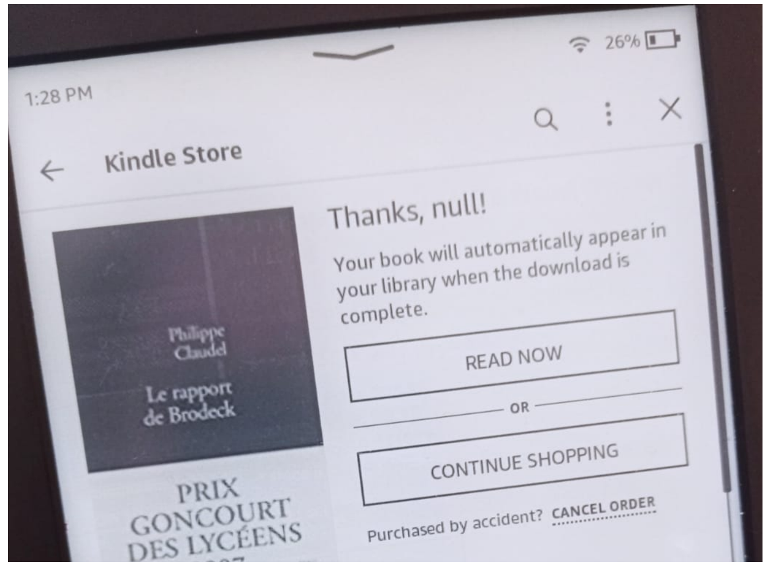
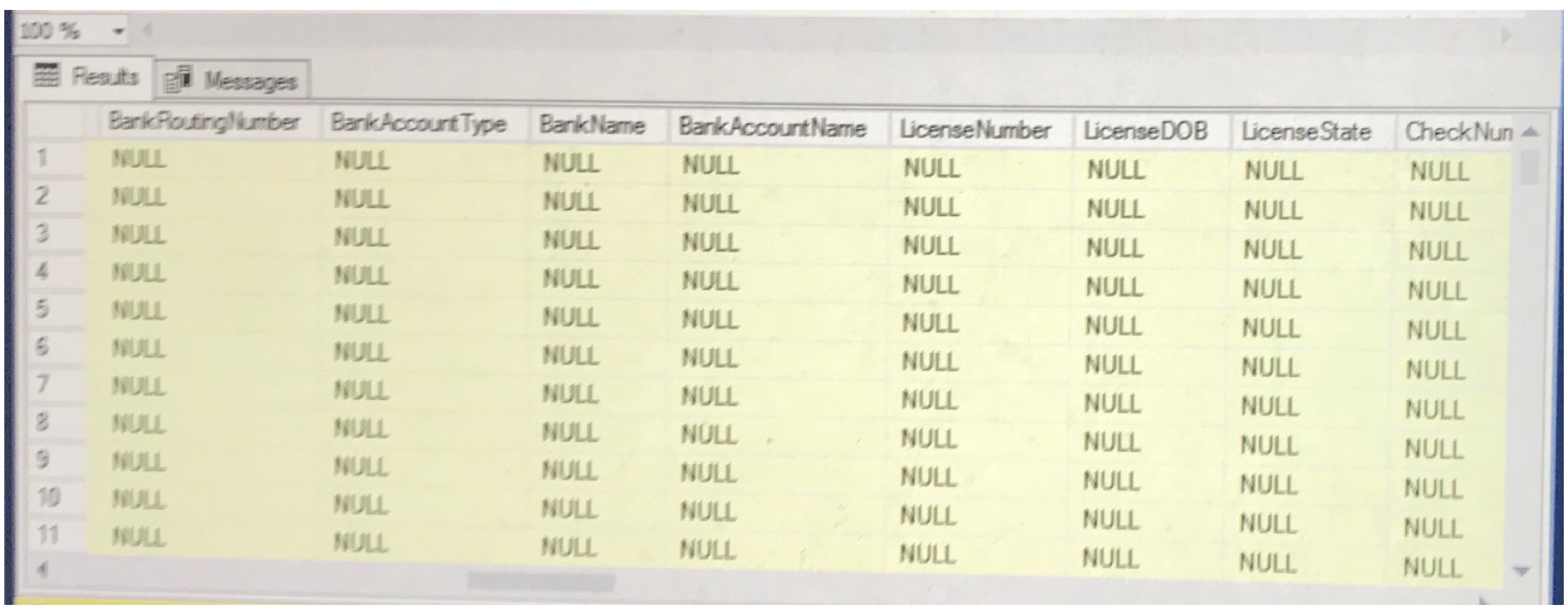
Incomplete Information

What this is about

- Incomplete information in general
- Its handling in SQL in particular
- Why?
    - Because SQL remains the main tool for handling incomplete information
    - Because incomplete information is everywhere
    - And because we know surprisingly little about providing correct answers when all data isn't there
    - Not in practice, and theory is largely lacking

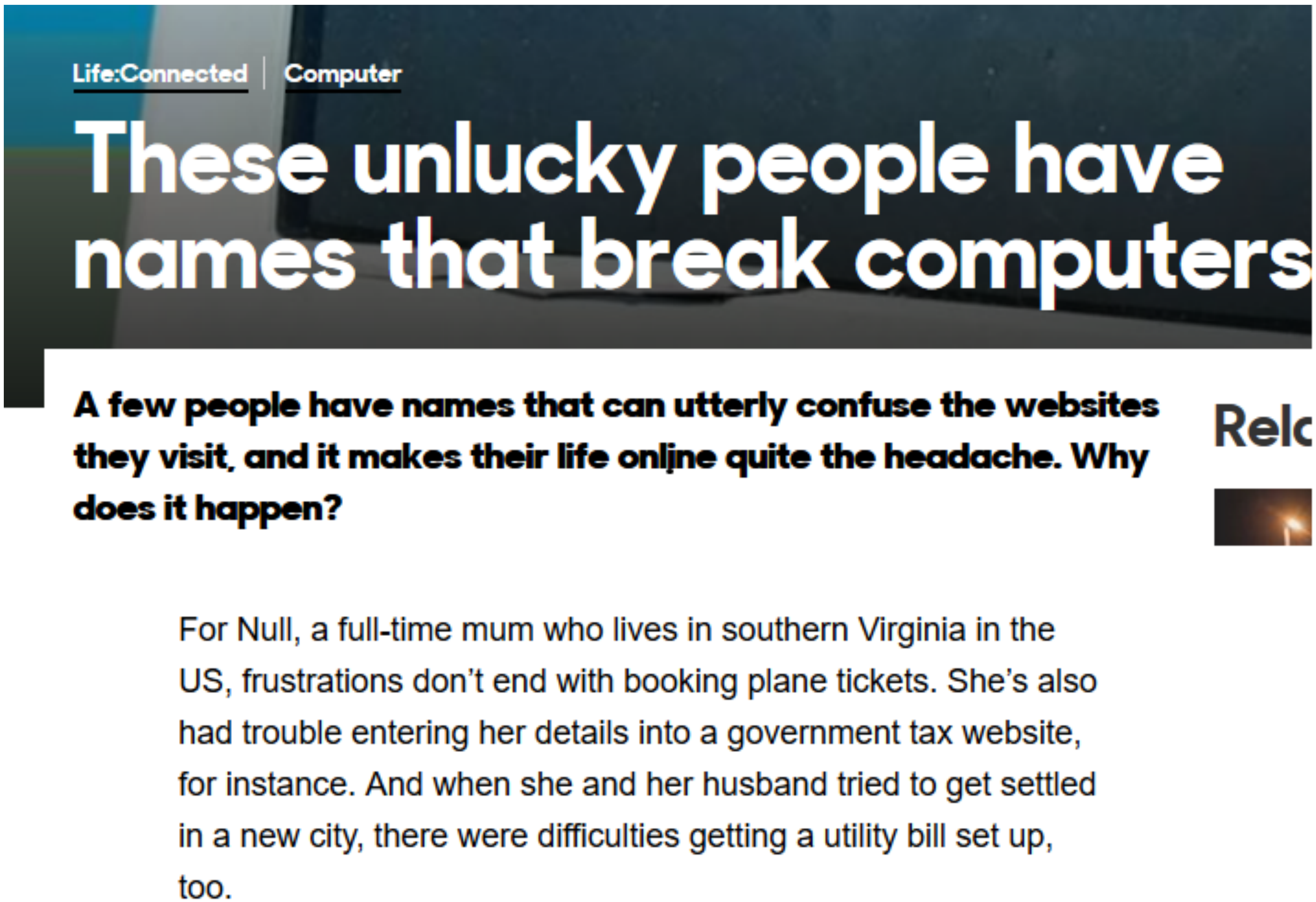
The problematic NULL





	BankRoutingNumber	BankAccountType	BankName	BankAccountName	LicenseNumber	LicenseDOB	LicenseState	CheckNun
1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
2	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
6	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
7	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
8	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
9	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

could create lots of trouble for people:



Life:Connected | Computer

These unlucky people have names that break computers

A few people have names that can utterly confuse the websites they visit, and it makes their life onljne quite the headache. Why does it happen?

Relc

For Null, a full-time mum who lives in southern Virginia in the US, frustrations don't end with booking plane tickets. She's also had trouble entering her details into a government tax website, for instance. And when she and her husband tried to get settled in a new city, there were difficulties getting a utility bill set up, too.

And when nulls appear, things go bad

Textbooks

“fundamentally at odds with the way the world behaves”

“cannot be explained”

Books for database professionals

“wrong answers to your queries”

“all results become suspect”

“can never trust the answers”

News headlines

“Leeds children's heart surgery halted by 'incomplete' data”

“non-existent bills because the companies have incomplete information”

TASK: Relations R(A), S(A)
Compute R - S.

R

A
1
null

S

A
null

Outputs:

Every student will write:

```
select R.A from R where R.A not in (select S.A from S)
```

A

And they are taught it is equivalent to :

```
select R.A from R
where not exists (select S.A from S where S.A=R.A)
```

A
1
null

and that they can do it directly in SQL:

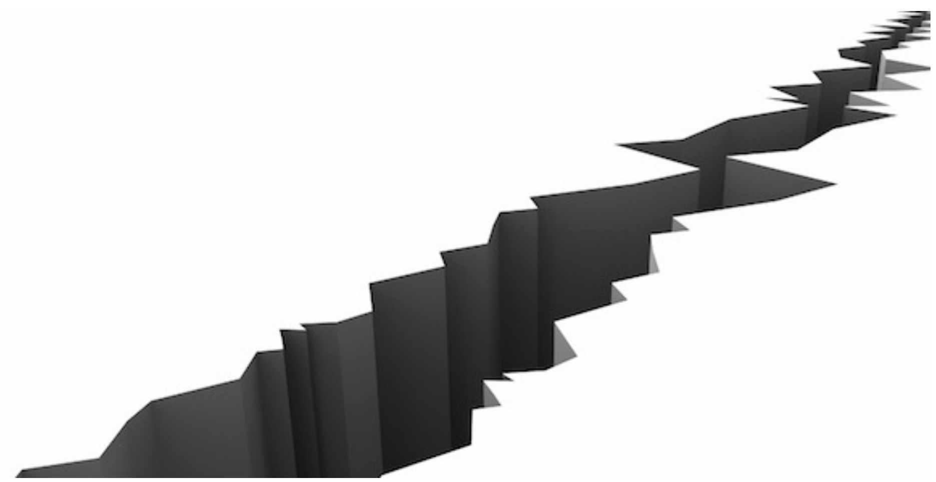
```
select * from r except select * from s
```

