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 $\rightsquigarrow$ 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 $A_1, \ldots, A_n$ (…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 $\rightarrow$ 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 $alias_1$.$attr_1$,$alias_2$.$attr_2$
  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 1990s.)

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 $R(\bar{X}) \setminus S(\bar{X})$:

- 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.

Syntactical Sugar: Join

- basic SQL syntax: list of relations in the FROM clause, cartesian product, conditions in the WHERE clause.
- explicit JOIN syntax in the FROM clause:

  `SELECT ...`
  `FROM` R_1 `NATURAL JOIN` R_2 `ON` $\mathit{join\text{-}cond}_{1,2}$ `[NATURAL JOIN` R_3 `ON` $\mathit{join\text{-}cond}_{1,2,3}$ `...]`
  `WHERE ...`
- usage of parentheses is optional,
- same translation to internal algebra.

Outer Join

- Syntax as above, as `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN` (and `FULL JOIN`, which is equivalent to `FULL OUTER JOIN`).
- usage of parentheses is optional, otherwise left-first application (!).
- can be translated to internal outer joins, much more efficient than handwritten outer join using UNION and NOT EXISTS.

Handling of Duplicates

In contrast to algebra relations, SQL tables may contain duplicates (cf. Slide 114):

- 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` *attr-list*: forms groups of tuples that have the same values for *attr-list*

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

:	:	:	:
Innsbruck	A	Tirol	118000
Vienna	A	Vienna	1761738
:	:	:	:
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): 2862618	MAX(population):1761738	COUNT(*):13
country: D	SUM(population):25333235	MAX(population):3292365	COUNT(*):85
:	:	:	:

- `HAVING` (condition on groups) AND `SELECT` use these values.

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

  $\Rightarrow$ 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

  $\Rightarrow$ 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).

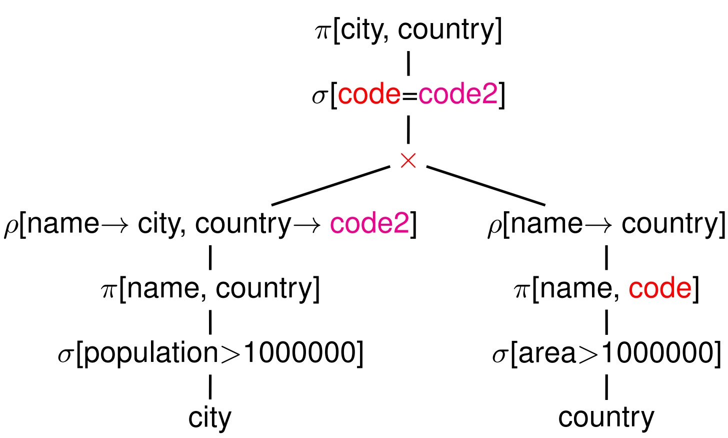
Note: the tuple variables can also be introduced in SQL explicitly as alias variables:

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 118): joined subtrees in the algebra

```
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;
```

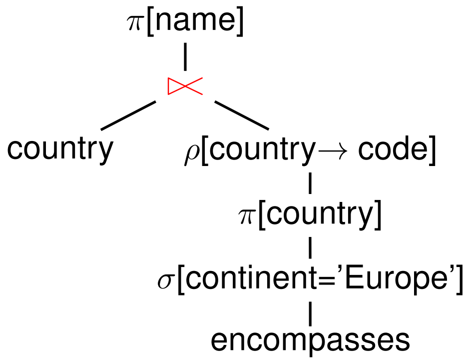


- the relation from evaluating the `from` clause has columns city, code2, country, code that can be used in the `where` clause and in the `select` clause.

Comparison: Subqueries in the WHERE clause

- WHERE ... IN uncorrelated-subquery (cf. Slide 120):
  Natural semijoin of outer tree with the subquery tree;

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

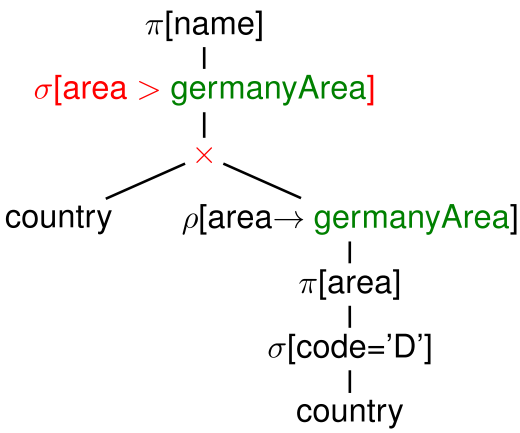


Note that the natural semijoin serves as an equi-selection where all tuples from the outer expression qualify that match an element of the result of the inner expression.

