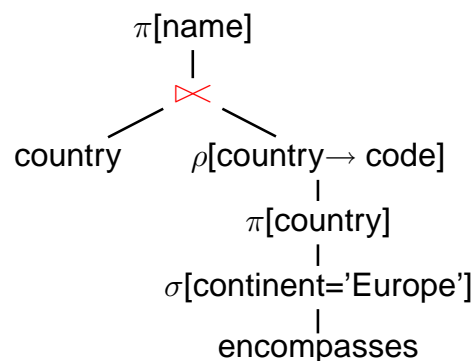


119

## Comparison: Subqueries in the WHERE clause

- WHERE ... IN uncorrelated-subquery (cf. Slide 113):  
Natural semijoin outer tree with the subquery tree;

```
SELECT name
FROM country
WHERE code IN
  (SELECT country
   FROM encompasses
   WHERE continent='Europe');
```



Note that the natural semijoin serves as an equi-selection where all tuples from the outer expression qualify that match an element of the result of the inner expression.

120



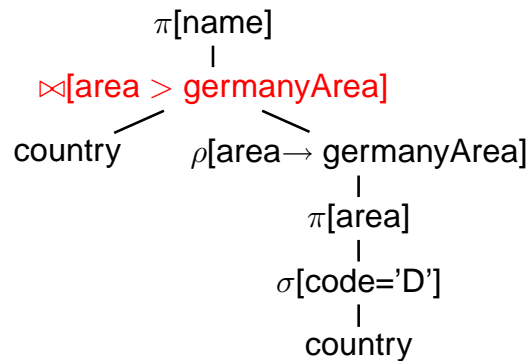
## Comparison: Subqueries

- WHERE value *op* uncorrelated-subquery:

(cf. Slide 113):

join of outer expression with subquery, selection, projection to outer attributes

```
SELECT name
FROM country
WHERE area >
  (SELECT area
   FROM country
   WHERE code='D');
```



Note: the table that results from the join has the format (name, code, area, population, ..., germanyArea).

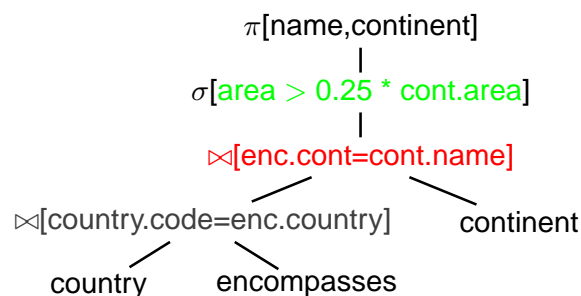
121

## Comparison: Correlated Subqueries

- WHERE value *op* correlated-subquery:

- tree<sub>1</sub>: outer expression
- tree<sub>2</sub>: subquery, uncorrelated
- natural join/semijoin of both trees contains the **correlating condition**
- afterwards: **WHERE condition**

```
SELECT name, continent
FROM country, encompasses enc
WHERE country.code=enc.country
AND area > 0.25 *
  (SELECT area
   FROM continent
   WHERE name=enc.continent);
```



- equivalent with semijoin:  $\bowtie [enc.cont=cont.name \wedge area > 0.25 * cont.area]$

122

## Comparison: Correlated Subqueries

... comment to previous slide:

- although the tree expression looks less target-oriented than the SQL correlated subquery, it does the same:
- instead of iterating over the tuples of the outer SQL expression and evaluating the inner one for each of the tuples,
- the results of the inner expression are “precomputed” and iteration over the outer result just fetches the corresponding one.
- effectiveness depends on the situation:
  - how many of the results of the subquery are actually needed (worst case: no tuple survives the outer local WHERE clause).
  - are there results of the subquery that are needed several times.

database systems are often able to internally choose the most effective solution (schema-based and statistics-based)

... see next section.

123

## Comparison: EXISTS-Subqueries

- WHERE EXISTS: similar to above:  
correlated subquery, no additional condition after natural semijoin
- SELECT ... FROM X,Y,Z WHERE NOT EXISTS (SFW):

```
SELECT ...  
FROM ((SELECT * FROM X,Y,Z) MINUS  
      (SELECT X,Y,Z WHERE EXISTS (SFW)))
```

## Results

- all queries (without NOT-operator) including subqueries without grouping/aggregation can be translated into SPJR-trees (selection, projection, join, renaming)
- they can even be flattened into a single broad cartesian product, followed by a selection and a projection.

