Part VI

The Relational Query Language SQL

The Relational Query Language SQL

The Relational Query Language SQL

The Relational Query Language SQL

The Relational Query Language SQL

The Relational Query Language SQL

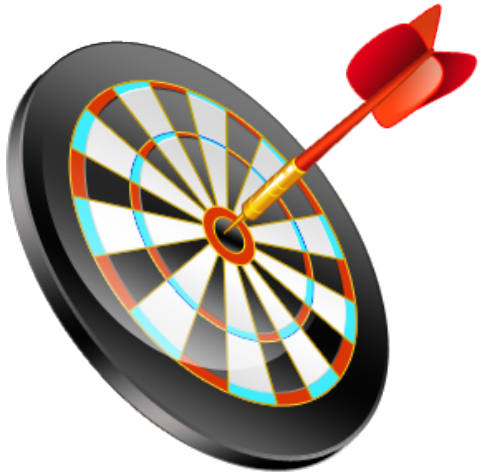
The Relational Query Language SQL

The Relational Query Language SQL

The Relational Query Language SQL

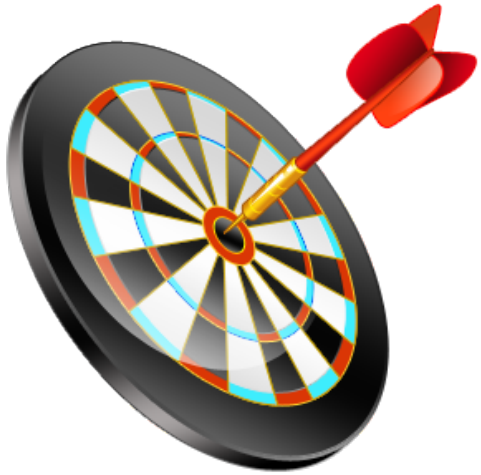
Educational Objective for Today . . .

- Advanced knowledge of the relational SQL



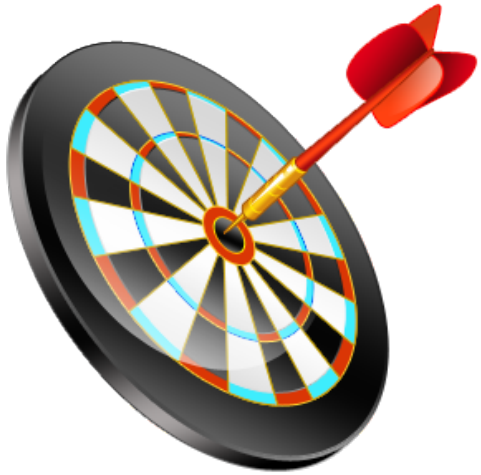
Educational Objective for Today ...

- 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



The SFW Block in Detail

Structure of an SQL Query

```
-- 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 `from` Clause

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

```
select *
from relations_list
```

- Example query:

```
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:

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

- Columns are then called:

```
w1.WineID, w1.Name, w1.Color, w1.Vintage, w1.Vineyard
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:

```
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

```
select *
from WINES natural join PRODUCER
```

Joins as Explicit Operators: `join`

- Join with arbitrary predicate:

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

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

```
select *
from WINES join PRODUCER
     using (Vineyard)
```

Joins as Explicit Operators: `cross join`

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

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

- As `cross join`

```
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

```
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 ...
```

- with

```
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:

Name
La Rose Grand Cru
Creek Shiraz
Zinfandel
Pinot Noir
Pinot Noir
Riesling Reserve
Chardonnay

`distinct` Eliminates Duplicates /2

```
select distinct Name from WINES
```

- Yields projection from the relational algebra:

Name
La Rose Grand Cru
Creek Shiraz
Zinfandel
Pinot Noir
Riesling Reserve
Chardonnay

Tuple Variables and Relation Names

- Query

```
select Name from WINES
```

- is equivalent to

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

- and

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

Prefixes for Unambiguousness

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

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

```
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:

```
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:

    $$\texttt{attribute}\ \theta\ \texttt{constant}$$

    possible comparison symbols θ depend on the domain; e.g., $=$, $<>$, $>$, $<$, $>=$ or $<=$.
  - Comparison between two attributes with compatible domains:

    $$\texttt{attribute1}\ \theta\ \texttt{attribute2}$$

  - Logical *connectors* **or**, **and** and **not**

Join Condition

- *Join condition* has the form:

```
relation1.attribute = relation2.attribute
```

- Example:

```
select Name, Vintage, PRODUCER.Vineyard
from WINES, PRODUCER
where WINES.Vineyard = PRODUCER.Vineyard
```

Range Selection

- *Range selection*

attrib **between** $constant_1$ **and** $constant_2$

is abbreviation for

attrib $\geq$ $constant_1$ **and**
attrib $\leq$ $constant_2$

- Restricts attribute values to the closed interval [$constant_1$, $constant_2$]
- Example:

```
select * from WINES
where Vintage between 2000 and 2005
```

Imprecise Selection

- Notation

```
attribute like 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 RoseB' or Name = 'La RoseBB' ...
      ...
      or Name = 'La Rose Grand Cru' ...
      or Name = 'La Rose Grand Cru Classe' ...
      ...
      or Name = 'La RoseZZZZZZZZZZZZ' ...
```

Set Operations

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

```
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:

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

equivalent to

```
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

```
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

R

A	B	C
1	2	3
2	3	4

S

A	C	D
2	3	4
2	4	5

R union S

A	B	C
1	2	3
2	3	4
2	4	5

R union all S

A	B	C
1	2	3
2	3	4
2	3	4
2	4	5

R union corresponding S

A	C
1	3
2	4
2	3

R union corresponding by (A) S

A
1
2

Nested Queries

Nesting Queries

- Necessary for comparing sets of values:
  - Standard comparisons in combination with the quantifiers **`all`** ($\forall$) or **`any`** ($\exists$)
  - Special predicates for working with sets, **`in`** and **`exists`**

`in` Predicate and Nested Queries

- Notation:

```
attribute in ( SFW-block )
```

- Example:

```
select Name
from WINES
where Vineyard in (
   select Vineyard from PRODUCER
   where Region='Bordeaux')
```

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')
```

Name
La Rose Grand Cru

Evaluation of Nested Queries /2

- Internal evaluation: transformation into a join

```
select Name
from WINES natural join PRODUCER
where Region = 'Bordeaux'
```

Negation of the `in` Predicate

- Simulation of the difference operator

$$\pi_{\texttt{Vineyard}}(\texttt{PRODUCER}) - \pi_{\texttt{Vineyard}}(\texttt{WINES})$$

using the SQL query

```
select Vineyard from PRODUCER
where Vineyard not in (
   select Vineyard from WINES )
```

Expressiveness of the SQL Kernel

Relational Algebra	**SQL**
Projection	**`select distinct`**
Selection	**`where`** without nesting
Join	**`from`**, **`where`** **`from`** with **`join`** or **`natural join`**
Renaming	**`from`** with tuple variable; **`as`**
Difference	**`where`** with nesting **`except corresponding`**
Intersection	**`where`** with nesting **`intersect corresponding`**
Union	**`union corresponding`**

Extensions of the SFW Block

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

```
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`

```
alter table WINES add column ProdDate date

update WINES set ProdDate = date '2004-08-13'
where Name = 'Zinfandel'
```

- Query:

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

Conditional Expressions

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

```
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

```
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

```
cast(expression as typname)
```

- Example: **int** values as strings for the concatenation operator

```
select cast(Vintage as varchar) || ' ' ||
       Name as Description
from WINES
```

Quantifiers and Set Comparisons

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

```
attribute θ { all | any | some } (
                                  select attribute
                                  from ...where ...)
```

- **all**: **where** condition is fulfilled if for all tuples of the inner SFW block, the θ-comparison with *attribute* evaluates to **true**
- **any** and **some**: **where** condition is fulfilled if the θ-comparison evaluates to **true** for at least one tuple of the inner SFW block

Conditions with Quantifiers: Examples

- Determine the oldest wine

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

- All vineyards that produce red wines

