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

Underlying Data Model

SQL uses the relational model:

- SQL relations are **multisets (bags)** of tuples (i.e., they can contain duplicates)
- Notions: Relation ⤳ Table
  Tuple ⤳ Row
  Attribute ⤳ 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` $A_1, \ldots, A_n$	(…corresponds to π in the algebra)
`FROM` $R_1, \ldots, R_m$	(…specifies the contributing relations)
`WHERE` F	(…corresponds to σ in the algebra)

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

- Note: cartesian product → 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` $alias_1$.$attr_1$,$alias_2$.$attr_2$
  `FROM` $table_1$ $alias_1$, $table_2$ $alias_2$
  `WHERE ...`

- Renaming of result columns of queries:

  `SELECT` $attr_1$ `AS` $name_1$, $attr_2$ `AS` $name_2$
  `FROM ...  WHERE ...`

  (formal algebra equivalent: renaming)

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);
```

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;
```

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 ...);
```

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);
```

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)

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';
```

Set Operations and Attribute Names

The relational algebra requires $\bar{X} = \bar{Y}$ for $R(\bar{X}) \cup S(\bar{X})$, $R(\bar{X}) \cap S(\bar{X})$, and $R(\bar{X}) - S(\bar{X})$:

- attributes are unordered, the tuple model is a “slotted” model.

In SQL,

```
(SELECT river, country, province FROM geo_river)
 UNION
(SELECT lake, country, province FROM geo_lake)
```

is allowed and the resulting table has the format (river, country, province) (note that the name of the first column may be indeterministic due to internal optimization).

- the SQL model is a “positional” model, where the name of the i-th column is just inferred “somehow”,
- cf. usage of column number in ... `ORDER BY 1`,
- note that column numbers can only be used if there is no ambiguity with numeric values, e.g.,
  `SELECT name, 3 FROM country`
  yields a table whose second column has always the value 3.

HANDLING OF DUPLICATES

In contrast to algebra relations, SQL tables may contain duplicates (cf. Slide 113):

- some applications require them
- duplicate elimination is relatively expensive ($O(n \log n)$)

⇒ do not do it automatically

⇒ SQL allows for *explicit* removal of duplicates:
Keyword: `SELECT DISTINCT` $A_1, \ldots, A_n$ `FROM ...`
The internal optimization can sometimes put it at a position where it does not incur additional costs.

GENERAL STRUCTURE OF SQL QUERIES:

SELECT [DISTINCT] $A_1, \ldots, A_n$	list of expressions
FROM $R_1, \ldots, R_m$	list of relations
WHERE F	condition(s)
GROUP BY $B_1, \ldots, B_k$	list of grouping attributes
HAVING G	condition on groups, same syntax as WHERE clause
ORDER BY H	sort order – only relevant for output

- `ORDER BY`: specifies output order of tuples

  ```
  SELECT name, population FROM city;
  ```

  full syntax: `ORDER BY` *attribute-list* `[ASC|DESC] [NULLS FIRST|LAST]`
  (ascending/descending)
  Multiple attributes allowed:

  ```
  SELECT * FROM city ORDER BY country, province;
  ```

  Next: How many people live in the cities in each country?

- `GROUP BY`: form groups of “related” tuples and generate one output tuple for each group
- `HAVING`: conditions evaluated on the groups

Grouping and Aggregation

- First Normal Form: all values in a tuple are atomic (string, number, date, . . .)
- `GROUP BY` *attribute-list*: forms groups of tuples that have the same values for *attribute-list*

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

:	:	:	:
Innsbruck	A	Tirol	118000
Vienna	A	Vienna	1583000
:	:	:	:
Graz	A	Steiermark	238000
:	:	:	:

- each group yields *one* tuple which may contain:
  - the group-by attributes
  - *aggregations* of all values in a column: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`

:	:	:	:
country: A	SUM(population): 2434525	MAX(population): 1583000	COUNT(*): 9
:	:	:	:

- `SELECT` and `HAVING`: use these terms.

Aggregation

- Aggregation can be applied to a whole relation:

```
SELECT COUNT(*), SUM(population), MAX(population)
FROM country;
```

- Aggregation with DISTINCT:

