Incomplete Information

SQL's handling of incompleteness is problematic

". . . this topic cannot be described in a manner that is simultaneously both comprehensive and comprehensible"
"Those SQL features are . . . fundamentally at odds with the way the world behaves"

C. Date & H. Darwen, 'A Guide to SQL Standard'

"If you have any nulls in your database, you're getting wrong answers to some of your queries. What's more, you have no way of knowing, in general, just which queries you're getting wrong answers to; all results become suspect. You can never trust the answers you get from a database with nulls"

C. Date, 'Database in Depth'

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	“Big Data”	30
Ord2	“SQL”	35
Ord3	“Logic”	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	“Big Data”	30
Ord2	“SQL”	35
Ord3	“Logic”	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

Typical queries, as we teach students to write them:

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	“Big Data”	30
Ord2	“SQL”	35
Ord3	“Logic”	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

Typical queries, as we teach students to write them:

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	“Big Data”	30
Ord2	“SQL”	35
Ord3	“Logic”	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

Typical queries, as we teach students to write them:

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	"Big Data"	30
Ord2	"SQL"	35
Ord3	"Logic"	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

Typical queries, as we teach students to write them:

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Answer: Ord3.

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

Answer: none.

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	"Big Data"	30
Ord2	"SQL"	35
Ord3	"Logic"	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

Typical queries, as we teach students to write them:

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	"Big Data"	30
Ord2	"SQL"	35
Ord3	"Logic"	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	“Big Data”	30
Ord2	“SQL”	35
Ord3	“Logic”	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	Ord2

Customer

CUST_ID	NAME
c1	John
c2	Mary

In the real world, information is often missing

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	"Big Data"	30
Ord2	"SQL"	35
Ord3	"Logic"	50

Pay

Customer

CUST_ID	NAME
c1	John
c2	Mary

In the real world, information is often missing

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	“Big Data”	30
Ord2	“SQL”	35
Ord3	“Logic”	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	--

Customer

CUST_ID	NAME
c1	John
c2	Mary

In the real world, information is often missing

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

A company database: orders, customers, payments

Orders

ORDER_ID	TITLE	PRICE
Ord1	"Big Data"	30
Ord2	"SQL"	35
Ord3	"Logic"	50

Pay

CUST_ID	ORDER
c1	Ord1
c2	--

Customer

CUST_ID	NAME
c1	John
c2	Mary

In the real world, information is often missing

Unpaid orders:

```
select O.order_id
from Orders O
where not exists
        (select * from Pay P
         where P.order=O.order_id)
```

Customers without an order:

```
select C.cust_id from Customer C
where not exists
        (select * from Orders O, Pay P
         where C.cust_id=P.cust_id
         and P.order=O.order_id)
```

Old Answer: Ord3   New: NONE!

Old answer: none   New: c2!

But it must be handled...

- Incomplete data is everywhere.
- Represented by nulls in relational databases.
- The more data we accumulate, the more incomplete data we accumulate.
- Sources:
  - Traditional (missing data, wrong entries, etc)
  - The Web
  - Integration/translation/exchange of data, etc
- The importance of it was recognized early
  - Codd, *"Understanding relations (installment #7)"*, 1975.
- And yet the state is very poor.

What it's blamed on: 3-valued logic

SQL used 3-valued logic, or 3VL, for databases with nulls.

Normally we have two truth values: true **t**, false **f**. But comparisons involving nulls evaluate to unknown (**u**): for instance, $5 = \text{null}$ is **u**.

They are propagated using 3VL rules:

$\wedge$	**t**	**f**	**u**
t	**t**	**f**	**u**
f	**f**	**f**	**f**
u	**u**	**f**	**u**

$\vee$	**t**	**f**	**u**
t	**t**	**t**	**t**
f	**t**	**f**	**u**
u	**t**	**u**	**u**

$\vee$	
t	**f**
f	**t**
u	**u**

- Committee design from 30 years ago, leads to many problems,
- but is efficient and used everywhere

What does theory have to offer?

The notion of correctness — certain answers.

- Answers independent of the interpretation of missing information.
- Typically defined as

$$\text{certain}(Q, D) = \bigcap Q(D')$$

  over all possible worlds D' described by D
- Standard approach, used in all applications: data integration and exchange, inconsistent data, querying with ontologies, data cleaning, etc.
- First need to define what an incomplete database can represent.

