**Natural Join in SQL: The Direct Way**

How to express a *natural join* in SQL?

The **direct way** is to „translate“ the definition of a natural join based on projection, selection and product, which are the only operators covered by SELECT-FROM-WHERE anyway:

\[
R \bowtie S = \pi_{A_1, \ldots, A_m, R.B_1, \ldots, R.B_k, C_1, \ldots, C_n} (\sigma_{R.B_1 = S.B_1 \land \ldots \land R.B_k = S.B_k} (R \times S))
\]

**in SQL:**

```sql
SELECT A_1, \ldots, A_m, R.B_1, \ldots, R.B_k, C_1, \ldots, C_n
FROM R, S
WHERE R.B_1 = S.B_1 \land \ldots \land R.B_k = S.B_k
```
JOIN Operators in SQL

- Instead of "simulating" a join operator in this potentially quite tedious way, it is possible to explicitly use one of the variants of the JOIN operator in SQL.

- JOIN operators can only be used in the FROM part of a block in order to avoid a selection condition altogether (in case of a natural join) or to place it more closely to the operator (in case of an inner join, see next slide).

- in standard SQL: special operator for natural join

```sql
SELECT * FROM table1 NATURAL JOIN table2
```

or even:

```sql
SELECT * FROM ( SELECT * FROM table1 WHERE A > 0 ) NATURAL JOIN table2
```

abbreviation for 'all columns'

embedded query
**Joins in SQL (2)**

- Theta join (or inner join) with **explicit** join-condition (in an extra ON part):

\[
\begin{align*}
\text{SELECT} & \quad X.A \\
\text{FROM} & \quad ( \text{SELECT} * \\
& \quad \quad \text{FROM} \quad \text{table1} \\
& \quad \quad \text{WHERE} \quad A > 0 ) \quad \text{AS} \quad X \\
& \quad \quad \text{JOIN} \\
& \quad \quad \text{table2} \quad \text{AS} \quad Y \\
& \quad \quad \text{ON} \quad X.A = Y.A
\end{align*}
\]

- **In general:**

\[
\begin{align*}
\text{SELECT} & \quad \ldots \quad \text{FROM} \quad R \quad \text{JOIN} \quad S \quad \text{ON} \quad <\text{condition}> \\
\text{SELECT} & \quad \ldots \quad \text{FROM} \quad R, S \quad \text{WHERE} \quad <\text{condition}>
\end{align*}
\]
Other RA Operators in SQL

- SELECT-FROM-WHERE (SFW-) blocks are the basic units from which a complex SQL query is composed (representing projection, selection and product).

- More complex queries can be constructed by combining simpler queries by means of one of the three RA operators UNION, INTERSECT, or MINUS (called EXCEPT in the SQL standard).

- When using these operators, union compatibility of the operand expression has to be guaranteed.

- Example:

```
( (SELECT Name FROM cities )
MINUS
(SELECT Name FROM cities WHERE country = 'Germany' ) )
UNION
(SELECT Capital FROM countries)
```

Find all cities, which are not in Germany, or which are capitals!
(This includes Berlin!!)
Intersection Expressed as Join

- Some dialects (e.g. MS Access) do not support the INTERSECT operator.

- It is possible, however, to express an intersection as a special case of an (inner or natural) join where all columns of the two operand tables are identified.

- Example: Find all cities which are capitals as well!

\[
\begin{align*}
\text{(SELECT Name} & \text{FROM cities) INTERSECT (SELECT Capital AS Name FROM countries)} \\
\text{in Standard SQL} & \\
\text{renaming of columns (union compatibility!)} \\
\text{(SELECT Name} & \text{FROM cities) JOIN (SELECT Capital FROM countries) ON Name = Capital} \\
\text{e.g. in Access SQL}
\end{align*}
\]
Block Nesting and the IN Operator

- SFW blocks can be nested in various ways. We already saw an example where an embedded block is used instead of a table name in the FROM part:

```
SELECT Name, Inhabitants
FROM (SELECT Capital FROM countries) JOIN cities ON Name=Capital
WHERE Inhabitants > 1000.
```

- But blocks can be contained in the WHERE part as well, nested by using the IN operator (resembling the element operator ∈ in set theory):

```
SELECT Inhabitants, Name
FROM cities
WHERE Name IN (SELECT Capital FROM countries)
```

- Both formulations are equivalent, thus IN is just a shorthand notation for joins. However, the IN version more properly reflects that 'countries' does not contribute to the target list of the query but is accessed for test purposes only.
NOT IN and MINUS

- The element operator `IN` can also be used negatively, combined with the (otherwise logical) operator `NOT`. `NOT IN` represents the non-element operator $\notin$ in set theory:

```sql
SELECT Inhabitants, Name
FROM cities
WHERE Name NOT IN (SELECT Capital
FROM countries)
```

- This is not an abbreviation for a join! However, `NOT IN` is able to „simulate“ `MINUS`:

```sql
(SELECT Name FROM cities) MINUS
(SELECT Capital FROM countries)
```

```sql
SELECT Name
FROM cities
WHERE Name NOT IN (SELECT Capital
FROM countries)
```
Conditional Expressions: Overview

- There is a second large class of SQL-expressions: **Conditions**
  (or: conditional expressions)

- Conditions are Boolean expressions, which are either true or false.

