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 $\mapsto$ Table
  - Tuple $\mapsto$ Row
  - Attribute $\mapsto$ 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 $A_1, \ldots, A_n$ (\ldots corresponds to $\pi$ in the algebra)
FROM $R_1, \ldots, R_m$ (\ldots specifies the contributing relations)
WHERE $F$ (\ldots corresponds to $\sigma$ 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**

```sql
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**:  
  ```sql
 SELECT alias1.attr1, alias2.attr2
 FROM table1 alias1, table2 alias2
 WHERE ...
  ```
- **Renaming of result columns of queries**:  
  ```sql
 SELECT attr1 AS name1, attr2 AS name2
 FROM ... WHERE ...
  ```
  (formal algebra equivalent: renaming)
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:

```sql
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:

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

```sql
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:

```sql
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:

```sql
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:

```sql
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

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

Often applied with renaming:

```sql
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,

\[
(\text{SELECT river, country, province FROM geo_river})
\quad \text{UNION}
\quad (\text{SELECT lake, country, province FROM geo_lake})
\]

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 \ldots ORDER BY 1,
- note that column numbers can only be used if there is no ambiguity with numeric values, e.g.,

\[
\text{SELECT name, 3 FROM country}
\]

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:

\[
\text{SELECT} \ldots \\
\text{FROM } R_1 \text{ NATURAL JOIN } R_2 \text{ ON } \text{join-cond}_{1,2} \ [\text{NATURAL JOIN } R_3 \text{ ON } \text{join-cond}_{1,2,3} \ldots ] \\
\text{WHERE} \ldots 
\]
- 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 \((O(n \log n))\)

\(\Rightarrow\) do not do it automatically

\(\Rightarrow\) 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

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

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

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

- Aggregation with DISTINCT:
  ```sql
  SELECT COUNT (DISTINCT country)
  FROM CITY
  WHERE population > 1000000;
  ```
### Altogether: Evaluation Strategy

**SELECT** [DISTINCT] \( A_1, \ldots, A_n \)  
**FROM** \( R_1, \ldots, R_m \)  
**WHERE** \( F \)  
**GROUP BY** \( B_1, \ldots, B_k \)  
**HAVING** \( G \)  
**ORDER BY** \( H \)

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
SQL:

\[
\text{SELECT } A_1, \ldots, A_n \text{ FROM } R_1, \ldots, R_m \text{ WHERE } F
\]

- equivalent expression in the relational algebra:

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

- Algorithm (nested-loop):
  
\[
\text{FOR each tuple } t_1 \text{ in relation } R_1 \text{ DO }
\]

\[
\text{FOR each tuple } t_2 \text{ in relation } R_2 \text{ DO }
\]

\[
\text{FOR each tuple } t_n \text{ in relation } R_n \text{ DO }
\]

\[
\text{IF tuples } t_1, \ldots, t_n \text{ satisfy the WHERE-clause THEN }
\]

\[
\text{evaluate the SELECT clause and generate the result tuple (projection).}
\]

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

\[
\text{SELECT } A_1, \ldots, A_n \text{ FROM } R_1 t_1, \ldots, R_m t_m \text{ WHERE } F
\]

(then optionally using \( t_i.\text{attr} \) in SELECT and WHERE)

Comparison: Subqueries

- Subqueries in the FROM-clause (cf. Slide 118): joined subtrees in the algebra

\[
\begin{align*}
\text{SELECT city, country} \\
\text{FROM (SELECT name AS city,} \\
\text{country AS code2} \\
\text{FROM city} \\
\text{WHERE population > 1000000} \\
\text{)}, \\
\text{(SELECT name AS country, code} \\
\text{FROM country} \\
\text{WHERE area > 1000000} \\
\text{)} \\
\text{WHERE code = code2;}
\end{align*}
\]

- the relation from evaluating the FROM clause has columns city, code2, country, code1 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;

  \[
  \pi_{\text{name}}
  \]

  \[
  \pi_{\text{[country]}}
  \]

  \[
  \sigma_{\text{continent='Europe'}}
  \]

  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

  \[
  \pi_{\text{name}}
  \]

  \[
  \pi_{\text{[area]}}
  \]

  \[
  \sigma_{\text{code='D'}}
  \]

  Note: the table that results from the join has the format (name, code, area, population, \ldots, 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

```sql
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: $\bowtie [enc.cont=cont.name \land area > 0.25 \ast cont.area]$

---

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 100):

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

\[
\text{orgOnCont} \div \pi[\text{name}](\text{continent}).
\]