A
1

What we have now

THEORY:

correctness, but at a huge cost



PRACTICE:

efficiency, but correctness sacrificed

Correctness: certain answers

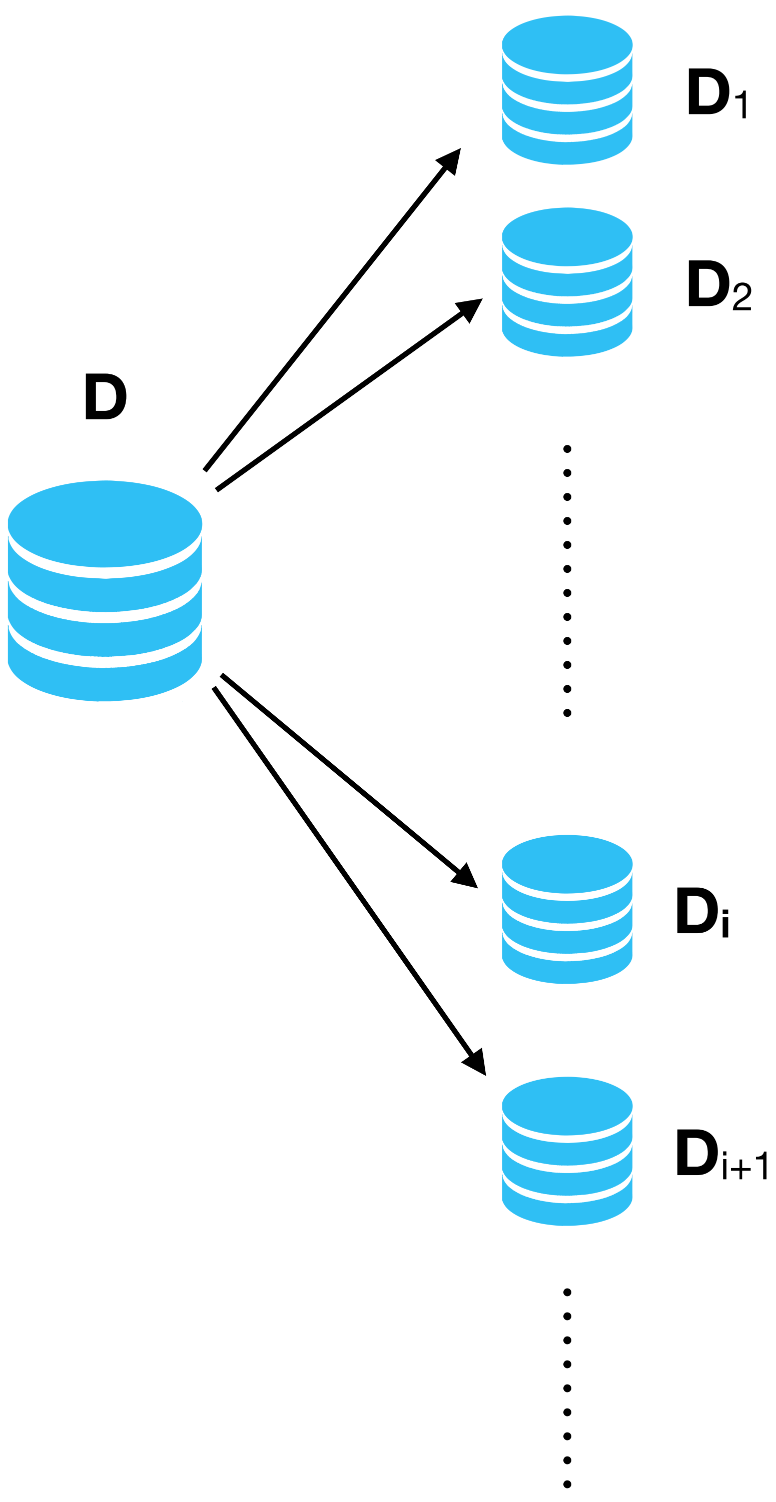
to be defined soon…

Just run queries and hope for the best….

even more than "just run": use a many-valued logic…

Theoretical Approaches

Incomplete data and certain answers

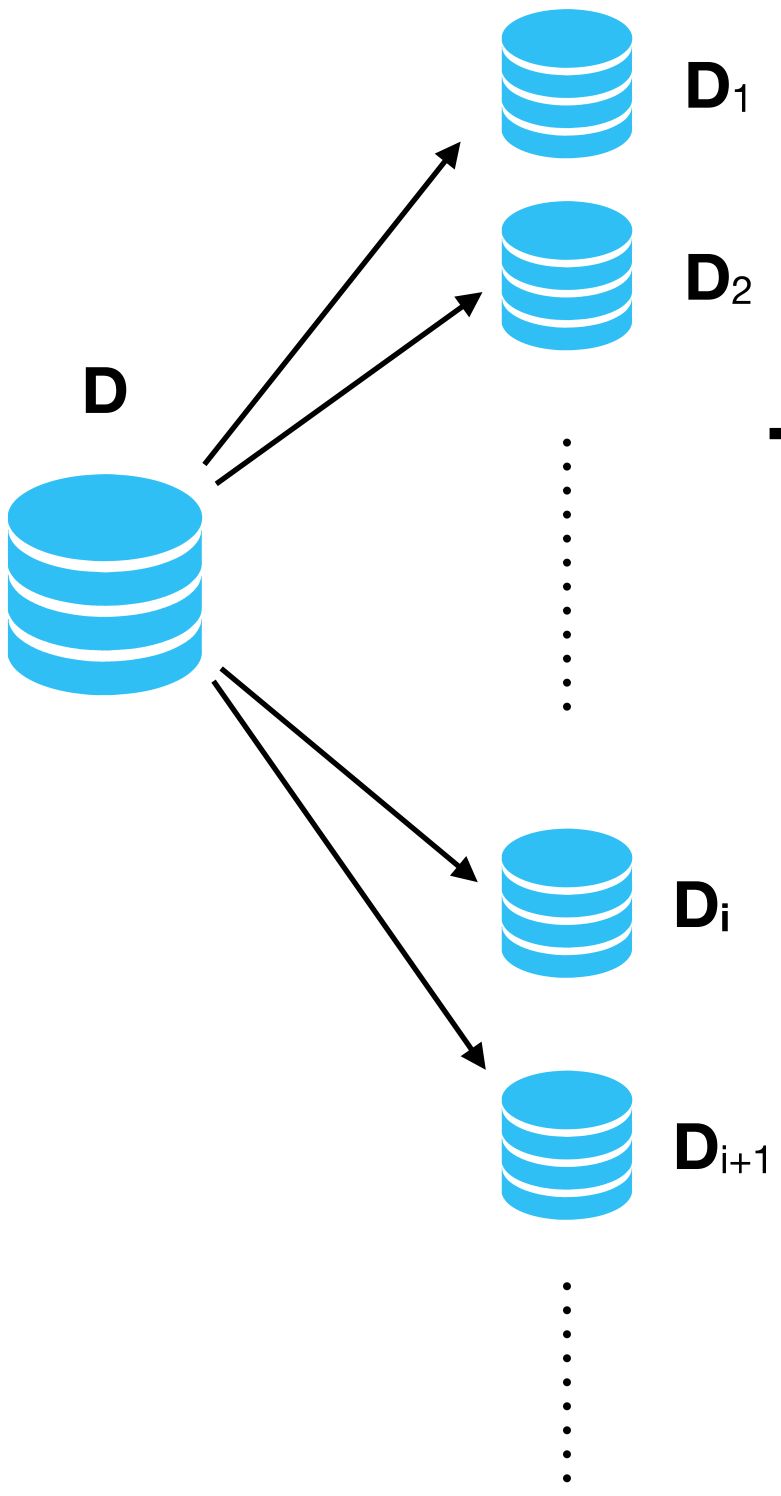


Incomplete database D represents many complete databases D_1, D_2, …

This is done by interpreting incompleteness

For example, by assigning values to every null that occurs in D

Incomplete data and certain answers



Tuple a is certain answer to query Q in D
⇔ a is an answer to Q in every D_i

Certainty is hard computationally:
coNP-hard for relational algebra (first-order logic) queries

The model

Marked nulls - **common in data integration, exchange, OBDA, generalize SQL nulls**

D

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

$\bot_1 \rightarrow 1$

$\bot_2 \rightarrow 2$

$\bot_3 \rightarrow 3$

A	B	C
1	1	2
2	2	3
3	3	1

⋮

$\bot_1 \rightarrow 5$

$\bot_2 \rightarrow 4$

$\bot_3 \rightarrow 7$

A	B	C
1	5	4
2	4	3
3	7	5

Valuations v: Nulls → Constants

Valuations are homomorphisms

- Database elements come from two sets:
  - constants (numbers, strings, etc)
  - nulls, denoted by $\bot_1$ $\bot_2$ $\bot_3$ .....
- Homomorphisms
  - $h(c)=c$ for constants, $h(\bot)$ is a constant or null
  - valuations v: in addition, $v(\bot)$ is always a constant
- **$[\![D]\!] = \{v(D) \mid v \text{ is a valuation}\}$**

Certain Answers

For Boolean queries: Q is certainly true in D ⇔

Q is true in ⟦D⟧ - that is, true in v(D) for each valuation v

For queries returning tuples, for tuples of constants:
c is a certain answer ⇔ c ∈ Q(v(D)) for each valuation v

An arbitrary tuple a is a certain answer ⇔

v(a) ∈ Q(v(D)) for each valuation v

A bit on the history of certain answers

- The definition for constant tuples is often given as $\bigcap\{Q(v(D)) \mid v \text{ is a valuation}\}$
- Issues: let Q that return R (a relation). If all tuples in R have nulls, big intersection is empty. But intuitively the answer should be R itself.
- The third definition, sometimes called certain answers with nulls, proposed in Lipski 1984, but then forgotten for decades in favour of the second (from Lipski 1979)

Certain answers are coNP-complete for first-order queries

- Boolean Q. Certainty is in coNP: Guess a valuation v so that Q is false in $v(D)$.
- Hardness for unions of CQ with negation. Take a graph G with nodes N and edges E.
- For each node $n \in N$, create a new null $\bot_n$. For an edge (n,n'), put $(\bot_n, \bot_{n'})$ in E.
- Query Q: $\exists x\ E(x,x) \vee \exists x,y,z,u$ (x,y,z,u are different)
- Q is certainly true iff the graph is not 3-colorable

A side remark: open world assumption

- The semantics is defined as
  - $[\![D]\!]_{owa} = \{v(D) \cup D' \mid v \text{ is a valuation, } D' \text{ has no nulls}\}$
- Alternatively, $D' \in [\![D]\!]_{owa} \Leftrightarrow D'$ is complete and there is a homomorphism from D to D'
- Then certainty becomes validity, hence undecidable for first-order queries
  - validity over infinite structures is not r.e.

Homomorphism preservation

- For simplicity, look at Boolean queries
- Q is preserved under homomorphisms if $D \models Q$ and $h: D \to D'$ imply $D' \models Q$
- Evaluate Q naively in D (as if nulls were constants). If it is false, then certain answer to Q is false
- If it is true, then it is true in every $D' \in [\![D]\!]$ because we have a homomorphism $D \to D'$, and certain answer is true.
- For queries preserved under homomorphisms, naive evaluation gives certain answers.
- For non-Boolean queries, it gives certain answers with nulls.

Queries preserved under homomorphisms

- Rossman's Theorem: a first-order (FO) query is preserved under homomorphism iff it is equivalent to a union of conjunctive queries

- Hence, for UCQs, naive evaluation gives certain answers.

- Under open world assumption, converse is true: if naive evaluation gives certain answers for an FO query, then is equivalent to a UCQ.

But generally we can do better

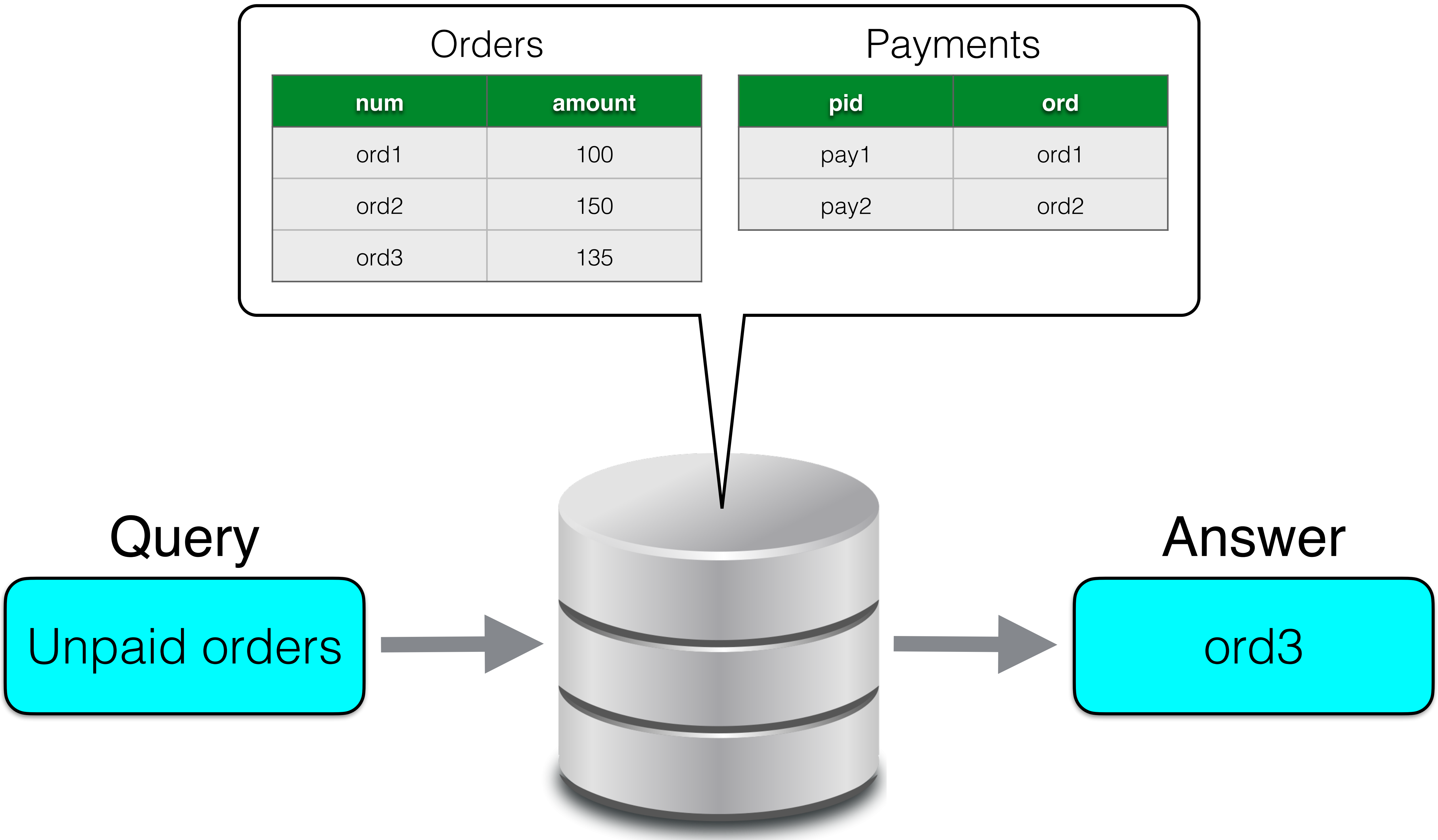
- Recall $[\![D]\!] = \{v(D) \mid v \text{ is a valuation}\}$
- We have special homomorphisms: $D \rightarrow v(D)$
- They are called strong onto homomorphisms in logic and model theory
- Theorem: If Q is preserved under strong onto homomorphisms, then naive evaluation produces certain answers with nulls

