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'*

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.

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

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

Queries, as we teach students to write them:

Unpaid orders

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

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)
```

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

Queries, as we teach students to write them:

Unpaid orders

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

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

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

Queries, as we teach students to write them:

Unpaid orders

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

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)
```

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

Queries, as we teach students to write them:

Unpaid orders

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

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~~ none

Answer: ~~none~~ c2

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	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	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	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

$$h(\perp_1) = 4$$
$$h(\perp_2) = 3$$
$$h(\perp_3) = 5$$
$$\Longrightarrow$$

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:

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

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

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.

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

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

Therefore:

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

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

Reduction from 3-colorability.

Take a graph $G = (V, E)$ and create a database D_G with nulls $\perp_v$ for each $v \in V$ and edges $(\perp_v, \perp_{v'})$ 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 $\mathsf{certain}_{\mathsf{CWA}}(Q, D_G) = \mathbf{t}$ iff G is not 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:

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 $\vee, \wedge, \neg, \forall, \exists$:

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

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

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

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

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

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

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, $\wedge, \vee, \exists$.

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

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

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

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.

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

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

Question:

- can we extend this, say to all of relational calculus? That is, get an evaluation without false positives?

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 $h(\bar{t}_1) = h(\bar{t}_2)$.

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

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

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

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

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

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

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

$h(\bar{a}) \in Q(h(D))$ for every valuation h 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:

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

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

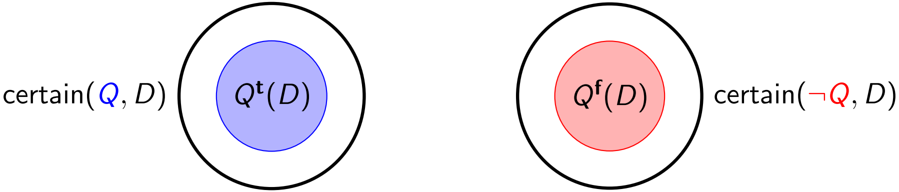
Same in relational algebra

There is an effective translation of queries

$$Q \quad \mapsto \quad (\, Q^{\mathbf{t}}, Q^{\mathbf{f}} \,)$$

such that:

- $Q^{\mathbf{t}}$ approximates certain answers to Q
- $Q^{\mathbf{f}}$ approximates certain answers to the negation of Q
- both queries have AC^0 data complexity



Relational algebra translations: $Q^{\mathbf{t}}$

For a relation R: $R^{\mathbf{t}} = R$

For op $\in \{\cap, \cup, \times\}$: $(Q_1 \text{ op } Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \text{ op } Q_2^{\mathbf{t}}$

For projection: $\pi_\alpha(Q)^{\mathbf{t}} = \pi_\alpha(Q^{\mathbf{t}})$

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

For selection: $\sigma_\theta(Q)^{\mathbf{t}} = \sigma_{\theta^*}(Q^{\mathbf{t}})$

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

Relational algebra translations: $Q^{\mathbf{f}}$

$$
\begin{aligned}
R^{\mathbf{f}} &= \left\{ \bar{r} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \bar{r} \text{ does not match any tuple in } R \right\} \\
(Q_1 \cup Q_2)^{\mathbf{f}} &= Q_1^{\mathbf{f}} \cap Q_2^{\mathbf{f}} \\
(Q_1 \cap Q_2)^{\mathbf{f}} &= Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{f}} \\
(Q_1 - Q_2)^{\mathbf{f}} &= Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{t}} \\
(\sigma_\theta(Q))^{\mathbf{f}} &= Q^{\mathbf{f}} \cup \sigma_{(\neg\theta)^*}\big(\mathsf{adom}^{\mathsf{ar}(Q)}\big) \\
(Q_1 \times Q_2)^{\mathbf{f}} &= Q_1^{\mathbf{f}} \times \mathsf{adom}^{\mathsf{ar}(Q_2)} \;\cup\; \mathsf{adom}^{\mathsf{ar}(Q_1)} \times Q_2^{\mathbf{f}} \\
\big(\pi_\alpha(Q)\big)^{\mathbf{f}} &= \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\big(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}}\big)
\end{aligned}
$$

Does it work in practice?

Not a chance: With as few as 1000 tuples and 3 attributes bad queries start computing relations with billions of tuples!