Comparison: Subqueries

- WHERE value op uncorrelated-subquery:
  (cf. Slide 120):
  join of outer expression with subquery, selection, projection to outer attributes

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

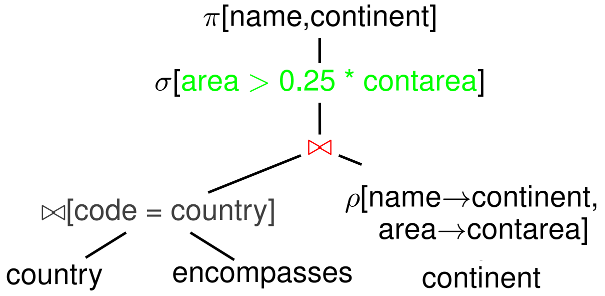


Note: the table that result table from the cartesian product has the format (name, code, area, population, . . . , germanyArea).

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);
```



- the internal (more operational) algebra evaluates the join and the condition as a semijoin that includes the condition.

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,
- so-called "SPJR-algebra" or "conjunctive queries", whose optimization plays an important role.

Comparison: the differences between Algebra and SQL

- The relational algebra has no notion of grouping and aggregate functions.
  Such operators can be defined as additional base operators (see Exercises)
- SQL has no clause that corresponds to relational division.
  Such queries must be constructed by the users, using the existing SQL constructs.

Example 3.16 (Relational Division in SQL)
Consider again Example 3.13 (Slide 100):

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

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

Exercise (DIV-1):

Use your commonsense logical reasoning to express this query in SQL.

Exercise (DIV-2): *Use the algebraic expression for $r \div s$ from Slide 99 for stating the query in SQL (use the SQL statement for orgOnCont from Slide 100):*

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

(try both now *before continuing with the video)*

Example 3.16 (Cont'd: Commonsense logical reasoning – Brain 1.0)

The relational division corresponds to the universal quantifier – "such that all ..." or "such that each of ...":

- *"... those organizations o such that that for each continent c, there is some country x such that x is a member of o and x is located on c. (here, x is a "witness")*
- *"... those organizations such that there is no continent c such that there is no country x such that x is a member of o and x is located on c.*
  (... no continent where no such witness can be found)

can be expressed by `not exists - not exists`*:*

```
select abbreviation
from organization o
where not exists
 (select *
  from continent c
  where not exists
   (select *
    from country x
    where (x.code, o.abbreviation) in (select country, organization from ismember)
      and (x.code, c.name)         in (select country, continent from encompasses)
   ))
```


Example 3.16 ((DIV-2) – inserting into $r \div s = \pi[\bar{Z}](r) \setminus \pi[\bar{Z}]((\pi[\bar{Z}](r) \times s) \setminus r)$ **)**

```
(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 that occur in $orgOnCont$ together with each of the $continent$ names",

or equivalently

"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 algebra-style set-theory-based.*

Oracle Query Plan Estimate: not-exists-not-exists: 339; copy-and-paste-solution: 707; minus-minus: 20; not-exists-minus: 341.

Example 3.16 (Cont'd)

Aside: logic-based querying with Datalog (see Lecture on "Deductive Databases") corresponding to the minus-minus solution:

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

```
% [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).
```

[Filename: Datalog/orgOnContsDiv.P]

... much shorter.

Algebra expression for it:

$\setminus$

π[abbrev](org)   *π[abbrev]*

$\setminus$

π[abbrev](org) $\times$ π[name](cont)   *ρ[org$\to$abbrev](π[org,cont](isMember $\bowtie$ encompasses))*

corresponds to the most efficient minus-minus solution.

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

Semantical/logical optimization: Consider integrity constraints in the database.