Preservation under strong onto homomorphisms

- In logic (FO), an extension of the positive fragment (without negation)
  - closure of atoms $R(\mathbf{x})$ and $x=y$ under $\vee$ $\wedge$ $\exists$ $\forall$ and the rule $\forall \mathbf{x}\ (R(\mathbf{x}) \rightarrow \alpha(\mathbf{x},\mathbf{y}))$
- In relational algebra (RA)
  - selection, projection, cartesian product, union, and division by a relation $(Q \div R)$
  - Division queries: "find students that take all courses"

But what do we do with more complex queries?

- First, let's see a bit what happens in everyday practice…



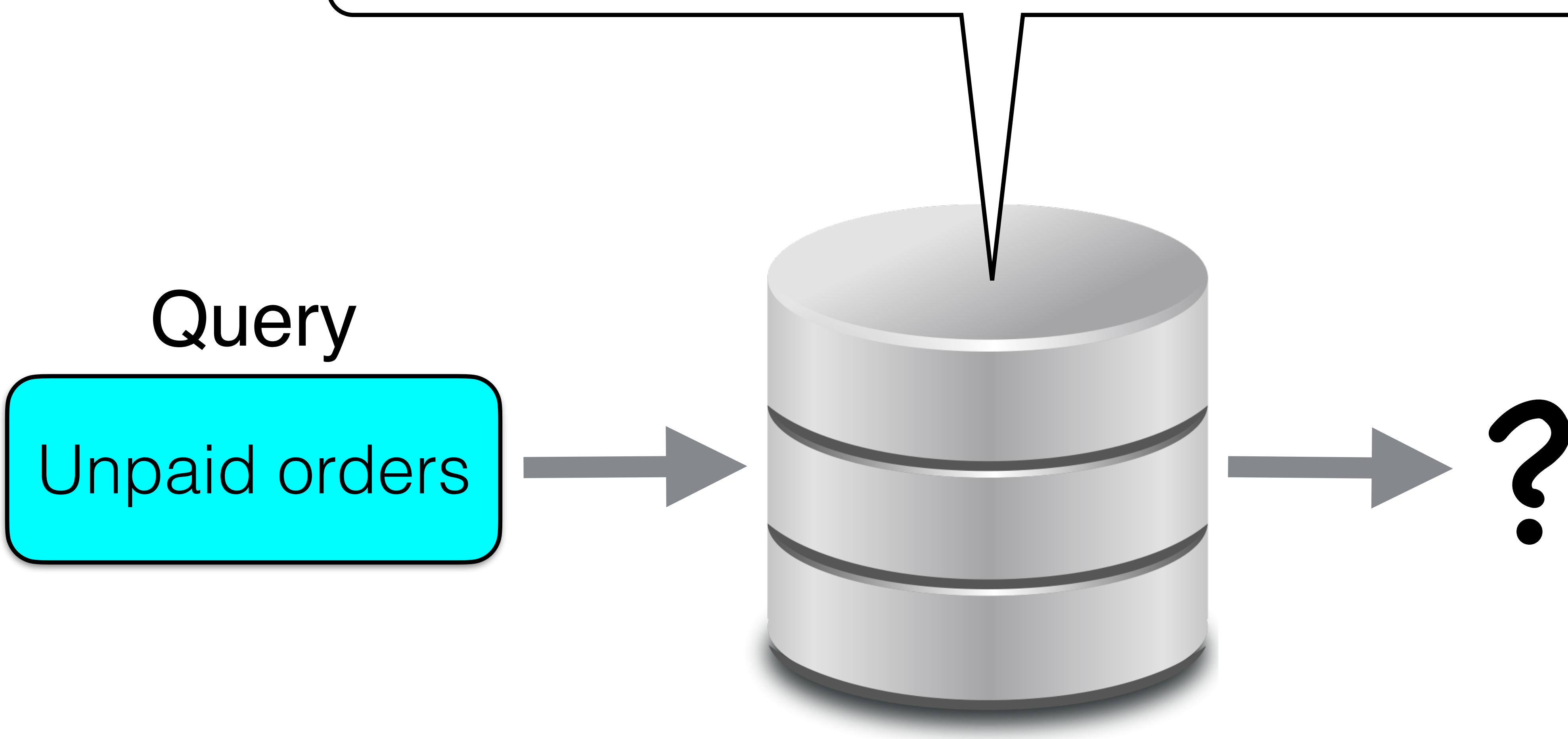
Orders
num
amount
ord1
100
ord2
150
ord3
135
Payments
pid
ord
pay1
ord1
pay2
ord2
Query
Unpaid orders
Answer
ord3

Orders

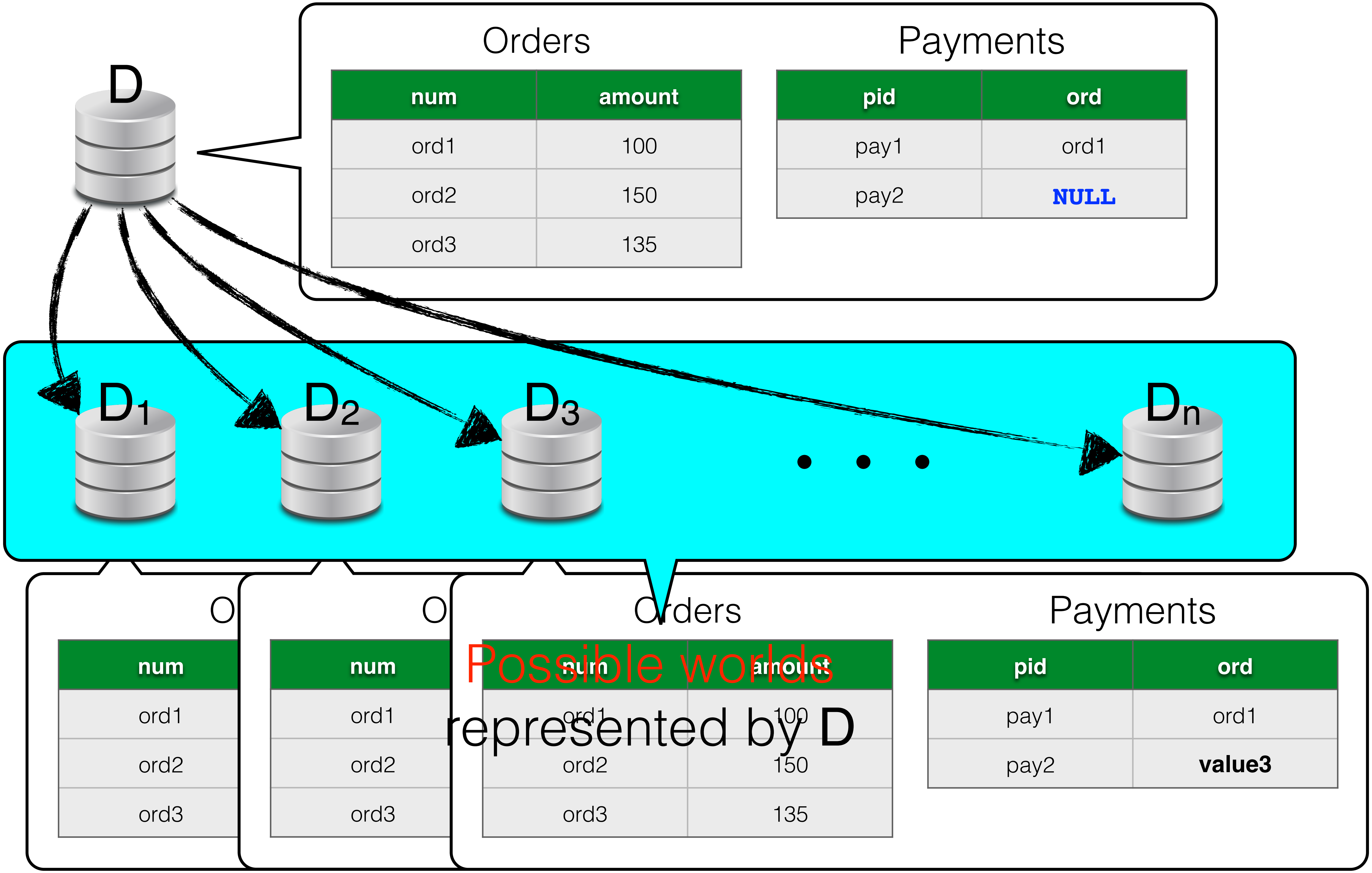
num	amount
ord1	100
ord2	150
ord3	135

Payments

pid	ord
pay1	ord1
pay2	NULL



Incomplete databases



Orders

num	amount
ord1	100
ord2	150
ord3	135

Payments

pid	ord
pay1	ord1
pay2	**NULL**

Orders

num	amount
ord1	100
ord2	150
ord3	135

Payments

pid	ord
pay1	ord1
pay2	**value3**

Querying incomplete databases

Certain answers:

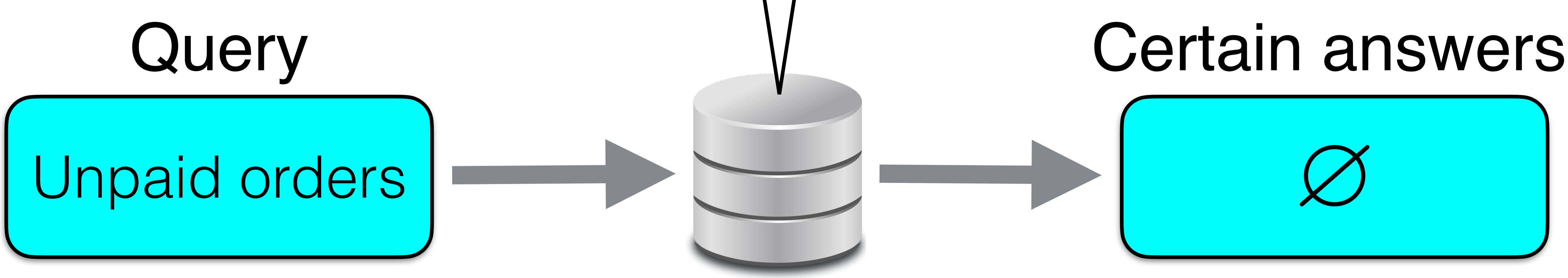
Answers that are true in all possible worlds

Orders

num	amount
ord1	100
ord2	150
ord3	135

Payments

pid	ord
pay1	ord1
pay2	NULL



Querying incomplete databases

Unpaid orders:

```
SELECT O.num FROM Orders O WHERE NOT EXISTS (
  SELECT * FROM Payments P WHERE P.ord = O.num )
```

