Lecture 4

SQL

Part 2: Set Operations and Aggregates

by

David Toman

University of Waterloo
Fall '05

List of Slides

Complex Queries in SQL

- so far we can write only $\exists, \wedge$ queries
  $\Rightarrow$ the SELECT BLOCK queries
  $\Rightarrow$ not sufficient to cover all RC queries
- remaining connectives:
  1. $\vee, \neg$: are expressed using **set operations**
     $\Rightarrow$ easy to enforce *safety requirements*
  2. $\forall$: use negation and $\exists$

Set Operations at Glance

Answers to *Select Block*s (and all SQL queries in general) are **relations** (sets of tuples).

$\Rightarrow$ we can apply **set operations** on them:

- set union: Q_1 **UNION** Q_2
  $\Rightarrow$ the set of tuples in Q_1 or Q_2.
  $\Rightarrow$ used to express "or".
- set difference: Q_1 **EXCEPT** Q_2
  $\Rightarrow$ the set of tuples in Q_1 but not in Q_2.
  $\Rightarrow$ used to express "and not".
- set intersection: Q_1 **INTERSECT** Q_2
  $\Rightarrow$ the set of tuples in both Q_1 and Q_2.
  $\Rightarrow$ used to express "and" (redundant, rarely used).

Q_1 and Q_2 must have **union-compatible** signatures:

$\Rightarrow$ same number and types of attributes

Q_1 and Q_2 are select blocks or set-operation queries; precedence and associativity defined as usual.

ORACLE 7 uses **MINUS** instead of **EXCEPT**.

Example: Union

List all publication ids for books or journals:

```
SQL> (select pubid from book)
  2  union
  3  (select pubid from journal);

PUBID
--------
ChSa98
JLP-3-98
```

Example: Set Difference

List all publication ids except those for articles:

```
SQL> (select pubid from publication)
  2  minus
  3  (select pubid from article);

PUBID
--------
ChSa98
DOOD97
JLP-3-98
```


Multisets and Duplicates

- SQL uses a **BAG** semantics rather than a **SET** semantics:
  ⇒ SQL tables are **multisets** of tuples
  ⇒ mainly for efficiency reasons
- this leads to additional (extra-logical) syntactic constructions:
  ⇒ a *duplicate elimination operator*
  in the SELECT BLOCK: **`SELECT DISTINCT ...`**
  ⇒ BAG operators
  ⇒ equivalents of *set operations*
  ⇒ but with multiset semantics.

Should be avoided like plague (MHO)!
But often more efficient execution...

Example

```
SQL> select r1.publication
  2  from wrote r1, wrote r2
  3  where r1.publication=r2.publication
  4    and r1.author<>r2.author;

PUBLICAT
--------
ChSa98
ChSa98
ChTo98
ChTo98
ChTo98a
ChTo98a
```

Note duplicate entries for publication id's
$\Rightarrow$ for publications with n authors we get $O(n^2)$ answers!

Bag Operations

- bag union **`UNION ALL`**
  $\Rightarrow$ additive union: bag containing all in Q_1 *and* Q_2.
- bag difference **`EXCEPT ALL`**
  $\Rightarrow$ subtractive difference (monus):
  $\Rightarrow$ a bag all tuples in Q_1 for which
  there is no "matching" tuple in Q_2.
- bag intersection **`INTERSECT ALL`**
  $\Rightarrow$ a bag of all tuples taking the
  maximal number either in Q_1 or in Q_2

Example

Foe every book and article list all authors:

```
SQL> ( select author
  2    from wrote, book
  3    where publication=pubid )
  4  union all
  5  ( select author
  6    from wrote, article
  7    where publication=pubid );

    AUTHOR
----------
         2
         3
         1
         2
         1
         2
         1
```

. . . a fragment of a more meaningful query (coming later).

What About Nesting of Queries?

Using the syntax (so far) we can use *SELECT Blocks* and *Set operations* inside (other) *Set operations*.

What is we need to use a **Set Operation** inside of a **SELECT Block**?

- use **distributive laws**
  $\Rightarrow (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$
  $\Rightarrow$ often **very** cumbersome
- nest set operation inside a select block.
  $\Rightarrow$ Views or extensions to the **`FROM`** clause.

Naming Queries and Views

Idea: Queries denote **relations**. We provide a **naming** mechanism that allows us to assign names to (results of) queries.

$\Rightarrow$ can be used later in place of base relations.

- Syntax:

```
CREATE VIEW foo [<opt-schema>] AS
            ( <query-goes-here> );
```

- Views are **permanently** added to the schema
  $\Rightarrow$ often used to define *External View* of the database
  $\Rightarrow$ you must have a permission to create them

Example

List all publication titles for books or journals:

```
SQL> create view bookorjournal as
  2    ( (select pubid from book)
  3      union
  4      (select pubid from journal)
  5    );

SQL> select title
  2  from publication, bookorjournal
  3  where publication.pubid=bookorjournal.pubid;

TITLE
-----------------------------------------
Logics for Databases and Information Systems
Journal of Logic Programming
```

FROM revisited

- using the view mechanism is often too cumbersome:

  ⇒ ad-hoc querying, program-generated queries:
  big overhead due to catalog access
  you must remember to discard (**DROP**) the views
  ⇒ you need a **CREATE VIEW** privilege

- SQL/92 allows us to **inline** queries in the **FROM** clause:

  ```
  FROM ..., ( <query-here> ) <id>,...
  ```

  ⇒ **<id>** stands for the result of **<query-here>**.
  ⇒ unlike for base relations, **<id>** is **mandatory**.

