Teil VI

The Relational Query Language SQL
The SFW Block in Detail
The Relational Query Language SQL

1. The SFW Block in Detail

2. Extensions of the SFW Block
The Relational Query Language SQL

1. The SFW Block in Detail
2. Extensions of the SFW Block
3. Recursion
Educational Objective for Today . . .

- Advanced knowledge of the relational SQL
Educational Objective for Today . . .

- Advanced knowledge of the relational SQL
- Knowledge of extensions of the SFW block
Educational Objective for Today . . .

- Advanced knowledge of the relational SQL
- Knowledge of extensions of the SFW block
- Understanding the formulation and evaluation of recursive queries
Structure of an SQL Query

```sql
-- query
select projection-list
from relations-list
[ where condition ]
```

**select**
- Projection list
- Arithmetic operations and aggregation functions

**from**
- Relations to use, optionally aliases (renamings)

**where**
- Selection and join conditions
- Nested queries (another SFW block)
Selection of Tables: The \textit{from} Clause

- Most simple form:
  - Each relation name may be followed by an optional tuple variable

```sql
select * 
from relations_list
```

- Example query:

```sql
select * 
from WINES
```
Cartesian Product

With more than one relation, the Cartesian product (a.k.a. cross product) is computed:

```
select *
from WINES, PRODUCER
```

All combinations are returned!
Tuple Variables for Repeated Access

- Using tuple variables, a relation can be accessed several times:

```sql
select *
from WINES w1, WINES w2
```

- Columns are then called:

```
w2.WineID, w2.Name, w2.Color, w2.Vintage, w2.Vineyard
```
Natural Join in SQL92

- Early versions of SQL
  - Standard that is usually implemented in current systems
  - Only know cross product, no explicit join operator
  - Join achieved with predicate after `where`

- Example for natural join:

```sql
select *
from WINES, PRODUCER
where WINES.Vineyard = PRODUCER.Vineyard
```
Joins as Explicit Operators: **natural join**

- Newer SQL versions
  - Know several explicit join operators
  - Can be seen as an abbreviation of the detailed query with cross product

```sql
select *
from WINES natural join PRODUCER
```
Joins as Explicit Operators: `join`

- Join with arbitrary predicate:

```sql
select *
from WINES join PRODUCER
  on WINES.Vineyard = PRODUCER.Vineyard
```

- Equi-joins on columns using the same name with `using`:

```sql
select *
from WINES join PRODUCER
  using (Vineyard)
```
Joins as Explicit Operators: cross join

- Cross product (a.k.a. Cartesian product)

```sql
select *
from WINES, PRODUCER
```

- As cross join

```sql
select *
from WINES cross join PRODUCER
```
Tuple Variable for Intermediate Results

- “Intermediate relations” from SQL operations or an SFW block can be named using tuple variables

```sql
select Result.Vineyard
from (WINES natural join PRODUCER) as Result
```

- For `from`, tuple variables are mandatory
- `as` is optional
The **select** Clause

- Determines projection attributes

```
select [distinct] projection-list
from ...
```

**projection-list** := `{attribute |
    arithmetic-expression |
    aggregation-function } [, ...]
```

- Attributes of the relation after the `from`, optionally with a prefix that specifies names of relations or names of tuple variables
- Arithmetic expressions over attributes of these relations, as well as constants
- Aggregation functions over attributes of these relations
The select Clause /2

- Special case of the projection list: *
  - Yields all attributes of the relation(s) from the from part

```
select *
from WINES
```
distinct Eliminates Duplicates

```
select Name from WINES
```

- Yields the result relation as a multi-set:

```
<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Zinfandel</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Riesling Reserve</td>
</tr>
<tr>
<td>Chardonnay</td>
</tr>
</tbody>
</table>
```
**distinct** Eliminates Duplicates /2

```
select distinct Name from WINES
```

- Yields projection from the relational algebra:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
<tr>
<td>Creek Shiraz</td>
</tr>
<tr>
<td>Zinfandel</td>
</tr>
<tr>
<td>Pinot Noir</td>
</tr>
<tr>
<td>Riesling Reserve</td>
</tr>
<tr>
<td>Chardonnay</td>
</tr>
</tbody>
</table>
Tuple Variables and Relation Names

- Query