Inefficient translations

$$R^{\mathsf{f}} = \left\{ \bar{r} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \bar{r} \text{ does not match any tuple in } R \right\}$$

$$(\sigma_\theta(Q))^{\mathsf{f}} = Q^{\mathsf{f}} \cup \sigma_{(\neg\theta)^*}\left(\mathsf{adom}^{\mathsf{ar}(Q)}\right)$$

$$(Q_1 \times Q_2)^{\mathsf{f}} = Q_1^{\mathsf{f}} \times \mathsf{adom}^{\mathsf{ar}(Q_2)} \cup \mathsf{adom}^{\mathsf{ar}(Q_1)} \times Q_2^{\mathsf{f}}$$

$$\left(\pi_\alpha(Q)\right)^{\mathsf{f}} = \pi_\alpha(Q^{\mathsf{f}}) - \pi_\alpha\left(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathsf{f}}\right)$$

With the best tricks we can only handle a few hundred tuples:

AC^0 and efficiency are NOT the same!

Does it work in practice?

Not a chance: With as few as 1000 tuples and 3 attributes bad queries start computing relations with billions of tuples!

Inefficient translations

$$R^{\mathbf{f}} = \left\{ \bar{r} \in \mathrm{adom}^{\mathrm{ar}(R)} \mid \bar{r} \text{ does not match any tuple in } R \right\}$$

$$(\sigma_\theta(Q))^{\mathbf{f}} = Q^{\mathbf{f}} \cup \sigma_{(\neg\theta)^*}\left(\mathrm{adom}^{\mathrm{ar}(Q)}\right)$$

$$(Q_1 \times Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \times \mathrm{adom}^{\mathrm{ar}(Q_2)} \cup \mathrm{adom}^{\mathrm{ar}(Q_1)} \times Q_2^{\mathbf{f}}$$

$$\left(\pi_\alpha(Q)\right)^{\mathbf{f}} = \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\left(\mathrm{adom}^{\mathrm{ar}(Q)} - Q^{\mathbf{f}}\right)$$

With the best tricks we can only handle a few hundred tuples:

AC^0 and efficiency are NOT the same!

Does it work in practice?

Not a chance: With as few as 1000 tuples and 3 attributes bad queries start computing relations with billions of tuples!

Inefficient translations

$$R^{\mathbf{f}} = \left\{ \bar{r} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \bar{r} \text{ does not match any tuple in } R \right\}$$

$$(\sigma_\theta(Q))^{\mathbf{f}} = Q^{\mathbf{f}}$$

$$(Q_1 \times Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \times \mathsf{adom}^{\mathsf{ar}(Q_2)} \cup \mathsf{adom}^{\mathsf{ar}(Q_1)} \times Q_2^{\mathbf{f}}$$

$$\left(\pi_\alpha(Q)\right)^{\mathbf{f}} = \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\left(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}}\right)$$

With the best tricks we can only handle a few hundred tuples:

AC^0 and efficiency are NOT the same!

Does it work in practice?

Not a chance: With as few as 1000 tuples and 3 attributes bad queries start computing relations with billions of tuples!

Inefficient translations

$$R^{\mathbf{f}} = \left\{ \bar{r} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \bar{r} \text{ does not match any tuple in } R \right\}$$

$$(\sigma_\theta(Q))^{\mathbf{f}} = Q^{\mathbf{f}}$$

$$(Q_1 \times Q_2)^{\mathbf{f}} = Q_1^{\mathbf{f}} \times Q_2^{\mathbf{f}}$$

$$\big(\pi_\alpha(Q)\big)^{\mathbf{f}} = \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\left(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}}\right)$$

With the best tricks we can only handle a few hundred tuples:

AC^0 and efficiency are NOT the same!

Does it work in practice?

Not a chance: With as few as 1000 tuples and 3 attributes bad queries start computing relations with billions of tuples!

Inefficient translations

$$
\begin{aligned}
R^{\mathbf{f}} &= \left\{ \bar{r} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \bar{r} \text{ does not match any tuple in } R \right\} \\
(\sigma_\theta(Q))^{\mathbf{f}} &= Q^{\mathbf{f}} \\
(Q_1 \times Q_2)^{\mathbf{f}} &= Q_1^{\mathbf{f}} \times Q_2^{\mathbf{f}} \\
\big(\pi_\alpha(Q)\big)^{\mathbf{f}} &= \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha \left(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}} \right)
\end{aligned}
$$

With the best tricks we can only handle a few hundred tuples:

AC^0 and efficiency are NOT the same!