Exercise: 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):

\[
r \div s = \pi[Z](r) \setminus \pi[Z]((\pi[Z](r) \times s) \setminus r).
\]
Example 3.16 (Cont’d – Solution to Exercise)

\[
\begin{align*}
&\text{(select org} \\
&\text{from (select distinct i.organization as org, e.continent as cont} \\
&\text{from ismember i, encompasses e} \\
&\text{where i.country = e.country ))} \\
&\text{minus} \\
&\text{( select o1} \\
&\text{from ((select o1,n1} \\
&\text{from (select org as o1} \\
&\text{from (select distinct i.organization as org, e.continent as cont} \\
&\text{from ismember i, encompasses e} \\
&\text{where i.country = e.country ))} \\
&\text{,} \\
&\text{(select name as n1 from continent) } \\
&\text{)} \\
&\text{minus} \\
&\text{(select distinct i.organization as org, e.continent as cont} \\
&\text{from ismember i, encompasses e} \\
&\text{where i.country = e.country )} \\
&\text{)}
\end{align*}
\]

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):

\[
\begin{align*}
\text{orgOnCont} \div \pi[\text{name}](\text{continent}) &= \\
\pi[\text{abbreviation}](\text{organization}) \setminus \\
\pi[\bar{Z}][\left(\pi[\text{abbreviation}](\text{organization}) \times \pi[\text{name}](\text{continent})\right) \setminus \text{orgOnCont}]
\end{align*}
\]

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

\[
\begin{align*}
&\text{(select abbreviation from organization)} \\
&\text{minus} \\
&\text{( select abbreviation} \\
&\text{from ((select o.abbreviation, c.name} \\
&\text{from organization o, continent c)} \\
&\text{minus} \\
&\text((select distinct i.organization as org, e.continent as cont} \\
&\text{from ismember i, encompasses e} \\
&\text{where i.country = e.country ) ) ) )}
\end{align*}
\]

... 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 \( \pi_{\text{name}}(\text{continent}) \) such that org
does not occur together with cont in orgOnCont”.

```sql
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)”

```sql
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))
```

Oracle Query Plan Estimate: copy-and-paste-solution: 568; minus-minus: 16;
not-exists-minus: 175; not-exists-not-exists: 295.
Example 3.16 (Cont’d)
Aside: logic-based querying with Datalog (see Lecture on “Deductive Databases”)

{o | organization(o, ...) \land \neg \exists \text{cont} : (continent(cont, ...) \land \neg \text{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:

\[
\begin{align*}
\pi[\text{abbrev}][\text{org}] \\
\pi[\text{abbrev}] \\
\pi[\text{abbrev}][\text{org}] \times \pi[\text{name}][\text{cont}] \\
\rho[\text{org} \to \text{abbrev}](\pi[\text{org}, \text{cont}](\text{isMember} \bowtie \text{encompasses}))
\end{align*}
\]

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: \text{SUM} (\text{SELECT} ...) \text{SUM} (\text{SELECT} \text{country}, \text{MAX} (\text{population}) \text{AS} \text{pop}_{-}\text{biggest})

\text{SELECT} \text{SUM} (\text{pop}_{-}\text{biggest})
\quad \text{FROM} (\text{SELECT} \text{country}, \text{MAX} (\text{population}) \text{AS} \text{pop}_{-}\text{biggest})
\quad \text{FROM} \text{City}
\quad \text{GROUP BY} \text{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: \( \text{population} \geq 0 \).

  Query plan for `select * from city where population < 0`:

<table>
<thead>
<tr>
<th>Operation</th>
<th>object</th>
<th>predicate</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT STATEMENT</td>
<td>0</td>
<td>NULL IS NOT NULL</td>
<td>0</td>
</tr>
<tr>
<td>_FILTER</td>
<td>NULL IS NOT NULL</td>
<td>CITY</td>
<td>POPULATION &lt; 0</td>
</tr>
</tbody>
</table>

  - (foreign key references activated)

  `select * from ismember where country not in (select code from country)`:

<table>
<thead>
<tr>
<th>Operation</th>
<th>object</th>
<th>predicate</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT STATEMENT</td>
<td>9</td>
<td>NULL IS NOT NULL</td>
<td>0</td>
</tr>
<tr>
<td>_FILTER</td>
<td>NULL IS NOT NULL</td>
<td>ISMEMBER</td>
<td></td>
</tr>
</tbody>
</table>

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.
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

---

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

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

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

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

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

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

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

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

(where \( \text{cond'} \) is obtained from \( \text{cond} \) 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.

---

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 \( \text{cond} = \text{cond}_X \wedge \text{cond}_Y \wedge \text{cond}_{XY} \) such that

- \( \text{cond}_X \) is concerned only with attributes in \( X \)
- \( \text{cond}_Y \) is concerned only with attributes in \( Y \)
- \( \text{cond}_{XY} \) is concerned both with attributes in \( X \) and in \( Y \).

Then,

\[ \sigma[\text{cond}](r \bowtie s) \equiv \sigma[\text{cond}_{XY}](\sigma[\text{cond}_X](r) \bowtie \sigma[\text{cond}_Y](s)) \]

**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. \( \blacksquare \)
Example 3.17 (Cont’d)

\[
\pi[\text{name}] \\
\sigma[\text{countrypop} < 2 \cdot \text{citypop}]
\]

\[
\rho[\text{capital} \rightarrow \text{city}, \text{population} \rightarrow \text{countrypop}] \\
\rho[\text{name} \rightarrow \text{city}, \text{population} \rightarrow \text{citypop}]
\]

\[
\pi[\text{name, code, capital, province, population}] \\
\pi[\text{name, province, country, population}]
\]

\[
\sigma[\text{area} > 1000000] \\
\sigma[\text{population} > 1000000]
\]

• Nevertheless, if \( \text{cond} \) is e.g. a complex mathematical calculation, it can be cheaper first to reduce the number of tuples by \( \cap, - \), or \( \bowtie \ \triangleright \)

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

Push Projections Down

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

Let \( \text{cond} = \text{cond}_{\bar{X}} \land \text{cond}_{\bar{Y}} \) such that

• \( \text{cond}_{\bar{Y}} \) is concerned only with attributes in \( \bar{Y} \)

• \( \text{cond}_{\bar{X}} \) is the remaining part of \( \text{cond} \) that is also concerned with attributes \( \bar{X} \setminus \bar{Y} \).

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

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

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

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

• Note that this does not hold for “\( \cap \)” and “\( - \)”!

• advantages of pushing “\( \sigma \)” vs. “\( \pi \)” are data-dependent

Default: push \( \sigma \) lower.

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

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

• complex interactions between reusing subexpressions and pushing selection/projection
Consider the query:

```sql
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:

```
π[oname,cname] (10000)
×
ρ[name→oname,abbreviation→organization]
π[name,abbreviation] organization (150)
ρ[name→cname,code→country]
π[name,code] country (250)
ismember (10000)
```

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

Minimize intermediate results (and number of comparisons):
... consider the equivalent query:

```sql
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)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Object</th>
<th>Pred(Index)</th>
<th>Pred(Filter)</th>
<th>COST</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>9968</td>
</tr>
<tr>
<td>_HASH JOIN</td>
<td>C.CODE=ISM.COUNTRY</td>
<td>13</td>
<td>9968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__VIEW</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>241</td>
</tr>
<tr>
<td>__HASH JOIN</td>
<td>ROWID=ROWID</td>
<td></td>
<td></td>
<td>1</td>
<td>241</td>
</tr>
<tr>
<td>__INDEX (FULL SCAN)</td>
<td>COUNTRYKEY</td>
<td></td>
<td></td>
<td>1</td>
<td>241</td>
</tr>
<tr>
<td>__INDEX (FULL SCAN)</td>
<td>SYS_C0030486</td>
<td></td>
<td></td>
<td>1</td>
<td>241</td>
</tr>
<tr>
<td>__HASH JOIN</td>
<td>ORG.ABBREV=ISM.ORG</td>
<td>11</td>
<td>9968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__VIEW</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>152</td>
</tr>
<tr>
<td>__HASH JOIN</td>
<td>ROWID=ROWID</td>
<td></td>
<td></td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>__INDEX (FULL SCAN)</td>
<td>ORGKEY</td>
<td></td>
<td></td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>__INDEX (FULL SCAN)</td>
<td>ORGNAMEUNIQ</td>
<td></td>
<td></td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>__SORT (UNIQUE)</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>9968</td>
</tr>
<tr>
<td>__INDEX (FULL SCAN)</td>
<td>MEMBERKEY</td>
<td></td>
<td></td>
<td>9</td>
<td>9968</td>
</tr>
</tbody>
</table>

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**

- \( \sigma[\text{country} = \text{"D"} \land \text{population} > 200000](\text{City}) \):  
  
  *Assume an index that supports the condition country = "D".*
  
  - *without pipelining*: compute \( \sigma[\text{country} = \text{"D"}](\text{City}) \) using the index, obtain City’. Then, compute \( \sigma[\text{population} > 200000](\text{City'}) \).
  
  - *with pipelining*: compute \( \sigma[\text{country} = \text{"D"}](\text{City}) \) using the index, and check **on-the-fly** each qualifying tuple against \( \sigma[\text{population} > 200000] \).
  
  - 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.*

\[\square\]
Pipelining

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

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

• $(R \bowtie S) \bowtie T$:
  
  – 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.