```
select Name from WINES
```

- is equivalent to

```
select WINES.Name from WINES
```

- and

```
select W.Name from WEINE W
```
Prefixes for Unambiguousness

```sql
select Name, Vintage, Vineyard -- (wrong!)
from WINES natural join PRODUCER
```

- Attribute Vineyard exists in both tables, WINES and PRODUCER!
- Correct with prefix:

```sql
select Name, Vintage, PRODUCER.Vineyard
from WINES natural join PRODUCER
```
Prefixes for Unambiguousness /2

- When using tuple variables, the name of a tuple variable can be used to qualify an attribute:

```sql
select w1.Name, w2.Vineyard
from WINES w1, WINES w2
```
The *where* Clause

```
select ... from ...
where condition
```

- **Forms of the condition:**
  - Comparing an attribute with a constant:
    \[
    \text{attribute} \; \theta \; \text{constant}
    \]
    possible comparison symbols \( \theta \) depend on the domain; e.g., \( =, \langle\rangle, >, <, >= \) or \( <= \).
  - Comparison between two attributes with compatible domains:
    \[
    \text{attribute1} \; \theta \; \text{attribute2}
    \]
  - Logical *connectors* **or**, **and** and **not**
Join Condition

- *Join condition* has the form:

\[ \text{relation1.attribute} = \text{relation2.attribute} \]

- Example:

\[
\begin{align*}
\text{select} & \quad \text{Name, Vintage, PRODUCER.Vineyard} \\
\text{from} & \quad \text{WINES, PRODUCER} \\
\text{where} & \quad \text{WINES.Vineyard} = \text{PRODUCER.Vineyard}
\end{align*}
\]
Range Selection

- Range selection

\[ \text{attrib between } \text{constant}_1 \text{ and } \text{constant}_2 \]

is

abbreviation for

\[ \text{attrib } \geq \text{constant}_1 \text{ and } \text{attrib } \leq \text{constant}_2 \]

- Restricts attribute values to the closed interval \([\text{constant}_1, \text{constant}_2]\)

- Example:

\[
\text{select } * \text{ from WINES}
\text{where Vintage between 2000 and 2005}
\]
Imprecise Selection

- **Notation**

  \[attribute \textit{like} \textit{special-constant}\]

- Pattern matching in strings (search for multiple substrings)
- Special constant can contain the wildcard characters ‘%’ and ‘_’
  - ‘%’ stands for no character or an arbitrary string of characters
  - ‘_’ stands for exactly one character
Imprecise Selection /2

Example

```
select * from WINES
where Name like 'La Rose%'
```

is shorthand for

```
select * from WINES
where Name = 'La Rose'
    or Name = 'La RoseA'
    or Name = 'La RoseAA'
    ...
    or Name = 'La Rose Grand Cru'
    ...
    or Name = 'La Rose Grand Cru Classe'
    ...
    or Name = 'La RoseZZZZZZZZZZZZZ'
    ...
```
Set Operations

- Set operation require compatible domains for pairs of corresponding attributes:
  - Both domains are equal, or
  - both domains are based on character (irrespective of the length of the strings), or
  - both domains are numeric (irrespective of the exact types), such as integer or float.

- Result schema := schema of the “left” relation

```sql
select A, B, C from R1
union
select A, C, D from R2
```
Set Operations in SQL

- *Union*, *intersection* and *difference* as *union*, *intersect* and *except*
- Can be used orthogonally:

```sql
select *
from (select Vineyard from PRODUCER
   except select Vineyard from WINES)
```

equivalent to

```sql
select *
from PRODUCER except corresponding WINES
```
Set Operations in SQL /2

- Via **corresponding by** clause: specification of the list of attributes over which to perform the set operation

```sql
select *
from PRODUCER except corresponding by (Vineyard) WINES
```

- When using union: Default case is duplicate removal (**union distinct**); **without** duplicate removal when using **union all**
### Set Operations in SQL /3

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R union S</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R union all S</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R union corresponding S</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R union corresponding by (A) S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
Nesting Queries

- Necessary for comparing sets of values:
  - Standard comparisons in combination with the quantifiers \textbf{all} (\(\forall\)) or \textbf{any} (\(\exists\))
  - Special predicates for working with sets, \textbf{in} and \textbf{exists}
in Predicate and Nested Queries

- Notation:

\[ attribute \text{ in ( SFW-block )} \]

- Example:

\[
\text{select Name} \\
\text{from WINES} \\
\text{where Vineyard in (} \\
\quad \text{select Vineyard from PRODUCER} \\
\quad \text{where Region='Bordeaux')} \]

Sattler / Saake
Database Systems
Last Edited: October 2014 6–29
Evaluation of Nested Queries

1. Evaluation of the inner query regarding the vineyards from Bordeaux
2. Insertion of the results as a set of constants in the outer query after `in`
3. Evaluation of the modified query