The model

Marked (aka labeled or naive) nulls. Idea: missing values, that can repeat.

Semantics: closed world

A	B	C
1	2	$\bot_1$
$\bot_2$	$\bot_1$	3
$\bot_3$	5	1
2	$\bot_3$	3

The model

Marked (aka labeled or naive) nulls. Idea: missing values, that can repeat.

Semantics: closed world

A	B	C
1	2	$\bot_1$
$\bot_2$	$\bot_1$	3
$\bot_3$	5	1
2	$\bot_3$	3

$$h(\bot_1) = 4$$
$$h(\bot_2) = 3$$
$$h(\bot_3) = 5$$
$$\Longrightarrow$$

A	B	C
1	2	4
3	4	3
5	5	1
2	5	3

The model

Marked (aka labeled or naive) nulls. Idea: missing values, that can repeat.

Semantics: closed world

A	B	C
1	2	$\bot_1$
$\bot_2$	$\bot_1$	3
$\bot_3$	5	1
2	$\bot_3$	3

$h(\bot_1) = 4$
$h(\bot_2) = 3$
$h(\bot_3) = 5$
$\Longrightarrow$

A	B	C
1	2	4
3	4	3
5	5	1
2	5	3

SQL model: a special case when all nulls are distinct.

Open worlds semantics

A	B	C
1	2	$\bot_1$
$\bot_2$	$\bot_1$	3
$\bot_3$	5	1
2	$\bot_3$	3

$h(\bot_1) = 4$
$h(\bot_2) = 3$
$h(\bot_3) = 5$
$\Longrightarrow$

A	B	C
1	2	4
3	4	3
5	5	1
2	5	3
7	9	12
11	8	10

Semantics via homomorphisms/valuations

Maps h are homomorphisms whose range does not include nulls. They are called valuation. A normal homomorphism:

- $h(c) = c$ for every constant value c
- $h(\perp)$ could be a null or a constant value

In a valuation,

- $h(c) = c$ for every constant value c
- $h(\perp)$ must be a constant value

They define open world semantics $[\![D]\!]_{\text{owa}}$ and closed world semantics $[\![D]\!]_{\text{cwa}}$

- semantics under open world and closed world assumptions

Semantics via homomorphisms/valuations

$$[\![D]\!]_{\mathsf{cwa}} = \{h(D) \mid h \text{ is a valuation}\}$$

$$[\![D]\!]_{\mathsf{owa}} = \{\text{complete } D' \mid \exists \text{ valuation } h : D \to D'\}$$

Alternatively:

$$[\![D]\!]_{\mathsf{owa}} = \{h(D) \cup D_0 \mid h \text{ is a valuation}, D_0 \text{ does not have nulls}\}$$

Certain answers

Tuples present in query answers in all possible world:

$$\text{certain}_{\text{CWA}}(Q, D) = \bigcap \{ Q(D') \mid D' \in [\![D]\!]_{\text{cwa}} \}$$

$$\text{certain}_{\text{OWA}}(Q, D) = \bigcap \{ Q(D') \mid D' \in [\![D]\!]_{\text{owa}} \}$$

Note that tuples in certain answers cannot have nulls, i.e. they only have constant values.

Can SQL evaluation and certain answers be the same?

No!

Complexity argument:

- Finding certain answers for relational calculus queries in coNP-hard
- SQL is very efficient (DLOGSPACE; even AC^0)

Complexity of certain answers

Query Q from relational calculus/algebra, i.e., first-order logic. Assume it is a sentence (yes/no query).

Look at OWA first.

$$\text{certain}_{\text{OWA}}(Q, D) = \mathbf{t} \;\Leftrightarrow\; \forall D' \in [\![D]\!]_{\text{owa}} : \; D' \models Q$$

But: $[\![\emptyset]\!]_{\text{owa}}$ = all databases!

Therefore:

$$\text{certain}_{\text{OWA}}(Q, \emptyset) = \mathbf{t} \;\Leftrightarrow\; \forall D' : \; D' \models Q$$

or Q' is a valid sentence.

Validity and certain answers

What do we know about validity of first-order sentences? It is undecidable!

This is a classical result in logic (INF1). But here we are in a different world, all databases are finite. Does it help?

No! It is even worse, not even recursively enumerable (even infinite time does not help!)