Orders

num	amount
ord1	100
ord2	150
ord3	135

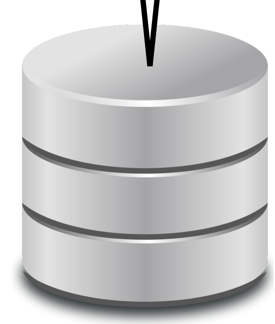
Payments

pid	ord
pay1	ord1
pay2	**NULL**

Query

Unpaid orders

SQL



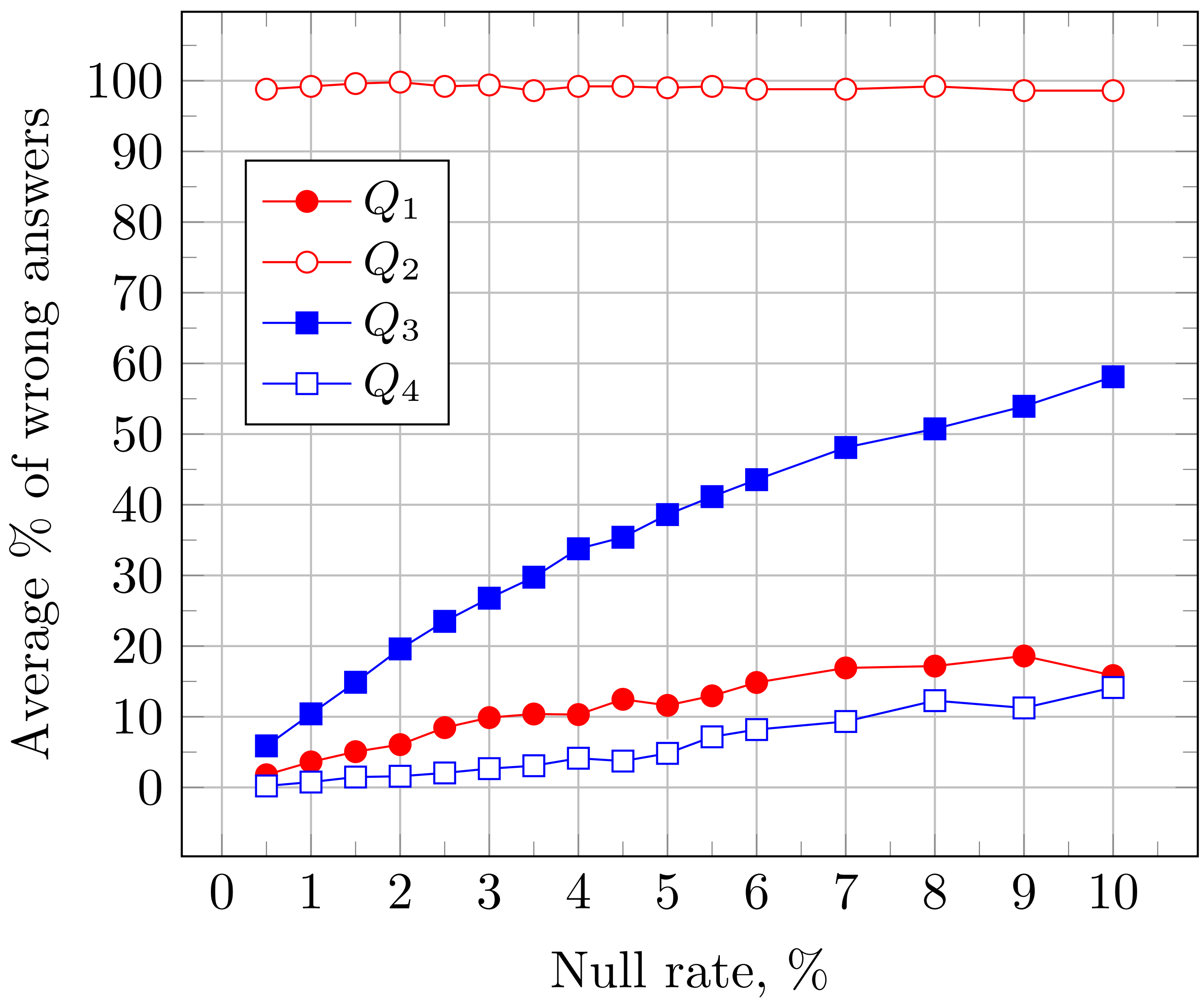
SQL

Certain answers $\neq$

ord2, ord3

Are wrong answers common in SQL?

Experiment on the TPC-H Benchmark:
models a business scenario with associated decision support queries



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 null

Customer

CUST_ID	NAME
c1	John
c2	Mary

Typical queries, as we teach students to write them:

In the real world, information is often missing

Unpaid orders:

```
select O.order_id
from Orders O
where O.order_id not in
        (select order from Pay P)
```

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: Ord3.

Old Answer: Ord3 New: NONE!

Answer: none.

Old answer: none New: c2!

What's the deal with nulls?

- Back in the 1980s, when SQL was standardized, it chose a 3-valued logic for handling nulls
  - truth values: **t, f, u**   **u** for unknown
  - conditions such as 1 = null evaluate to **u**
  - propagated using Kleene's logic:

	$\neg$
t	**f**
f	**t**
u	**u**

$\wedge$	**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**

Types of errors

- False negatives: miss some of the correct answers
- False positives: return answers that are false
- False positives are worse: blatant lie vs hiding some of the truth
- Correct answers: those that are **certain**
  - don't depend on the interpretation of missing data
- SQL gives both types of errors

Avoiding wrong answers

- Nothing prevents us from finding an efficient query evaluation that avoids false positives
- Surprisingly not known until very recently
- Idea: translate query Q into queries Q^t that returns certainly true answers and Q^f that returns certainly false answers.
- Underapproximates certainly true/false answers, overapproximates unknown

The Q^t translation

$$
\begin{aligned}
R^{\mathbf{t}} &= R \\
(\sigma_\theta(Q))^{\mathbf{t}} &= \sigma_{\theta^*}(Q^{\mathbf{t}}) \\
(\pi_\alpha(Q))^{\mathbf{t}} &= \pi_\alpha(Q^{\mathbf{t}}) \\
(Q_1 \times Q_2)^{\mathbf{t}} &= Q_1^{\mathbf{t}} \times Q_2^{\mathbf{t}} \\
(Q_1 \cup Q_2)^{\mathbf{t}} &= Q_1^{\mathbf{t}} \cup Q_2^{\mathbf{t}} \\
(Q_1 \cap Q_2)^{\mathbf{t}} &= Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{t}} \\
(Q_1 - Q_2)^{\mathbf{t}} &= Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}}
\end{aligned}
$$

$$
\begin{aligned}
(A = B)^* &= (A = B) \\
(A \neq B)^* &= (A \neq B) \land \text{not\_null}(A) \land \text{not\_null}(B) \\
(\theta_1 \text{ op } \theta_2)^* &= \theta_1^* \text{ op } \theta_2^* \qquad \text{for op} \in \{\land, \lor\}
\end{aligned}
$$

A tuple is certainly in $Q_1 - Q_2$ if it is certainly in Q_1 and certainly not in Q_2

The problematic Q^f translation

Need an extra operation of left unification (anti)semijoin

$$R \ltimes_u S \quad = \quad \{\, \bar{r} \in R \mid \exists\, \bar{s} \in S : \bar{r} \text{ unifies with } \bar{s} \,\}$$

$$R \,\overline{\ltimes}_u\, S \quad = \quad R - R \ltimes_u S$$

Inefficient translations:

$$R^{\boldsymbol{f}} = adom^{\text{arity(R)}} \,\overline{\ltimes}_u\, R$$

$$(\sigma_\theta(Q))^{\boldsymbol{f}} = Q^{\boldsymbol{f}} \cup \sigma_{(\neg\theta)^*}(adom^{\text{arity(Q)}})$$

$$(Q_1 \times Q_2)^{\boldsymbol{f}} = Q_1^{\boldsymbol{f}} \times adom^{\text{arity}(Q_2)} \;\cup\; adom^{\text{arity}(Q_1)} \times Q_2^{\boldsymbol{f}}$$

$$(\pi_{\boldsymbol{\alpha}}(Q))^{\boldsymbol{f}} = \pi_{\boldsymbol{\alpha}}(Q^{\boldsymbol{f}}) - \pi_{\boldsymbol{\alpha}}(adom^{\text{arity}(Q)} - Q^{\boldsymbol{f}})$$

Has no chance of working in practice

A different perspective

$$(Q_1 - Q_2)^{\mathbf{t}} \ = \ Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}}$$

A tuple is certainly in **Q_1 – Q_2** if it is certainly in **Q_1** **and** certainly not in **Q_2**

But this is not the only possibility

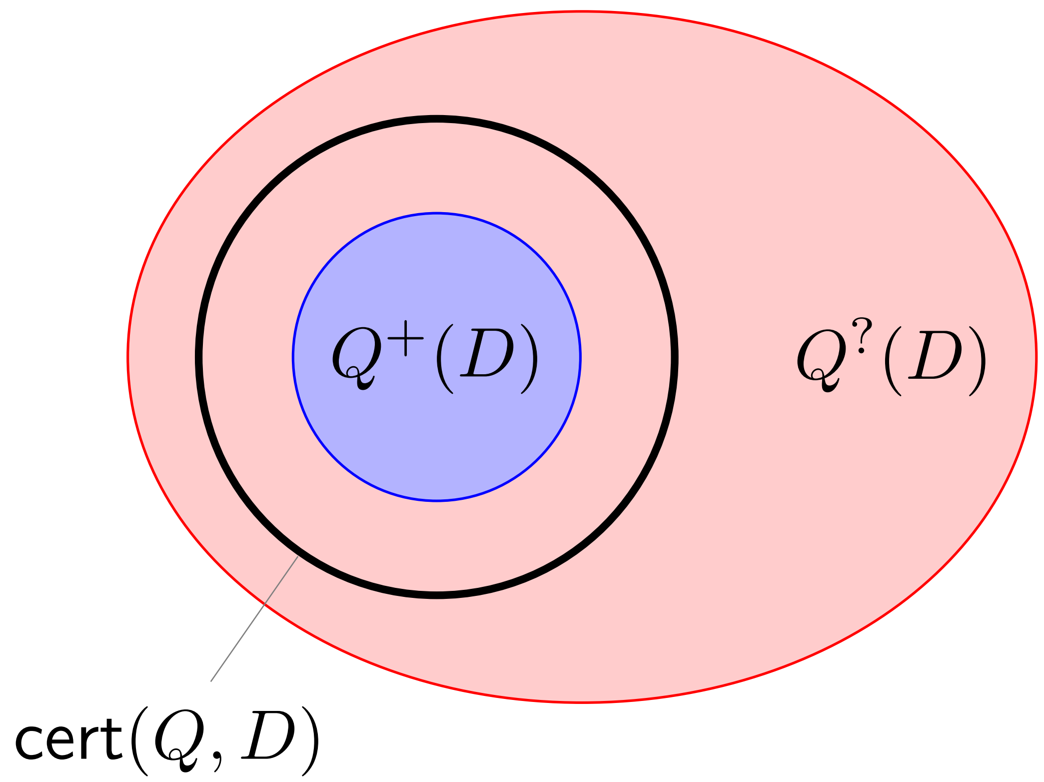
A tuple is certainly in **Q_1 – Q_2** if

- it is certainly in **Q_1** and
- it does not match any tuple that could be in **Q_2**

Improved translation

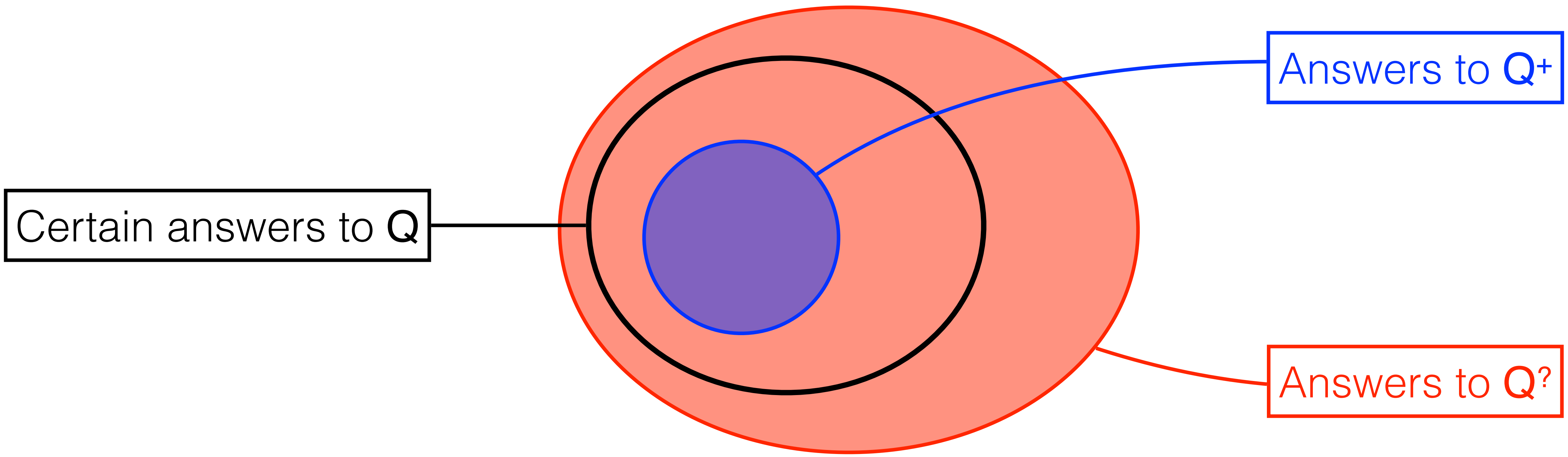
Translate **Q** into (Q^{+}, $Q^{?}$) where