```
select Name
from WINES
where Vineyard in ('Château La Rose', 'Château La Pointe')
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>La Rose Grand Cru</td>
</tr>
</tbody>
</table>
Evaluation of Nested Queries /2

- Internal evaluation: transformation into a join

```
select Name
from WINES natural join PRODUCER
where Region = 'Bordeaux'
```
Negation of the \textbf{in} Predicate

- Simulation of the difference operator

\[ \pi_{\text{Vineyard}}(\text{PRODUCER}) \setminus \pi_{\text{Vineyard}}(\text{WINES}) \]

using the SQL query

```sql
select Vineyard from PRODUCER
where Vineyard not in (
    select Vineyard from WINES)
```
# Expressiveness of the SQL Kernel

<table>
<thead>
<tr>
<th>Relational Algebra</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projection</td>
<td><code>select distinct</code></td>
</tr>
<tr>
<td>Selection</td>
<td><code>where</code> without nesting</td>
</tr>
<tr>
<td>Join</td>
<td><code>from, where</code></td>
</tr>
<tr>
<td></td>
<td><code>from with join</code> or <code>natural join</code></td>
</tr>
<tr>
<td>Renaming</td>
<td><code>from with tuple variable; as</code></td>
</tr>
<tr>
<td>Difference</td>
<td><code>where with nesting</code></td>
</tr>
<tr>
<td></td>
<td><code>except corresponding</code></td>
</tr>
<tr>
<td>Intersection</td>
<td><code>where with nesting</code></td>
</tr>
<tr>
<td></td>
<td><code>intersect corresponding</code></td>
</tr>
<tr>
<td>Union</td>
<td><code>union corresponding</code></td>
</tr>
</tbody>
</table>
Additional Notes on SQL

- Extensions of the SFW block
  - Further join operations inside the `from` clause (outer join),
  - Other kinds of conditions and conditions using quantifiers inside the `where` clause,
  - Application of scalar operations and aggregation functions inside the `select` clause,
  - Additional clauses `group by` and `having`

- Recursive queries
Scalar Expressions

- Renaming of columns: `expression as new-name`
- Scalar operations on
  - Numeric domains: for instance `+`, `−`, `∗` and `/`,
  - Strings: Operations such as `char_length` (current length of a string), concatenation `∥` and the `substring` operation (extract a substring starting at a certain position in the string),
  - Dates and time intervals: operations such as `current_date` (current date), `current_time` (current time), `+`, `−` and `∗`

Conditional expressions
Type conversion
Notes:
  - Scalar expressions can comprise multiple attributes
  - Application is performed tuple-wise: one output tuple is created for each input tuple
Scalar Expressions /2

- Return the names of all Grand-Cru wines

```sql
select substring(Name from 1 for
       (char_length(Name) - position('Grand Cru' in Name)))
from WINES where Name like '%Grand Cru'
```

- Assumption: additional attribute ProdDate in WINES

```sql
alter table WINES add column ProdDate date
update WINES set ProdDate = date '2004-08-13'
where Name = 'Zinfandel'
```

- Query:

```sql
select Name, year(current_date - ProdDate) as Age
from WINES
```
Conditional Expressions

- **case expression**: return a value depending on the Evaluation of predicate

```sql
case
  when predicate_1 then expression_1
  ...
  when predicate_{n-1} then expression_{n-1}
  [ else expression_n ]
end
```

- Use in **select**- and **where** clause

```sql
select case
  when Color = 'Red' then 'Red wine'
  when Color = 'White' then 'White wine'
  else 'Other'
end as WineType, Name from WINES
```
Type Conversion

- Explicit conversion of the types of expressions

\[
\text{cast}(\text{expression as typname})
\]