124

## Comparison: the differences between Algebra and SQL

- The relational algebra has no notion of grouping and aggregate functions
- SQL has no clause that corresponds to relational division

### Example 3.17

Consider again Example 3.10 (Slide 86).

The corresponding SQL formulation that implements division corresponds to the textual

“all countries that occur in  $\pi[\text{country}](\text{enc})$ , with the additional condition that they occur in  $\text{enc}$  together with each of the continent values that occur in  $\text{cts}$ ”,

or equivalent

“all countries  $c$  in  $\pi[\text{country}](\text{enc})$  such that there is no continent value  $\text{cont}$  in  $\text{cts}$  such that  $c$  does not occur together with  $\text{cont}$  in  $\text{enc}$ ”: □

125

### Example 3.17 (Continued)

“all countries  $c$  in  $\pi[\text{country}](\text{enc})$  such that there is no continent value  $\text{cont}$  in  $\text{cts}$  such that  $c$  does not occur together with  $\text{cont}$  in  $\text{enc}$ ”:

SELECT  $\text{enc1.country}$

FROM  $\text{enc}$   $\text{enc1}$

— consider  $\text{enc1.country}=\text{“R”}$  and  $\text{enc1.country}=\text{“D”}$

WHERE NOT EXISTS

— correlated subquery

( ( SELECT  $\text{ct}$   
FROM  $\text{cts}$ )

— always

“Europe”
“Asia”

MINUS

( SELECT  $\text{ct}$

FROM  $\text{enc}$   $\text{enc2}$

WHERE  $\text{enc1.country} = \text{enc2.country}$

for “R”:

“R”	“Asia”
“R”	“Europe”

for “D”:

“D”	“Europe”
-----	----------

)  
)

— remains: for “R”: nothing  $\leadsto$  “R” belongs to the result

for D: “Asia”  $\leadsto$  “D” does not belong to the result

126

## Orthogonality

Full orthogonality means that an expression that results in a relation is allowed everywhere, where an input relation is allowed

- subqueries in the FROM clause
- subqueries in the WHERE clause
- subqueries in the SELECT clause (returning a single value)
- combinations of set operations

But:

- Syntax of aggregation functions is not fully orthogonal:

Not allowed: `SUM(SELECT ...)`

```
SELECT SUM(pop_biggest)
      FROM (SELECT country, MAX(population) AS pop_biggest
            FROM City
            GROUP BY country);
```

- The language OQL (Object Query Language) uses similar constructs and is fully orthogonal.

127

## 3.3 Efficient Algebraic Query Evaluation

Queries are formulated *declaratively* (e.g., SQL or algebra trees), actually built over a small set of basic operations (cf. the definition of the relational algebra).

**Semantical optimization:** consider integrity constraints in the database.

Example:  $population > 0$ , thus, a query that asks for negative values can be answered without explicit computation.

- not always obvious
- general case: first-order theorem proving.
- special cases: [see lecture on Database Theory]

**Logical/algebraic optimization:** search for an equivalent algebra expression that performs better:

- size of intermediate results,
- implementation of operators as algorithms,
- presence of indexes and order.

128

## ALGEBRAIC OPTIMIZATION

The operator tree of an algebra expression provides a base for several optimization strategies:

- reusing intermediate results
- equivalent restructuring of the operator tree
- “shortcuts” by melting several operators into one (e.g., join + equality predicate  $\rightarrow$  equijoin)
- combination with actual situation: indexes, properties of data