Does CWA help? Somewhat....

$$\mathsf{certain}_{\mathsf{CWA}}(Q, D) = \mathbf{t} \;\Leftrightarrow\; \forall \text{ valuations } h : h(D) \models Q$$

There are still infinitely many valuations, but actually only finitely many suffice (blackboard).

This means checking whether $\mathsf{certain}_{\mathsf{CWA}}(Q, D) = \mathbf{f}$ is in NP:

- guess a valuation h
- check if $h(D) \models \neg Q$ (in PTIME in data complexity)

Thus checking whether $\mathsf{certain}_{\mathsf{CWA}}(Q, D) = \mathbf{t}$ is in coNP.

CWA: can we do better? No...

Checking whether $\text{certain}_{\text{CWA}}(Q, D) = \mathbf{t}$ is coNP-complete.

Reduction from 3-colorability.

Take a graph $G = (V, E)$ and create a database D_G with nulls $\bot_v$ for each $v \in V$ and edges $(\bot_v, \bot_{v'})$ whenever $(v, v') \in E$.

$Q = 4$ different vertices $\vee \exists x E(x, x)$

where 4 different vertices is $\exists x, y, z, u\ (x \neq y \wedge y \neq z \wedge \ldots)$

Then $\text{certain}_{\text{CWA}}(Q, D_G) = \mathbf{t}$ iff G is 3-colorable.

The bottom line

SQL is very efficient (for the relational calculus fragment, AC^0)

Certain answers range from coNP-complete to undecidable for different semantics.

Hence provably SQL cannot compute certain answers.

Wrong behaviors: false negatives and false positives

False negatives: missing some of the certain answers

False positives: giving answers which are not certain

Complexity tells us:

$\boxed{\text{SQL query evaluation cannot avoid both!}}$

False positives are worse: they tell you something blatantly false rather than hide part of the truth

And we have seen SQL generates both.

What to do?

We now analyze evaluation procedures.

Goal: to see when we can effectively

- compute or
- approximate

certain answers.

So first we need to define evaluation procedures.

Evaluation procedures for first-order queries

Given a database D, a query $Q(\bar{x})$, a tuple $\bar{a}$

$$\mathrm{Eval}(D, Q(\bar{a})) \quad \in \quad \text{set of truth values}$$

- 2-valued logic: truth values are **t** (true) and **f** (false)
- 3-valued logic: **t**, **f**, and **u** (unknown)

Meaning: if $\mathrm{Eval}(D, Q(\bar{a}))$ evaluates to

- **t**, we know $\bar{a} \in Q(D)$
- **f**, we know $\bar{a} \notin Q(D)$
- **u**, we don't know whether $\bar{a} \in Q(D)$ or $\bar{a} \notin Q(D)$

Evaluation procedures and queries results

A procedure defines the result of evaluation:

$$\mathsf{Eval}(Q, D) \quad = \quad \{\bar{a} \;\mid\; \mathsf{Eval}(D, Q(\bar{a})) = \mathbf{t}\}$$

Think of the `WHERE` clause in SQL: we only look at values that make it true (and discard those that make it false or unknown).

Standard semantics for logical connectives

All evaluation procedures are completely standard for $\vee, \wedge, \neg, \forall, \exists$:

$$\begin{array}{rcl}
\mathsf{Eval}(D, Q \vee Q') & = & \mathsf{Eval}(D, Q) \vee \mathsf{Eval}(D, Q') \\
\mathsf{Eval}(Q \wedge Q', D) & = & \mathsf{Eval}(D, Q) \wedge \mathsf{Eval}(D, Q') \\
\mathsf{Eval}(D, \neg Q) & = & \neg \mathsf{Eval}(D, Q) \\
\mathsf{Eval}(D, \exists x\ Q(x, \bar{a})) & = & \bigvee \{\mathsf{Eval}(D, Q(a', \bar{a})) \mid a' \in adom(D)\} \\
\mathsf{Eval}(D, \forall x\ Q(x, \bar{a})) & = & \bigwedge \{\mathsf{Eval}(D, Q(a', \bar{a})) \mid a' \in adom(D)\}
\end{array}$$

Standard semantics for logical connectives cont'd

Of course $\vee, \wedge, \neg$ are given by truth tables for the logic: the usual Boolean logic for relational calculus, or the 3-valued logic for SQL.

So we just need to define rules for atoms, $R(\bar{x})$ and basic comparisons.

We assume comparisons are just equalities $a = b$.