- in old SQL (SQL/89) views were the only option. . .

Example

List all publication titles for books or journals:

```
SQL> select title
  2  from publication,
  3       ( (select pubid from book)
  4         union
  5         (select pubid from journal) ) bj
  6  where publication.pubid=bj.pubid;

TITLE
-----------------------------------------
Logics for Databases and Information Systems
Journal of Logic Programming
```

Can't we just use `OR` instead of `UNION`?

- A **common** mistake:

  ⇒ use of **`OR`** in the **`WHERE`** clause
  instead of the **`UNION`** operator

- An incorrect solution:

```
SELECT title
FROM   publication, book, journal
WHERE  publication.pubid=book.pubid
   OR  publication.pubid=journal.pubid
```

- often works; but imagine there are no **book**s. . .

Summary on First-Order SQL

- SQL introduced so far captures
  all of **Relational Calculus**

  ⇒ optionally with duplicate semantics
  ⇒ powerful (many queries can be expressed)
  ⇒ efficient (PTIME, LOGSPACE)

- Shortcomings:

  ⇒ some queries are hard to write (syntactic sugar)
  ⇒ no *"counting"* (aggregation)
  ⇒ no *"path in graph"* (recursion)

min and max can be expressed in RC; the formulation is nasty (go and try).

Aggregation

Standard and most useful extension of First-Order Queries.

- Aggregate (column) functions are introduced to
  $\Rightarrow$ find number of tuples in a relation
  $\Rightarrow$ add values of an attribute (over the whole relation)
  $\Rightarrow$ find minimal/maximal values of an attribute
- Can apply to *groups* of tuples that with equal values in (some) attributes
- Generally, can **NOT** be written in Relational Calculus

Operational Reading

1. partition the input relation to groups with equal values of **grouping** attributes
2. on each of these partitions apply the **aggregate function**
3. collect the results and form the answer

Aggregation (cont).

Formal Definition [Klug]:

$$\{ (x'_1, \ldots, x'_k, f_1, \ldots, f_l) : \\ f_i := \text{agg}_i\{ (x_1, \ldots, x_k, y_1, \ldots, y_n) : \\ Q((x_1, \ldots, x_k, y_1, \ldots, y_n) \\ \wedge x_1 = x'_1 \wedge \ldots \wedge x_k = x'_k\} \\ \wedge (\exists y_1, \ldots, y_n . Q)(x'_1, \ldots, x'_k) \quad \}$$

where

- $x'_1, \ldots, x'_k$ are the **grouping** attributes.
- agg_i are the **aggregate functions**
  $\Rightarrow$ e.g., **`count`**, **`sum`**, **`min`**, **`max`**, or **`avg`**.
- Q is the query on which *aggregation* is applied.

Aggregation (cont.)

The same in SQL syntax:

```
SELECT   x1,...,xk, agg1,...,aggl
FROM     Q
GROUP BY x1,...,xk
```

Restrictions:

- all attributes in the **`SELECT`** clause that are **NOT** in the scope of an aggregate function **MUST** appear in the **`GROUP BY`** clause.
- **`aggi`** are of the form **`count(y)`**, **`sum(y)`**, **`min(y)`**, **`max(y)`**, or **`avg(y)`** where **`y`** is an attribute of **`Q`** (usually not in the **`GROUP BY`** clause).

Example (count)

For each publication count the number of authors:

```
SQL> select publication, count(author)
  2  from wrote
  3  group by publication

PUBLICAT COUNT(AUTHOR)
-------- -------------
ChSa98               2
ChTo98               2
ChTo98a              2
Tom97                1
```


Example (sum)

For each author count the number of article pages:

```
SQL> select author,sum(endpage-startpage+1) as pages
  2  from wrote, article
  3  where publication=pubid
  4  group by author

    AUTHOR      PAGES
---------- ----------
         1         87
         2         68
```

. . . not quite correct: it doesn't list 0 pages for author 3.

`HAVING` clause

- the **`WHERE`** clause can't impose conditions on values of aggregates as the **`WHERE`** clause has to be used **before `GROUP BY`**
- SQL allows a **`HAVING`** clause instead

  $\Rightarrow$ like **`WHERE`**, but for aggregates. . .
- The aggregate functions used in the **`HAVING`** clause may be different from those in the **`SELECT`** clause; the grouping, however, is common.

Example

List publications with exactly one author:

```
SQL> select publication, count(author)
  2  from wrote
  3  group by publication
  4  having count(author)=1

PUBLICAT COUNT(AUTHOR)
-------- -------------
Tom97                1
```

This query *can* be written without aggregation as well.

Example (revisited.)

For every author count the number of books and articles:

```
SQL> select name, count(aid)
  2  from author, (
  3   ( select author
  4     from wrote, book
  5     where publication=pubid )
  6   union all
  7   ( select author
  8     from wrote, article
  9     where publication=pubid ) ) ba
 10  where aid=author
 11  group by name,aid;

NAME                 COUNT(AID)
-------------------- ----------
Toman, David                  3
Chomicki, Jan                 3
Saake, Gunter                 1
```


Summary

- SQL covered so far:
  1. Simple SELECT BLOCK (naming of attributes, allowed expressions, etc.)
  2. Set operations
  3. Duplicates and Bag operations
  4. Formulation of complex queries, nesting of queries, and views
  5. Aggregation

  . . . this covers ALL of SQL queries (i.e., they can be expressed in the syntax introduced so far, but it might be cumbersome)

  $\Rightarrow$ (lots of) syntactic sugar coming next . . .