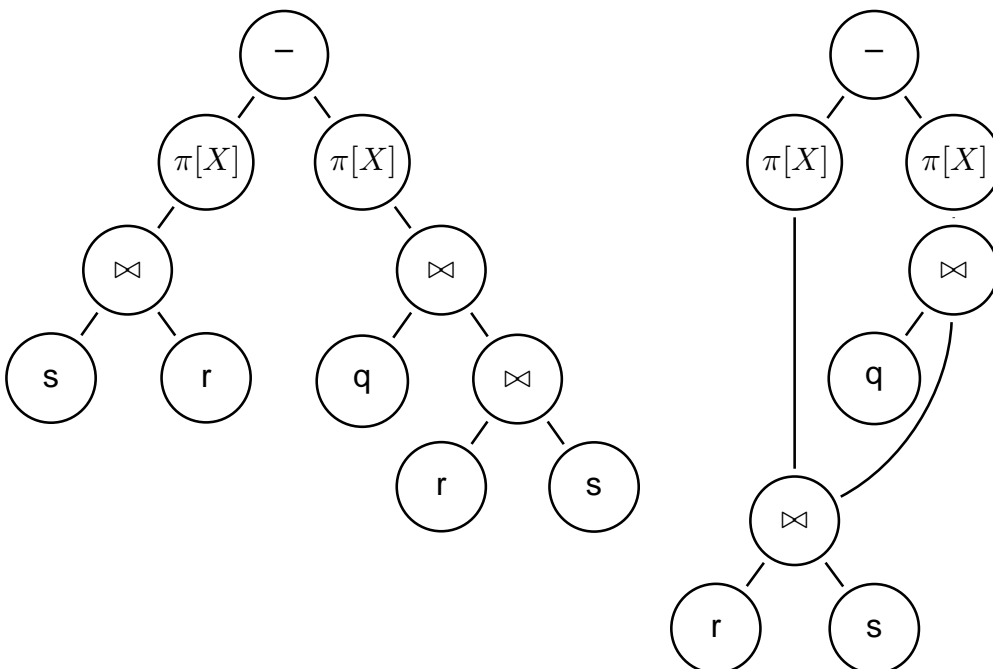
Real-life databases implement this functionality.

- SQL: **declarative** specification of a query
- internal: algebra tree + optimizations

129

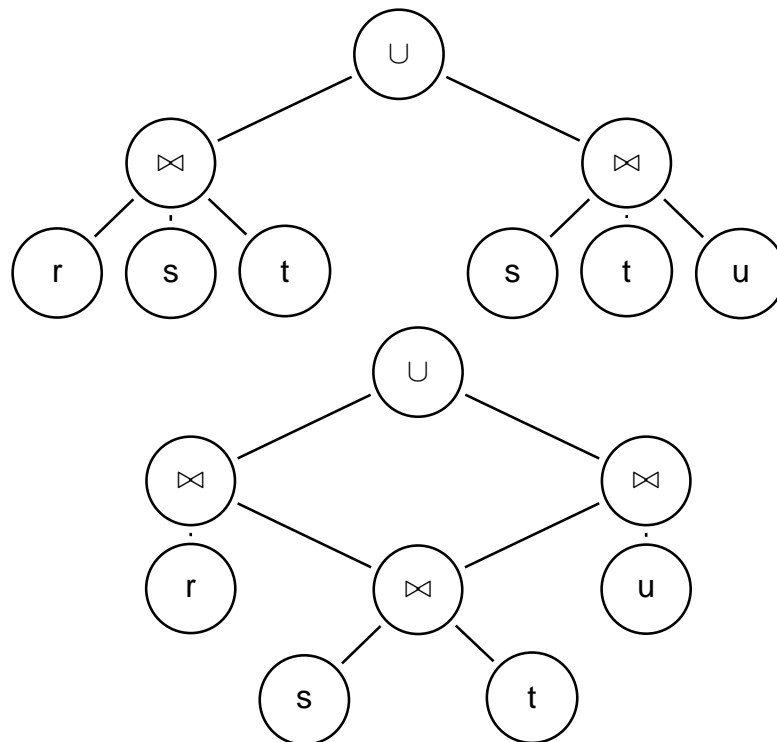
## REUSING INTERMEDIATE RESULTS

- Multiply occurring subtrees can be reused (directed acyclic graph (DAG) instead of algebra tree)



130

## Reusing intermediate results



131

## OPTIMIZATION BY TREE RESTRUCTURING

- Equivalent transformation of the operator tree that represents an expression
- Based on the equivalences shown on Slide 102.
- minimize the size of intermediate results  
(reject tuples/columns as early as possible during the computation)
- selections reduce the number of tuples
- projections reduce the size of tuples
- apply both as early as possible (i.e., before joins)
- different application order of joins
- semijoins instead of joins (in combination with implementation issues; see next section)

