SQL: Duplicate Semantics and NULL Values

Fall 2015

David Toman

School of Computer Science
University of Waterloo

Databases CS338

Multisets and Duplicates

- SQL uses a MULTISET/BAG semantics rather than a SET semantics:
  - SQL tables are multisets of tuples
  - originally for efficiency reasons
- What does “allows duplicates” mean?

<table>
<thead>
<tr>
<th>part</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>bolt</td>
<td>3</td>
</tr>
<tr>
<td>bolt</td>
<td></td>
</tr>
<tr>
<td>bolt</td>
<td></td>
</tr>
<tr>
<td>nut</td>
<td>2</td>
</tr>
</tbody>
</table>

How does this impact Queries?

Example (Cheap Quantification–Projection)

<table>
<thead>
<tr>
<th>EMP</th>
<th>Dept</th>
<th>Dept cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>Fred</td>
<td>PMath</td>
<td></td>
</tr>
<tr>
<td>Barb</td>
<td>Stats</td>
<td></td>
</tr>
<tr>
<td>Jim</td>
<td>Stats</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EMP</th>
<th>Dept</th>
<th>Dept cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>CS</td>
<td>2</td>
</tr>
<tr>
<td>Sue</td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>Fred</td>
<td>PMath</td>
<td>1</td>
</tr>
<tr>
<td>Barb</td>
<td>Stats</td>
<td>2</td>
</tr>
</tbody>
</table>

Duplicates and Queries

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

**Ideas**

1. an answer tuple can appear *k* times (*k > 0*) in *Q(D)*
2. the number of duplicates is a function of the numbers of duplicates in subqueries

**Definition (Duplicate Semantics (for Relational Calculus))**

we write *t*{*k*} ∈ *Q* for “the tuple *t* appears *k* times in the answer to *Q*”

- *t*{*k*} ∈ *R*(*x*) if *t* ∈ *R*{*k*}
- *t*{*1*} ∈ (*x*{*i*} → *x*) if *t*(*x*{*i*}) = *t*(*x*)
- *t*{*m *- n*} ∈ *Q*{*1*} ∧ *Q*{*2*} if *t*{*m*} ∈ *Q*{*1*} and *t*{*n*} ∈ *Q*{*2*}
- *t*{*∑*ν∈D*νv*} ∈ *∃x*{*i*}.*Q* if *t*[*x*{*i*} → *ν*{*νv*}] ∈ *Q*
- *t*{*m + n*} ∈ *Q*{*1*} ∨ *Q*{*2*} if *t*{*m*} ∈ *Q*{*1*} and *t*{*n*} ∈ *Q*{*2*}
- *t*{*max(0, m - n)*} ∈ *Q*{*1*} ∧ ¬*Q*{*2*} if *t*{*m*} ∈ *Q*{*1*} and *t*{*n*} ∈ *Q*{*2*}
- *t*{*1*} ∈ DISTINCT(*Q*{*1*} if *t*{*n*} ∈ *Q*
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.

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 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

⇒ for publications with \( n \) authors we get \( O(n^2) \) answers!
```

```
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

This covers pretty much all of standard SQL queries (i.e., they can be expressed in the syntax introduced so far, but it might be quite cumbersome)

⇒ (lots of) syntactic sugar coming next ...

Value Inapplicable

- Essentially poor schema design.
- Better design:

<table>
<thead>
<tr>
<th>Office Phone</th>
<th>Home Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Phone</td>
</tr>
<tr>
<td>Joe</td>
<td>1234</td>
</tr>
<tr>
<td>Sue</td>
<td>1235</td>
</tr>
<tr>
<td>Name</td>
<td>Phone</td>
</tr>
<tr>
<td>Joe</td>
<td>3456</td>
</tr>
</tbody>
</table>

- Queries should behave as if asked over the above decomposition.

⇒ (relatively) easy to implement

What is a “null” value?

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

Value Unknown

Idea

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

⇒ many possibilities (possible worlds)
Value Unknown and Queries

How do we answer queries?

Idea

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

Certain Answer

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

⇒ answer common to all possible worlds.

Is this (computationally) feasible?

⇒ 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*
- $1 + \text{NULL} \rightarrow \text{NULL}$, ‘foo’ | | | NULL → 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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>TRUE</td>
</tr>
<tr>
<td>1</td>
<td>NULL</td>
<td></td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Still short of *proper logical* behaviour:

$$x = 0 \lor x \neq 0$$

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

---

UNKNOWN and Boolean Connectives

Idea

*Boolean operations have to handle* UNKNOWN

⇒ *extended truth tables for boolean connectives*

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\land$</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>$\lor$</td>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>$\neg$</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
</tbody>
</table>

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

$$x = 0 \lor x \neq 0 \rightarrow \text{UNKNOWN} \lor \text{UNKNOWN} \rightarrow \text{UNKNOWN}$$

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`  
  ⇒ 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

<table>
<thead>
<tr>
<th>AID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Saake, Gunter</td>
</tr>
</tbody>
</table>
```

**Counting NULLS**

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

- **count (URL)** counts only non-NULL URLs  
  ⇒ **count(*)** counts “rows”

```
db2 => select count(*) as RS, count(url) as US
   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**  
  ⇒ FROM R <j-type> JOIN S ON C  
  ⇒ 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 \(\{(x, y, z) : R(x, y) \land S(y, z)\}\) for LEFT and FULL  
  2 \(\{(x, y, NULL) : R(x, y) \land \neg(\exists z.S(y, z))\}\) for LEFT and FULL  
  3 \(\{(NULL, y, z) : S(y, z) \land \neg(\exists x.R(x, y))\}\) for RIGHT and FULL  
  ⇒ **syntactic sugar** for UNION ALL

**Example**

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

<table>
<thead>
<tr>
<th>AID</th>
<th>PUBLICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ChTo98</td>
</tr>
<tr>
<td>1</td>
<td>ChTo98a</td>
</tr>
<tr>
<td>1</td>
<td>Tom97</td>
</tr>
<tr>
<td>2</td>
<td>ChTo98</td>
</tr>
<tr>
<td>2</td>
<td>ChTo98a</td>
</tr>
<tr>
<td>2</td>
<td>ChSa98</td>
</tr>
<tr>
<td>3</td>
<td>ChSa98</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
</tr>
</tbody>
</table>

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
```

<table>
<thead>
<tr>
<th>AID</th>
<th>PUBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

4 record(s) selected.

Summary

- **NULLs are necessary evil**
  - used to account for (small) irregularities in data
  - should be used sparingly

- can be **always** avoided
  - however some of the solutions may be inefficient

- you can’t escape **NULLs** in practice
  - *easy fix* for blunders in schema design
  - . . . also due to schema evolution, etc.