6. Analytical SQL

Vlastislav Dohnal

Contents

- Recursive queries

- Ranking functions
- Windowing functions

- OLAP

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in the book (Database Systems Concepts)

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

Advanced Aggregate Functions

- General functions
  - min, max, count, sum, avg
  - array_agg(expression)
    - packs all input values into one array
- Statistical functions
  - stddev_samp(expression)
    - calculates the (sample) standard deviation over the values
  - var_samp(expression)
    - calculates the (sample) variance over the values
  - corr(a,b)
    - correlation coefficient between the two sets of values
  - regr_slope(y,x)
    - slope of the least-squares-fit linear function determined by the (x, y) pairs
  - regr_intercept(y, x)
    - y-intercept of the least-squares-fit linear equation determined by the (x, y) pairs

Advanced Aggregate Functions

- (Inverse) Distribution functions
  - mode() WITHIN GROUP (ORDER BY expression)
    - returns the most frequent input value
      - choosing the first one arbitrarily if there are multiple equally-frequent results
  - percentile_cont(fraction) WITHIN GROUP (ORDER BY expression)
    - *continuous* percentile: returns a value corresponding to the specified fraction in the ordering,
    - interpolating between adjacent input items if needed
  - percentile_disc(fraction) WITHIN GROUP (ORDER BY expression)
    - *discrete* percentile: returns the first input value whose position in the ordering equals or exceeds the specified fraction

fraction ∈ <0;1>

Advanced Aggregate Functions

- Hypothetical-set functions
  - rank(value) WITHIN GROUP (ORDER BY expr)
    - rank of the hypothetical value, with gaps for duplicate rows, over all values of expr.
  - dense_rank(value) WITHIN GROUP (ORDER BY expr)
    - rank of the hypothetical value, without gaps
  - percent_rank(value) WITHIN GROUP (ORDER BY expr)
    - relative rank of the hypothetical value, ranging from 0 to 1
  - cume_dist(value) WITHIN GROUP (ORDER BY expr)
    - relative rank of the hypothetical value, ranging from 1/N to 1

Analytic Functions

- provide the ability to perform calculations across sets of rows that are related to the current query row
- generally called *Window functions*
- <aggregate function>
  OVER ([PARTITION BY <column list>]
  ORDER BY <sort column list>
  [<aggregation grouping>])
- E.g.,
  SELECT … ,
  AVG(sales) OVER (PARTITION BY region
  ORDER BY month ASC ROWS 2 PRECEDING), …
  FROM …
  - moving/rolling average over 3 rows

Analytic Functions

- Ranking operators
  - Row numbering is the most basic ranking function
    - E.g.,

```
SELECT SalesOrderID , CustomerID ,
       ROW_NUMBER() OVER (ORDER BY SalesOrderID )
                    as RunningCount
FROM Sales WHERE SalesOrderID > 10000
ORDER BY SalesOrderID
```

SalesOrderID	CustomerID	RunningCount
43659	543	1
43660	234	2
43661	143	3
43662	213	4
43663	312	5

Analytic Functions

- ROW_NUMBER does not consider tied values
  - Each 2 equal values get 2 different row numbers

SalesOrderID	RunningCount
43659	1
43659	2
43660	3
43661	4

  - The behavior is nondeterministic
    - Each tied value could have its number switched!
- We need something deterministic
  - RANK() and DENSE_RANK()

Analytic Functions

- RANK and DENSE_RANK functions
  - Allow ranking items in a group
  - Syntax:
    - RANK ( ) OVER ([query_partition_clause] order_by_clause)
    - DENSE_RANK ( ) OVER ([query_partition_clause] order_by_clause)
  - DENSE_RANK
    - leaves no gaps in ranking sequence when there are ties
  - PERCENT_RANK ←→ (rank - 1) / (total rows - 1)
  - CUME_DIST - the cumulative distribution
    - the number of partition rows preceding (or peers with) the current row / total partition rows
    - The value ranges from 1/N to 1