132

## Push Selections Down

Assume  $r, s \in \text{Rel}(\bar{X})$ ,  $\bar{Y} \subseteq \bar{X}$ .

$$\sigma[\text{cond}](\pi[\bar{Y}](r)) \equiv \pi[\bar{Y}](\sigma[\text{cond}](r))$$

(condition: *cond* does not use attributes from  $\bar{X} - \bar{Y}$ ,  
otherwise left term is undefined)

$$\sigma_{\text{pop} > 1E6}(\pi[\text{name, pop}](\text{country})) \equiv \pi[\text{name, pop}](\sigma_{\text{pop} > 1E6}(\text{country}))$$

$$\sigma[\text{cond}](r \cup s) \equiv \sigma[\text{cond}](r) \cup \sigma[\text{cond}](s)$$

$$\begin{aligned} \sigma_{\text{pop} > 1E6}(\pi[\text{name, pop}](\text{country}) \cup \pi[\text{name, pop}](\text{city})) \\ \equiv \sigma_{\text{pop} > 1E6}(\pi[\text{name, pop}](\text{country})) \cup \sigma_{\text{pop} > 1E6}(\pi[\text{name, pop}](\text{city})) \end{aligned}$$

$$\sigma[\text{cond}](\rho[N](r)) \equiv \rho[N](\sigma[\text{cond}'](r))$$

(where *cond'* is obtained from *cond* by renaming according to *N*)

$$\sigma[\text{cond}](r \cap s) \equiv \sigma[\text{cond}](r) \cap \sigma[\text{cond}](s)$$

$$\sigma[\text{cond}](r - s) \equiv \sigma[\text{cond}](r) - \sigma[\text{cond}](s)$$

$\pi$  : see comment above. Optimization uses only left-to-right.

133

## Push Selections Down (Cont'd)

Assume  $r \in \text{Rel}(\bar{X})$ ,  $s \in \text{Rel}(\bar{Y})$ . Consider  $\sigma[\text{cond}](r \bowtie s)$ .

Let  $\text{cond} = \text{cond}_{\bar{X}} \wedge \text{cond}_{\bar{Y}} \wedge \text{cond}_{\overline{\bar{X}\bar{Y}}}$  such that

- $\text{cond}_{\bar{X}}$  is concerned only with attributes in  $\bar{X}$
- $\text{cond}_{\bar{Y}}$  is concerned only with attributes in  $\bar{Y}$
- $\text{cond}_{\overline{\bar{X}\bar{Y}}}$  is concerned both with attributes in  $\bar{X}$  and in  $\bar{Y}$ .

Then,

$$\sigma[\text{cond}](r \bowtie s) \equiv \sigma[\text{cond}_{\overline{\bar{X}\bar{Y}}}] (\sigma[\text{cond}_{\bar{X}}](r) \bowtie \sigma[\text{cond}_{\bar{Y}}](s))$$

### Example 3.18

*Names of all countries that have been founded earlier than 1970, their capital has more than 1.000.000 inhabitants, and more than half of the inhabitants live in the capital.*  $\square$

134

### Example 3.18 (Continued)

(Solution)

$$\begin{aligned}
 & \pi[Name](\sigma[establ < "01 01 1970" \wedge city.pop > 1.000.000 \wedge country.pop < 2 \cdot city.pop] \\
 & \quad (country \times_{country.(capital,prov,code)=city(name,prov,country)} city)) \\
 & \equiv \pi[Name](\sigma[country.pop < 2 \cdot city.pop] \\
 & \quad (\sigma[establ < "01 01 1970"](country) \\
 & \quad \quad \times_{country.(capital,prov,code)=city(name,prov,country)} \\
 & \quad \quad \sigma[city.pop > 1.000.000](city))) \quad \square
 \end{aligned}$$

- Nevertheless, if  $cond$  is e.g. a complex mathematical calculation, it can be cheaper first to reduce the number of tuples by  $\cap$ ,  $-$ , or  $\bowtie$

$\Rightarrow$  data-dependent strategies (see later)

135

### Push Projections Down

Assume  $r, s \in \text{Rel}(\bar{X})$ ,  $\bar{Y} \subseteq \bar{X}$ .