- Q^{+} approximates certain answers
- $Q^{?}$ represents possible answers
- Both queries have AC^0 data complexity



The + / ? approximation scheme

$$Q \mapsto (\, Q^{+},\ Q^{?} \,)$$



The + / ? approximation scheme

$$
\begin{aligned}
R^{+} &= R\\
\big(\sigma_\theta(Q)\big)^{+} &= \sigma_{\theta^*}(Q^{+})\\
\big(\pi_\alpha(Q)\big)^{+} &= \pi_\alpha(Q^{+})\\
\big(Q_1 \times Q_2\big)^{+} &= Q_1^{+} \times Q_2^{+}\\
\big(Q_1 \cup Q_2\big)^{+} &= Q_1^{+} \cup Q_2^{+}\\
\big(Q_1 \cap Q_2\big)^{+} &= Q_1^{+} \cap Q_2^{+}\\
\big(Q_1 - Q_2\big)^{+} &= Q_1^{+}\ \overline{\ltimes}_{\mathsf{u}}\ Q_2^{?}
\end{aligned}
$$

$$
\begin{aligned}
R^{?} &= R\\
\big(\sigma_\theta(Q)\big)^{?} &= \sigma_{\neg(\neg\theta)^*}(Q^{?})\\
\big(\pi_\alpha(Q)\big)^{?} &= \pi_\alpha(Q^{?})\\
\big(Q_1 \times Q_2\big)^{?} &= Q_1^{?} \times Q_2^{?}\\
\big(Q_1 \cup Q_2\big)^{?} &= Q_1^{?} \cup Q_2^{?}\\
\big(Q_1 \cap Q_2\big)^{?} &= Q_1^{?} \ltimes_{\mathsf{u}} Q_2^{?}\\
\big(Q_1 - Q_2\big)^{?} &= Q_1^{?} - Q_2^{+}
\end{aligned}
$$

The + / ? approximation: performance

- Normally one would not expect to outperform native SQL that does not care about correctness.

- We observed 3 types of behaviour:

  - most commonly, a small overhead (3-4%), very acceptable

  - sometimes it outperforms SQL significantly (when the original query spends all the time looking for wrong answers)

  - Sometimes it lags behind. Reason: case analysis, what is null and what is not, and this leads to disjunction in queries. SQL's well-kept secret: it does not optimize disjunctions.

SQL and 3VL (3-valued logic)

- Constant source of confusion for programmers
- Committee design, just to handle nulls
- Heavily criticized ever since
- But was the right many-valued logic chosen?
- First one more example of confusion.

R

A
1
2

S

A
2

Compute **R – S**

Answer

Q$_1$

```
SELECT R.A FROM R
EXCEPT
SELECT S.A FROM S
```

A
1

Q$_2$

```
SELECT R.A FROM R
WHERE  R.A NOT IN (
  SELECT S.A FROM S )
```

A
1

Q$_3$

```
SELECT R.A FROM R
WHERE  NOT EXISTS (
  SELECT S.A FROM S
  WHERE  S.A=R.A )
```

A
1

R

A
1
NULL

S

A
NULL

Compute **R – S**

Answer

Q_1

```
SELECT R.A FROM R
EXCEPT
SELECT S.A FROM S
```

A
1

Q_2

```
SELECT R.A FROM R
WHERE  R.A NOT IN (
  SELECT S.A FROM S )
```

A

Q_3

```
SELECT R.A FROM R
WHERE  NOT EXISTS (
  SELECT S.A FROM S
  WHERE  S.A=R.A )
```

A
1
NULL

Why this happens

- EXCEPT treats NULL syntactically: this is the usual set difference, hence {1,NULL} EXCEPT {NULL} = {1}

- NOT IN uses 3VL:
  1 NOT IN {NULL} = NOT (1 IN {NULL}) = NOT (1=NULL) = NOT(UNKNOWN) = UNKNOWN
  and hence 1 is not selected.

- NOT EXISTS: mix of 2VL and 3VL. First,
  (SELECT A FROM S WHERE A=1) =
  (SELECT A FROM S WHERE NULL=1)
  returns empty table as NULL=1 is UNKNOWN. Then
  NOT EXISTS (SELECT A FROM S WHERE A=1) returns true and 1 is selected.

Questions about SQL's 3VL

- **Did they choose the right many-valued logic?**
- **Did they really have to use a many-valued logic?**
    - people prefer to think — and write programs — with just true and false

Which logic we are talking about?

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
or not(O.date = $today))

Predicate Logic

Propositional Logic

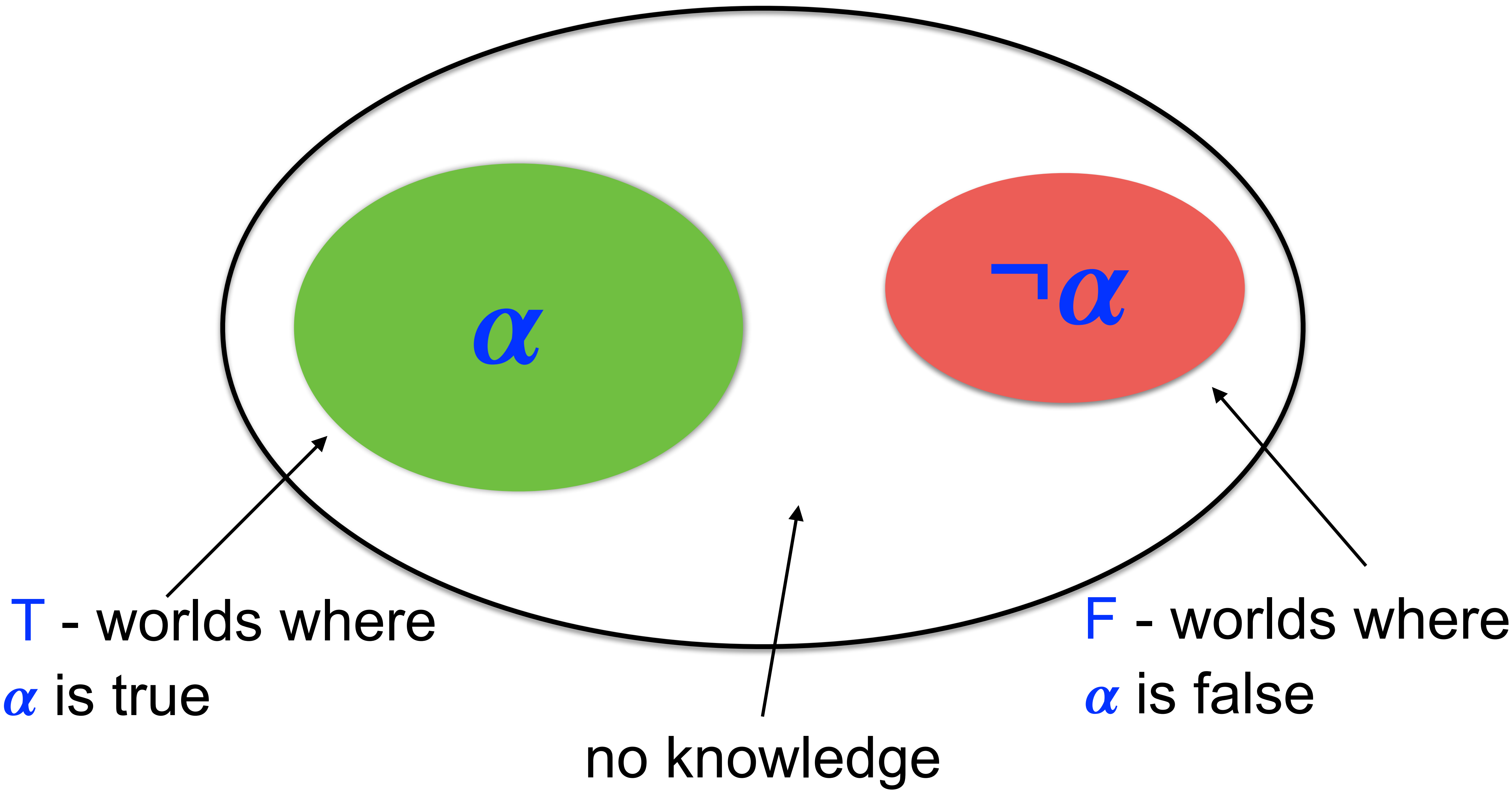
not exists: ¬∃ (or ∀)   select = ∃

Core SQL = First-Order Predicate Logic
Conditions in Queries = Propositional Logic

Choosing Propositional Logic: Idea

- An incomplete database can represent many completions — possible worlds
- Let's look at what can be known about an atomic proposition α in those worlds