```
SELECT COUNT (DISTINCT country)
FROM CITY
WHERE population > 1000000;
```

ALTOGETHER: EVALUATION STRATEGY

SELECT [DISTINCT] $A_1, \ldots, A_n$	list of expressions
FROM $R_1, \ldots, R_m$	list of relations
WHERE F	condition(s)
GROUP BY $B_1, \ldots, B_k$	list of grouping attributes
HAVING G	condition on groups, same syntax as WHERE clause
ORDER BY H	sort order – only relevant for output

1. evaluate FROM and WHERE,
2. evaluate GROUP BY $\rightarrow$ yields groups,
3. generate a tuple for each group containing all expressions in HAVING and SELECT,
4. evaluate HAVING on groups,
5. evaluate SELECT (projection, removes things only needed in HAVING),
6. output result according to ORDER BY.

CONSTRUCTING QUERIES

For each problem there are multiple possible equivalent queries in SQL (cf. Example 3.14). 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

Comparison

SQL:

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

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

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

- **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 $t_1, \ldots, t_n$ satisfy the WHERE-clause THEN
                    evaluate the SELECT clause and generate the result tuple (projection).
```

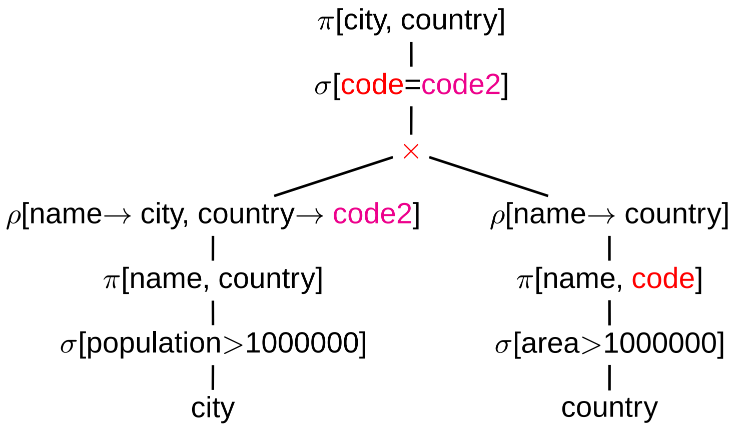
Note: the tuple variables can also be introduced in SQL explicitly as alias variables:

`SELECT` $A_1, \ldots, A_n$ `FROM` R_1 $t_1, \ldots, R_m$ t_m `WHERE` F
(then optionally using $t_i.attr$ in `SELECT` and `WHERE`)

Comparison: Subqueries

- Subqueries in the FROM-clause (cf. Slide 117): 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;
```



π[city, country]

σ[code=code2]

×

ρ[name$\rightarrow$ city, country$\rightarrow$ code2]

π[name, country]

σ[population$>$1000000]

city

ρ[name$\rightarrow$ country]

π[name, code]

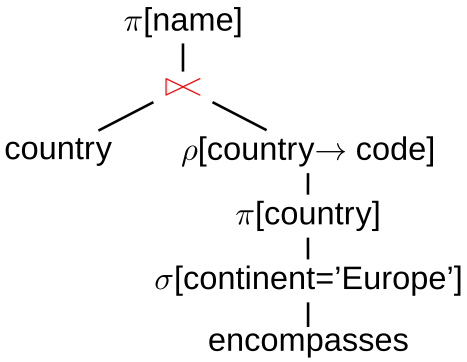
σ[area$>$1000000]

country

Comparison: Subqueries in the WHERE clause

- WHERE ... IN uncorrelated-subquery (cf. Slide 119):
  Natural semijoin outer tree with the subquery tree;

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



π[name]

$\ltimes$

country

ρ[country$\rightarrow$ code]

π[country]

σ[continent=’Europe’]

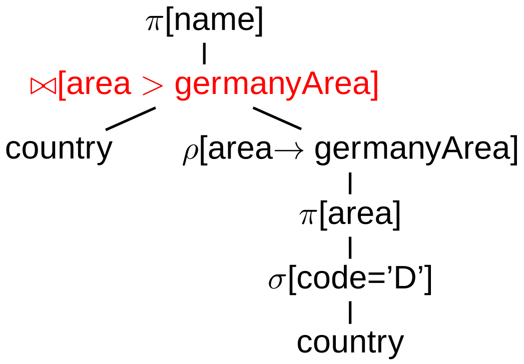
encompasses

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.

Comparison: Subqueries

- WHERE value op uncorrelated-subquery:
  (cf. Slide 119):
  join of outer expression with subquery, selection, projection to outer attributes

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



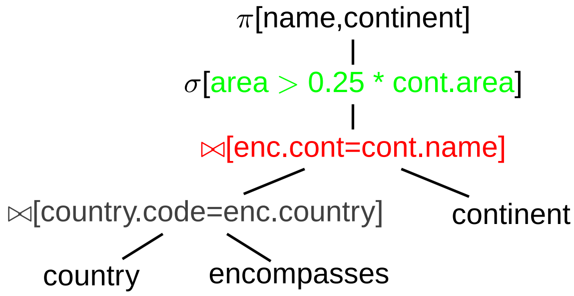
π[name]
$\bowtie$[area $>$ germanyArea]
country
ρ[area$\to$ germanyArea]
π[area]
σ[code='D']
country

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

Comparison: Correlated Subqueries

- WHERE value op correlated-subquery:
  - tree$_1$: outer expression
  - tree$_2$: 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);
```



π[name,continent]
σ[area $>$ 0.25 * cont.area]
$\bowtie$[enc.cont=cont.name]
$\bowtie$[country.code=enc.country]
continent
country
encompasses

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

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.

