SQL: Duplicate Semantics and NULL Values

Fall 2017

School of Computer Science
University of Waterloo

Databases CS348
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>bolt</th>
<th>bolt</th>
<th>bolt</th>
<th>nut</th>
<th>nut</th>
</tr>
</thead>
</table>

$$ \iff $$

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

(University of Waterloo)
How does this impact Queries?

Example (Cheap Quantification–Projection)

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

\(\{y \mid \exists x. \text{EMP}(x, y)\}\)  \(\rightarrow\)  

<table>
<thead>
<tr>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
</tr>
<tr>
<td>CS</td>
</tr>
<tr>
<td>PMath</td>
</tr>
<tr>
<td>Stats</td>
</tr>
</tbody>
</table>

\(\iff\)  

<table>
<thead>
<tr>
<th>Dept</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>2</td>
</tr>
<tr>
<td>PMath</td>
<td>1</td>
</tr>
<tr>
<td>Stats</td>
<td>2</td>
</tr>
</tbody>
</table>

(University of Waterloo) SQL: Duplicates and NULLs
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 \( t\{k\} \in Q \) for “the tuple \( t \) appears \( k \) times in the answer to \( Q \)”

\[
\begin{align*}
    t\{k\} &\in R(x) \quad \text{if} \quad t \in R^D \quad \text{k times} \\
    t\{1\} &\in (x_i = x_j) \quad \text{if} \quad t(x_i) = t(x_j) \\
    t\{m \cdot n\} &\in Q_1 \land Q_2 \quad \text{if} \quad t\{m\} \in Q_1 \quad \text{and} \quad t\{n\} \in Q_2 \\
    t\{\sum_{v \in D} n_v\} &\in \exists x_i.Q \quad \text{if} \quad t[x_i \mapsto v]\{n_v\} \in Q \\
    t\{m + n\} &\in Q_1 \lor Q_2 \quad \text{if} \quad t\{m\} \in Q_1 \quad \text{and} \quad t\{n\} \in Q_2 \\
    t\{\max(0, m - n)\} &\in Q_1 \land \neg Q_2 \quad \text{if} \quad t\{m\} \in Q_1 \quad \text{and} \quad t\{n\} \in Q_2 \\
    t\{1\} &\in \text{DISTINCT}(Q)(D) \quad \text{if} \quad t\{n\} \in Q
\end{align*}
\]
Allowing duplicates leads to additional \textit{syntax}. 

- a \textit{duplicate elimination operator} \\
  \Rightarrow \text{"SELECT DISTINCT x" v.s. "SELECT x" in SELECT-blocks}

- MULTISET (BAG) operators \\
  \Rightarrow \text{equivalents of set operations} \\
  \Rightarrow \text{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
```

⇒ 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
```

Duplicates and NULLs
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 . . .
What is a “null” value?

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

- *Sue doesn’t have home phone* (value inapplicable)
- *Sue has home phone, but we don’t know her number* (value unknown)
Essentially *poor schema design*.

**Better design:**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Home Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>3456</td>
</tr>
</tbody>
</table>

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

⇒ (relatively) easy to implement
**Value Unknown**

**Idea**

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

⇒ *many possibilities (possible worlds)*

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

<table>
<thead>
<tr>
<th>Phone</th>
<th>Name</th>
<th>Office</th>
<th>Home</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>1234</td>
<td></td>
<td>3456</td>
</tr>
<tr>
<td>Sue</td>
<td>1235</td>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phone</th>
<th>Name</th>
<th>Office</th>
<th>Home</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>1234</td>
<td></td>
<td>3456</td>
</tr>
<tr>
<td>Sue</td>
<td>1235</td>
<td></td>
<td>9999</td>
</tr>
</tbody>
</table>
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
- \[1 + \text{NULL} \rightarrow \text{NULL}, \ '\text{foo}' \mid \text{NULL} \rightarrow \text{NULL}, \text{etc.}\]

**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 \texttt{NULL} value return \texttt{UNKNOWN}

Example

\begin{align*}
1 &= 1 & \text{TRUE} \\
1 &= \text{NULL} & \text{UNKNOWN} \\
1 &= 2 & \text{FALSE}
\end{align*}

Still short of \textit{proper logical} behaviour:

\[ x = 0 \lor x \neq 0 \]

should be always \texttt{true} (no matter what \( x \) is, including \texttt{NULL}!), but. . .
Idea

Boolean operations have to handle unknown
⇒ extended truth tables for boolean connectives

\[
\begin{array}{c|cccc}
\land & T & U & F \\
T & T & U & F \\
U & U & U & F \\
F & F & F & F \\
\end{array}
\quad
\begin{array}{c|cccc}
\lor & T & U & F \\
T & T & T & T \\
U & T & U & U \\
F & T & U & F \\
\end{array}
\quad
\begin{array}{c|c}
\neg & \\
T & F \\
U & U \\
F & T \\
\end{array}
\]

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

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

(University of Waterloo)  SQL: Duplicates and NULLs
Counting NULLS

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

- \( \text{count (URL)} \) counts only non-NULL URL’s
  
  \( \Rightarrow \) \( \text{count (*)} \) counts “rows”

\[
\begin{align*}
\text{db2} & \Rightarrow \text{select count(*) as RS, count(url) as US} \\
\text{db2 (cont.)} & \Rightarrow \text{from author}
\end{align*}
\]

<table>
<thead>
<tr>
<th>RS</th>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

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
  \[ \Rightarrow \text{FROM } R \ <j-type> \ JOIN \ S \ ON \ C \]  
  \[ \Rightarrow \text{the } <j-type> \text{ is one of FULL, LEFT, RIGHT, or INNER} \]

- semantics (for \( R(x, y) \), \( S(y, z) \), and \( C = (r.y = s.y) \)).
  1. \( \{(x, y, z) : R(x, y) \land S(y, z)\} \)
  2. \( \{(x, y, \text{NULL}) : R(x, y) \land \neg(\exists z. S(y, z))\} \) for LEFT and FULL
  3. \( \{(\text{NULL}, y, z) : S(y, z) \land \neg(\exists x. R(x, y))\} \) for RIGHT and FULL

  \[ \Rightarrow \text{syntactic sugar for UNION ALL} \]
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

- **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.