Constrained Validity and Querying Uncertain Data
Exact and Approximate Solutions

Leonid Libkin (University of Edinburgh)

Logic and Big Data

It's 9:15am.

People might be a bit too slow to arrive.

The talk starts with important definitions.

So let's spend the first few minutes on the role of logic in the Big Data world.

Logic and Big Data cont'd

What's the most successful application of logic (other than propositional) in CS?

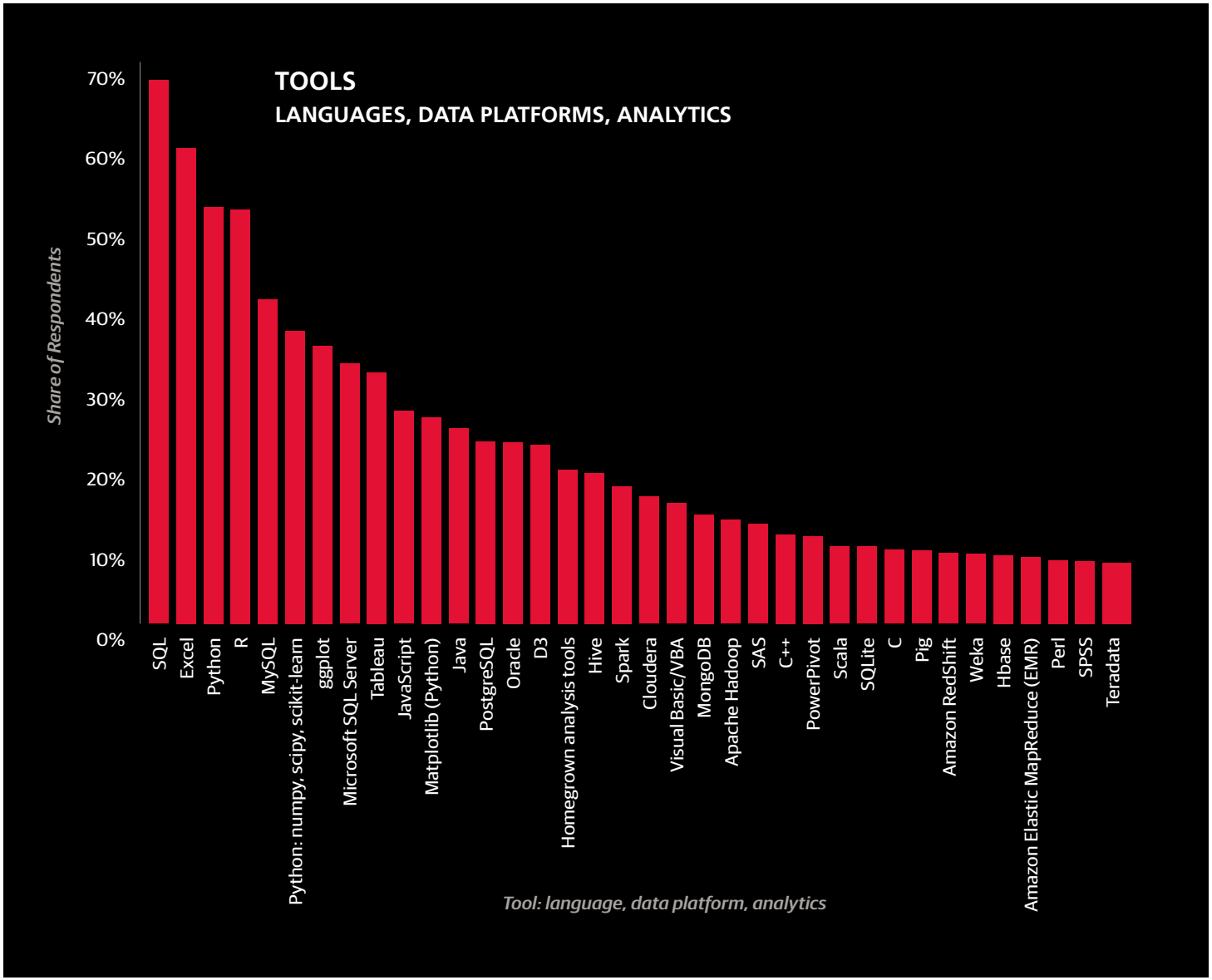
Databases. Nothing else comes close to $25B/year.

Main language: SQL. The core of it: first-order logic.