- Example: int values as strings for the concatenation operator

\[
\begin{align*}
\text{select } \text{cast} & (\text{Vintage as varchar}) \mid \mid ' ' \mid \mid \\
& \text{Name as Description} \\
\text{from WINES}
\end{align*}
\]
Quantifiers and Set Comparisons

- Quantifiers: all, any, some and exists
- Notation

\[
\text{attribute } \theta \{ \text{all} \mid \text{any} \mid \text{some} \} (\text{select attribute from ...}
\text{where ...})
\]

- **all**: where condition is fulfilled if for all tuples of the inner SFW block, the \( \theta \)-comparison with attribute evaluates to \text{true}
- **any** and **some**: where condition is fulfilled if the \( \theta \)-comparison evaluates to \text{true} for at least one tuple of the inner SFW block
Conditions with Quantifiers: Examples

- Determine the oldest wine

```sql
select *
from WINES
where Vintage <= all (
    select Vintage from WINES)
```

- All vineyards that produce red wines

```sql
select *
from PRODUCER
where Vineyard = any (
    select Vineyard from WINES
    where Color = 'Rot')
```
Comparison of Sets of Values

- Test for equality of two sets impossible with quantifiers alone
- Example: “Return all producers that produce both, red and white wines.”
- Wrong query

```sql
select Vineyard
from WINES
where Color = 'Red' and Color = 'White'
```

- Correct query

```sql
select w1.Vineyard
from WINES w1, WINES w2
where w1.Vineyard = w2.Vineyard
  and w1.Color = 'Red' and w2.Color = 'White'
```
The **exists/not exists** Predicate

- Simple form of nesting

\[ \text{exists} (\ SFW\text{-}block\ ) \]

- Yields **true** if the result of the inner query is **not** empty
- Especially useful for correlated subqueries (a.k.a. synchronized subqueries)
  - In the inner query, the relation names and tuple variable names from the `from` part of the outer query are used
Synchronized Subqueries

- Vineyards with 1999 red wine

```sql
select * from PRODUCER
where 1999 in ( 
    select Vintage from WINES
    where Color='Red' and WINES.Vineyard = PRODUCER.Vineyard)
```

- Conceptual evaluation
  1. Examination of the first PRODUCER tuple the outer query (Creek) and insertion into the inner query
  2. Evaluation of the inner query

```sql
select Vintage from WINES
where Color='Red' and WINES.Vineyard = 'Creek'
```

3. Continue at step 1. with second tuple ...

- Alternative: reformulation as join
Example for `exists`

- Vineyards from Bordeaux without known wines

```sql
select * from PRODUCER e
where Region = 'Bordeaux' and not exists (
    select * from WINES
    where Vineyard = e.Vineyard)
```
Aggregation Functions and Grouping

- Aggregation functions calculate new values for the whole column, such as the sum or the average of the values of a column.

- Example: Determination of the average price of articles or the total sales of all sold products.

- With additional grouping: calculation of functions per group, e.g., the average price per Product group or the total sales per customer.
Aggregation Functions

Aggregation functions in Standard-SQL:

- **count**: calculates the number of values in a column or alternatively (in a special case \texttt{count(*)}) the number of tuples of a relation
- **sum**: calculates the sum of all values in a column (only for numeric values)
- **avg**: calculates the arithmetic mean of the values of a column (only for numeric domains)
- **max** resp. **min**: calculate the biggest or smallest value of a column
Aggregation Functions /2

- Arguments of a aggregation function:
  - an attribute of the `from`-clause specified relation,
  - a valid scalar expression or,
  - in the clause of the `count`-function also the symbol `*`
Aggregation Functions /3

Before the argument (except of the case $\text{count}(\ast)$) optional also the keywords $\text{distinct}$ or $\text{all}$

- $\text{distinct}$: before application of aggregation functions, duplicate values are removed from the set of values on which the function is applied
- $\text{all}$: duplicates are used in calculations (default setting)
- null values are always eliminated before the function is applied (except of the case of $\text{count}(\ast)$)
Aggregation Functions – Examples

- **Number of wines**

```sql
select count(*) as Number
from WINES
```

<table>
<thead>
<tr>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Aggregation Functions – Examples /2

- Number of distinct wine regions:

```sql
select count(distinct Region)
from PRODUCER
```

- Wines that are older than the average:

```sql
select Name, Vintage
from WINES
where Vintage < (select avg(Vintage) from WINES)
```

