# SQL: Ordering Results, Duplicate Semantics and NULL Values

Spring 2018

School of Computer Science
University of Waterloo

Databases CS348

# Ordering Results

- No particular ordering on the rows of a table can be assumed when queries are written. (This is important!)
- No particular ordering of rows of an intermediate result in the query can be assumed either.
- However, it is possible to order the final result of a query, using the `ORDER BY` clause at the end of the query.

General form:

$$\texttt{ORDER BY}\ \ e_1\ [Dir_1], \ldots, e_k\ [Dir_k]$$

where $Dir_i$ is either `ASC` or `DESC`; `ASC` is the default.

# Example

List all authors in the database in ascending order of their name:

```
SQL> select distinct *
  2  from author
  3  order by name;

AID NAME
--- --------------
  2 Chomicki, Jan
  3 Saake, Gunter
  1 Toman, David
```

Again, the `asc` keyword is optional, and is assumed by default.
A *descending* order is obtained with the `desc` keyword.
Minor sorts, minor minor sorts, etc., can be added.

# Multisets and Duplicates

- SQL has always had a **MULTISET/BAG** semantics rather than a **SET** semantics:

  $\Rightarrow$ SQL tables are **multisets** of tuples

  $\Rightarrow$ originally for efficiency reasons

- What does "*allows duplicates*" mean?

| part |
|---|
| bolt |
| bolt |
| bolt |
| nut |
| nut |

$\Longleftrightarrow$

| part | cnt |
|---|---|
| bolt | 3 |
| nut | 2 |

# How does this impact Queries?

## Example (Cheap Quantification–Projection)

EMP

| Name | Dept |
|---|---|
| Bob | CS |
| Sue | CS |
| Fred | PMath |
| Barb | Stats |
| Jim | Stats |

$\xrightarrow{\{y \mid \exists x.\mathrm{EMP}(x,y)\}}$

| Dept |
|---|
| CS |
| CS |
| PMath |
| Stats |
| Stats |

$\Longleftrightarrow$

| Dept | cnt |
|---|---|
| CS | 2 |
| PMath | 1 |
| Stats | 2 |

# Range-restricted Queries for Multisets

## Definition (Range restricted formulas with DISTINCT)

A formula (condition) $\varphi$ is *range restricted* when, for $\varphi_i$ that are also range restricted, $\varphi$ has the form

$$
\begin{array}{ll}
R(x_{i_1}, \ldots, x_{i_k}), & \\
\varphi_1 \wedge \varphi_2\,, & \\
\varphi_1 \wedge (x_i = x_j) & (\{x_i, x_j\} \cap FV(\varphi_1) \neq \emptyset), \\
\exists x_i . \varphi_1 & (x_i \in FV(\varphi_1)), \\
\varphi_1 \vee \varphi_2 & (FV(\varphi_1) = FV(\varphi_2)), \\
\varphi_1 \wedge \neg \varphi_2 & (FV(\varphi_2) = FV(\varphi_1)), \text{ or} \\
\text{DISTINCT}(\varphi). &
\end{array}
$$

# Duplicates and Queries

How do we define what an answer to a query is now?

## Ideas

1. A finite valuation can appear $k$ times ($k > 0$) as a query answer.
2. The number of duplicates is a function of the numbers of duplicates in formulas.
3. $\mathbf{DB}, \theta, k \models \varphi$ reads "finite valuation $\theta$ appears $k$ times in $\varphi$'s answer".

## Definition (Multiset Semantics for the Relational Calculus)

$$
\begin{array}{lll}
\mathbf{DB}, \theta, 0 \models \varphi & \text{if} & \mathbf{DB}, \theta \not\models \varphi \\
\mathbf{DB}, \theta, k \models R(x_1, \ldots, x_k) & \text{if} & (\theta(x_1), \ldots, \theta(x_k)) \in \mathbf{R} \ \ k \text{ times} \\
\mathbf{DB}, \theta, m \cdot n \models \varphi \wedge \psi & \text{if} & \mathbf{DB}, \theta, m \models \varphi \text{ and } \mathbf{DB}, \theta, n \models \psi \\
\mathbf{DB}, \theta, m \models \varphi \wedge (x_i = x_j) & \text{if} & \mathbf{DB}, \theta, m \models \varphi \text{ and } \theta(x_i) = \theta(x_j) \\
\mathbf{DB}, \theta, \sum_{v \in D} n_v \models \exists x.\varphi & \text{if} & \mathbf{DB}, \theta[x := v], n_v \models \varphi \\
\mathbf{DB}, \theta, m + n \models \varphi \vee \psi & \text{if} & \mathbf{DB}, \theta, m \models \varphi \text{ and } \mathbf{DB}, \theta, n \models \psi \\
\mathbf{DB}, \theta, m - n \models \varphi \wedge \neg\psi & \text{if} & \mathbf{DB}, \theta, m \models \varphi, \mathbf{DB}, \theta, n \models \psi \text{ and } m > n
\end{array}
$$

# Duplicates and SQL

Allowing duplicates leads to additional *syntax*.

- A *duplicate elimination operator*
  ⇒ "`SELECT DISTINCT x`" v.s. "`SELECT x`" in `SELECT`-blocks
- MULTISET (BAG) operators
  ⇒ equivalents of *set operations*
  ⇒ but with multiset semantics.