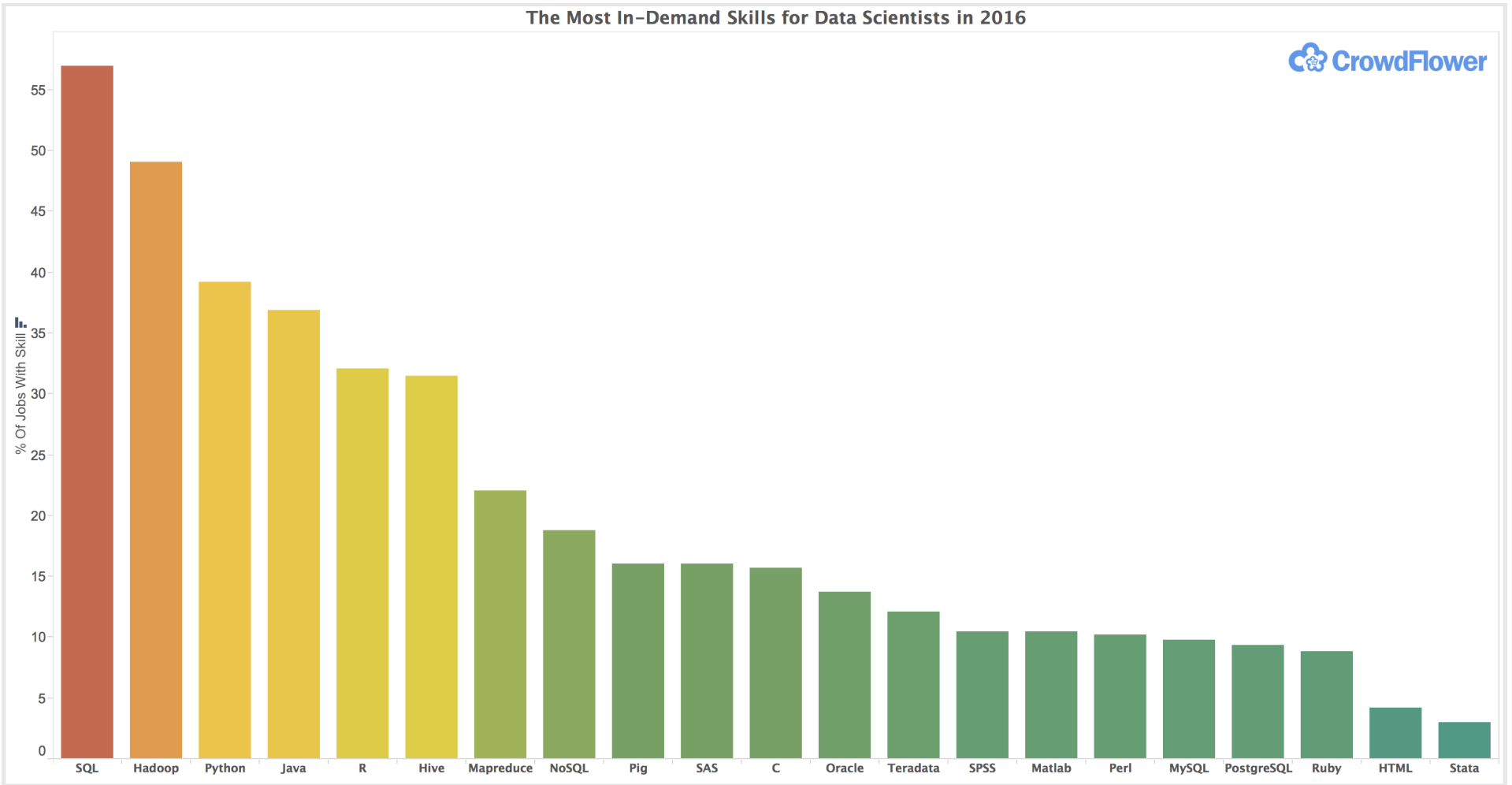
What do we often hear from data science, big data, etc crowds? That it's all about data mining, machine learning, statistics. Logic? Not really.

Is it true? Absolutely not! Not now, not in the future.

Logic in the big data world: SQL is the preferred tool



Logic in the big data world: SQL will remain the preferred tool



A quick introduction

We deal with first-order logic, FO over finite structures.

Model-checking: Given a structure S and a formula φ, is φ true in S?

- is $S \models \varphi$ true?

Validity: Given a a formula φ, is it true in every (finite) structure S?

- is $\models \varphi$ true?

A quick introduction

We deal with first-order logic, FO over finite structures.

Model-checking: Given a structure S and a formula φ, is φ true in S?

- is $S \models \varphi$ true?

Validity: Given a a formula φ, is it true in every (finite) structure S?