```
select *
from PRODUCER
where Vineyard = any (
    select Vineyard from WINES
    where Color = 'Red')
```

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

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

- Correct query

```
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

```
exists ( SFW-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

```
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

```
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

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

Aggregation and Grouping

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 **`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 **count**(∗)) optional also the keywords **distinct** or **all**
  - **distinct**: before application of aggregation functions, duplicate values are removed from the set of values on which the function is applied
  - **all**: duplicates are used in calculations (default setting)
  - null values are always eliminated before the function is applied (except of the case of **count**(∗))

Aggregation Functions – Examples

- Number of wines

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

results in

Number
7

Aggregation Functions – Examples /2

- Number of distinct wine regions:

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

- Wines that are older than the average:

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

- All producers that deliver exactly one wine:

```
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

```
select f1(f2(A)) as Result
from R ...   -- (Wrong!)
```

- Possible formalization:

```
select f1(Temp) as Result
from ( select f2(A) as Temp from R ...)
```

Aggregation Functions in `where` Clause

- Aggregation functions give only one value $\rightsquigarrow$ Application in Constants-""Selections of the **`where`**-""Clause possible
- All producers that deliver exactly one wine:

```
select * from PRODUCER e
where 1 = (
   select count(*) from WINES w
   where w.Vineyard = e.Vineyard)
```

`group by` and `having`

- Notation

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

Grouping: Scheme

- Relation REL:

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7
...			

- Query:

```
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`**

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7
...			

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7

Grouping: Step 2

- **group by** A, B

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7

A	B	N	
		C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7

Grouping: Step 3

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

A	B	N	
		C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7

A	**sum**(D)	N	
		C	D
1	9	3	4
		4	5
2	4	3	4
3	12	4	5
		6	7

Grouping: Step 4

- **having** A<4 **and** **sum**(D)<10 **and** **max**(C)=4

A	**sum**(D)	N	
		C	D
1	9	3	4
		4	5
2	4	3	4
3	12	4	5
		6	7

A	**sum**(D)
1	9

Grouping - Example

- Number of red and white wines:

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

- Result relation:

Color	Number
red	5
white	2

`having` - Example

- Region with more than one wine

```
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

Special Joins, Sorting, Null Values

Outer Joins

- Additionally to classic join (**`inner join`**): in SQL-92 also outer join
  $\rightsquigarrow$ 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

LEFT

A	B
1	2
2	3

RIGHT

B	C
3	4
4	5

NATURAL JOIN

A	B	C
2	3	4

OUTER

A	B	C
1	2	$\perp$
2	3	4
$\perp$	4	5

LEFT

A	B	C
1	2	$\perp$
2	3	4

RIGHT

A	B	C
2	3	4
$\perp$	4	5

Outer Join: Example

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

Region	Number
Barossa Valley	2
Napa Valley	3
Saint-Emilion	1
Pomerol	0
Rheingau	1

Simulation of the Outer Join

- Left outer join

```
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

```
order by attribute-list
```

- Example:

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

- Sorting ascending (**asc**) or descending (**desc**)
- Sorting as last operation of a query $\rightsquigarrow$ Sort attribute must be contained in the **select** clause

Sorting /2

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

```
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

```
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
```

Name	Rank
Zinfandel	1
Creek Shiraz	2
Chardonnay	3
Pinot Noir	4

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 ≤ 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

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

not	
true	false
unknown	unknown
false	true

Selection of Null Values

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

```
attribute is null
```

- Example

```
select * from PRODUCER
where Region is null
```

Recursion

Named Queries

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

```
with query-name [(column-list) ] as
( query-expression )
```

- Query without **with**

```
select *
from WINES
where Vintage - 2 >= (
      select avg(Vintage) from WINES)
   and Vintage + 2 <= (
      select avg(Vintage) from WINES)
```

Named Queries /2

- Query with **with**

```
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:

BUSLINE

Departure	Arrival	Distance
Nuriootpa	Penrice	7
Nuriootpa	Tanunda	7
Tanunda	Seppeltsfield	9
Tanunda	Bethany	4
Bethany	Lyndoch	14

Recursive Queries /2

- Bus trips with max. two transfers