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.

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 3.16
Consider again Example 3.13 (Slide 99):

“Compute those organizations that have at least one member on each continent”:

$orgOnCont \div \pi[name](continent).$

Exercise: *Use the algebraic expression for $r \div s$ from Slide 98 for stating the query in SQL (use the SQL statement for orgOnCont from Slide 99):*

$$r \div s = \pi[\bar{Z}](r) \setminus \pi[\bar{Z}]((\pi[\bar{Z}](r) \times s) \setminus r).$$

Example 3.16 (Cont’d – Solution to Exercise)

```
(select org
 from (select distinct i.organization as org, e.continent as cont
       from ismember i, encompasses e
       where i.country=e.country ))
minus
( select o1
  from ((select o1,n1
         from (select org as o1
               from (select distinct i.organization as org, e.continent as cont
                     from ismember i, encompasses e
                     where i.country=e.country ))
              ,
              (select name as n1 from continent)
        )
        minus
        (select distinct i.organization as org, e.continent as cont
         from ismember i, encompasses e
         where i.country=e.country )
       )
)
```

Nobody would do this:

- *learn this formula,*
- *copy&paste and fight with parentheses!*

Example 3.16 (Cont’d)

- *Instead of $\pi[\bar{Z}](r)$, a simpler query yielding the $\bar{Z}$ values can be used.
These often correspond to the keys of some relation that represents the instances of some entity type (here: the organizations):*

$$orgOnCont \div \pi[name](continent) =$$
$$\pi[abbreviation](organization) \setminus$$
$$\underbrace{\pi[\bar{Z}](\underbrace{(\underbrace{\pi[abbreviation](organization) \times \pi[name](continent)}_{orgs \times conts}) \setminus orgOnCont}_{\text{the “missing” pairs}})}_{\text{organizations that have a missing pair}}$$

- *the corresponding SQL query is much smaller, and can be constructed intuitively:*

```
(select abbreviation from organization)
minus
( select abbreviation
  from ((select o.abbreviation, c.name
         from organization o, continent c)
        minus
        (select distinct i.organization as org, e.continent as cont
         from ismember i, encompasses e
         where i.country=e.country ) ) )
```

... the structure is the same as the previous one!

Example 3.16 (Cont’d)

The corresponding SQL formulation that implements division corresponds to the textual

“all organizations such that they occur in $orgOnCont$ together with each of the $continent$ names”,

or equivalent

“all organizations org such that there is no value $cont$ in $\pi[name](continent)$ such that org does not occur together with $cont$ in $orgOnCont$”.

```
 select abbreviation
 from organization o
 where not exists
  ((select name from continent)
   minus
   (select cont
    from (select distinct i.organization as org, e.continent as cont
          from ismember i, encompasses e
          where i.country=e.country )
    where org = o.abbreviation))
```

- *the query is still set-theory-based.*
- *there is also a logic-based way:*

Example 3.16 (Cont’d)

“all organizations such that there is no continent such that the organization has no member on this continent (i.e., does not occur in $orgOnCont$ together with this continent)”

```
select abbreviation
from organization o
where not exists
 (select name
  from continent c
  where not exists
   (select *
    from (select distinct i.organization as org, e.continent as cont
          from ismember i, encompasses e
          where i.country=e.country )
    where org = o.abbreviation
      and cont = c.name))
```

Example 3.16 (Cont’d)

Aside: logic-based querying with Datalog

$$\{o \mid \textsf{organization}(o, \ldots) \wedge \neg\exists cont : (\textsf{continent}(cont, \ldots) \wedge \neg\textsf{orgOnCont}(o, cont))\}$$

```
% [mondial].
orgOnCont(O,C,Cont) :- isMember(C,O,_), encompasses(C, Cont,_).
notResult(O) :- organization(O,_,_,_,_,_), continent(Cont,_), not orgOnCont(O,_,Cont).
result(O) :- organization(O,_,_,_,_,_), not notResult(O).
% ?- result(O).
% ?- findall(O, result(O), L).
```

... much shorter.