- is $\models \varphi$ true?

We are interested in a constrained version of validity:

- Given a formula φ, and a class $\mathcal{K}$ of structures, is φ true in every structure $S \in \mathcal{K}$?

A quick introduction cont'd

How to define $\mathcal{K}$?

Let's assume it is given by a formula ψ.

Then we are just checking validity of $\psi \rightarrow \varphi$.

But what if ψ – and $\mathcal{K}$ – is somehow given by another structure D?

Then we want to reduce constrained validity to model-checking:

- replace $\models \psi \rightarrow \varphi$ with $D \models \varphi$

A quick introduction cont'd

How to define $\mathcal{K}$?

Let's assume it is given by a formula ψ.

Then we are just checking validity of $\psi \to \varphi$.

But what if ψ – and $\mathcal{K}$ – is somehow given by another structure D?

Then we want to reduce constrained validity to model-checking:

- replace $\models \psi \to \varphi$ with $D \models \varphi$

A quick introduction cont'd

How to define $\mathcal{K}$?

Let's assume it is given by a formula ψ.

Then we are just checking validity of $\psi \rightarrow \varphi$.

But what if ψ – and $\mathcal{K}$ – is somehow given by another structure D?

Then we want to reduce constrained validity to model-checking:

- replace $\models \psi \rightarrow \varphi$ with $D \models \varphi$

A quick introduction cont'd

How to define $\mathcal{K}$?

Let's assume it is given by a formula ψ.

Then we are just checking validity of $\psi \rightarrow \varphi$.

But what if ψ – and $\mathcal{K}$ – is somehow given by another structure D?

Then we want to reduce constrained validity to model-checking:

- replace $\models \psi \rightarrow \varphi$ with $D \models \varphi$

A quick introduction cont'd

How to define $\mathcal{K}$?

Let's assume it is given by a formula ψ.

Then we are just checking validity of $\psi \rightarrow \varphi$.

But what if ψ – and $\mathcal{K}$ – is somehow given by another structure D?

Then we want to reduce constrained validity to model-checking:

- replace $\models \psi \rightarrow \varphi$ with $D \models \varphi$

Validity vs model-checking: flavors

- Problem to solve: is $\psi \to \varphi$ valid?
- Known: ψ is given with the help of a structure D

Possibilities:

Ideal solution — validity is model-checking:

$$\models \psi \to \varphi \quad \Leftrightarrow \quad D \models \varphi$$

A semi-ideal situation — validity can be checked algorithmically:

$$\models \psi \to \varphi \quad \Leftrightarrow \quad \text{algorithm}(D, \varphi) = \text{true}$$

An approximate solution — replace φ with φ' so that:

$$D \models \varphi' \quad \Rightarrow \quad \models \psi \to \varphi$$

Validity vs model-checking: flavors

- Problem to solve: is $\psi \to \varphi$ valid?
- Known: ψ is given with the help of a structure D

Possibilities:

Ideal solution — validity is model-checking:

$$\models \psi \to \varphi \quad \Leftrightarrow \quad D \models \varphi$$

A semi-ideal situation — validity can be checked algorithmically:

$$\models \psi \to \varphi \quad \Leftrightarrow \quad \mathsf{algorithm}(D, \varphi) = \mathsf{true}$$

An approximate solution — replace φ with φ' so that:

$$D \models \varphi' \quad \Rightarrow \quad \models \psi \to \varphi$$

Validity vs model-checking: flavors

- Problem to solve: is $\psi \to \varphi$ valid?
- Known: ψ is given with the help of a structure D

Possibilities:

Ideal solution — validity is model-checking:

$$\models \psi \to \varphi \quad \Leftrightarrow \quad D \models \varphi$$

A semi-ideal situation — validity can be checked algorithmically:

$$\models \psi \to \varphi \quad \Leftrightarrow \quad \mathsf{algorithm}(D, \varphi) = \mathsf{true}$$

An approximate solution — replace φ with φ' so that:

$$D \models \varphi' \quad \Rightarrow \quad \models \psi \to \varphi$$

Restricted validity: how to get $\mathcal{K}$ and ψ

Class $\mathcal{K}$ will typically be of the form:

$$\mathcal{K}_D = \{S \mid \text{there is a } \_\_\_\_\_ \text{ homomorphism } h : D \to S\}$$

- a usual homomorphism;
- or an onto homomorphism: universe of S = h(universe of D)
- or a strong onto homomorphism: $S = h(D)$