W — set of possible worlds

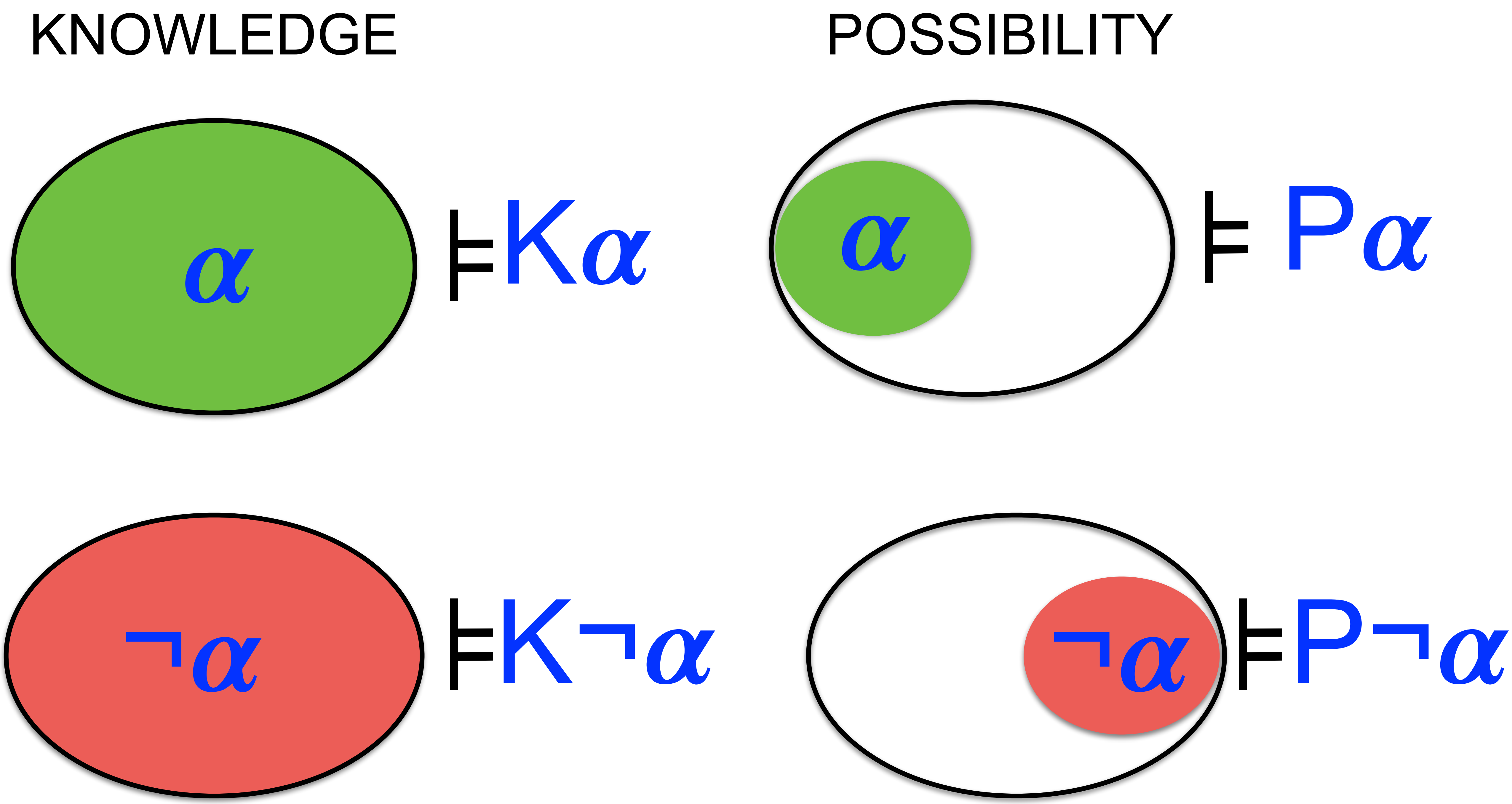


(W, T, F) — describes what we know about α

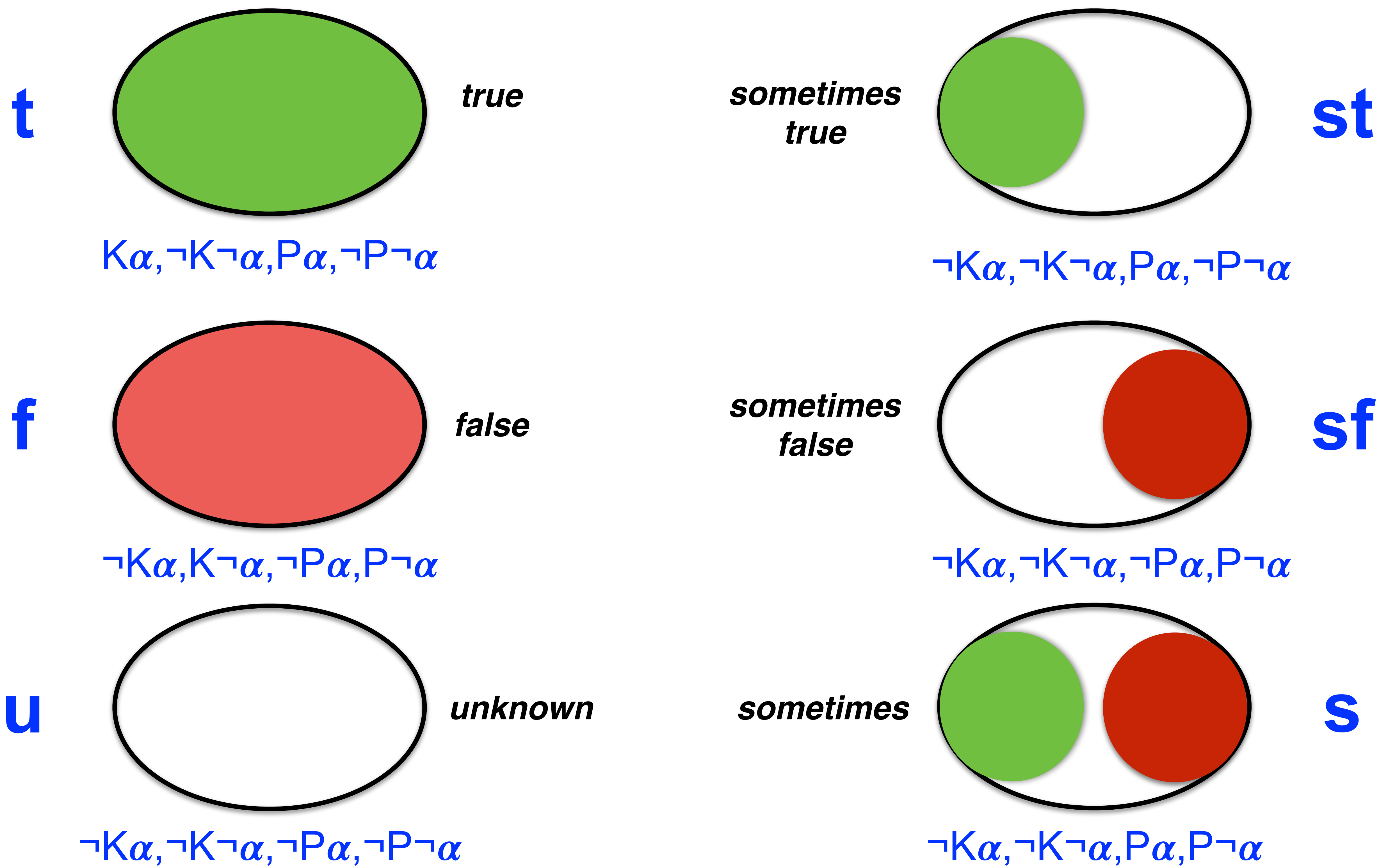
This idea was used before

- Work on bilattice-based many-valued logics
  - Each such description is treated as truth value
- Too many values that convey the same information
- A better idea: a truth value is the epistemic theory of a description (W, T, F)
  - maximally consistent theory

Building Blocks



Truth Values



Truth tables

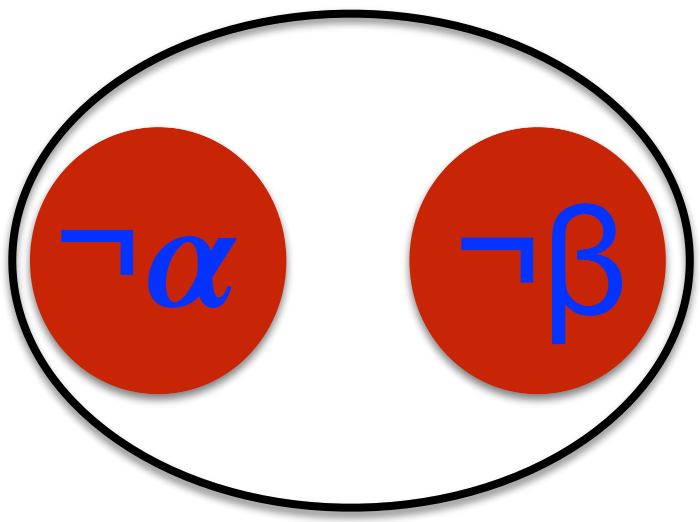
- $Th(\tau,\alpha)$ - the maximally consistent theory for truth value τ and proposition α
- If $\sigma = \omega(\tau,\tau')$, then

  $Th(\tau,\alpha) \wedge Th(\tau',\beta) \wedge Th(\sigma,\omega(\alpha,\beta))$

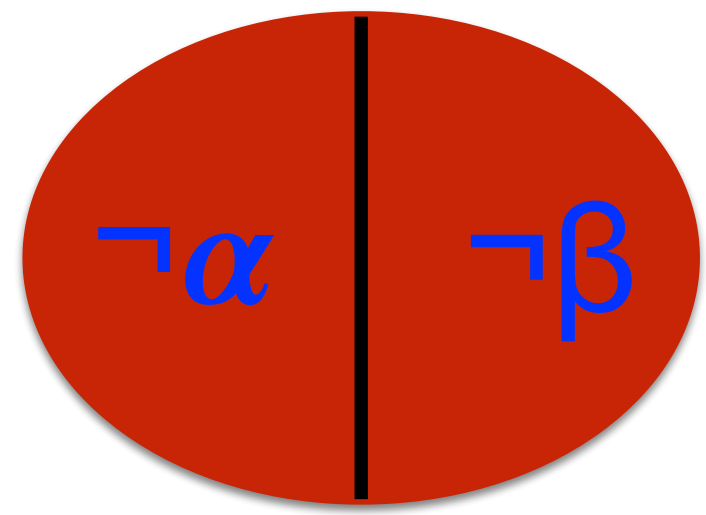
  must be consistent for all α and β.
- Such σ is not unique
  - but we need the most general one

More general truth value: **sf** ∧ **sf**

α : **sf** β : **sf**



$\alpha \wedge \beta$: **sf**



$\alpha \wedge \beta$: **f**

sf ∧ **sf** is consistent with both **sf** and **f**

but **sf** is more general than **f**

Truth tables for 6-valued logic

∧	t	f	s	st	sf	u
t	t	f	s	st	sf	u
f	f	f	f	f	f	f
s	s	f	sf	sf	sf	sf
st	st	f	sf	u	sf	u
sf	sf	f	sf	sf	sf	sf
u	u	f	sf	u	sf	u

∨	t	f	s	st	sf	u
t	t	t	t	t	t	t
f	t	f	s	st	sf	u
s	t	s	st	st	st	st
st	t	st	st	st	st	st
sf	t	sf	st	st	u	u
u	t	u	st	st	u	u

	¬
t	f
f	t
s	s
st	sf
sf	st
u	u

Do SQL programmers need to memorize this now?

Not yet: these truth tables break distributivity and idempotence

And database optimizers need them (for elimination of redundant subexpressions and operations)

$$\mathbf{sf} = \mathbf{s} \wedge (\mathbf{s} \vee \mathbf{s}) \neq (\mathbf{s} \wedge \mathbf{s}) \vee (\mathbf{s} \wedge \mathbf{s}) = \mathbf{u}$$

The propositional answer

The only maximal sublogic of the 6-valued logic that

(a) has truth value **t**
(b) $\wedge$ and $\vee$ are idempotent and distributive

is SQL's 3-valued Kleene's logic

So it appears ISO JTC1 SC32 WG3 was right after all?
Wait a bit…

Reminder

```
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
                or not( O.date = $today) )
```

Predicate Logic

Propositional Logic

Core SQL = First-Order Predicate Logic

over….

What are nulls?

- SQL has a single null value — NULL
- In applications (OBDA, data integration, etc) one uses marked nulls $\bot_1, \bot_2, \bot_3, \ldots$

How to interpret atoms?

Standard 2-valued semantics: $R(\mathbf{a}) = \begin{cases} \mathbf{t} & \text{if } \mathbf{a} \in R \\ \mathbf{f} & \text{if } \mathbf{a} \notin R \end{cases}$

SQL semantics: $(a{=}b) = \begin{cases} \mathbf{t} & \text{if } a,b \neq \text{NULL and } a=b \\ \mathbf{f} & \text{if } a,b \neq \text{NULL and } a \neq b \\ \mathbf{u} & \text{if } a \text{ or } b \text{ is NULL} \end{cases}$

Unification semantics

$$R(\mathbf{a}) = \begin{cases} \mathbf{t} & \text{if } \mathbf{a} \in R \\ \mathbf{f} & \text{if does not unify with any } \mathbf{b} \in R \\ \mathbf{u} & \text{if } \mathbf{a} \notin R \text{ and } \mathbf{a} \text{ unifies with some } \mathbf{b} \in R \end{cases}$$

Let’s look at SQL first…

- A single null value
- 2-valued semantics for R(**a**), SQL semantics for (a=b)
- … and imagine we can rewrite history

A logician's approach

- First Order Logic (FO)
  - domain has usual values and NULL
  - Syntactic equality: NULL = NULL but NULL ≠ 1 etc
  - Boolean logic rules for $\wedge$, $\vee$, $\neg$
  - Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction
- Why would one even think of anything else??

What did SQL do?

- 3-valued FO (a textbook version)
  - domain has usual values and NULL
  - comparisons with NULL result in **unknown**
  - Kleene logic rules for $\wedge$, $\vee$, $\neg$
  - Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction
- Seemingly more expressive.
- But does it correspond to reality?

SQL logic is NOT 2-valued or 3-valued: it's a mix

- Conditions in WHERE are evaluated under 3-valued logic. But then only those evaluated to true matter.

- Studied before for propositional logic:

  - In 1939, Russian logician Bochvar wanted to give a formal treatment of logical paradoxes. To assert that something is true, he introduced a new connective: $\uparrow$**p** means that p is true.

- Amazingly, 40 years later SQL adopted the same idea.

What did SQL really do?

- 3-valued FO with ↑:
  - As textbook version but with the extra connective ↑

$$\uparrow\varphi = \begin{cases} \mathbf{t}, & \text{if } \varphi \text{ is } \mathbf{t} \\ \mathbf{f}, & \text{if } \varphi \text{ is } \mathbf{f} \text{ or } \mathbf{u} \end{cases}$$

What *is* the logic of SQL?

- We have:
  - logician's 2-valued FO
  - 3-valued FO (Kleene logic)
  - 3-valued FO + Bochvar's assertion (SQL logic)
- **AND THEY ARE ALL THE SAME!**

Collapse to Boolean FO

- There is a much more general result
  - Any set of nulls: SQL, marked…
  - Any propositional many-valued logic $\mathcal{L}$
  - Any semantics — Boolean, SQL, unification, can mix and use different ones for different atoms
- **First-Order predicate logic based on $\mathcal{L}$ collapses to the usual Boolean FO predicate logic**

2-valued SQL

Idea — 3 simultaneous translations:

- conditions $P \longrightarrow P^t$ and P^f
- Queries $Q \longrightarrow Q'$

P^t and P^f are Boolean conditions: P^t / P^f is true iff P under 3-valued logic is true / false.