- Conditions appear
  - as selection criteria in the WHERE-part of a SELECT-block and
  - as integrity constraints in CHECK-clauses (to be discussed later).

- There are two fundamental forms of conditions not otherwise expressible in SQL:
  - comparisons
  - existential conditions

- Complex conditions can be composed from simpler conditions by means of the Boolean operators AND, OR, NOT as in propositional logic.

- Various special forms of conditions can be equivalently expressed by means of the two basic types of conditions (comparisons and existential conditions) and thus are dispensable as far as pure expressive power is concerned.
Comparisons have been discussed in the context of the WHERE-part of an SQL-block on an earlier slide: Attribute values of a tuple can be compared with other attribute values or with constant values using one of the six comparison operators:

\[
\begin{array}{|c|c|}
\hline
= & <> \\
< & > \\
\leq & \geq \\
\hline
\end{array}
\]

Arguments in complex comparisons may be computed by means of a subquery (provided it can be guaranteed that the answer set contains one element only):

- **simple**: e.g. \(P\text{.age} = 30\) or \(P\text{.age} > Q\text{.age}\)
- **complex**: e.g. \(X\text{.age} > (\text{SELECT } Y\text{.age} \text{ FROM person } Y \text{ WHERE } Y\text{.name} = \text{'John'}\)\)

**Further special operators** in elementary comparisons in standard SQL:

- \(X\text{.name LIKE } \text{'Man%'\) (\%: „wildcard“)
  („pattern matching“ operator: not otherwise expressible)
- \(X\text{.age BETWEEN 40 AND 50\)
  (interval operator; alternatively expressible via '=<' and '=>')
Existential Conditions

- Existential conditions are used for checking whether the answer table of a sub-query is empty, or not:

\[
\text{SELECT Name} \\
\text{FROM city} \\
\text{WHERE EXISTS \ ( SELECT River } \\
\text{FROM city_at_river } \\
\text{WHERE City = Name )}
\]

Which cities are situated close to a river?

- Existential conditions can be negated as well: \( \text{NOT EXISTS} \)

- Positive existential conditions are expressible without an explicit quantifier by eliminating the nesting of SELECT-blocks:

\[
\text{SELECT Name} \\
\text{FROM city, city_at_river} \\
\text{WHERE City = Name}
\]

- This unnesting technique is not applicable for NOT EXISTS cases, however!
Existential Quantifier and Duplicates

- Avoiding an existential quantifier is potentially dangerous, as EXISTS is not treated in the same way as product construction in the FROM part by some commercial DBMS.

- In Access-SQL, e.g., an existential quantifier causes automatic elimination of duplicates from the answer to the enclosing SELECT expression. The standard (and the book of Date) interpret the semantics of EXISTS differently:

```
SELECT Name
FROM city, city_at_river
WHERE City = Name
```

![Result](image1.png)

```
SELECT Name
FROM city
WHERE EXISTS (SELECT River
               FROM city_at_river
               WHERE City = Name)
```

![Result](image2.png)
Duplicate Elimination in SQL

- SQL answer tables are no relations in the general case: They may be duplicate-free, but this is not guaranteed, even though all input tables of a query are free of duplicate rows.

- Fortunately, duplicates can be explicitly eliminated by using the keyword DISTINCT after SELECT:

  ```sql
  SELECT DISTINCT Name
  FROM city, city_at_river
  WHERE City = Name
  ```

- It is recommendable to always use SELECT DISTINCT as soon as a „real“ projection occurs, except if the SELECT part refers to a key column only.

- There is no convincing reason for working with duplicates in SQL!
Simulation of FORALL via NOT EXISTS

- SQL has no keyword for universal quantification (no 'FORALL'!).

