SQL

Part 3: Where Subqueries and other Syntactic Sugar

Part 4: Unknown Values and NULLs

University of Waterloo

List of Slides

1
2 More on **"where"** conditions
3 Esoteric Predicates: Example
4 WHERE Subqueries
5 Overview of Where Subqueries
6 Example: **"attr IN (Q)"**
7 “Pure” SQL Equivalence
8 Example: **"attr NOT IN (Q)"**
9 **"attr NOT IN (Q)"** (cont.)
10 Example: **"attr op SOME/ALL (Q)"**
11 Parametric Subqueries
12 Example: **EXISTS**
13 Example: **NOT EXISTS**
14 Example: **IN**
15 More levels of Nesting
16 Example
17 Summary
18 Unknown values: **NULL**s
19 What can we do with **NULL**s?
20 Example
21 Counting NULLS
22 Outer Join
23 Example
24 Counting with OJ
25 Summary

More on `"where"` conditions

- Most queries are SELECT BLOCK queries.
- Early SQL developments tried to cram **everything** in the SELECT block
  1. Esoteric predicates (atomic)

     $\Rightarrow$ range searches (for ordered types)

     ```
     Name BETWEEN 'C' AND 'EZ'
     ```

     $\Rightarrow$ "almost" matches (for strings)

     ```
     Name LIKE '_A%'
     ```

     `"_"` matches one character

     `"%"` matches any number of characters
  2. Predicates defined by **subqueries**

     $\Rightarrow$ allow for *really tricky* formulation of queries.

many many others

$\Rightarrow$ usually DBMS/implementation dependent

Esoteric Predicates: Example

Find all book titles starting with “Logic” and published between 1980 and 2000:

```
SQL> select title
  2  from publication, book
  3  where publication.pubid=book.pubid
  4    and title like 'Logic%'
  5    and year between 1980 and 2000

TITLE
----------------------------------------
Logics for Databases and Information Systems
```

WHERE Subqueries

- Additional (complex) search conditions

  $\Rightarrow$ query-based search predicates

- Advantages

  $\Rightarrow$ simplifies writing queries with negation

- Drawbacks

  $\Rightarrow$ complicated semantics
  (especially when duplicates are involved)

  $\Rightarrow$ **very** easy to make mistakes

- **VERY COMMONLY** used to formulate queries

Overview of Where Subqueries

- presence/absence of a *single* value in a query

```
Attr IN ( Q )
Attr NOT IN ( Q )
```

- relationship of a value to some/all values in a query

```
Attr op SOME ( Q )
Attr op ALL ( Q )
```

- emptiness/non-emptiness of a query

```
EXISTS ( Q )
NOT EXISTS ( Q )
```

In the first two cases `Q` has to be unary.

Example: "attr IN (Q)"

```
SQL> select title
  2  from publication
  3  where pubid in (
  4      select pubid from article
  5  )

TITLE
----------------------------------------
Temporal Logic in Information Systems
Datalog with Integer Periodicity Constraints
Point-Based Temporal Extension of Temporal SQL
```

but we could have written

```
select title
from   publication, article
where  publication.pubid=article.pubid
```

as we know that pubid is a unique identifier of publications.

"Pure" SQL Equivalence

Nesting in the WHERE clause is mere syntactic sugar:

```
SELECT r.b
FROM   r
WHERE  r.a IN (
  SELECT b
  FROM s
)
```

```
SELECT r.b
FROM   r, (
  SELECT DISTINCT b
  FROM            s
        ) s
WHERE  r.a=s.b
```

Every of the remaining constructs can be rewritten in similar fashion. . .

Please think about this (as I may ask on the exam)

Example: "attr NOT IN (Q)"

Find all author-publication id's pairs for all publications except books and journals:

```
SQL> select *
  2  from wrote
  3  where publication not in (
  4      ( select pubid from book )
  5     union
  6      ( select pubid from journal )
  7  )

    AUTHOR PUBLICAT
---------- --------
         1 ChTo98
         1 ChTo98a
         1 Tom97
         2 ChTo98
         2 ChTo98a
```

. . . search conditions may contain complex queries,

"attr NOT IN (Q)" (cont.)