# 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
```

$\Rightarrow$ for publications with $n$ authors we get $O(n^2)$ answers!

# Bag Operations

- Bag union: `UNION ALL`

  ⇒ additive union: bag containing all in $Q_1$ *and* $Q_2$.

- Bag difference: `EXCEPT ALL`

  ⇒ subtractive difference (monus):
  ⇒ a bag all tuples in $Q_1$ for which
  there is no "matching" tuple in $Q_2$.

- Bag intersection: `INTERSECT ALL`

  ⇒ a bag of all tuples taking the
  maximal number common to $Q_1$ and $Q_2$

# Example

```
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
```

# Summary

- SQL covered so far:
  1. simple SELECT BLOCK
  2. set operations
  3. duplicates and multiset operations
  4. formulation of complex queries, nesting of queries, and views
  5. aggregation
- Note that duplicates in subqueries occurring in `where` clauses will not change the results computed by the top-level query, but that this is not true for subqueries in `with` or `from` clauses.

# "Pure" SQL Equivalence, Revisited

Recall how nesting in the WHERE clause is 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
) AS s
WHERE  r.a = s.b
```

Rewriting does not generally hold if `DISTINCT` is removed.

# What is a "null" value?

| Phone | | |
|---|---|---|
| Name | Office | Home |
| Joe | 1234 | 3456 |
| Sue | 1235 | ? |

- *Sue doesn't have home phone* (value inapplicable)
- *Sue has home phone, but we don't know her number* (value unknown)

# Value Inapplicable

- Essentially *poor schema design*.
- Better design:

| Office Phone | |
|---|---|
| Name | Office |
| Joe | 1234 |
| Sue | 1235 |

| Home Phone | |
|---|---|
| Name | Home |
| Joe | 3456 |

- Queries should behave *as if asked* over the above decomposition.
  $\Rightarrow$ (relatively) easy to implement

# Value Unknown

## Idea

*Unknown values can be replaced by any domain value (that satisfies integrity constraints).*

$\Rightarrow$ *many possibilities (possible worlds)*

| Phone | | |
|---|---|---|
| Name | Office | Home |
| Joe | 1234 | 3456 |
| Sue | 1235 | ? |

$\rightarrow$

| Phone | | |
|---|---|---|
| Name | Office | Home |
| Joe | 1234 | 3456 |
| Sue | 1235 | 0000 |

⋮

| Phone | | |
|---|---|---|
| Name | Office | Home |
| Joe | 1234 | 3456 |
| Sue | 1235 | 9999 |

# Value Unknown and Queries

How do we answer queries?

**Idea**

*Answers true in all possible worlds $W$ of an incomplete $D$.*

**Certain Answer**

$$Q(D) = \bigcap_{W \text{ world of } D} Q(W)$$

$\Rightarrow$ answer common to all possible worlds.

**Is this (computationally) feasible?**

$\Rightarrow$ NO (NP-hard to *undecidable* except in trivial cases)

**SQL's solution:** a (crude) approximation

# What can we do with NULLs in SQL?

expressions

- general rule: a NULL as a parameter to an operation makes (should make) the result NULL
- $1 + \texttt{NULL} \rightarrow \texttt{NULL}$, $\texttt{'foo'||NULL} \rightarrow \texttt{NULL}$, etc.

predicates/comparisons

- three-valued logic (crude approximation of “value unknown”)

set operations

- unique *special value* for *duplicates*

aggregate operations

- doesn’t “*count*” (i.e., “value inapplicable”)

# Comparisons Revisited

**Idea**

*Comparisons with a* `NULL` *value return* `UNKNOWN`

**Example**

$$
\begin{array}{ll}
1 = 1 & \texttt{TRUE} \\
1 = \texttt{NULL} & \texttt{UNKNOWN} \\
1 = 2 & \texttt{FALSE}
\end{array}
$$

Still short of *proper logical* behaviour:

$$x = 0 \vee x \neq 0$$

should be always true (no matter what $x$ is, including `NULL`!), but...

# UNKNOWN and Boolean Connectives

## Idea

*Boolean operations have to handle* UNKNOWN
$\Rightarrow$ *extended truth tables for Boolean connectives*

| $\wedge$ | T | U | F |
|---|---|---|---|
| T | T | U | F |
| U | U | U | F |
| F | F | F | F |

| $\vee$ | T | U | F |
|---|---|---|---|
| T | T | T | T |
| U | T | U | U |
| F | T | U | F |

| $\neg$ | |
|---|---|
| T | F |
| U | U |
| F | T |

. . . for tuples in which $x$ is assigned the NULL value we get:

$$x = 0 \vee x \neq 0 \rightarrow \texttt{UNKNOWN} \vee \texttt{UNKNOWN} \rightarrow \texttt{UNKNOWN}$$

which is not the same as TRUE.

# `UNKNOWN` in `WHERE` Clauses

How is this used in a `WHERE` clause?

- Additional syntax `IS TRUE`, `IS FALSE`, and `IS UNKNOWN`
  $\Rightarrow$ `WHERE <cond>` shorthand for `WHERE <cond> IS TRUE`
- Special comparison `IS NULL`

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

  ⇒ `count(*)` counts "rows"

```
db2 => select count(*) as RS, count(url) as US
db2 => from author;

RS          US
----------------
3           2

  1 record(s) selected.
```

# Outer Join

## Idea

*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 $C = (r.y = s.y)$).
  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`
  3. $\{(\text{NULL}, y, z) : S(y, z) \wedge \neg(\exists x.R(x, y))\}$ for `RIGHT` and `FULL`

  $\Rightarrow$ syntactic sugar for `UNION ALL`

# Example

```
db2 => select aid,publication
db2 => from author left join wrote
db2 =>                     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 => from author left join wrote
db2 =>                          on aid=author
db2 => group by aid;

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

  4 record(s) selected.
```

# Summary

- NULLs 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 NULLs in practice

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