- All producers that deliver exactly one wine:

```sql
select * from PRODUCER e
where 1 = (select count(*) from WINES w
            where w.Vineyard = e.Vineyard)
```
Aggregation Functions /2

- Nesting of aggregation functions is not allowed

```sql
select \( f_1(f_2(A)) \) as Result
from R ...  -- (Wrong!)
```

- Possible formalization:

```sql
select \( f_1(\text{Temp}) \) as Result
from ( select \( f_2(A) \) as Temp from R ... )
```
Aggregation Functions in \texttt{where} Clause

- Aggregation functions give only one value $\mapsto$ Application in Constants-Selections of the \texttt{where}-Clause possible
- All producers that deliver exactly one wine:

\begin{verbatim}
select * from PRODUCER e
where 1 = ( 
    select count(*) from WINES w 
    where w.Vineyard = e.Vineyard)
\end{verbatim}
group by and having

Notation

```
select ...
from ...
[where ...]
[group by attribute-list ]
[having condition ]
```
Grouping: Scheme

- Relation REL:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- Query:

```sql
select A, sum(D) from REL where ...
group by A, B
having A < 4 and sum(D) < 10 and max(C) = 4
```
**Grouping: Step 1**

- **from** and **where**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

...
### Grouping: Step 2

**group by** `A, B`

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
### Grouping: Step 3

**select** `A, sum(D)`

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>sum(D)</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

| Sattler / Saake Database Systems | Last Edited: October 2014 6–57 |
Grouping: Step 4

- having $A < 4$ and $\text{sum}(D) < 10$ and $\text{max}(C) = 4$

<table>
<thead>
<tr>
<th>A</th>
<th>sum(D)</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Next: A 9
Grouping - Example

- Number of red and white wines:

```sql
select Color, count(*) as Number
from WINES
group by Color
```

- Result relation:

<table>
<thead>
<tr>
<th>Color</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>5</td>
</tr>
<tr>
<td>white</td>
<td>2</td>
</tr>
</tbody>
</table>
having - Example

- Region with more than one wine

```sql
select Region, count(*) as Number
from PRODUCER natural join WINES
group by Region
having count(*) > 1
```
Attributes for Aggregation resp. `having`

- Valid attributes after `select` at grouping on relation with scheme $R$
  - Grouping attributes $G$
  - Aggregations on non-grouping attributes $R - G$
- Valid attributes for `having`
  - dito
Outer Joins

- Additionally to classic join (inner join): in SQL-92 also outer join;
  Adoption of “dangling tuples” into the result and completion with null values
- outer join takes all tuples of both operands (long version: full outer join)
- left outer join resp. right outer join takes all tuples of the left resp. right operand
- Outer natural join each with keyword natural, e.g. natural left outer join
## Outer Joins /2

<table>
<thead>
<tr>
<th>LEFT</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RIGHT</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NATURAL JOIN</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTER</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>⊥</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LEFT</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RIGHT</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>⊥</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

| Sattler / Saake | Database Systems | Last Edited: October 2014 | 6–63 |
Outer Join: Example

```
select Region, count(WineID) as Number
from PRODUCER natural left outer join WINES
group by Region
```

<table>
<thead>
<tr>
<th>Region</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barossa Valley</td>
<td>2</td>
</tr>
<tr>
<td>Napa Valley</td>
<td>3</td>
</tr>
<tr>
<td>Saint-Emilion</td>
<td>1</td>
</tr>
<tr>
<td>Pomerol</td>
<td>0</td>
</tr>
<tr>
<td>Rheingau</td>
<td>1</td>
</tr>
</tbody>
</table>
Simulation of the Outer Join

- Left outer join

```sql
select *
from PRODUCER natural join WINES
union all
select PRODUCER.*, cast(null as int),
cast(null as varchar(20)),
cast(null as varchar(10)), cast(null as int),
cast(null as varchar(20))
from PRODUCER e
where not exists (
    select *
    from WINES
    where WINES.Vineyard = e.Vineyard)
```
Sorting with `order by`

- Notation
  
  ```sql
  order by attribute-list
  ```

- Example:

  ```sql
  select *
  from WINES
  order by Vintage
  ```

- Sorting ascending (**asc**) or descending (**desc**)