Find all author-publication id's pairs for all publications except books and journals (another formulation):

```
SQL> select *
  2  from wrote
  3  where publication not in (
  4        select pubid from book
  5  ) and publication not in (
  6        select pubid from journal
  7  )

    AUTHOR PUBLICAT
---------- --------
         1 ChTo98
         1 ChTo98a
         1 Tom97
         2 ChTo98
         2 ChTo98a
```

. . . and may be combined using boolean connectives.

Example: "attr op SOME/ALL (Q)"

Find article with most pages:

```
SQL> select pubid
  2  from article
  3  where endpage-startpage>=all (
  4        select endpage-startpage
  5        from   article
  6  )

PUBID
--------
ChTo98
```

. . . another way of saying **max**

attr = SOME (Q) is the same as **attr IN (Q)**

attr <> ALL (Q) is the same as **attr NOT IN (Q)**

Parametric Subqueries

- so far *subqueries* were **independent** on the *main* query

  $\Rightarrow$ not correlated
  $\Rightarrow$ not much fun (good only for simple queries)

- SQL allows **parametric** (correlated) subqueries:

  $\Rightarrow$ of the form `Q(p1,...pk)` where
  `pi`s are attributes in the main query.

  $\Rightarrow$ The truth of the predicate defined by the subquery is determined for each tuple in the main query by instantiating the parameters and then checking for the truth value as before...

Example: `EXISTS`

Parametric subqueries are most common for “existential” subqueries:

```
SQL> select *
  2  from wrote r
  3  where exists (
  4        select *
  5        from wrote s
  6        where r.publication=s.publication
  7          and r.author<>s.author
  8  )

    AUTHOR PUBLICAT
---------- --------
         1 ChTo98
         1 ChTo98a
         2 ChTo98
         2 ChTo98a
         2 ChSa98
         3 ChSa98

6 rows selected.
```

Example: `NOT EXISTS`

. . . and it is easier to complement conditions:

```
SQL> select *
  2  from wrote r
  3  where not exists (
  4        select *
  5        from wrote s
  6        where r.publication=s.publication
  7          and r.author<>s.author
  8  )

    AUTHOR PUBLICAT
---------- --------
         1 Tom97
```

Example: `IN`

. . . but works in the other cases too:

```
SQL> select *
  2  from wrote r
  3  where publication in (
  4        select publication
  5        from wrote s
  6        where r.author<>s.author
  7  )

    AUTHOR PUBLICAT
---------- --------
         1 ChTo98
         1 ChTo98a
         2 ChTo98
         2 ChTo98a
         2 ChSa98
         3 ChSa98

6 rows selected.
```

More levels of Nesting

- WHERE subqueries are **just queries**

  $\Rightarrow$ we can nest again and again and . . .
  $\Rightarrow$ every nested subquery can use attributes
  from the enclosing queries as parameters.
  $\Rightarrow$ correct naming is imperative

- used to formulate very complex **search conditions**

  $\Rightarrow$ attributes present **in the subquery only**
  CANNOT be used to construct the result(s).

In old SQL (SQL/89) only one level of nesting was allowed (and the nested subquery had to be a “simple select block”). You could (and had to) cheat by using views (as we have seen earlier.

Not a substitute for nesting in the FROM clause!

Example

List all authors who always publish with someone else:

```
SQL> select a1.name, a2.name
  2  from author a1, author a2
  3  where not exists (
  4     select *
  5     from   publication p, wrote w1
  6     where  p.pubid=w1.publication
  7       and  a1.aid=w1.author
  8       and  a2.aid not in (
  9              select author
 10              from   wrote
 11              where  publication=p.pubid
 12                and  author<>a1.aid
 13       )
 14  )

NAME                 NAME
-------------------- --------------------
Saake, Gunter        Chomicki, Jan
```

Summary

- WHERE subqueries support easy formulation of queries of the form

  "All x in R such that (a part of) x doesn't appear in S".

  $\Rightarrow$ subqueries stand for **conditions**

  CANNOT be used to produce results

  $\Rightarrow$ you can use input parameters

  but these must be *bound* in the main query

  $\Rightarrow$ easy to make mistakes though (be **very** careful)

- all of these are just a syntactic sugar and can be expressed using queries nested in the FROM clause

  $\Rightarrow$ but it might be quite hard. . .

the elimination of WHERE nesting only works if we have duplicate operations that work correctly.