Let  $cond = cond_{\bar{X}} \wedge cond_{\bar{Y}}$  such that

- $cond_{\bar{Y}}$  is concerned only with attributes in  $\bar{Y}$
- $cond_{\bar{X}}$  is the remaining part of  $cond$  that is also concerned with attributes  $\bar{X} \setminus \bar{Y}$ .

$$\pi[\bar{Y}](\sigma[cond](r)) \equiv \sigma[cond_{\bar{Y}}](\pi[\bar{Y}](\sigma[cond_{\bar{X}}](r)))$$

$$\pi[\bar{Y}](\rho[N](r)) \equiv \rho[N](\pi[\bar{Y}'](r))$$

(where  $\bar{Y}'$  is obtained from  $\bar{Y}$  by renaming according to  $N$ )

$$\pi[\bar{Y}](r \cup s) \equiv \pi[\bar{Y}](r) \cup \pi[\bar{Y}](s)$$

- Note that this does *not* hold for “ $\cap$ ” and “ $-$ ”!
- advantages of pushing “ $\sigma$ ” vs. “ $\pi$ ” are data-dependent  
Default: push  $\sigma$  lower.

Assume  $r \in \text{Rel}(\bar{X})$ ,  $s \in \text{Rel}(\bar{Y})$ .

$$\pi[\bar{Z}](r \bowtie s) \equiv \pi[\bar{Z}](\pi[\bar{X} \cap \bar{Z}\bar{Y}](r) \bowtie \pi[\bar{Y} \cap \bar{Z}\bar{X}](s))$$

- complex interactions between reusing subexpressions and pushing selection/projection

136



## Application Order of Joins

Minimize intermediate results:

```
SELECT organization.name, country.name
FROM organization, country, is_member
WHERE organization.abbrev = is_member.organization
      AND country.code = is_member.country
```

Exploit selectivity of join:

- $(\underbrace{\text{org} \times \text{country}}_{200 \cdot 200 = 40000}) \bowtie \text{is\_member}$   
7000
- $(\underbrace{\text{org} \bowtie \text{is\_member}}_{200, 7000 \rightsquigarrow 7000}) \bowtie \text{country}$   
7000

If indexes on country.code and organization.abbrev are available:

- loop over is\_member
- extend each tuple with matching country and organization by using the indexes.

137

## Example/Exercise

Consider the equivalent (to the previous example) query:

```
SELECT organization.name, country.name
FROM organization, country,
WHERE EXISTS
  (SELECT *
   FROM is_member
   WHERE organization.abbrev = is_member.organization
         AND country.code = is_member.country)
```

- suggests the non-optimal evaluation!
- transform the above query into algebra
- ... yields the same “broad” join as before ...
- ... and leads to the optimized join ordering.

138

## OPERATOR EVALUATION BY PIPELINING

- above, each algebra operator has been considered separately
- if a query consists of several operators, the materialization of intermediate results should be avoided
- **Pipelining** denotes the immediate propagation of tuples to subsequent operators

### Example 3.19

- $\sigma[A = 5 \wedge B > 6]R$ :

Assume an index that supports the condition  $A = 5$ .

- *without pipelining*: compute  $\sigma[A = 5]R$  using the index, obtain  $R'$ . Then, compute  $\sigma[B > 6]R'$ .
- *with pipelining*: compute  $\sigma[A = 5]R$  using the index, and check **on-the fly** each qualifying tuple against  $\sigma[B > 6]$ . □

- **Unary** (i.e., selection and projection) operations can always be pipelined with the next lower binary operation (e.g., join)

139

### Example 3.19 (Continued)

- $\sigma[cond](R \bowtie S)$ :

- *without pipelining*: compute  $R \bowtie S$ , obtain  $RS$ , then compute  $\sigma[cond](RS)$ .
- *with pipelining*: during computing  $(R \bowtie S)$ , each tuple is immediately checked whether it satisfies *cond*.

- $(R \bowtie S) \bowtie T$ :

- *without pipelining*: compute  $R \bowtie S$ , obtain  $RS$ , then compute  $RS \bowtie T$ .
- *with pipelining*: during computing  $(R \bowtie S)$ , each tuple is immediately propagated to one of the described join algorithms for computing  $RS \bowtie T$ . □

Most database systems combine materialization of intermediate results, iterator-based implementation of algebra operators, and pipelining.

140