```
select Departure, Arrival
from BUSLINE
where Departure = 'Nuriootpa'
   union
select B1.Departure, B2.Arrival
from BUSLINE B1, BUSLINE B2
where B1.Departure = 'Nuriootpa' and B1.Arrival = B2.Departure
   union
select B1.Departure, B3.Arrival
from BUSLINIE B1, BUSLINIE B2, BUSLINIE B3
where B1.Departure = 'Nuriootpa' and B1.Arrival = B2.Departure
and B2.Arrival = B3.Departure
```

Recursion in SQL:2003

- Formulation via extended **`with recursive`**-query
- Notation

```
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

Departure	Arrival
Nuriootpa	Penrice
Nuriootpa	Tanunda

Step 1

Departure	Arrival
Nuriootpa	Penrice
Nuriootpa	Tanunda
Nuriootpa	Seppeltsfield
Nuriootpa	Bethany

Step 2

Departure	Arrival
Nuriootpa	Penrice
Nuriootpa	Tanunda
Nuriootpa	Seppeltsfield
Nuriootpa	Bethany
Nuriootpa	Lyndoch

Recursion: Example /2

- Arithmetic operations in the recursion step

```
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

```
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

```
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 *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

```
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)
   cycle Arrival set Cycle to '*' default '-'
select Way, Cycle from TOUR
```

Way	Cyle
Nuriootpa-Penrice	-
Nuriootpa-Tanunda	-
Nuriootpa-Tanunda-Seppeltsfield	-
Nuriootpa-Tanunda-Bethany	-
Nuriootpa-Tanunda-Bethany-Lyndoch	-
Nuriootpa-Tanunda-Bethany-Lyndoch-Tanunda	*

History and Summary

SQL-Versions

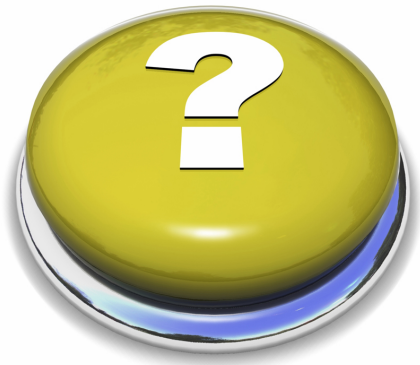
- History
  - SEQUEL (1974, IBM Research Labs San Jose)
  - SEQUEL2 (1976, IBM Research Labs San Jose)
  - SQL (1982, IBM)
  - ANSI-SQL (SQL-86; 1986)
  - ISO-SQL (SQL-89; 1989; three Languages Level 1, Level 2, + IEF)
  - (ANSI / ISO) SQL2 (as SQL-92 adopted)
  - (ANSI / ISO) SQL3 (as SQL:1999 adopted)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

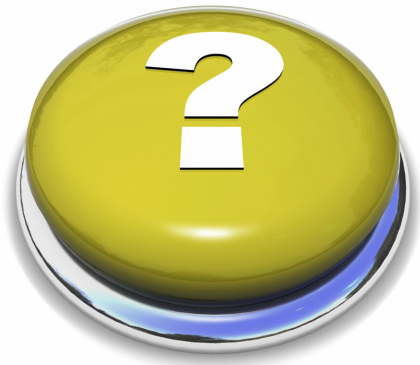
Control Questions

- What are the options to formalize joins?



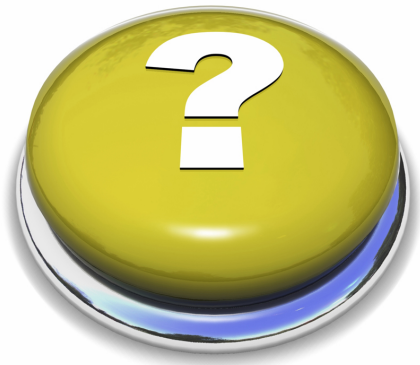
Control Questions

- 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?