FO evaluation procedure

$$\mathrm{Eval}_{\mathrm{FO}}(D, R(\bar{a})) \quad = \quad \begin{cases} \mathbf{t} & \text{if } \bar{a} \in R \\ \mathbf{f} & \text{if } \bar{a} \notin R \end{cases}$$

$$\mathrm{Eval}_{\mathrm{FO}}(D, a = b) \quad = \quad \begin{cases} \mathbf{t} & \text{if } a = b \\ \mathbf{f} & \text{if } a \neq b \end{cases}$$

Correctness via $\mathsf{Eval}_{\mathsf{FO}}$

Recall:

$$\mathsf{Eval}_{\mathsf{FO}}(Q, D) \quad = \quad \{\bar{a} \;\mid\; \mathsf{Eval}_{\mathsf{FO}}(D, Q(\bar{a})) = \mathbf{t}\}$$

We want at least simple correctness guarantees

$$\text{constant tuples in } \mathsf{Eval}_{\mathsf{FO}}(Q, D) \quad \subseteq \quad \mathsf{certain}(Q, D)$$

Correctness via $\mathsf{Eval}_{\mathsf{FO}}$

Recall:

$$\mathsf{Eval}_{\mathsf{FO}}(Q, D) \quad = \quad \{\bar{a} \mid \mathsf{Eval}_{\mathsf{FO}}(D, Q(\bar{a})) = \mathbf{t}\}$$

We want at least simple correctness guarantees

$$\text{constant tuples in } \mathsf{Eval}_{\mathsf{FO}}(Q, D) \quad \subseteq \quad \mathsf{certain}(Q, D)$$

Ideally:

$$\text{constant tuples in } \mathsf{Eval}_{\mathsf{FO}}(Q, D) \ = \ \mathsf{certain}(Q, D)$$

Correctness for CQs

UCQ: unions of conjunctive queries, or positive relational algebra $\pi, \sigma, \bowtie, \cup$.

For UCQs,

$$\text{constant tuples in } \mathsf{Eval}_{\mathsf{FO}}(Q, D) \;=\; \mathsf{certain}(Q, D)$$

for both open and closed world semantics.

First, $[\![D]\!]_{\mathsf{cwa}}$ and $[\![D]\!]_{\mathsf{owa}}$ have a "copy" of D (replace all nulls by new constants) so if $\mathsf{certain}_{\mathsf{OWA}}(D, Q) = \mathbf{t}$ or $\mathsf{certain}_{\mathsf{CWA}}(D, Q) = \mathbf{t}$ then $D \models Q$.

Correctness for CQs cont'd

Now we need the converse: if $D \models Q$, then $\text{certain}_{\text{OWA}}(D, Q) = \text{certain}_{\text{CWA}}(D, Q) = \mathbf{t}$.

Idea: let's look at a Boolean CQ Q with a tableau T_Q. Then

$$\begin{array}{ll} & D \models Q \\ \Rightarrow & T_Q \mapsto D \\ \Rightarrow & \forall D' : D \mapsto D' \text{ implies } T_Q \mapsto D' \\ \Rightarrow & \forall D' : D \mapsto D' \text{ implies } D' \models Q \\ \Rightarrow & \forall D' \in [\![D]\!]_{\text{owa}}(\text{ or in } [\![D]\!]_{\text{cwa}}) : \; D' \models Q \\ \Rightarrow & \text{certain}_{\text{OWA}}(D, Q) = \text{certain}_{\text{CWA}}(D, Q) = \mathbf{t} \end{array}$$

Same idea works for UCQs with free variables.

Correctness for CQs cont'd

Can the class of UCQs be extended? Answer:

- no under open world semantics, and
- yes under closed world semantics.

If Q is a relational calculus query, and

$$D \models Q \quad \Leftrightarrow \quad \text{certain}_{\text{OWA}}(D, Q) = \mathbf{t}$$

for all D, then Q is equivalent to a UCQ.

Correctness for CQs under closed world

Recall: UCQ is the fragment of relational calculus without $\forall$ and $\neg$.
That is, $\wedge, \vee, \exists$.

$\mathsf{RelCalc}_{\mathsf{certain}}$ — UCQs extended with the formation rule:

if $\varphi(\bar{x}, \bar{y}))$ is a query in $\mathsf{RelCalc}_{\mathsf{certain}}$, then so is:

$$\forall \bar{y}\ (\mathsf{atom}(\bar{y}) \rightarrow \varphi(\bar{x}, \bar{y}))$$

Here atom is $R(\bar{y})$ or $y_1 = y_2$.