Let's rethink the basics

We only needed $Q^{\mathbf{f}}$ to handle difference: $(Q_1 - Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}}$

Intuition: A tuple is for sure in $Q_1 - Q_2$ if

- it is certainly in Q_1 and
- it is certainly not in Q_2

This is not the only possibility

A tuple is for sure in $Q_1 - Q_2$:

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

Let's rethink the basics

We only needed $Q^{\mathbf{f}}$ to handle difference: $(Q_1 - Q_2)^{\mathbf{t}} = Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}}$

Intuition: A tuple is for sure in $Q_1 - Q_2$ if

- it is certainly in Q_1 and
- it is certainly not in Q_2

This is not the only possibility

A tuple is for sure in $Q_1 - Q_2$:

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

What is "match"?

Unification: Two tuples unify if there is an instantiation of nulls with constants that makes them equal

Left unification antijoin

$$R \overline{\ltimes}_u S = \{ \bar{r} \in R \mid \nexists \bar{s} \in S \colon \bar{s} \text{ unifies with } \bar{r} \}$$

What is "match"?

Unification: Two tuples unify if there is an instantiation of nulls with constants that makes them equal

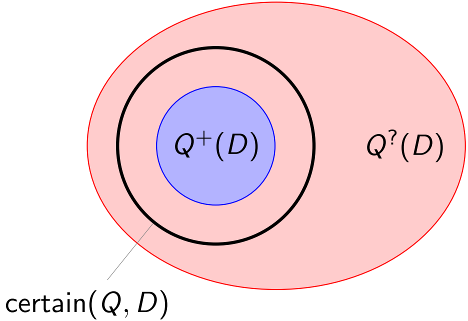
Left unification antijoin

$$R \mathbin{\overline{\ltimes}_u} S = \left\{ \bar{r} \in R \mid \nexists \bar{s} \in S \colon \bar{s} \text{ unifies with } \bar{r} \right\}$$

Second translation

Translate Q into $\big(\ Q^{+}, Q^{?}\ \big)$ where:

- Q^{+} approximates certain answers
- $Q^{?}$ represents possible answers



$$(Q_1 - Q_2)^{+} = Q_1^{+} \mathbin{\overline{\ltimes}_u} Q_2^{?}$$

$$R^{?} = R$$

$$(Q_1 \cup Q_2)^{?} = Q_1^{?} \cup Q_2^{?}$$

$$(Q_1 \cap Q_2)^{?} = Q_1^{?} \ltimes_u Q_2^{?}$$

$$(Q_1 - Q_2)^{?} = Q_1^{?} - Q_2^{+}$$

$$\big(\sigma_\theta(Q)\big)^{?} = \sigma_{\neg(\neg\theta)^*}\big(Q^{?}\big)$$

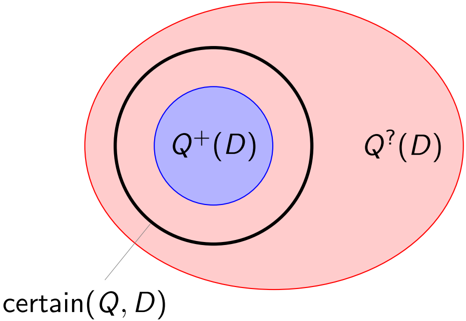
$$(Q_1 \times Q_2)^{?} = Q_1^{?} \times Q_2^{?}$$

$$\big(\pi_\alpha(Q)\big)^{?} = \pi_\alpha\big(Q^{?}\big)$$

Second translation

Translate Q into $\left(Q^+, Q^? \right)$ where:

- Q^+ approximates certain answers
- $Q^?$ represents possible answers



$$(Q_1 - Q_2)^+ = Q_1^+ \overline{\ltimes}_u Q_2^?$$

$$R^? = R$$

$$(Q_1 \cup Q_2)^? = Q_1^? \cup Q_2^?$$

$$(Q_1 \cap Q_2)^? = Q_1^? \ltimes_u Q_2^?$$

$$(Q_1 - Q_2)^? = Q_1^? - Q_2^+$$

$$\big(\sigma_\theta(Q)\big)^? = \sigma_{\neg(\neg\theta)^*}\big(Q^?\big)$$

$$(Q_1 \times Q_2)^? = Q_1^? \times Q_2^?$$

$$\big(\pi_\alpha(Q)\big)^? = \pi_\alpha\big(Q^?\big)$$

Certain and possible answers

For every valuation h of nulls:

$$h(Q^{+}(D)) \quad \subseteq \quad Q(h(D))$$

$$Q(h(D)) \quad \subseteq \quad h(Q^{?}(D))$$

- in particular, $Q^{+}(D) \subseteq \mathsf{certain}_{\perp}(Q, D)$

New translation: example

For queries with difference, Q^{+} is much more efficient than $Q^{\boldsymbol{t}}$.

$$Q = R - \big(\pi_\alpha(T) - \sigma_\theta(S)\big)$$

of arity k.

Translations:

$$Q^{\boldsymbol{t}} \ = \ R \cap \big((\pi_\alpha(\mathrm{adom}^k \,\overline{\ltimes}_{\boldsymbol{u}}\, T) - \pi_\alpha(\mathrm{adom}^k \ltimes_{\boldsymbol{u}} T)) \cup \sigma_{\theta^*}(S)\big)$$

(uncomputable in practice) but

$$Q^{+} \ = \ R \,\overline{\ltimes}_{\boldsymbol{u}}\, \big(\pi_\alpha(T) - \sigma_{\theta^*}(S)\big)$$

(easy to compute)

Does it work in practice?

Run queries and translations on TPC-H instances with nulls and measure the relative runtime performance of Q^+ w.r.t. Q

- SQL was designed for efficiency
  $\Longrightarrow$ we cannot expect to outperform native SQL
- but we can hope for the overhead to be acceptable

We observed the following behaviors:

- **The good:** small overhead
  (less than $< 4\%$)
- **The fantastic:** significant speed-up
  (more than 10^3 times faster)
- **The tolerable:** moderate slow-down
  (half the speed on 1GB instances, a quarter on 10GB ones)

Does it work in practice?

Run queries and translations on TPC-H instances with nulls and measure the relative runtime performance of Q^+ w.r.t. Q

- SQL was designed for efficiency
  $\Longrightarrow$ we cannot expect to outperform native SQL
- but we can hope for the overhead to be acceptable

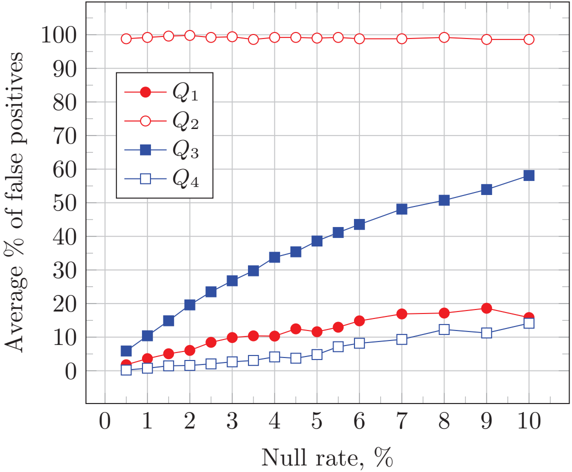
We observed the following behaviors:

- **The good:** small overhead
  (less than $< 4\%$)
- **The fantastic:** significant speed-up
  (more than 10^3 times faster)
- **The tolerable:** moderate slow-down
  (half the speed on 1GB instances, a quarter on 10GB ones)

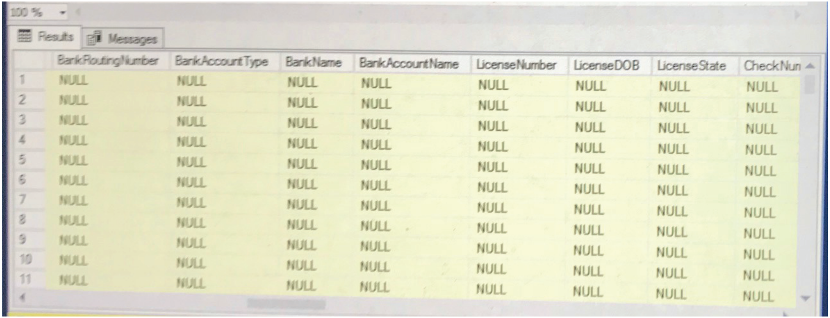
But do we solve a real poroblem?