Analytic Functions

- Example

```
SELECT channel, calendar,
  TO_CHAR(TRUNC(SUM(amount_sold), -6), '9,999,999’) AS sales,
  RANK() OVER (ORDER BY TRUNC(amount_sold, -6)) DESC) AS rank,
  DENSE_RANK() OVER (ORDER BY TRUNC(SUM(amount_sold), -6)) DESC) AS dense_rank
FROM sales, products
… GROUP BY channel, calendar ORDER BY sales DESC
```

CHANNEL	CALENDAR	SALES	RANK	DENSE_RANK
Direct sales	02.2015	10,000	1	1
Direct sales	03.2015	9,000	2	2
Internet	02.2015	6,000	3	3
Internet	03.2015	6,000	3	3
Partners	03.2015	4,000	5	4

Analytic Functions

- Group ranking - RANK function can operate within groups: the rank gets reset whenever the group changes
  - A single query can contain more than one ranking function, each partitioning the data into different groups.
  - PARTITION BY clause

```
SELECT … RANK() OVER (PARTITION BY channel ORDER BY SUM(amount_sold) DESC) AS rank_by_channel
```

CHANNEL	CALENDAR	SALES	RANK _BY_CHANNEL
Direct sales	02.2016	10,000	1
Direct sales	03.2016	9,000	2
Internet	02.2016	6,000	1
Internet	03.2016	6,000	1
Partners	03.2016	4,000	1

Analytic Functions

- NTILE splits a set into equal-sized groups
  - It divides an ordered partition into buckets and assigns a bucket number to each row in the partition
  - Buckets are calculated so that each bucket has exactly the same number of rows assigned to it or at most 1 row more than the others

```
SELECT … NTILE(3) OVER (ORDER BY sales) NT_3 FROM …
```

CHANNEL	CALENDAR	SALES	NT_3
Direct sales	02.2016	10,000	1
Direct sales	03.2016	9,000	1
Internet	02.2016	6,000	2
Internet	03.2016	6,000	2
Partners	03.2016	4,000	3

  - NTILE(4) - quartile
  - NTILE(100) - percentage

- Not a part of the SQL99 standard, but adopted by major vendors

More on Ranking…

- Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

```
select ID, (1 + (select count(*)
                 from student_grades B
                 where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

- More efficient solution with advanced SQL:

```
select ID, rank() over (order by GPA desc) as s_rank
from student_grades
```

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: "Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day"
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

select *date*, ***sum***(*value*) **over**
(**order by** *date* **between rows** 1 **preceding and** 1 **following**)
from *sales*

Windowing

- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value –10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row

Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal
  - "Find total balance of each account after each transaction on the account"

select *account_number, date_time*,
  sum (*value*) **over**
    (**partition by** *account_number*
    order by *date_time*
    rows unbounded preceding)
  as *balance*
from *transaction*
order by *account_number, date_time*

Windowing (Cont.)

- Obtain a value of a particular row of a *window frame* defined by window clause (PARTITION BY…)
  - first_value(expression)
  - last_value(expression)
  - nth_value (expression)

CHANNEL	CALENDAR	SALES	LOWEST_SALE
Direst sales	02.2016	10,000	4,000
Direst sales	03.2016	9,000	4,000
Internet	02.2016	6,000	4,000
Internet	03.2016	6,000	4,000
Partners	03.2016	4,000	4,000

SELECT … FIRST_VALUE(sales) OVER (ORDER BY sales) AS lowest_sale

SELECT … FIRST_VALUE(sales) OVER (PARTITION BY **channel** ORDER BY sales) AS lowest_sales

Windowing (Cont.)

- Access to a row that comes before the current row at a specified physical offset with the current window frame (partition)
  - LAG(expression [,offset [,default_value]])
- … after the current row
  - LEAD(expression [,offset [,default_value]]

CHANNEL	CALENDAR	SALES	PREV_SALE
Direst sales	02.2016	10,000	NULL
Direst sales	03.2016	9,000	10,000
Internet	02.2016	6,000	NULL
Internet	03.2016	6,000	6,000
Partners	03.2016	4,000	NULL

SELECT … LAG(sales, 1) OVER (PARTITION BY channel ORDER BY calendar) AS prev_sales

Data Aggregations

- Used in GROUP BY clause instead of mere list of attributes
- ROLLUP (e1, e2, e3, …)
  - represents the given list of expressions and all prefixes of the list including the empty list
- CUBE (e1, e2, e3, …)
  - represents the given list and all of its possible subsets (i.e., the power set)
- GROUPING SETS ((e1,e2), (e4,e5), (e6), () …)
  - rows are grouped separately by each specified grouping set

- Function to obtain which “GROUP BY” takes place
  - GROUPING(args...)
    - Integer bit mask indicating which arguments are not being included in the current grouping set

Data Aggregations

- Pivoting table for make and model over sales data
  - SELECT make, model, sum(amount) FROM sales GROUP BY CUBE (make, model)

	BMW	Mercedes	By model
SUV			
Sedan			
Sport			
By maker			

SUM

Data Aggregations

- Example of CUBE on table of car sales (year, make, model, amount)
  - GROUP BY CUBE (year, make, model) calculates:

Aggregate

Sum

Group By
(with total)

By model

SUV
SEDAN
SPORT

Sum

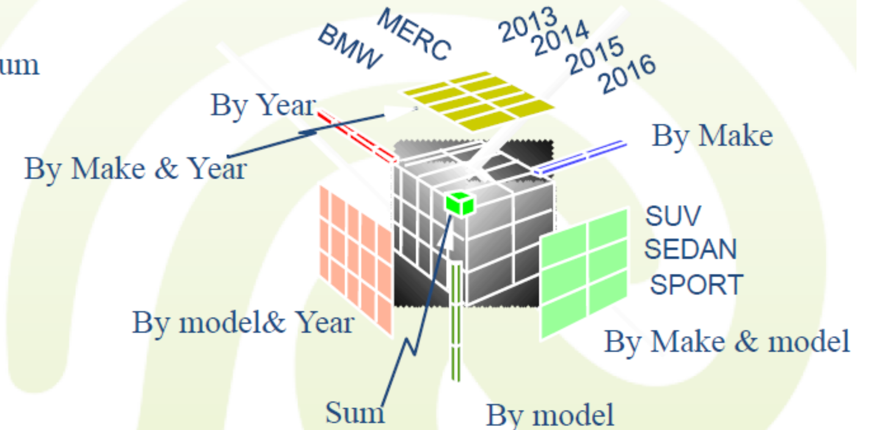
Cross Tab

BMW MERC By model

SUV
SEDAN
SPORT

By Make

Sum



Example sales relation

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6

Cross Tabulation of *sales* by *item_name* and *color*

clothes_size: **all**

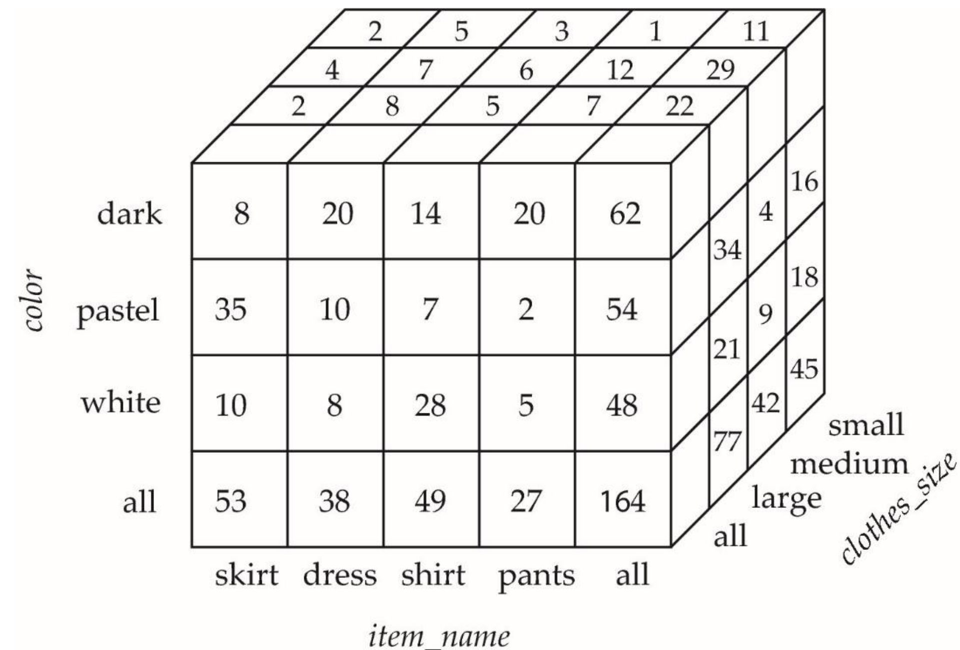
color (columns), *item_name* (rows)

	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

- The table above is an example of a **cross-tabulation** (**cross-tab**), also referred to as a **pivot-table**.
  - Values for one of the dimension attributes form the row headers
  - Values for another dimension attribute form the column headers
  - Other dimension attributes are listed on top
  - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab
- Can have *n* dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube



Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
  - Can drill down or roll up on a hierarchy

clothes_size: **all**

category	*item_name*	*color* dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
  - We use the value all is used to represent aggregates.
  - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.
    - The function **grouping()** can be applied on an attribute
      - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	White	all	28
shirt	all	all	49
pant	dark	all	20
pant	pastel	all	2
pant	white	all	5
pant	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164

Relational Representation of Cross-tabs (cont.)

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
  - E.g., replace *item_name* in the query by
    decode(**grouping**(item_*name*), 1, ‘all’, *item_name*)
    - By analogy for color and clothes_size

- In PostgreSQL, CASE WHEN … THEN … ELSE … END must be used.
- E.g., replace *item_name* in the query by
  case when grouping(item_*name*) = 1 **then** ‘all’ **else** *item_name* **end** **as** *item_name*

Extended Aggregation

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
- Example relation for this section
  sales(*item_name, color, clothes_size, quantity*)
- E.g., consider the query

  select *item_name, color, size,* **sum**(*number*)
  from *sales*
  group by cube(*item_name, color, size*)

  This computes the union of eight different groupings of the *sales* relation:

  { (*item_name, color, size*), (*item_name, color*),
  (*item_name, size*), (*color, size*),
  (*item_name*), (*color*),
  (*size*), ( ) }

  where ( ) denotes an empty **group by** list.
- For each grouping, the result contains the null value for attributes not present in the grouping.

Extended Aggregation (Cont.)

- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,

  select *item_name*, *color*, *size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name, color, size*)

  - Generates union of four groupings:

    { (*item_name, color, size*), (*item_name, color*), (*item_name*), ( ) }
- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory*(*item_name, category*) gives the category of each item. Then

  select *category, item_name*, **sum**(*number*)
  from *sales, itemcategory*
  where *sales.item_name = itemcategory.item_name*
  group by rollup(*category, item_name*)

  would give a hierarchical summary by *item_name* and by *category.*

Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

  select *item_name, color, size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name*), **rollup**(*color, size*)

  generates the groupings

  {*item_name, ()*} X {*(color, size), (color), ()*}

  = { (*item_name, color, size*), (*item_name, color*), (*item_name*), (*color, size*), (*color*), ( ) }

Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes, are called **multidimensional data**.
  - **Measure attributes**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - **Dimension attributes**
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., attributes *item_name, color,* and *size* of the *sales* relation

Types of analytical queries in OLAP

- **Pivoting**: changing the dimensions used in a cross-tab is called
- **Slicing**: creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup**: moving from finer-granularity data to a coarser granularity
- **Drill down**: The opposite operation - that of moving from coarser-granularity data to finer-granularity data

Takeaways

- Single SELECT command can inspect table rows “multiple” times
  - in recursive queries, windowing functions
  - but much faster than multiple specific SELECTs
- Statistical functions
  - variance, deviation, percentile, median
  - sliding statistics using windowing functions
- OLAP as a concept