Correctness for CQs under closed world cont'd

Also recall: UCQs are positive relational algebra, $\pi, \sigma, \bowtie, \cup$.

$\mathsf{RelCalc}_{\mathsf{certain}}$ is its extension with the division operator $\div$

- but only $Q \div R$ queries
- meaning: find tuples $\bar{a}$ that occur in $Q(D)$ together with every tuple $\bar{b}$ in R

For $\mathsf{RelCalc}_{\mathsf{certain}}$ queries,

$$\text{constant tuples in } \mathsf{Eval}_{\mathsf{FO}}(Q, D) \;=\; \mathsf{certain}_{\mathsf{CWA}}(Q, D)$$

SQL evaluation procedure

All that changes is the rule for comparisons.

SQL's rule: if one attribute of a comparison is null, the result is unknown.

$$\boxed{\mathsf{Eval}_{\mathsf{SQL}}(D, a = b) \quad = \quad \begin{cases} \mathbf{t} & \text{if } a = b \text{ and } \mathsf{NotNull}(a, b) \\ \mathbf{f} & \text{if } a \neq b \text{ and } \mathsf{NotNull}(a, b) \\ \mathbf{u} & \text{if } \mathsf{Null}(a) \text{ or } \mathsf{Null}(b) \end{cases}}$$

We write $\mathsf{Null}(a)$ if a is a null and $\mathsf{NotNull}(a)$ if it is not.

When does it work?

For UCQs,

constant tuples in $\mathsf{Eval}_{\mathsf{SQL}}(Q, D) \subseteq \mathsf{certain}(Q, D)$

Questions:

- can we extend this, say to all of relational calculus? That is, get an evaluation without false positives, and
- do we really need 3 truth values in SQL?

Is there a Boolean solution?

Perhaps the committee design missed something and we don't need 3-valued logic? Actually, we do....

Every query evaluation that uses the Boolean semantics for $\wedge, \vee, \neg$ generates false positives on databases with nulls.

Every evaluation means: complete freedom for

- Eval$(R(\bar{x}))$
- Eval$(x = y)$
- Eval$(\forall x\ \varphi)$
- Eval$(\exists x\ \varphi)$;

the only restriction is that $\wedge, \vee, \neg$ come from the usual Boolean logic.

No Boolean solution: by contradiction

Take a Boolean evaluation Eval with $\text{Eval}(Q, D) \subseteq \text{certain}(\varphi, D)$.

Take $D = \{R(1), S(\bot)\}$.

If $\text{Eval}\big(R(1), D\big) = \mathbf{f}$, then $\text{Eval}\big(\neg R(1), D\big) = \mathbf{t}$, but
$1 \notin \text{certain}\big(\neg R(x), D\big) \quad \Longrightarrow \quad \text{Eval}\big(R(1), D, a\big) = \mathbf{t}$.

If $\text{Eval}\big(\neg S(1), D\big) = \mathbf{f}$, then $\text{Eval}\big(S(1), D\big) = \mathbf{t}$. But
$1 \notin \text{certain}\big(S(x), D\big) \quad \Longrightarrow \quad \text{Eval}\big(\neg S(1), D\big) = \mathbf{t}$.

Now take $Q(x) = R(x) \wedge \neg S(x)$; we have $\text{Eval}\big(Q(1), D\big) = \mathbf{t}$, but
$1 \notin \text{certain}(Q, D)$:

- just take a complete database where $\bot$ becomes 1; in it $Q(1)$ is false.

What's wrong with SQL's 3VL?

It gives us both false positives and false negatives. Can we eliminate false positives?

SQL is too eager to say no.

If we say no to a result that ought to be unknown, when negation applies, no becomes yes! And that's how false positives creep in.

Consider R =

A	B
1	null

What about $(\text{null}, \text{null}) \in R$?

SQL says no but correct answer is unknown: what if null is really 1?

Towards a good evaluation: unifying tuples

Two tuples $\bar{t}_1$ and $\bar{t}_2$ unify if there is a mapping h of nulls to constants such that $h(\bar{t}_1) = h(\bar{t}_2)$.

$$\begin{array}{cccccc} (& 1 & \bot & 1 & 3 &) \\ (& \bot' & 2 & \bot' & 3 &) \end{array} \quad \Longrightarrow \quad (\; 1 \;\; 2 \;\; 1 \;\; 3 \;)$$

but $\begin{array}{cccccc} (& 1 & \bot & 2 & 3 &) \\ (& \bot' & 2 & \bot' & 3 &) \end{array}$ do not unify.