- constraint on table city: $population \geq 0$.
  Query plan for `select * from city where population < 0`:

Operation	object	predicate	cost
SELECT STATEMENT			0
_FILTER		NULL IS NOT NULL	
__TABLE ACCESS (FULL)	CITY	POPULATION < 0	7

- (foreign key references activated)
  `select * from ismember where country not in (select code from country)`:

Operation	object	predicate	cost
SELECT STATEMENT			0
_FILTER		NULL IS NOT NULL	
__TABLE ACCESS (FULL)	ISMEMBER		9

Semantical/logical optimization (Cont'd): Consider integrity constraints in the database.

- (foreign key references activated)

  `select country from ismember where country in (select code from country)`:

Operation	object	predicate	cost
SELECT STATEMENT			9
_TABLE ACCESS (FULL)	ISMEMBER		9

  No lookup of country.code at all (because guaranteed by foreign key)

- not always obvious
- general case: first-order theorem proving.

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:

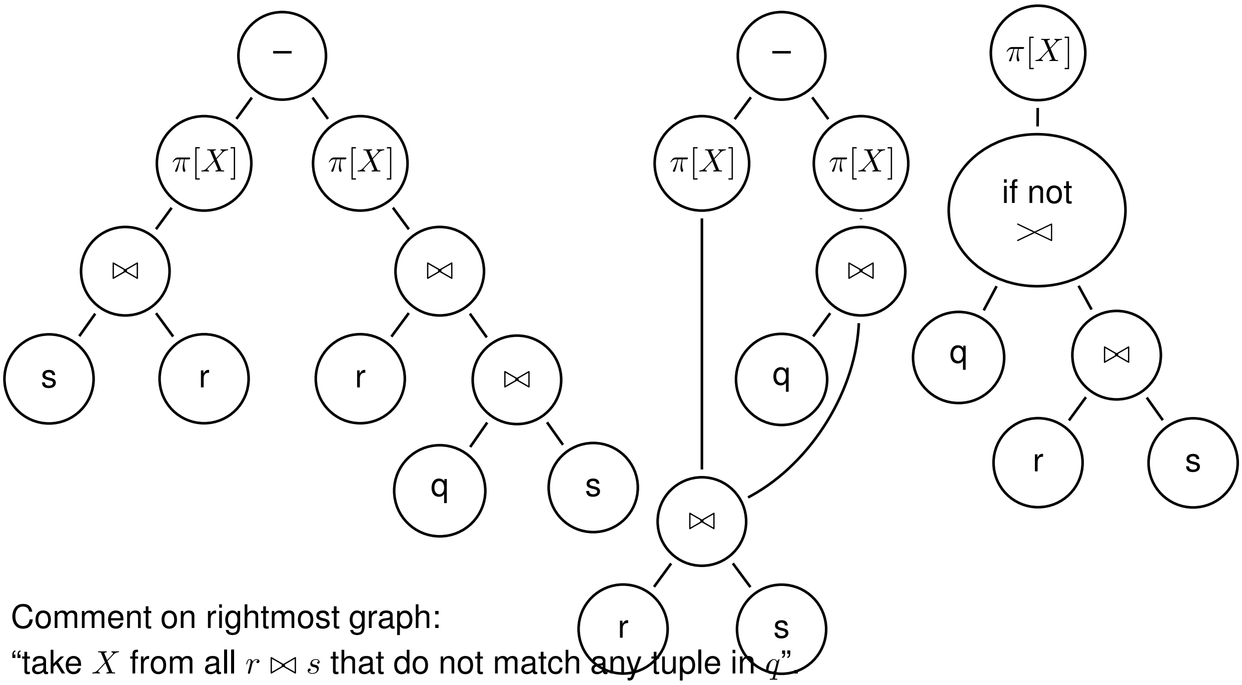
- reusing intermediate results
- equivalent restructuring of the operator tree
- "shortcuts" by melting several operators into one
  (e.g., join + equality predicate $\to$ equijoin)
- combination with actual situation: indexes, properties of data

Real-life databases implement this functionality.