In Q' we simply replace P by P^t

2-valued SQL: translation

$$
\begin{aligned}
P(\bar{t})^{\mathbf{t}} &= P(\bar{t}) & P(t_1,\ldots,t_k)^{\mathbf{f}} &= \texttt{NOT}\ P(t_1,\ldots,t_k)\ \texttt{AND}\ \bar{t}\ \texttt{IS NOT NULL} \\
(\texttt{EXISTS}\ Q)^{\mathbf{t}} &= \texttt{EXISTS}\ Q' & (\texttt{EXISTS}\ Q)^{\mathbf{f}} &= \texttt{NOT EXISTS}\ Q' \\
(\theta_1 \wedge \theta_2)^{\mathbf{t}} &= \theta_1^{\mathbf{t}} \wedge \theta_2^{\mathbf{t}} & (\theta_1 \wedge \theta_2)^{\mathbf{f}} &= \theta_1^{\mathbf{f}} \vee \theta_2^{\mathbf{f}} \\
(\theta_1 \vee \theta_2)^{\mathbf{t}} &= \theta_1^{\mathbf{t}} \vee \theta_2^{\mathbf{t}} & (\theta_1 \vee \theta_2)^{\mathbf{f}} &= \theta_1^{\mathbf{f}} \wedge \theta_2^{\mathbf{f}} \\
(\neg\theta)^{\mathbf{t}} &= \theta^{\mathbf{f}} & (\neg\theta)^{\mathbf{f}} &= \theta^{\mathbf{t}} \\
(t\ \texttt{IS NULL})^{\mathbf{t}} &= t\ \texttt{IS NULL} & (t\ \texttt{IS NULL})^{\mathbf{f}} &= t\ \texttt{IS NOT NULL} \\
(\bar{t}\ \texttt{IN}\ Q)^{\mathbf{t}} &= \bar{t}\ \texttt{IN}\ Q' & \big((t_1,\ldots,t_n)\ \texttt{IN}\ Q\big)^{\mathbf{f}} &= \texttt{NOT EXISTS}\ \big(\texttt{SELECT}\ \star\ \texttt{FROM}\ Q'\ \texttt{AS}\ N(A_1,\ldots,A_n)\ \texttt{WHERE} \\
& & & \quad (t_1\ \texttt{IS NULL OR}\ A_1\ \texttt{IS NULL OR}\ t_1 = N.A_1)\ \texttt{AND}\ \cdots \\
& & & \quad \cdots\ \texttt{AND}\ (t_n\ \texttt{IS NULL OR}\ A_n\ \texttt{IS NULL OR}\ t_n = N.A_n)\big)
\end{aligned}
$$

Idea of the translation

- When does (A=B) evaluate to false in SQL?
  - When A, B are not nulls and A ≠ B
- Hence translation (A IS NOT NULL) AND (B IS NOT NULL) AND NOT (A=B)
- Bottom line: case analysis with IS NULL and IS NOT NULL makes it possible to eliminate 3VL.

Predicate logic answer

- **No, they did not need to use many-valued logic!**
- But what now?
  - We can’t change the way SQL is: too much legacy code, issues with optimization
  - But new languages are being designed, and they do not need to follow the SQL path

More on nulls in SQL

- SQL: not marked nulls
- A single NULL for all purposes
  - Unknown value
  - Value inapplicable (e.g., in outerjoins)
  - No information null
- Still uses 3-valued logic

Basic rules for nulls

- Any comparison involving NULLs results in **unknown**
    - 5 < NULL, NULL > NULL, even NULL=NULL
- Any operation involving NULLs results in NULL
    - 5+NULL = NULL, NULL || ‘abc’ = NULL
    - BUT: the condition 5+NULL = NULL evaluates to **unknown**

Nulls as Booleans

What is the output of these queries if S={1}?

```
SELECT 1 FROM S
WHERE (null = ((null =
       ((null = ((null = null) is null))
        is null)) is null)) is null
```

1

```
SELECT 1 FROM S
WHERE (null = ((null =
       ((null = ((null = null) is null))
        is null)) is null))
```

∅

NULLs as Booleans

- As a Boolean value, NULL is viewed as unknown
- null=null is unknown, hence null
- (null=null) is null is hence true
- ((null=null) is null)=null is unknown hence null etc

NULLs and Aggregation

- Remember the rule: NULLs in operations result in NULL as result
  - 1+2+NULL is thus NULL, but:
- ```
  SELECT SUM(A)
  FROM ( VALUES (1), (2), (NULL)) AS R(A)
  ```
- which adds 1, 2, and NULL gives 3
- SQL rule for aggregates: ignore NULLs and then apply the aggregate (except COUNT(*))
  ```

Some systems do weird things…Is empty string equal to itself?