This can be checked in linear time.

Proper 3-valued procedure

$$\mathsf{Eval}_{3v}(D, R(\bar{a})) \;=\; \begin{cases} \mathbf{t} & \text{if } \bar{a} \in R \\ \mathbf{f} & \text{if } \bar{a} \text{ does not unify with any tuple in } R \\ \mathbf{u} & \text{otherwise} \end{cases}$$

$$\mathsf{Eval}_{3v}(D, a = b) \;=\; \begin{cases} \mathbf{t} & \text{if } a = b \\ \mathbf{f} & \text{if } a \neq b \text{ and } \mathsf{NotNull}(a, b) \\ \mathbf{u} & \text{otherwise} \end{cases}$$

Simple correctness guarantees: no false positives

If $\bar{a}$ is a tuple without nulls, and $\mathsf{Eval}_{3v}(D, Q(\bar{a})) = 1$ then $\bar{a} \in \mathsf{certain}(Q, D)$.

Simple correctness guarantees:

$$\text{constant tuples in } \mathsf{Eval}_{3v}(Q, D) \subseteq \mathsf{certain}_{\mathsf{CWA}}(Q, D)$$

Thus:

- Fast evaluation (checking $\mathsf{Eval}_{3v}(D, Q(\bar{a})) = 1$ in AC^0)
- Correctness guarantees: no false positives

Strong correctness guarantees: involving nulls

How can we give correctness guarantees for tuples with nulls? By a natural extension of the standard definition (proposed in 1984 but quickly forgotten).

A tuple without nulls $\bar{a}$ is a certain answer if

$$\bar{a} \in Q(h(D)) \text{ for every valuation } h \text{ of nulls.}$$

Strong correctness guarantees: involving nulls

How can we give correctness guarantees for tuples with nulls? By a natural extension of the standard definition (proposed in 1984 but quickly forgotten).

A tuple without nulls $\bar{a}$ is a certain answer if

$$\bar{a} \in Q(h(D)) \text{ for every valuation } h \text{ of nulls.}$$

An arbitrary tuple $\bar{a}$ is a certain answers with nulls if

$$h(\bar{a}) \in Q(h(D)) \text{ for every valuation } h \text{ of nulls.}$$

Notation: $\text{certain}_{\perp}(Q, D)$

Certain answers with nulls: properties

$$\mathsf{certain}(Q,D) \quad \subseteq \quad \mathsf{certain}_\perp(Q,D) \quad \subseteq \quad \mathsf{Eval}_{\mathsf{FO}}(Q,D)$$

Moreover:

- $\mathsf{certain}(Q,D)$ is the set of null free tuples in $\mathsf{certain}_\perp(Q,D)$
- $\mathsf{certain}_\perp(Q,D) = \mathsf{Eval}_{\mathsf{FO}}(Q,D)$ for $\mathsf{RelCalc}_{\mathsf{certain}}$ queries

Correctness with nulls: strong guarantees

- D – a database,
- $Q(\bar{x})$ – a first-order query
- $\bar{a}$ – a tuple of elements from D.

Then:

- $\text{Eval}_{3v}(D, Q(\bar{a})) = \mathbf{t} \quad \Longrightarrow \quad \bar{a} \in \text{certain}_{\perp}(Q, D)$
- $\text{Eval}_{3v}(D, Q(\bar{a})) = \mathbf{f} \quad \Longrightarrow \quad \bar{a} \in \text{certain}_{\perp}(\neg Q, D)$

3-valuedness extended to answers: certainly true, certainly false, don't know.

Summary: incomplete information

- Often disregarded and leads to huge problems
- If you write SQL queries, think in 3-valued logic
- Cannot avoid errors, so need to choose which errors to tolerate
- Some types of errors can be eliminated

Inconsistent databases

- Often arise in data integration.
- Suppose have a functional dependency name $\rightarrow$ salary and two tuples (John, 10K) in source 1, and (John, 20K) in source 2.
- One may want to clean data before doing integration.
- This is not always possible.
- Another solution: keep inconsistent records, and try to address the issue later.
- Issue = query answering.