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

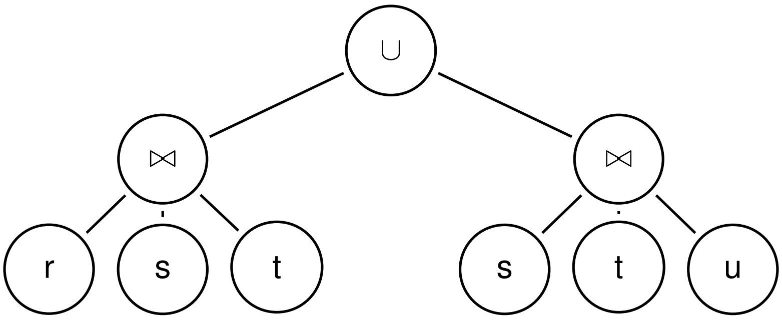
Reusing intermediate results

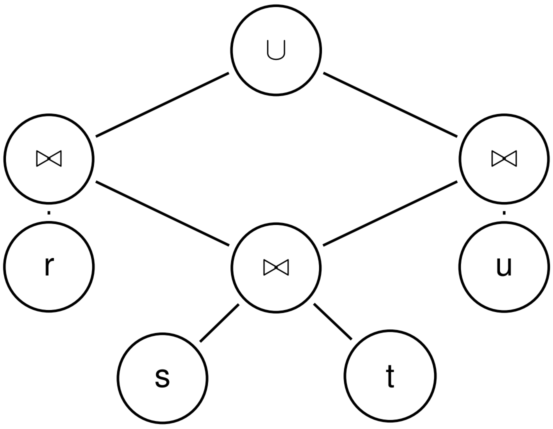
- Multiply occurring subtrees can be reused
  (directed acyclic graph (DAG) instead of algebra tree)



Comment on rightmost graph:
"take X from all $r \bowtie s$ that do not match any tuple in q".

Reusing intermediate results





for each tuple in $s \bowtie t$, computation can be forked, joining it with r and u and contributing to the union in parallel

OPTIMIZATION BY TREE RESTRUCTURING

- Equivalent transformation of the operator tree that represents an expression
- Based on the equivalences shown on Slide 109.
- 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 $r, s \in \text{Rel}(\bar{X})$, $\bar{Y} \subseteq \bar{X}$.

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

(condition: $cond$ does not use attributes from $\bar{X} - \bar{Y}$,

otherwise left term is undefined)

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

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

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

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

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

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

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

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

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 an area of more than 1,000,000 km^2, their capital has more than 1,000,000 inhabitants, and more than half of the inhabitants live in the capital. □

Example 3.17 (Cont'd) $\square$

π[name]

σ[countrypop $<$ 2·citypop]

$\bowtie$

- ρ[capital$\to$city,population$\to$countrypop]
  - π[name,code,capital,province,population]
    - σ[area $>$ 1000000]
      - country
- ρ[name$\to$city,population$\to$citypop]
  - π[name,province,country,population]
    - σ[population $>$ 1000000]
      - city

- 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 $\bar{X} \setminus Y$.

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

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

(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 "σ" vs. "π" are data-dependent
  Default: push σ lower.

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

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

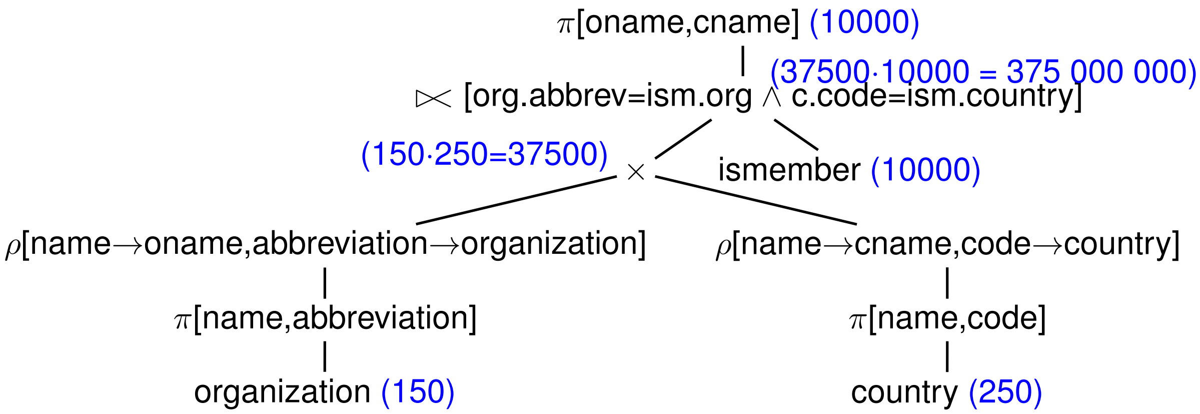
- complex interactions between reusing subexpressions and pushing selection/projection