- Sorting as last operation of a query ~~~ Sort attribute must be contained in the `select` clause
Sorting /2

- Sorting also with calculated attributes (aggregates) as sort criterion

```sql
select Vineyard, count(*) as Number
from PRODUCER natural join WINES
group by Vineyard
order by Number desc
```
Sorting: Top-k-Queries

- Query, that gives the best $k$ elements for a ranking function

```sql
select w1.Name, count(*) as Rank
from WINES w1, WINES w2
where w1.Vintage <= w2.Vintage -- Step 1
group by w1.Name, w1.WineID -- Step 2
having count(*) <= 4 -- Step 3
order by Rank -- Step 4
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zinfandel</td>
<td>1</td>
</tr>
<tr>
<td>Creek Shiraz</td>
<td>2</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>3</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>4</td>
</tr>
</tbody>
</table>
Sorting: Top-k-Queries

- Determination of the \( k = 4 \) youngest wines
- Explanation
  - Step 1: assignment of all wines that are older
  - Step 2: grouping by names, determination of the rank
  - Step 3: restriction to ranks \( \leq 4 \)
  - Step 4: sorting by rank
Handling of Null Values

- Scalar Expressions: Result null, when null value is used in calculation
- In all aggregation functions (except of count(*)) null values are removed before the function is applied
- Almost all comparisons with null values result in unknown (instead of true or false)
- Exception: is null gives true and is not null gives false
- Boolean expressions are then based on three-valued logic
## Handling of Null Values /2

<table>
<thead>
<tr>
<th>and</th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>or</th>
<th>true</th>
<th>unknown</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection of Null Values

- *Null-Selection* selects tuples that contain null values for a certain attribute.

**Notation**

attribute *is null*

**Example**

```sql
select * from PRODUCER
where Region *is null*
```
Known Queries

- Query expression that can be referenced multiple times in a query

**Notation**

\[
\text{with } \text{query-name} \ [(\text{column-list})] \ \text{as} \\
( \text{query-expression} )
\]

- Query without `with`

\[
\text{select } * \\
\text{from WINES} \\
\text{where Vintage - 2 } \geq ( \\
\quad \text{select avg(Vintage) from WINES}) \\
\text{and Vintage + 2 } \leq ( \\
\quad \text{select avg(Vintage) from WINES})
\]
Known Queries /2

- Query with `with`

```sql
with AGE(Average) as (  
    select avg(Vintage) from WINES)
select *  
from WINES, AGE  
where Vintage - 2 >= Average  
and Vintage + 2 <= Average
```
Recursive Queries

- Application: *Bill of Material*-Queries, Calculation of the transitive closure (flight connection etc.)

- Example:

<table>
<thead>
<tr>
<th>BUSLINE</th>
<th>Departure</th>
<th>Arrival</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Penrice</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Nuriootpa</td>
<td>Tanunda</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Tanunda</td>
<td>Seppeltsfield</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Tanunda</td>
<td>Bethany</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Bethany</td>
<td>Lyndoch</td>
<td>14</td>
</tr>
</tbody>
</table>
Recursive Queries /2

- Bus trips with max. two transfers

```sql
select Departure, Arrival
from BUSLINE
where Departure = 'Nuriootpa'
union
from BUSLINE B1, BUSLINE B2
union
select B1.Departure, B3.Arrival
from BUSLINE B1, BUSLINE B2, BUSLINE B3
```
Recursion in SQL:2003

- Formulation via extended with recursive-query
- Notation

```sql
with recursive recursive-table as (  
    query-expression -- recursive part
)  
[traversal-clause] [cycle-clause]  
query-expression -- non-recursive part
```

- Non-recursive part: query of recursion table
Recursion in SQL:2003 /2

- Recursive part:

  -- Initialization
  select ... 
  from table where ...
  -- Recursion step
  union all 
  select ...
  from table, recursion table
  where recursion condition
Recursion in SQL:2003: Example

with recursive TOUR(Departure, Arrival) as (  
   select Departure, Arrival  
   from BUSLINE  
   where Departure = 'Nuriootpa'  
   union all  
   select T.Departure, B.Arrival  
   from TOUR T, BUSLINE B  
   where T.Arrival = B.Departure)  
select distinct * from TOUR
### Step-Wise Composition of the Recursion Table TOUR

#### Initialization

<table>
<thead>
<tr>
<th>Departure</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuriootpa</td>
<td>Penrice</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Tanunda</td>
</tr>
</tbody>
</table>

#### Step 1

<table>
<thead>
<tr>
<th>Departure</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuriootpa</td>
<td>Penrice</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Tanunda</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Seppeltsfield</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Bethany</td>
</tr>
</tbody>
</table>

#### Step 2

<table>
<thead>
<tr>
<th>Departure</th>
<th>Arrival</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuriootpa</td>
<td>Penrice</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Tanunda</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Seppeltsfield</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Bethany</td>
</tr>
<tr>
<td>Nuriootpa</td>
<td>Lyndoch</td>
</tr>
</tbody>
</table>
Recursion: Example /2

- Arithmetic operations in the recursion step