- *see Lecture on “Database Theory”.* □

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.

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.

ALGEBRAIC OPTIMIZATION

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

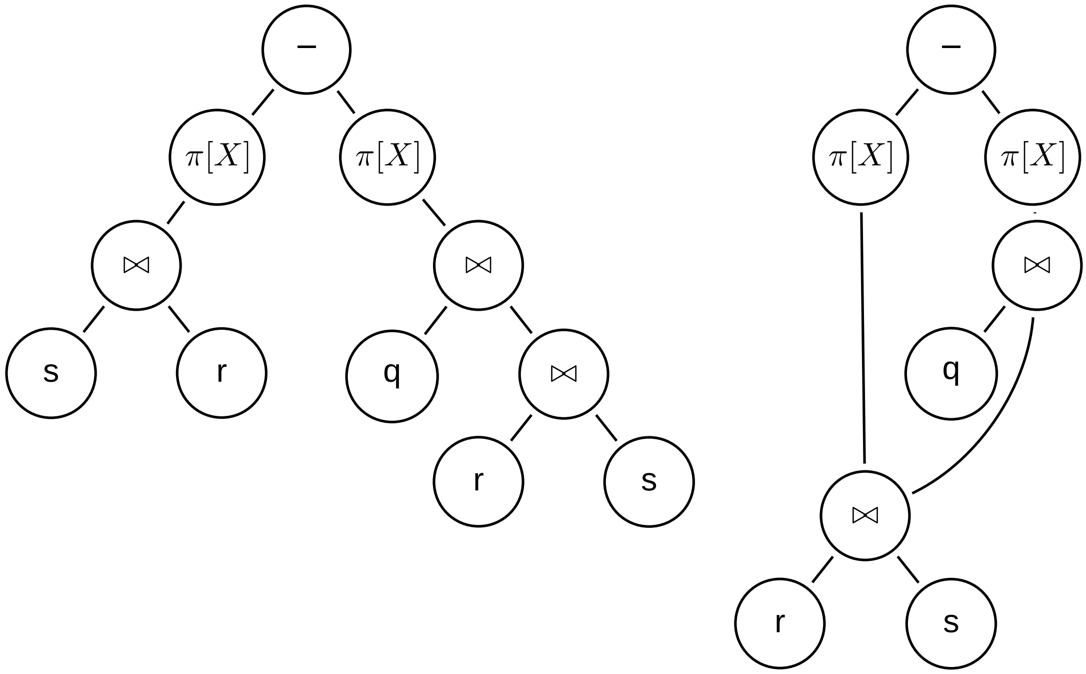
- reusing intermediate results
- equivalent restructuring of the operator tree
- "shortcuts" by melting several operators into one
  (e.g., join + equality predicate → equijoin)
- combination with actual situation: indexes, properties of data

Real-life databases implement this functionality.

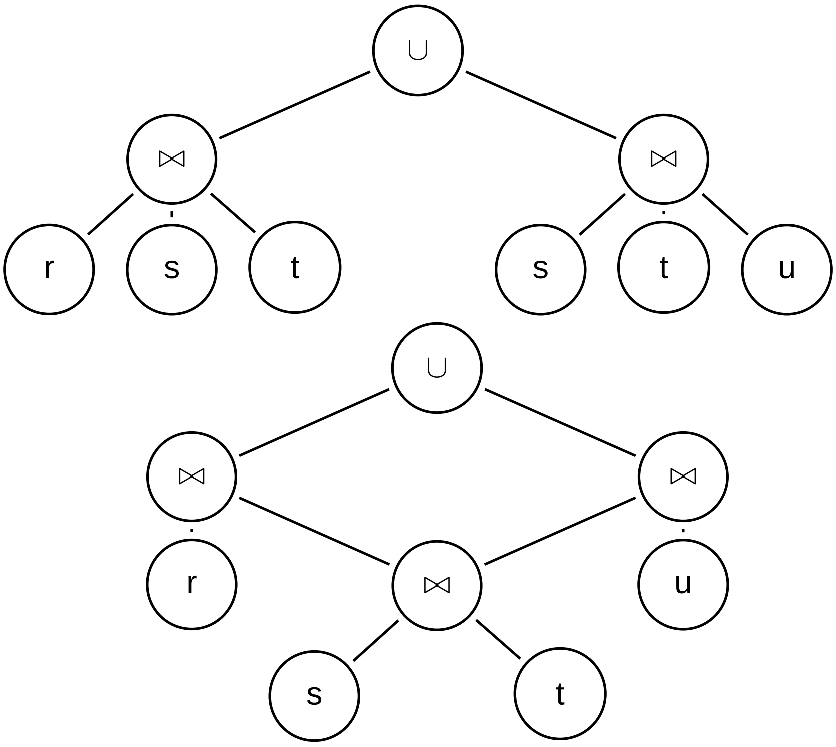
- SQL: **declarative** specification of a query
- internal: algebra tree + optimizations

