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)

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, ..., A_n        (... corresponds to π in the algebra)
   FROM R_1, ..., R_m       (... specifies the contributing relations)
   WHERE F                  (... corresponds to σ in the algebra)
```

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 ...
```

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
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'
```

Grouping and Aggregation

General Structure of SQL Queries

SELECT $A_1, \ldots, A_n$	list of attributes
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

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.

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

  $\Rightarrow$ 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).

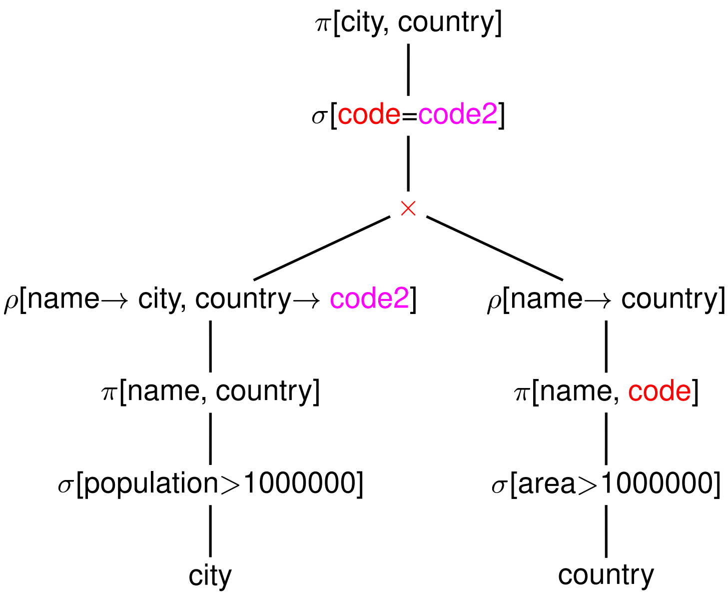
Note: the tuple variables can also be introduced in SQL explicitly as alias variables:

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 110): joined subtrees in the algebra

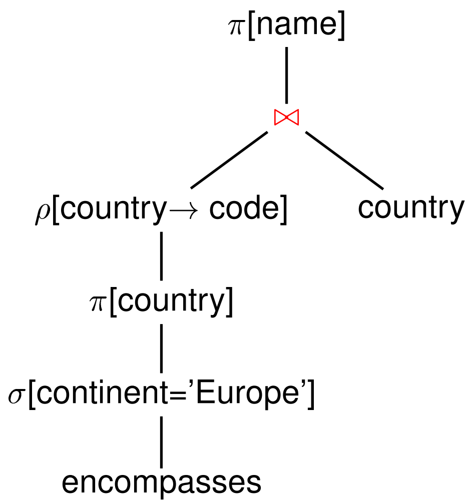
```
SELECT city, country.name
FROM (SELECT name AS city,
             country AS code2
      FROM city
      WHERE population > 1000000
     ),
     (SELECT name AS country, code
      FROM country
      WHERE area > 1000000
     )
WHERE code = code2;
```



Comparison: Subqueries in the WHERE clause

- WHERE ... IN uncorrelated-subquery (cf. Slide 112):
  Natural join of the subtree with the outer tree; possibly with renaming

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

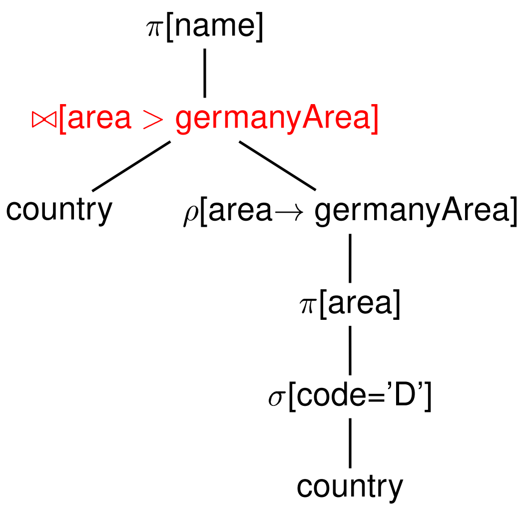


Note that the natural join serves as an equi-selection where all tuples from the outer expression qualify that match an element of the result of the inner expression.

Comparison: Subqueries

- WHERE value op uncorrelated-subquery:
  (cf. Slide 112):
  join of outer expression with subquery, selection, projection to outer attributes

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

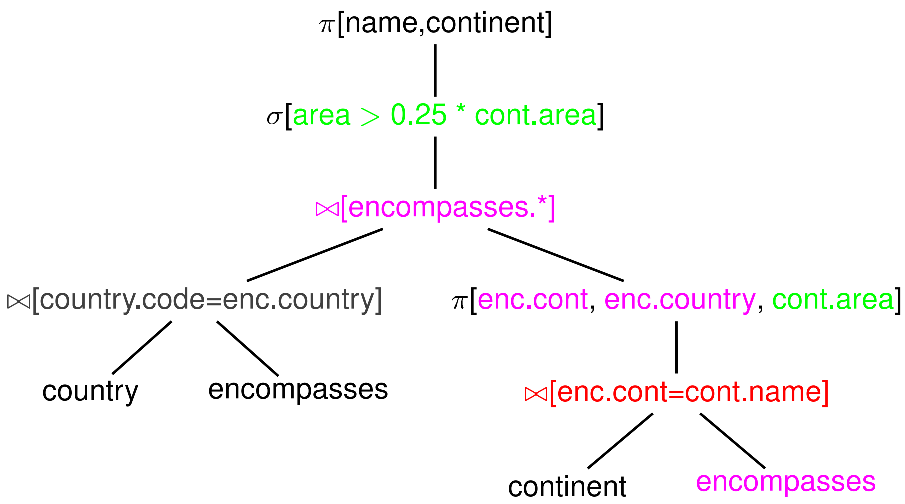


Note: the table that results from the join has the format (name, code, area, population, . . . , germanyArea).

Comparison: Correlated Subqueries

- WHERE value op correlated-subquery:
  - tree$_1$: outer expression
  - tree$_2$: subquery with “own” copies of all correlating relations; projection to keys of copied relations and comparison attributes
  - natural join of both trees over keys of correlating relations
  - correlating WHERE as additional condition

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



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 join
- 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 3.17
Consider again Example 3.10 (Slide 86).

The corresponding SQL formulation that implements division corresponds to the textual

"all countries that occur in $\pi[country](enc)$*, with the additional condition that they occur in* enc *together with each of the* $continent$ *values that occur in* cts*",*

or equivalent

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

Example 3.17 (Continued)

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

SELECT enc1.country

FROM enc enc1 — *consider enc1.country="R" and enc1.country="D"*

WHERE NOT EXISTS — *correlated subquery*

((SELECT ct — *always*

FROM cts)

"Europe"
"Asia"

MINUS

(SELECT ct

FROM enc enc2

WHERE enc1.country = 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

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.