```sql
with recursive TOUR(Departure, Arrival, Route) as (
    select Departure, Arrival, Distance as Route
    from BUSLINE
    where Departure = 'Nuriootpa'
    union all
    select T.Departure, B.Arrival, Route + Distance as Route
    from TOUR T, BUSLINE B
    where T.Arrival = B.Departure)
select distinct * from TOUR
```
Safety of Recursive Queries

- Safety (= finiteness of the calculation) is the most important requirement on a query language
- Problem: cycles in the recursion

```sql
insert into BUSLINE (Departure, Arrival, Distance)
values ('Lyndoch', 'Tanunda', 12)
```

- Handling in SQL
  - Limitation of the recursion depth
  - Cycle detection
Safety of Recursive Queries /2

- Restriction on the recursion depth

```sql
with recursive TOUR(Departure, Arrival, Transitions) as ( 
    select Departure, Arrival, 0 
    from BUSLINE 
    where Departure = 'Nuriootpa' 
    union all 
    select T.Departure, B.Arrival, Transitions + 1 
    from TOUR T, BUSLINE B 
    where T.Arrival = B.Departure and Transitions < 2) 
select distinct * from TOUR
```
Safety through Cycle Detection

- Cycle Clause
  - at detection of duplicates in the calculation path $\text{attrib: Cycle} = '*'$
    (Pseudo column of type char(1))
  - Guarantee the finiteness of the result “by hand”

```
cycle attrib set marke to '∗' default '−'
```
Safety through Cycle Detection

```sql
with recursive TOUR(Departure, Arrival, Way) as (
    select Departure, Arrival, Departure || '-' || Arrival as Way
    from BUSLINIE where Departure = 'Nuriootpa'
    union all
    select T.Departure, B.Arrival, Way || '-' || B. Arrival as Way
    from TOUR T, BUSLINIE B where T.Arrival = B.Departure)

select Arrival set Cycle to '*' default '-'

select Way, Cycle from TOUR
```

<table>
<thead>
<tr>
<th>Way</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuriootpa-Penrice</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Seppeltsfield</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Bethany</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Bethany-Lyndoch</td>
<td>-</td>
</tr>
<tr>
<td>Nuriootpa-Tanunda-Bethany-Lyndoch-Tanunda</td>
<td>*</td>
</tr>
</tbody>
</table>
SQL-Versions

- History
  - SEQUEL (1974, IBM Research Labs San Jose)
  - SEQUEL2 (1976, IBM Research Labs San Jose)
  - SQL (1982, IBM)
  - ANSI-SQL (SQL-86; 1986)
  - ISO-SQL (SQL-89; 1989; three Languages Level 1, Level 2, + IEF)
  - (ANSI / ISO) SQL2 (as SQL-92 adopted)
  - (ANSI / ISO) SQL3 (as SQL:1999 adopted)
  - (ANSI / ISO) SQL:2003 ... current SQL:2011

- Despite of standardization: partly incompatible among systems of certain producers
Summary

- SQL as standard language
- SQL-Core with reference to relational algebra
- Extensions: Grouping, Recursion etc.
Control Questions

- What are the options to formalize joins?
Control Questions

- What are the options to formalize joins?
- What do aggregations and grouping calculate?
Control Questions

- What are the options to formalize joins?
- What do aggregations and grouping calculate?
- Which operations can be used for the handling of null values?
Control Questions

- What are the options to formalize joins?
- What do aggregations and grouping calculate?
- Which operations can be used for the handling of null values?
- What is the purpose of recursive queries in SQL?