```
SELECT *
FROM   R
WHERE  ‘’=‘’
```

- Usually it is, but not in Oracle: the above query always returns the empty table.
- Because Oracle implements NULL as ‘’
- Madness? Yes. With a string operation that produces ‘’ you deal with 3-valued logic before you realize it!

Last topic: almost certain answers

- Do we really need to insist on certainty?

- Often, “sufficiently close” is good enough. Certainly better than what SQL can give you.

- Does it make finding answers to queries over incomplete data easier?

Naive Evaluation

- **Treat nulls as new constants**
- **Evaluate query using standard techniques**
- **Heavily used: data integration/exchange, OBDA etc**

Orders

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

Pay

CUST_ID	ORDER
c1	Ord1
c2	$\bot$

Customer

CUST_ID	NAME
c1	John
c2	Mary

Unpaid orders:

select O.order_id
from Orders O
where O.order_id not in
(select order from Pay P)

Answer: Ord2, 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: c2.

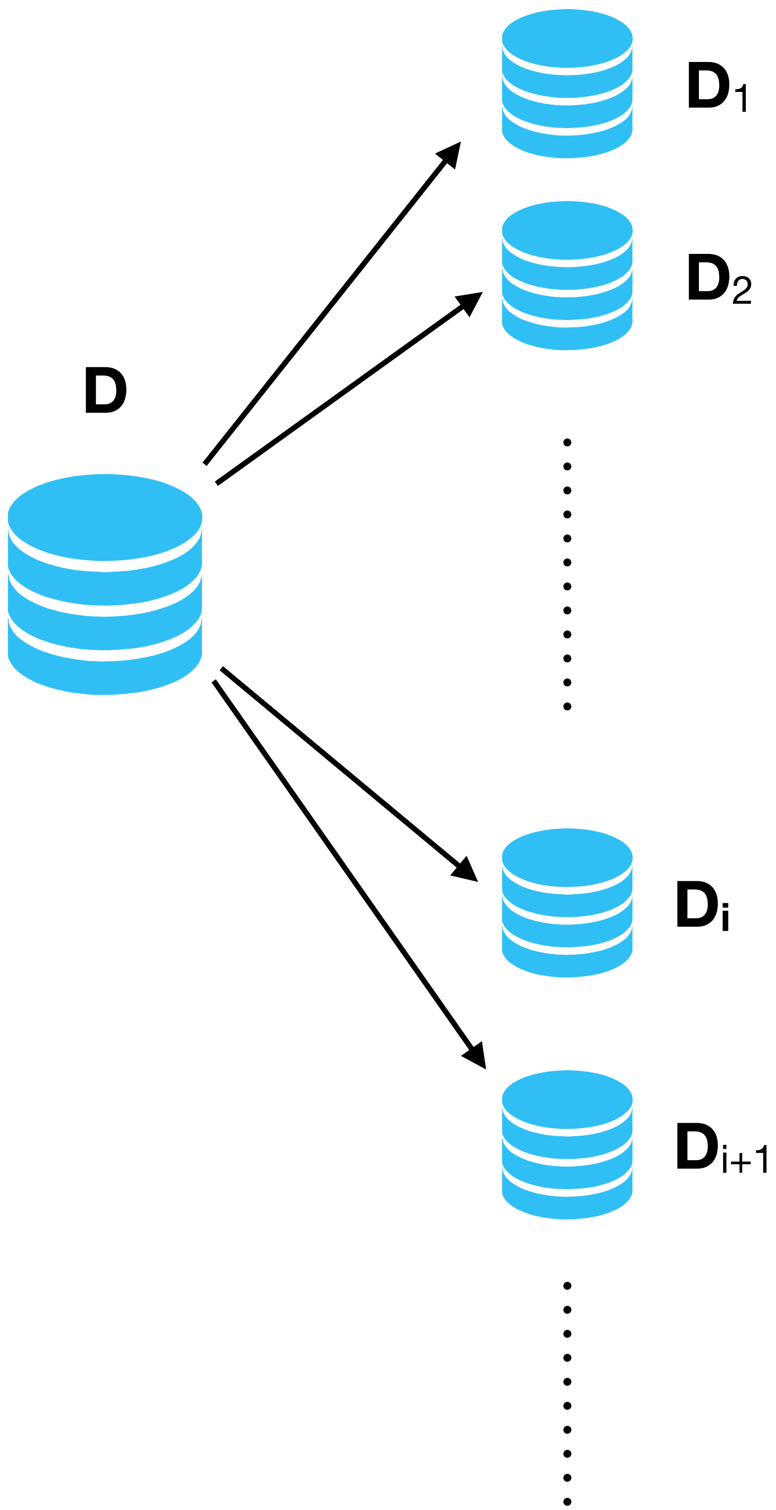
How bad are bad answers?

- What if the real value of ⊥ is an order different from Ord1, Ord2, Ord3?
  - Then naive evaluation actually produces correct answers!
  - If we know nothing about ⊥ this isn't an unreasonable assumption: there could be many orders.
- But what if we know ⊥ $\in$ {Ord1,Ord2,Ord3}?
  - Then answer to the first query is Ord2 with 50% chance and Ord3 with 50% chance. Answer to the second query is empty.

Questions

- Is naive evaluation always good without constraints on nulls, or we just got lucky?
    - Yes, it always is
- Can we get the second type of answers, with constraints?
    - Yes, but with more work
- Now revisit certain answers, and connect them with a well know subject in logic and probability

Incomplete data and certain answers

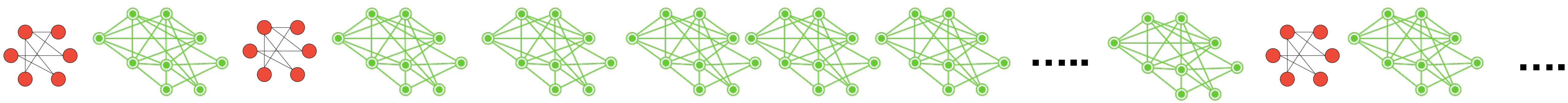


Incomplete database D represents many complete databases D_1, D_2, …

Tuple a is certain answer to query Q in D
⇔ a is an answer to Q in every D_i

Zero-One Laws

A formula α over graphs; green = true; red = false



α is almost surely valid: true in almost all graphs

- **pick a graph G at random**
- **calculate the probability $\mu(\alpha)$ that α is true in G**
- **$\mu(\alpha) = 1 \Leftrightarrow \alpha$ is almost surely valid**

Examples:

- **μ(has an isolated node) = 0**
- **μ(is a tree)=0**
- **μ(connected) = 1**
- **μ(has diameter at most 2) = 1**

Zero-One Laws

Fagin 1976:
if α is first-order, then $\mu(\alpha)$ is 0 or 1

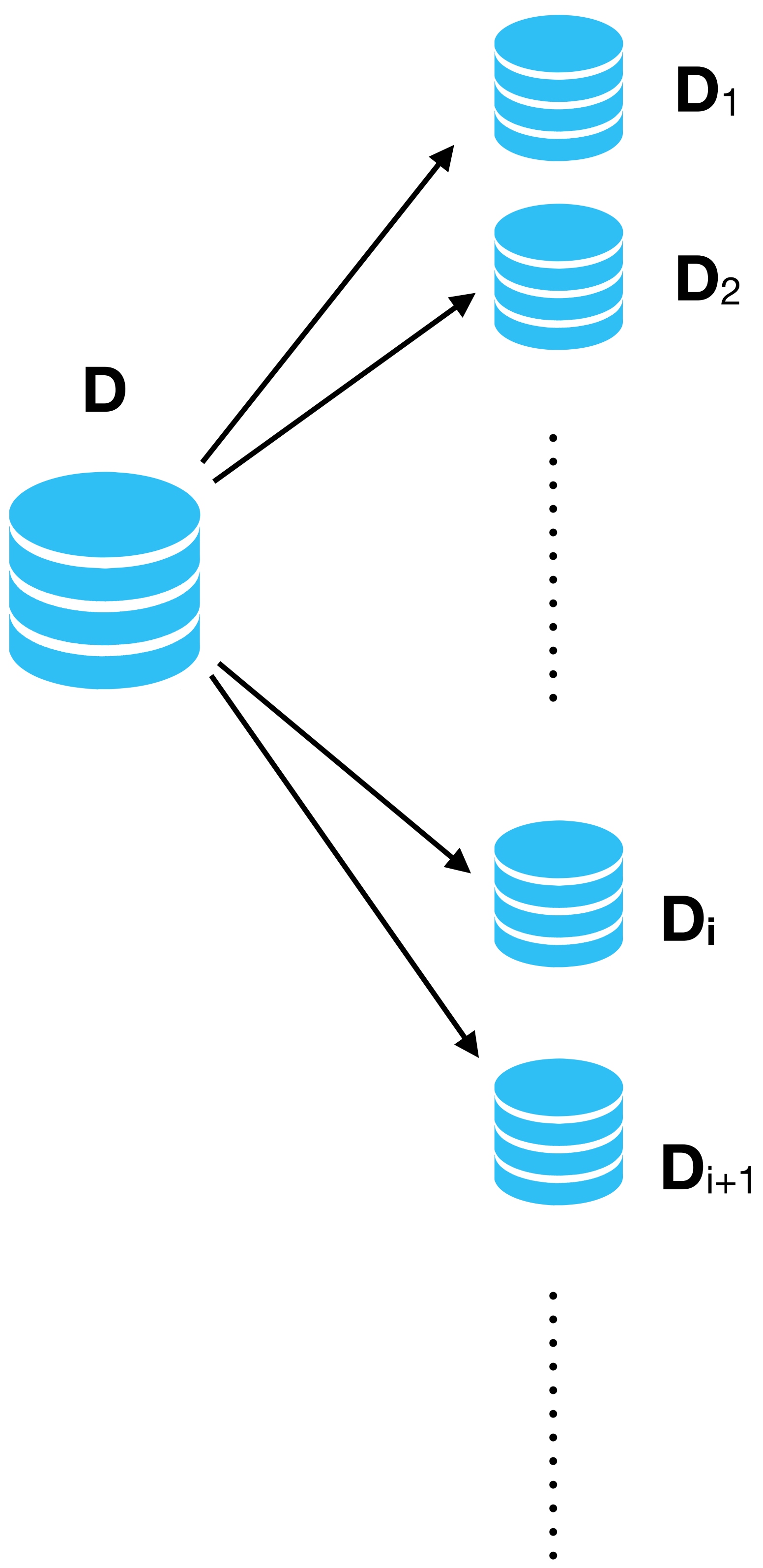
α is valid (true in all graphs) - undecidable.

α is almost surely valid ($\mu(\alpha) = 1$) - easy to decide.

Extended to many other logics: Fixed-point, Infinitary logics, Fragments of second-order logic; Other distributions too

A very active subject in logic/combinatorics

Certainty and Zero-One Laws



For query Q:

- **pick a complete database D_i at random**
- **$\mu(Q,D,a)$: probability that $a \in Q(D_i)$**

$$\mu(Q,D,a) = 1 \Rightarrow$$

a = almost certainly true answer to Q in D

Questions

1. **When is $\mu(Q,D,a) = 1$?**
2. **How easy is it to compute?**
3. **Can an answer be 50% true?**
4. **Is one tuple a better answer than another?**

Certain Answers

A tuple of constants c is a certain answer:
$c \in Q(v(D))$ for each valuation v

An arbitrary tuple a is a certain answer:
$v(a) \in Q(v(D))$ for each valuation v

Support of a:

$$\text{Supp}(Q,D,a) = \{\text{valuations } v \mid v(a) \in Q(v(D))\}$$

Certain Answers

Support of a: $\mathrm{Supp}(Q,D,a) = \{\text{valuations } v \mid v(a) \in Q(v(D))\}$

Answer a is certain ⇔ every valuation v is in Supp(Q,D,a)

Idea: answer a is almost certainly true
⇔ a randomly chosen valuation v is in Supp(Q,D,a)

A small problem: there are infinitely many valuations.
But techniques from zero-one laws help: look at finite approximations.

Measuring Certainty

Constants (non-nulls) = $\{c_1, c_2, c_3, \ldots\ \}$

$Valuation_k$ **=** **finite** **set of valuations with range** $\subseteq \{c_1, \ldots, c_k\}$

$$Supp_k(Q,D,a) = Supp(Q,D,a) \cap Valuation_k$$

$$\mu_k(Q,D,a) = \frac{|Supp_k(Q,D,a)|}{|Valuation_k|}$$

(a number in [0,1])

Interpretation: Probability that a randomly chosen valuation with range in $\{c_1, \ldots, c_k\}$ **witnesses that** a **is an answer to** Q

Measuring Certainty

$$\mu(Q,D,a) = \lim_{k\to\infty} \mu_k(Q,D,a)$$

Interpretation: Probability that a randomly chosen valuation witnesses that *a* is an answer to *Q*

Observation: the value $\mu(Q,D,a)$ does not depend on a particular enumeration of $\{c_1, c_2, c_3, \ldots\}$

Zero-One Law

- **Q: any reasonable query**
  - **definable in a query language such as relational algebra, datalog, second-order logic etc - formally, generic**
- **Theorem: μ(Q,D,a) is either 0 or 1**
  - **every answer is either almost certainly true or almost certainly false**

Zero-One Law and Naive Evaluation

- **$\mu(Q,D,a) = 1 \Leftrightarrow a$ is returned by the naive evaluation of Q**
  - **thus almost certainly true answers are much easier to compute than certain answers**
  - **and naive evaluation is justified as being very close to certainty**

Naive evaluation: treat nulls as values

A	B
1	$\bot_1$
2	$\bot_1$
2	$\bot_2$

-

A	B
1	$\bot_2$
2	$\bot_1$

=

A	B
1	$\bot_1$
2	$\bot_2$

Certain answer is empty because of valuations $\bot_1, \bot_2 \to c$

If the range of nulls is infinite, such valuations are unlikely.
Returned tuples are almost certainly true answers - but not certain.

In general, naive evaluation ≠ certain answers as we have seen, except

- **unions of conjunctive queries**
- **their extension with $Q \div R$ where R is a relation**

Proof idea

- Let $\bot_1 \bot_2 \ldots\ldots \bot_m$ enumerate all nulls in database D
- Consider all k^m mappings $f: \{\bot_1 \bot_2 \ldots\ldots \bot_m\} \rightarrow \{1,\ldots,k\}$. For how many $f(i)=f(j)$ for some i,j?
- Choose i,j; select value of $f(i)$; find an arbitrary mapping on the remaining $m-2$ nulls:
  - $\text{Choose}(m,2) \cdot k \cdot k^{m-2} = O(m^2 \cdot k^{m-1})$
  - $(m^2 \cdot k^{m-1}) / k^m \rightarrow 0$ when $k \rightarrow \infty$
- Thus most mappings assign distinct values to nulls, and hence we use naive evaluation

Naive evaluation: treat nulls as values

A	B
1	$\bot_1$
2	$\bot_1$
2	$\bot_2$

-

A	B
1	$\bot_2$
2	$\bot_1$

=

A	B
1	$\bot_1$
2	$\bot_2$

What if:

1. **We have a functional dependency A→B, forcing $\bot_1 = \bot_2$, or**
2. **there is a restriction on the range of B?**

The reasoning that valuations $\bot_1, \bot_2 \rightarrow c$ are unlikely no longer works

This is due to the presence of constraints.

Certainty with constraints

- **Only interested in databases satisfying integrity constraints Σ - for example, keys or foreign keys**

- **Standard approach: find certain answers to $\Sigma \rightarrow Q$**

- **Not very successful: if we have Q from a good class (certain answers can be computed efficiently) and Σ from a common class of constraints, the syntactic shape of $\Sigma \rightarrow Q$ makes existing results on finding certain answers inapplicable.**

Certainty with constraints

- In addition, this approach is not very informative
  - $\Sigma \rightarrow Q$ is $\neg\Sigma \vee Q$
  - if $\mu(\Sigma,D) = 0$, then $\mu(\Sigma \rightarrow Q,D,a) = 1$
  - if $\mu(\Sigma,D) = 1$, then $\mu(\Sigma \rightarrow Q,D,a) = \mu(Q,D,a)$

Certainty with constraints

- A better idea: use conditional probability $\mu(Q \mid \Sigma, D, a)$
  - probability that a randomly chosen valuation that satisfies Σ also witnesses that a is an answer to Q
- Still defined as a limit since there are infinitely many valuations

Measuring certainty with constraints

$Supp_k(Q,D,a)$ = {valuations $v \in Valuation_k \mid v(a) \in Q(v(D))$ }

$$\mu_k(Q \mid \Sigma, D, a) = \frac{|Supp_k(Q \wedge \Sigma, D, a)|}{|Supp_k(\Sigma,D,a)|}$$

Interpretation: Probability that a randomly chosen valuation with range in $\{c_1, \ldots, c_k\}$ that witnesses constraints Σ also witnesses that a is an answer to Q

Measuring certainty with constraints

$$\mu(Q \mid \Sigma, D, a) = \lim_{k\to\infty} \mu_k(Q \mid \Sigma, D, a)$$

Interpretation: Probability that a randomly chosen valuation that witnesses constraints Σ also witnesses that a is an answer to Q

Observation: the value $\mu(Q \mid \Sigma, D, a)$ does not depend on a particular enumeration of $\{c_1, c_2, c_3, \ldots\}$

Zero-One Law fails with constraints

- **Database D: R = {⊥}, S = {1}, U = {1,2}**
- **Constraint: R ⊆ U**
- **Query Q: is R ⊆ S ?**
- ***μ(Q | Σ, D) = 0.5***

What if zero-one fails?

- **The best next thing: convergence**
- **Consider, for example, ordered graphs.**
- **Zero-one law fails: μ(edge between the smallest and the largest element) = 0.5**
- **But $\mu(\alpha)$ exists for every first-order α**
  - **and is a rational of the form $n/2^m$ (Lynch 1980)**

Convergence with constraints

- **Q: any reasonable query, Σ: any reasonable constraints (both generic)**
- **Theorem: $\mu(Q \mid \Sigma, D, a)$ always exists**
    - **$\mu(Q \mid \Sigma, D, a)$ is a rational number between 0 and 1**
- **Every rational number in [0,1] can appear as $\mu(Q \mid \Sigma, D, a)$ for a conjunctive query Q and an inclusion constraint Σ**

Computing $\mu(Q \mid \Sigma, D, a)$

- **A rational number - need a function complexity class**
- **It can be computed in $FP^{\#P}$**
    - functions computable in polynomial time with access to a #P oracle
- **#P: counting solutions to NP problems**
    - How many satisfying assignments does a formula have?
    - How many 3-colorings a graph has? etc

Constraints and zero-one laws

- **Zero-one law still holds for some constraints, e.g., functional dependencies**
- **Σ: a set of functional dependencies.**
- **certain answers under Σ : Answers true in every database satisfying Σ**
- **We can compute them easily for conjunctive queries using the Chase procedure**

What is Chase?

- A procedure often used in databases to enforce integrity constraints or to check their implication.
- $A \rightarrow B$ and $B \rightarrow C$
-

A	B	C
1	$\perp_1$	$\perp_3$
1	$\perp_2$	$\perp_4$
2	$\perp_1$	$\perp_2$

→

A	B	C
1	$\perp_1$	$\perp_3$
1	$\perp_1$	$\perp_4$
2	$\perp_1$	$\perp_1$

→

A	B	C
1	$\perp_1$	$\perp_3$
1	$\perp_1$	$\perp_3$
2	$\perp_1$	$\perp_1$

→

A	B	C
1	$\perp_1$	$\perp_3$
2	$\perp_1$	$\perp_1$

$A \rightarrow B$

$\perp_1 = \perp_2$

$B \rightarrow C$

$\perp_3 = \perp_4$

eliminate duplicate rows

Result: chase(D, Σ)

Constraints and zero-one laws

- **If Q is a conjunctive query, then**
  - **certain answers under Σ = Q(chase(D,Σ))**
- **If Q is an arbitrary query, then almost certainly true answers under Σ = Q(chase(D,Σ))**
  - **$\mu(Q \mid \Sigma, D, a) = \mu(Q, \text{chase}(D, \Sigma), a)$**

Qualitative Measures

- **We can also use supports Supp(Q,D,a) to define qualitative measures:**
  - **a is at least as good an answer as b, to query Q if Supp(Q,D,b) $\subseteq$ Supp(Q,D,a)**
  - **a is a better answer than b, to query Q if Supp(Q,D,b) $\subsetneq$ Supp(Q,D,a)**
  - **a is a best answer to Q if there is no better answer**

Qualitative measure: example

A	B
1	$\bot_1$
2	$\bot_1$
2	$\bot_2$

-

A	B
1	$\bot_2$
2	$\bot_1$

- **No certain answers**
- **Naive evaluation gives (1, $\bot_1$) and (2, $\bot_2$)**
- **(2, $\bot_2$) is a better answer than (1, $\bot_1$)**
- **Best answer = (2, $\bot_2$)**

Unlike certain answers, best answers always exist

Qualitative measures: complexity

- **Fix a query Q of relational algebra/calculus**
- **Input: database D, tuples a and b**

Is a at least as good as b?	**coNP-complete**
Is a better than b?	**DP-complete**
Identify the set of best answers	**$P^{NP[\log n]}$-complete**

- **For unions of conjunctive queries, all in PTIME.**
  - Does not go via naive evaluation; the algorithm is of very different nature

Measuring complexity

Question	CERTAIN ANSWER	BEST ANSWER
Given a tuple a, is a ∈ Answer ?	coNP-complete	$P^{NP[\log n]}$-complete
Given a set X, is X = Answer ?	DP-complete	$P^{NP[\log n]}$-complete
Given a family of sets *F*, is Answer ∈ F ?	$P^{NP[\log n]}$-complete	$P^{NP[\log n]}$-complete

BIG open questions

- How to handle aggregation
- How to handle bag semantics
- How to handle more complex constraints
- How to implement these algorithms inside DBMSs
- How to convince designers of new languages to drop SQL’s approach
- and crucially: **WHAT DO USERS ACTUALLY WANT FROM NULL?**