REUSING INTERMEDIATE RESULTS

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



Reusing intermediate results



OPTIMIZATION BY TREE RESTRUCTURING

- Equivalent transformation of the operator tree that represents an expression
- Based on the equivalences shown on Slide 108.
- 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)

Push Selections Down

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

$$
\begin{aligned}
\sigma[cond](\pi[\bar{Y}](r)) &\equiv \pi[\bar{Y}](\sigma[cond](r)) \\
&\quad \text{(condition: } cond \text{ does not use attributes from } \bar{X} - \bar{Y}, \\
&\quad \text{otherwise left term is undefined)} \\
\sigma_{pop>1E6}(\pi[name, pop](country)) &\equiv \pi[name, pop](\sigma_{pop>1E6}(country)) \\
\sigma[cond](r \cup s) &\equiv \sigma[cond](r) \cup \sigma[cond](s) \\
\sigma_{pop>1E6}(\pi[name, pop](country) \cup \pi[name, pop](city)) & \\
&\equiv \sigma_{pop>1E6}(\pi[name, pop](country)) \cup \sigma_{pop>1E6}(\pi[name, pop](city)) \\
\sigma[cond](\rho[N](r)) &\equiv \rho[N](\sigma[cond'](r)) \\
&\quad \text{(where } cond' \text{ is obtained from } cond \text{ by renaming according to } N\text{)} \\
\sigma[cond](r \cap s) &\equiv \sigma[cond](r) \cap \sigma[cond](s) \\
\sigma[cond](r - s) &\equiv \sigma[cond](r) - \sigma[cond](s)
\end{aligned}
$$

π : see comment above. Optimization uses only left-to-right.

Push Selections Down (Cont'd)

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

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

- $cond_{\bar{X}}$ is concerned only with attributes in $\bar{X}$
- $cond_{\bar{Y}}$ is concerned only with attributes in $\bar{Y}$
- $cond_{\overline{XY}}$ is concerned both with attributes in $\bar{X}$ and in $\bar{Y}$.

Then,

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

Example 3.17

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. $\square$

Example 3.17 (Continued)

(Solution)

$$\pi[Name](\sigma[establ < \text{“01 01 1970”} \wedge city.pop > 1.000.000 \wedge country.pop < 2 \cdot city.pop]$$
$$(country \times_{country.(capital,prov,code)=city(name,prov,country)} city)$$
$$\equiv \pi[Name](\sigma[country.pop < 2 \cdot city.pop]$$
$$(\sigma[establ < \text{“01 01 1970”}](country)$$
$$\times_{country.(capital,prov,code)=city(name,prov,country)}$$
$$\sigma[city.pop > 1.000.000](city))) \qquad \square$$

- 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)

Push Projections Down

Assume $r, s \in$ 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 $\bar{X} \setminus Y$.

$$\begin{array}{rcl} \pi[\bar{Y}](\sigma[cond](r)) & \equiv & \sigma[cond_{\bar{Y}}](\pi[\bar{Y}](\sigma[cond_{\bar{X}}](r))) \\ \pi[\bar{Y}](\rho[N](r)) & \equiv & \rho[N](\pi[\bar{Y}'](r)) \\ & & \text{(where } \bar{Y}' \text{ is obtained from } \bar{Y} \text{ by renaming according to } N\text{)} \\ \pi[\bar{Y}](r \cup s) & \equiv & \pi[\bar{Y}](r) \cup \pi[\bar{Y}](s) \end{array}$$

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

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

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

- complex interactions between reusing subexpressions and pushing selection/projection

Application Order of Joins

Minimize intermediate results (and number of comparisons):

```
SELECT organization.name, country.name
FROM organization, country, isMember
WHERE organization.abbreviation = isMember.organization
  AND country.code = isMember.country
```

Exploit selectivity of join:

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

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

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

Example/Exercise

Consider the equivalent (to the previous example) query:

```
SELECT organization.name, country.name
FROM organization, country
WHERE EXISTS
  (SELECT *
   FROM isMember
   WHERE organization.abbreviation = isMember.organization
     AND country.code = isMember.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.

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 3.18

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

  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)

Example 3.18 (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.