That is, do false positives occur?

Nullrate: the probability a null occurs in an attribute that has not been declared as not null



And do nulls occur?



	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

The good

Q_3 *Find orders supplied entirely by a specific supplier*

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT   *
       FROM     lineitem
       WHERE    l_orderkey = o_orderkey


         AND    l_suppkey <> $supp_key

)
```

In relational algebra: $\pi_{\texttt{o\_orderkey}}\big(\texttt{orders}\ \overline{\ltimes}_{\theta}\ \texttt{lineitem}\big)$

becomes $\pi_{\texttt{o\_orderkey}}\big(\texttt{orders}\ \overline{\ltimes}_{\neg(\neg\theta)^*}\ \texttt{lineitem}\big)$

The good

Q_3 *Find orders supplied entirely by a specific supplier*

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT   *
       FROM     lineitem
       WHERE  ( l_orderkey = o_orderkey
                OR l_orderkey IS NULL
                OR o_orderkey IS NULL )
         AND  ( l_suppkey <> $supp_key
                OR l_suppkey IS NULL )
)
```

In relational algebra: $\pi_{\texttt{o\_orderkey}}\big(\texttt{orders}\ \overline{\ltimes}_{\theta}\ \texttt{lineitem}\big)$
becomes $\pi_{\texttt{o\_orderkey}}\big(\texttt{orders}\ \overline{\ltimes}_{\neg(\neg\theta)^*}\ \texttt{lineitem}\big)$

The good

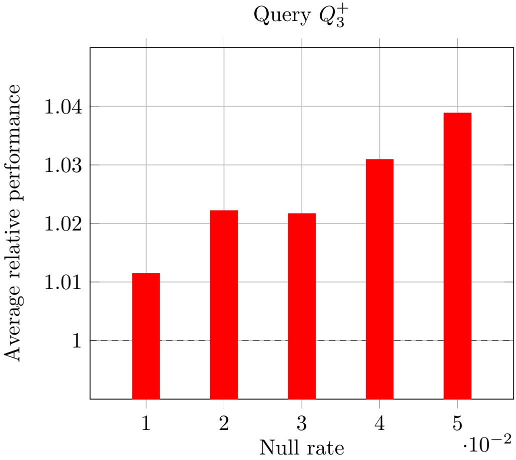
Q_3 *Find orders supplied entirely by a specific supplier*

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT   *
       FROM     lineitem
       WHERE    l_orderkey = o_orderkey



         AND    ( l_suppkey <> $supp_key
                  OR l_suppkey IS NULL )
)
```

In relational algebra: $\pi_{\texttt{o\_orderkey}}\big(\texttt{orders}\ \overline{\ltimes}_{\theta}\ \texttt{lineitem}\big)$
becomes $\pi_{\texttt{o\_orderkey}}\big(\texttt{orders}\ \overline{\ltimes}_{\neg(\neg\theta)^*}\ \texttt{lineitem}\big)$

The good: Results

$<$ 4% overhead (the same behavior scales up to 10GB instances)



The fantastic

Q_2 *Identify countries where there are customers who may be likely to make a purchase*

```
SELECT c_custkey, c_nationkey
FROM   customer
WHERE  c_nationkey IN ($countries)
  AND  c_acctbal > (
       SELECT avg(c_acctbal) FROM customer
       WHERE  c_acctbal > 0.00
         AND  c_nationkey IN ($countries) )
  AND  NOT EXISTS (
       SELECT * FROM orders
       WHERE  o_custkey = c_custkey )
```

The fantastic

Q_2 *Identify countries where there are customers who may be likely to make a purchase*