Application Order of Joins

Consider the query:

```
SELECT organization.name as oname, country.name as cname
FROM organization, country
WHERE (abbreviation,code) IN (SELECT organization, country
                              FROM isMember)
```

- transforming into the relational algebra suggests a very costly evaluation:



- evaluation: semijoin uses an index (on the key of ismember) or nested-loop.

Application Order of Joins

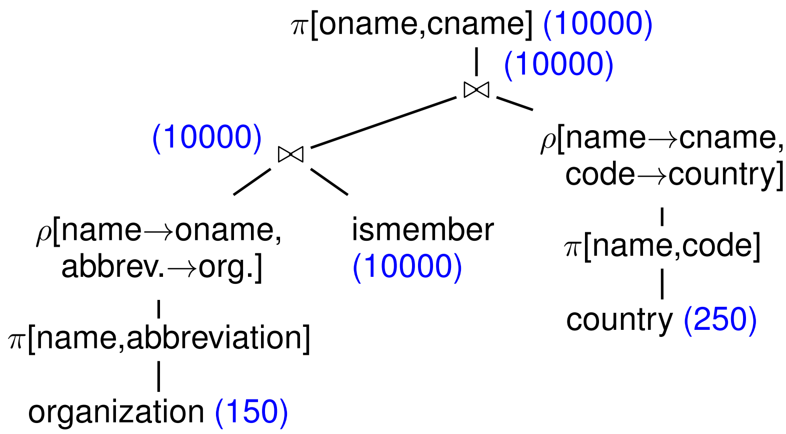
Minimize intermediate results (and number of comparisons):

... consider the equivalent query:

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

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

- loop over isMember
- extend each tuple with matching organization and country by using the indexes.
- Oracle query plan shows an extremely efficient evaluation of both of the above queries using indexes and ad-hoc views.



Aside: the real query plan

(see Slide 160 ff. for details)

Operation	Object	Pred(Index)	Pred(Filter)	COST	Rows
SELECT STATEMENT				13	9968
_HASH JOIN		C.CODE=ISM.COUNTRY		13	9968
__VIEW	$v2$			2	241
___HASH JOIN		ROWID=ROWID			
____INDEX (FULL SCAN)	COUNTRYKEY			1	241
____INDEX (FULL SCAN)	SYS_C0030486			1	241
__HASH JOIN		ORG.ABBREV=ISM.ORG		11	9968
___VIEW	$v1$			2	152
____HASH JOIN		ROWID=ROWID			
_____INDEX (FULL SCAN)	ORGKEY			1	152
_____INDEX (FULL SCAN)	ORGNAMEUNIQ			1	152
___SORT (UNIQUE)				9	9968
____INDEX (FULL SCAN)	MEMBERKEY			9	9968

No access to actual tables, ism(org,country) from key index, org(abbrev,name) from indexes via rowid-join, country(code,name) from indexes via rowid-join; both materialized as ad-hoc-views, combined by two hash-joins.

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

- σ[country = "D" $\wedge$ population $>$ 200000](City)*:*

  Assume an index that supports the condition country = "D"*.*

  - *without pipelining: compute* σ[country = "D"](City) *using the index, obtain* City'*. Then, compute* σ[population $>$ 200000](City')*.*
  - *with pipelining: compute* σ[country = "D"](City) *using the index, and check* ***on-the fly*** *each qualifying tuple against* σ[population $>$ 200000]*.*
  - *extreme case: when there is also an index on population (tree index, allows for range scan):*
    obtain set S_1 of all tuple-ids for german cities from index on code, obtain set S_2 of all tuple-ids of cities with more than 2 million inhabitants from population index, intersect S_1 and S_2 and access only the remaining cities. □

Pipelining

- **Unary** (i.e., selection and projection) operations can always be pipelined with the next lower binary operation (e.g., join)
- $\sigma[cond](R \bowtie S)$:
  - 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, indexes, and pipelining.