Inconsistent databases cont'd

- Setting:
  - a database D;
  - a set of integrity constraints IC, e.g. keys, foreign keys, functional dependencies etc
  - a query Q
- D violates IC
- What is a proper way of answering Q?
- Certain Answers :

$$\text{certain}_{IC}(Q, D) \;=\; \bigcap_{D_r \text{ is a repair of } D} Q(D_r)$$

Repairs

- How can we repair an instance to make it satisfy constraints?
- If constraints are functional dependencies: say $A \to B$ and we have

A	B	C
a1	b1	c1
a1	b2	c2

  we have to delete one of the tuples.
- If constraints are referential constraints, e.g. $R[A] \subseteq S[B]$ and we have

R:

A	C
a1	c1
a2	c2

S:

B	D
a1	d1
a3	d2

  then we have to add a tuple to S.

Repairs cont'd

- Thus to repair a database to make it satisfy IC we may need to add or delete tuples.
- Given D and D', how far are they from each other?
- A natural measure: the minimum number of deletions/insertions of tuples it takes to get to D' from D.
- In other words,

$$\delta(D, D') \;=\; (D - D') \cup (D' - D)$$

- A repair is a database D' so that
  - it satisfies constraints IC, and
  - there is no D'' satisfying constraints IC with $\delta(D, D'') \subset \delta(D, D')$

How many repairs are there?

Can easily be exponential even for keys: i.e. $\sqrt{2}^N$.

A	B
1	0
1	1
2	0
2	1
...	...
...	...
n	0
n	1

plus key $A \rightarrow B$ $\overset{\text{REPAIR}}{\Rightarrow}$

A	B
1	$\cdot$
2	$\cdot$
...	...
n	$\cdot$

I.e. for $N = 2n$ tuples we have $2^n = \sqrt{2}^N$ repairs.
(A side remark: this construction gives us $\sqrt[c]{c}^n$ repairs for any number c. What is the maximum of $\sqrt[c]{c}$?)

Query answering

- Recall $\text{certain}_{IC}(Q, D) \;=\; \bigcap_{D_r \text{ is a repair of } D} Q(D_r)$.
- Computing all repairs is impractical.
- Hence one tries to obtain a rewriting Q':

$$Q'(D) \;=\; \text{certain}_{IC}(Q, D).$$

- Is this always possible?

Query rewriting: a good case

- One relation $R(A, B, C)$
- Functional dependency $A \rightarrow B$
- Query Q: just return R
- If an instance may violate $A \rightarrow B$, then we can rewrite Q to $R(x, y, z) \wedge \forall u \forall v \ \big(R(x, u, v) \rightarrow u = y\big)$ or

```
SELECT * FROM R
WHERE NOT EXISTS (SELECT * FROM R R1
                  WHERE R.A=R1.A AND R.B ≠ R1.B)
```

- This technique applies to a small class of queries: conjunctive queries without projections, i.e.
  SELECT * FROM R1, R2 ...
  WHERE $\bigwedge R_i.A_j = R_l.A_k$

Query rewriting: a mildly bad case

- One relation $R(A, B)$; attribute A is a key
- Query $Q \ = \ \exists x, y, z \ \big(R(x, z) \wedge R(y, z) \wedge (x \neq y)\big)$
- When are certain answers false ?
- If there is a repair in which the negation of Q is true.
- What is the negation of Q?
  - $\neg Q \ = \ \forall x, y, z \ \big((R(x, z) \wedge R(y, z)) \rightarrow x = y\big)$
- This happens precisely when R contains a perfect matching
- But checking for a perfect matching cannot be expressed in SQL.
- Hence, no SQL rewriting for $\text{certain}_{IC}(Q)$.

Query rewriting: the worst

- One can find an example of a rather simple relational algebra query Q and a set of constraints IC so that the problem of finding

  $$\text{certain}_{IC}(Q, D)$$

  is coNP -complete.
- In general for most types of constraints one can limit the number of repairs but they give rather high complexity bounds
  - typically classes "above" PTIME and contained in PSPACE – hence almost certainly requiring exponential time.

Other approaches

- Repair attribute values.
  - A common example: census data. Don't get rid of tuples but change the values.
  - Distance: sum of absolute values of squares of differences new value – old value
  - Typically one considers aggregate queries and looks for approximations or ranges of their values
- A different notion of repair.
  - Most commonly: the cardinality of $(D - D') \cup (D' - D)$ must be minimum.
  - This is a reasonable measure but the complexity of query answering is high.