Unknown values: `NULL`s

- a tool to cope with "unknown" values

  $\Rightarrow$ e.g., we don't know a URL (or a phone #)
  $\Rightarrow$ can be avoided by careful schema design
  $\Rightarrow$ **very** common in practice

- some attributes are **guaranteed** not to contain a **`NULL`** value (usually identifiers of entities)

  $\Rightarrow$ in our database: **`pubid`**, **`aid`**, **`...`**

- but if **`NULL`**s are allowed in a attribute, what does it mean for queries?

What can we do with `NULL`s?

- in selection conditions
  - ⇒ general rule: a **`NULL`** as a parameter to an atomic predicate makes the result **unknown**
  - ⇒ based on three-valued logic
  - ⇒ a special **`attr IS [NOT] NULL`** predicate
- in expressions: more complicated,
  usually depends on the data type of the attribute:
  - ⇒ **||** assumes a **`NULL`** is an empty string
  - ⇒ **+** results in **`NULL`** (so **`NULL`** is not **`0`**)

Example

List all authors for which we don't know a URL of their home page:

```
SQL> select aid, name
  2  from author
  3  where url IS NULL

       AID NAME
---------- --------------------
         3 Saake, Gunter
```

Counting NULLS

How do NULLs interact with counting (and aggregates in general)?

- **`count(URL)`** counts only non-NULL URL's

  $\Rightarrow$ **`count(*)`** counts "rows"

```
db2 => select count(*) as ROWS, count(url) as URLS \
db2 (cont.) => from author

ROWS        URLS
----------- -----------
          3           2

  1 record(s) selected.
```

or number of URLs per author—always 0 (NULL) or 1 (not NULL)

```
db2 => select aid,count(url) from author group by aid

AID         2
----------- -----------
          1           1
          2           1
          3           0

  3 record(s) selected.
```

Outer Join

IDEA: to allow "NULL-padded" answers that "fail to satisfy" a conjunct in a conjunction

- extension of syntax for the **`FROM`** clause

  $\Rightarrow$ **`FROM`** R **`<j-type> JOIN`** S **`ON`** C

  $\Rightarrow$ the **`<j-type>`** is one of

  `FULL`, **`LEFT`**, **`RIGHT`**, or **`INNER`**

- semantics (for $R(x, y)$, $S(y, z)$, and $C = (r.y = s.y)$). The result is the union of:

  1. $\{(x, y, z) : R(x, y) \wedge S(y, z)\}$
  2. $\{(x, y, \text{NULL}) : R(x, y) \wedge \neg(\exists z.S(y, z))\}$
     for **`LEFT`** and **`FULL JOIN`**
  3. $\{(\text{NULL}, y, z) : S(y, z) \wedge \neg(\exists x.R(x, y))\}$
     for **`RIGHT`** and **`FULL JOIN`**

  $\Rightarrow$ syntactic sugar for **`UNION ALL`**

Example

```
db2 => select aid,publication \
db2 (cont.) => from author left join wrote \
db2 (cont.) =>                          on aid=author

AID         PUBLICATION
----------- -----------
          1 ChTo98
          1 ChTo98a
          1 Tom97
          2 ChTo98
          2 ChTo98a
          2 ChSa98
          3 ChSa98
          5 -

  8 record(s) selected.
```

Counting with OJ

For every author count the number of publications:

```
db2 => select aid, count(publication) as pubs \
db2 (cont.) => from author left join wrote \
db2 (cont.) =>                          on aid=author \
db2 (cont.) => group by aid

AID         PUBS
----------- -----------
          1           3
          2           3
          3           1
          5           0

  4 record(s) selected.
```

Summary

- **`NULL`**s are necessary evil

  $\Rightarrow$ used to account for (small) irregularities in data
  $\Rightarrow$ should be used sparingly

- can be **always** avoided

  $\Rightarrow$ however some of the solutions may be inefficient

- you can't escape **`NULL`**s in practice

  $\Rightarrow$ *easy fix* for blunders in schema design
  $\Rightarrow$ . . . also due to schema evolution, etc.