```
SELECT c_custkey, c_nationkey
FROM   customer
WHERE  c_nationkey IN ($countries)
  AND  c_acctbal > (
       SELECT avg(c_acctbal) FROM customer
       WHERE  c_acctbal > 0.00
         AND  c_nationkey IN ($countries) )
  AND  NOT EXISTS (
       SELECT * FROM orders
       WHERE  o_custkey = c_custkey
              OR o_custkey IS NULL )
```

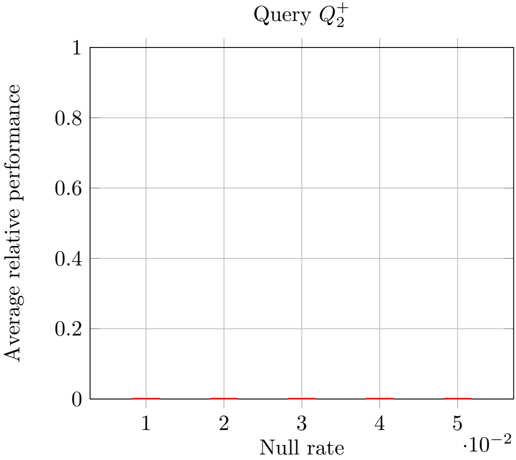
The fantastic

Q_2 *Identify countries where there are customers who may be likely to make a purchase*

```
SELECT c_custkey, c_nationkey
FROM   customer
WHERE  c_nationkey IN ($countries)
  AND  c_acctbal > (
       SELECT avg(c_acctbal) FROM customer
       WHERE  c_acctbal > 0.00
         AND  c_nationkey IN ($countries) )
  AND  NOT EXISTS (
       SELECT * FROM orders
       WHERE  o_custkey = c_custkey )
  AND  NOT EXISTS (
       SELECT * FROM orders
       WHERE  o_custkey IS NULL )
```

The fantastic: Results

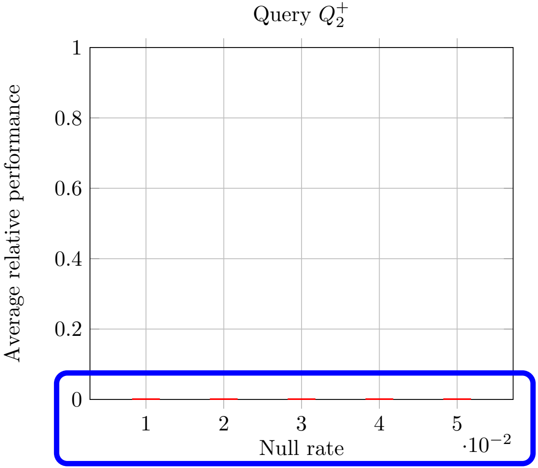
Over 10^3 times faster (same or better up to 10GB)



The original query spends most of the time looking for wrong answers

The fantastic: Results

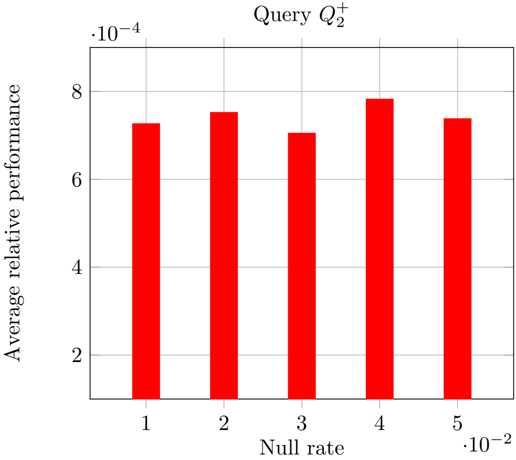
Over 10^3 times faster (same or better up to 10GB)



The original query spends most of the time looking for wrong answers

The fantastic: Results

Over 10^3 times faster (same or better up to 10GB)



The original query spends most of the time looking for wrong answers

The tolerable

Q_4 *Orders not supplied with any part of a specific color by any supplier from a specific country*

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT    *
       FROM      lineitem, part, supplier, nation
       WHERE     l_orderkey = o_orderkey

       AND       l_suppkey = s_suppkey

       AND       p_name LIKE '%'||$color||'%'

       AND       s_nationkey = n_nationkey

       AND       n_name = $nation )
```

The tolerable

Q_4 Orders not supplied with any part of a specific color by any supplier from a specific country

```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT   *
       FROM     lineitem, part, supplier, nation
       WHERE  ( l_orderkey = o_orderkey
                OR l_orderkey IS NULL )
       AND    ( l_suppkey = s_suppkey
                OR l_suppkey IS NULL )
       AND    ( p_name LIKE '%'||$color||'%'
                OR p_name IS NULL )
       AND    ( s_nationkey = n_nationkey
                OR s_nationkey IS NULL )
       AND      n_name = $nation )
```

The tolerable: Problems with the optimizer

Query times out. Reason: optimizer resorts to a nested loop plan.