- Universal conditions have to be „simulated“ by means of logical transformations using double negation and existential quantification based on the following law of predicate logic:

\[
\forall x: F \iff \neg \exists x: \neg F
\]

- **Example:** „Which river runs through every federal state in Germany?“

- In logic, e.g. in tuple relational calculus, this query can be formalized as follows:

\[
\{ x.Name \mid \text{river}(x) \land \forall y: (\text{state}(y) \Rightarrow \text{river_through_state}(x,y)) \}
\]

- If no „forall“ is available, as in SQL, application of the above mentioned law results in the following more complex formulation:

\[
\{ x.Name \mid \text{river}(x) \land \neg \exists y: \neg (\text{state}(y) \Rightarrow \text{river_through_state}(x,y)) \}
\]
Simulation of FORALL via NOT EXISTS (2)

- Applying two more transformation laws of propositional logic eliminates the implication and pushes the inner negation even more inward, thus resulting in a slightly more intuitive formalization:

$$\{x.\text{Name} \mid \text{river}(x) \land \neg \exists y: (\text{state}(y) \land \neg \text{river-through-state}(x,y))\}$$

- If this simple query is to be expressed in SQL, it is necessary to go exactly this way (involving quite a bit of logic) in order to be able to „simulate“ FORALL:

```sql
( SELECT X.Name
  FROM river AS X
  WHERE NOT EXISTS
    ( SELECT *
      FROM state AS Y
      WHERE NOT EXISTS
        ( SELECT *
          FROM river_through_state AS Z
          WHERE X.Name = Z.River AND
            Y.State = Z.State ) )
```


Aggregate functions are an important class of "built-in"-functions in SQL:

- **COUNT** - Cardinality
- **SUM** - Sum
- **AVG** - Average
- **MAX** - Maximum
- **MIN** - Minimum

Computation of **one** scalar value from a **set** of scalar values (the aggregate) originating from **one** column of **one** table:
Aggregate functions (2)

Examples of aggregate expressions in the SELECT-part:

Compute the overall salary of all C3-professors!

```sql
SELECT SUM ( P.salary ) AS Total
FROM professors AS P
WHERE P.Rank = 'C3'
```

Which C3-professors are older than all C4-professors?

```sql
SELECT P.Name
FROM professors AS P
WHERE P.Rank = 'C3' AND P.Age > ( SELECT MAX (Q.Age) 
                                   FROM professors AS Q 
                                   WHERE Q.Rank = 'C4' )
```
Aggregate functions (3)

- Often used in connection with aggregate functions: Extended SELECT-blocks with subdivision of the resulting tables into groups

- **Syntactic extension:** GROUP BY- and (possibly) HAVING-part in SELECT-blocks

- **Basic idea:** The result of the evaluation of SELECT-FROM-WHERE (a table) is divided into „subtables“ (groups) with identical values for certain **grouping columns** (specified in the GROUP BY-part)

- **optional:** Groups not satisfying a certain additional condition (HAVING-part), are eliminated.

- Aggregate functions are applied to groups (as aggregates), if GROUP BY has been specified, e.g.:

```
SELECT        P.Rank,   AVG(P.Age ) AS AvgAge
FROM           professors AS P
GROUP BY   P.Rank
HAVING        P. Rank > 'C2'
```
Aggregate functions (4)

Illustration with example data:

```
SELECT P.Rank, AVG(P.Age) AS AvgAge
FROM professors AS P
WHERE P.Name <> 'Ken'
GROUP BY P.Rank
HAVING P.Rank > 'C2'
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>C4</td>
<td>43</td>
</tr>
<tr>
<td>John</td>
<td>C3</td>
<td>33</td>
</tr>
<tr>
<td>Ken</td>
<td>C4</td>
<td>57</td>
</tr>
<tr>
<td>Lisa</td>
<td>C4</td>
<td>39</td>
</tr>
<tr>
<td>Tom</td>
<td>C2</td>
<td>32</td>
</tr>
<tr>
<td>Eva</td>
<td>C3</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>AvgAge</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>34.5</td>
</tr>
<tr>
<td>C4</td>
<td>41.0</td>
</tr>
</tbody>
</table>
Sorting tables in SQL

- Sorting of the result table can be specified at the end of a SELECT-block (after GROUP BY or HAVING, resp., if present at all)

- Example:

  ```sql
  SELECT X.Rank, X.Salary
  FROM professors AS X
  ORDER BY X.Rank DESC, X.Salary ASC
  ```

- „Direction“ of sorting: ASC (ascending, default value if unspecified)
  DESC (descending)

- The order of columns is always respected when sorting, thus introducing multiple sorting criteria.

- Sorting can be specified independent of aggregation.
Null values

- SQL offers a predefined, universal null value NULL, intended to represent unknown or missing information in a systematic way.

- Correct usage of NULL is difficult, partly because there are a number of inconsistent design decisions in the SQL standard.

- Null values can be interpreted in a number of different ways. Possible interpretations are:
  - Value exists, but is presently unknown.
  - It is known that in this row no value exists in the respective column.
  - It is not known if a value exists or if so, what it is like.

- Intended interpretation of NULL in SQL: *Value exists, but is unknown!*

- Thus: NULL is denoted a „value“! Each two occurrences of NULL represent different „real“ values presently (still) unknown.

- However: NULL itself doesn’t have a type but always takes the type of the resp. column under consideration.
Null values (2)

- NULL can, however, **not** be used like a „normal“ value in several cases, e.g.
  - NULL may **not** occur as a parameter of a function (e.g.: X+NULL)
  - NULL may **not** occur in comparisons (e.g.: X=NULL)

- For testing whether a column contains NULL a special syntax is offered:
  - \( X.<\text{column name}> \text{ IS NULL} \)
  - \( X.<\text{column name}> \text{ IS NOT NULL} \)

- If the evaluation of a subexpression returns NULL, then the entire expression returns NULL as a result, too, e.g.:

```
SELECT (65 - Age) AS Rest
FROM person
WHERE Name = 'Tom'
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>33</td>
</tr>
<tr>
<td>Tom</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Null values (3)

- **Exception**: Aggregate functions ignore NULL “on purpose”!

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>33</td>
</tr>
<tr>
<td>Tom</td>
<td>NULL</td>
</tr>
</tbody>
</table>

  - SUM (Age): 33
  - COUNT (Age): 1
  - AVG(Age): 33

- „Exception from the exception“: COUNT(*) does **not** ignore NULL tuples!

- In comparisons (and other conditions) NULL leads to usage of a **three-valued logic**, i.e. a logic with three rather than two truth values:
  - TRUE
  - FALSE
  - UNKNOWN

- **Example**: If A=3, B=4 and C=NULL, then . . .
  - A > B AND B > C results in FALSE
  - A > B OR B > C results in UNKNOWN
### Null values (4)

- Truth tables of Boolean operators in three-valued logic:

<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

- Respecting NULL when evaluating SELECT-expressions means:
  - All FROM-rows are eliminated for which the WHERE-part does **not** return TRUE.
  - **Consequently:** A row is eliminated as soon as the WHERE-part returns FALSE or UNKNOWN !!!
> Null values (5)

- SQL offers a special syntax for testing the truth value of a condition:

\[
<\text{conditional-expression}> \ IS \ [ \ NOT ] \ { \text{TRUE} \ | \ \text{UNKNOWN} \ | \ \text{FALSE} }\]

- **Semantics** of such IS-expressions: TRUE if and only if the evaluation of the left-hand expression returns the truth value on the right-hand side; FALSE else.

- **Consequence:** \( p \ IS \ NOT \ \text{TRUE} \) is no longer equivalent with \( \text{NOT} \ p \)!
  (if \( p \) is \( \text{UNKNOWN} \), then \( \text{NOT} \ p \) returns \( \text{UNKNOWN} \), too)

- **Further „logical trap“:**
  EXISTS doesn’t behave like an existential quantifier in three-valued logic
  EXISTS ( \( <\text{table-expression}> \) ) returns FALSE, if \( <\text{table expression}> \) results in an empty table, TRUE else -- but never UNKNOWN!

- Chapter 16 in Date’s SQL-book closes with the following (very brief) section:

16.6 A RECOMMENDATION
Avoid nulls.
Outer join

- Automatic generation of null values when using an OUTER JOIN-operator:
  \{ \text{LEFT} \mid \text{RIGHT} \mid \text{FULL} \} \ [ \text{OUTER} ] \ JOIN

- Semantics: „Normal“ join extended by rows filled up with NULLs, containing values which would otherwise not appear in a join.

- Example:

  \begin{array}{c|c}
  \text{p} & \text{A} & \text{B} \\
  \hline
  1 & 2 \\
  1 & 3 \\
  2 & 1 \\
  \end{array}

  \begin{array}{c|c}
  \text{q} & \text{B} & \text{C} \\
  \hline
  2 & 5 \\
  3 & 4 \\
  5 & 3 \\
  \end{array}

  SELECT * 
  FROM p FULL OUTER JOIN q ON p.B >= q.B

  \begin{array}{c|c|c|c}
  \text{A} & \text{p.B} & \text{q.B} & \text{C} \\
  \hline
  1 & 2 & 2 & 5 \\
  1 & 3 & 2 & 5 \\
  1 & 3 & 3 & 4 \\
  2 & 1 & \text{NULL} & \text{NULL} \\
  \text{NULL} & \text{NULL} & 5 & 3 \\
  \end{array}

Always contains INNER JOIN as subtable!
Outer join (2)

- **LEFT and RIGHT OUTER JOIN**: Only the "non-joining" elements of the left or right table, resp., are filled up with NULLs.

- **Example**:

<p>| p (left) |   |   |</p>
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| q (right) |   |   |</p>
<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

  Access-SQL: Only LEFT JOIN and RIGHT JOIN are supported, no FULL OUTER JOIN; "OUTER" is omitted.