On the smallest benchmark instance, we have relations with

- $6,000,000$ tuples,
- $200,000$ tuples,
- $10,000$ tuples,
- 100 tuples.

Nested loop: look at $1,200,000,000,000,000,000$ tuples.

No chance.

Join processing by example

$R(A, B),\ S(B, C)$

$$R \bowtie S \ = \ \{(x, y, z) \mid (x, y) \in R,\ (y, z) \in S\ \}$$

- Nested loop: look at all tuples $(x, y) \in R,\ (y', z) \in S$ and check if $y = y'$.
  - Hopelessly $O(n^2)$ — terrible on large data.
- Sort-merge join: Sort on B in $O(n \log n)$ and merge sorted lists.
  - Without too many repetitions of values of B, sort dominates, merge is fast, i.e., often $O(n \log n)$.
- Hash-join: apply a (good) hash function on the B attribute, only join tuples with the same hash value.
  - As sort-merge, often $O(n \log n)$ under some assumptions. Most commonly used in query processing.

The tolerable: Problems with the optimizer

Joins with disjunctions in correlated subqueries

$$R \,\overline{\ltimes}_{R.A=S.A} \left(\underbrace{S \bowtie_{S.B=T.B \,\vee\, \text{null}(S.B)} T}_{\text{nested-loop join}}\right)$$

As bad as computing a Cartesian product

We can do better

$$R \,\overline{\ltimes}_{R.A=S.A} \left(\underbrace{S \bowtie_{S.B=T.B} T}_{\text{hash join}}\right) \cap R \,\overline{\ltimes}_{\text{null}(S.B)} \left(\underbrace{S \ltimes T}_{\text{decorrelated EXISTS}}\right)$$

The tolerable: Problems with the optimizer

Joins with disjunctions in correlated subqueries

$$R \,\overline{\ltimes}_{R.A=S.A} \left(\underbrace{S \bowtie_{S.B=T.B \,\vee\, \mathrm{null}(S.B)} T}_{\text{nested-loop join}}\right)$$

As bad as computing a Cartesian product

We can do better

$$R \,\overline{\ltimes}_{R.A=S.A} \left(\underbrace{S \bowtie_{S.B=T.B} T}_{\text{hash join}}\right) \cap R \,\overline{\ltimes}_{\mathrm{null}(S.B)} \left(\underbrace{S \ltimes T}_{\text{decorrelated EXISTS}}\right)$$

Towards an improved translation

Conditions NOT EXISTS (.... OR .... OR .....)

$$\neg\exists(\ldots \vee \ldots \vee \ldots) \quad \Rightarrow \quad \neg\exists \bigvee \varphi_i \quad \Rightarrow \quad \bigwedge_i \neg\exists\varphi_i$$

Eliminate ORs and get conjunctions of nested NOT EXISTS subqueries.

Note: exponential blowup!

The tolerable: translation

Instructions: don't read.

```
WITH part_view  AS (SELECT p_partkey FROM part WHERE  p_name IS NULL
  UNION SELECT p_partkey FROM part WHERE  p_name LIKE '%'||$color||'%' ),
   supp_view  AS (SELECT s_suppkey FROM supplier WHERE s_nationkey IS NULL
  UNION SELECT s_suppkey FROM supplier, nation WHERE s_nationkey=n_nationkey
      AND n_name='$nation' )
SELECT o_orderkey FROM   orders
WHERE  NOT EXISTS (SELECT *
  FROM  lineitem, part_view, supp_view
  WHERE l_orderkey=o_orderkey AND l_partkey=p_partkey AND l_suppkey=s_suppkey)
  AND  NOT EXISTS (SELECT *
  FROM  lineitem, supp_view
  WHERE l_orderkey=o_orderkey AND  l_partkey IS NULL AND l_suppkey=s_suppkey
         AND  EXISTS (SELECT * FROM part_view))
  AND  NOT EXISTS (SELECT *
  FROM  lineitem, part_view
  WHERE l_orderkey=o_orderkey AND  l_partkey=p_partkey AND l_suppkey IS NULL
        AND  EXISTS (SELECT * FROM supp_view))
  AND  NOT EXISTS (SELECT * FROM   lineitem
  WHERE l_orderkey=o_orderkey AND l_partkey IS NULL  AND l_suppkey IS NULL
  AND EXISTS (SELECT * FROM part_view) AND EXISTS (SELECT * FROM supp_view))
```

What we've done

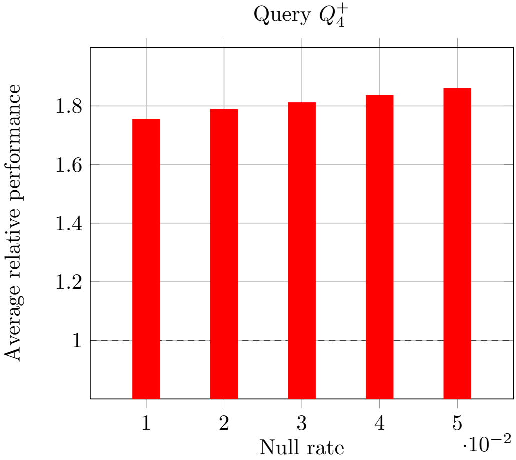
- Exponential blowup of the query.
- Complexity went from $|D|^{O(|Q|)}$ to $|D|^{2^{O(|Q|)}}$.
  - Double-exponential query complexity!
  - Theory teaches us that this is impossible to evaluate.
- Split one nested subquery into several ones.
  - Practice teaches us that this is much harder to evaluate.
- What happens in real life?
  - The query becomes several orders of magnitude faster!

What we've done

- Exponential blowup of the query.
- Complexity went from $|D|^{O(|Q|)}$ to $|D|^{2^{O(|Q|)}}$.
  - Double-exponential query complexity!
  - Theory teaches us that this is impossible to evaluate.
- Split one nested subquery into several ones.
  - Practice teaches us that this is much harder to evaluate.
- What happens in real life?
  - The query becomes several orders of magnitude faster!

The tolerable: Results

Half the speed (on 1GB; a quarter on 10GB instances)



The bad and the ugly

- Optimizers (we used PostgreSQL, others seem to be similar).
- Many translations amount to

$$\mathsf{A} = \mathsf{B} \quad \mapsto \quad \mathsf{A} = \mathsf{B}\ \mathsf{OR}\ \ \mathsf{B}\ \mathsf{IS}\ \mathsf{NULL}.$$

- They can't handle it, throw away the original plan and resort to nested loops!
- Why?
- We saw part of the reason above but there is more to it.

Join size estimate

- We observed that the query planner often underestimates the size of joins.

- Actually, this is known:
  Leis, Gubichev, Boncz, Kemper, Neumann: *How Good Are Query Optimizers, Really?* VLDB 2015
- All major ones (Microsoft, Oracle, IBM) and Postgres underestimate join sizes, sometimes by several orders of magnitude.
- If they wrongly think the join is small, $O(n^2)$ nested loop is no big deal to them compared to $O(n \log n)$

Disjunctions

- It is not just the IS NULL condition that is problematic, it is also the OR.

- Take some TPC-H queries, and change conditions like R.A=S.B into (R.A=S.B OR S.B=0)
- Basic benchmark queries: good plans, low costs
- Modified benchmark queries: nested-loops, high costs, queries dont terminate.

- In fact optimizers don't optimize with ORs!

SQL nulls vs marked nulls

- All theoretical translations assumed the model of marked nulls – these are special values distinct from the usual ones:

1	2	$\bot_1$
$\bot_2$	$\bot_3$	3
$\bot_4$	5	1

- Subtle differences with SQL nulls: comparing a SQL null with itself is unknown, comparing a marked null with itself is true
- SELECT R.A FROM R WHERE R.B=R.B
- On

1	null

it returns nothing.

- On

1	$\bot_1$

it returns 1

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 → 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 $A \to B$ $\overset{\text{REPAIR}}{\Rightarrow}$

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 $A \to B$
- Query Q: just return R
- If an instance may violate $A \to B$, then we can rewrite Q to $R(x, y, z) \wedge \forall u \forall v \ \big(R(x, u, v) \to u = y\big)$ 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 $Q \;=\; \exists x,y,z\ \big(R(x,z) \land R(y,z) \land (x \neq y)\big)$
- When are certain answers false ?
- If there is a repair in which the negation of Q is true.
- What is the negation of Q?
  - $\neg Q \;=\; \forall x,y,z\ \big((R(x,z) \land R(y,z)) \rightarrow x = y\big)$
- 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.