For usual homomorphisms, ψ is just the positive diagram of D:

$$\psi_D = \exists \text{ universe of } D \bigwedge \text{ facts of } D$$

Restricted validity: how to get $\mathcal{K}$ and ψ

Class $\mathcal{K}$ will typically be of the form:

$$\mathcal{K}_D = \{S \mid \text{there is a } \_\_\_\_\_ \text{ homomorphism } h : D \to S\}$$

- a usual homomorphism;
- or an onto homomorphism: universe of S = h(universe of D)
- or a strong onto homomorphism: $S = h(D)$

For usual homomorphisms, ψ is just the positive diagram of D:

$$\psi_D = \exists \text{ universe of } D \bigwedge \text{ facts of } D$$

Why? This is how we answer queries over incomplete data. But first...

Complexity of model-checking

- Problem: Given φ and S, is $S \models \varphi$?
- Two types of complexity:
  - Combined complexity: both φ and S are the input.
  - Data complexity: φ is fixed, only S is the input.
  - Often an exponential gap between them.
  - Also parameterized complexity, but we don't deal with it here.
- For FO:
  - Combined – PSPACE.
  - Data – AC^0.

Complexity of model-checking lowered

Conjunctive queries – the $\exists, \wedge$-fragment of first-order logic:

$$\varphi(\bar{x}) \quad = \quad \exists \bar{y} \, \left(R_1(\bar{u}_1) \wedge \ldots \wedge R_n(\bar{u}_n) \right)$$

- where $\bar{u}_i$ list variables among $\bar{x}, \bar{y}$
- they capture select-project-join queries; the best studied class of database queries.
- Combined complexity: NP.
- The same is true for unions of conjunctive queries, i.e., existential positive formulae (the $\exists, \wedge, \vee$-fragment of FO).

Validity: Complexity

- Version 1: Given φ, it is true in every finite structure?
- Version 2: Given D and φ, is $\psi_D \rightarrow \varphi$ true in every finite structure?
- For FO: both are undecidable.
- But if φ is a conjunctive query, it is in NP (Chandra/Merlin 1978):

$$\models \psi_D \rightarrow \varphi \quad \Leftrightarrow \quad D \models \varphi$$

- A similar argument works for unions of conjunctive queries.
- Thus, validity is reduced to model-checking

Incomplete information

- It is everywhere.
- 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
- One tries to clean but not always possible
- The importance of it was recognized early
  - Codd, *"Understanding relations (installment #7)"*, 1975.
- And yet the state is very poor:
  - Both practice and theory

Incomplete information: basic idea

We have an incomplete database D.

It represents many possible complete databases D'.

Refer to them as the semantics of an incomplete database:

$$[\![D]\!] \quad = \quad \{ \text{ all } D' \text{ that } D \text{ represents } \}$$

Models of incompleteness: missing data

Semantics is most commonly defined via homomorphisms.

Query answering: Given φ and an incomplete D, want to check if φ is true with certainty:

$$\forall D' \in [\![D]\!] : \quad D' \models \varphi$$

If $[\![D]\!]$ is given by ψ_D, this is $\models \psi_D \to \varphi$ — validity!

Want to reduce to model checking, or query evaluation, ideally $D \models \varphi$

Turning a highly intractable problem into a highly tractable one.

The idea goes back to the 1980s (Imielinski/Lipski 1984, Reiter 1986, Vardi 1986)

Definitions

Vocabularies: relation names $R_1, \ldots, R_n$ and their arities.

Relational structures, also known as databases:

$$D = \langle U, R_1, \ldots, R_n \rangle$$

- U is the universe
- $R_i^D \subseteq U^{\text{arity}(R_i)}$
- Convention: every element of U occurs in some relation R_i^D
- Convention: often omit superscript ^{D} from R_i^D

Relational databases

Finite first-order structures of relational vocabulary. In the example below, we have R_1 of arity 3, and R_2 of arity 2.

In databases, columns are named (they are attributes of a relation).

R_1:

A	B	C
1	2	5
3	4	3
2	5	1
2	6	3

R_2:

B	D
2	7
3	5
4	1

Queries: fragments of first-order logic (FO)

A very common fragment – conjunctive queries – the $\exists, \wedge$ fragment of FO

Example: $\varphi(x) \;=\; \exists y, z \; R_1(x, y, z) \wedge R_2(z, y)$

Relational databases with missing information

Domain: disjoint union of

- constants like 1,2, etc
- nulls, denoted by $\perp_1$, $\perp_2$, etc.
- Meaning: a value is missing, unknown at present.

R_1:

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

R_2:

B	D
2	$\perp_2$
3	5
4	$\perp_4$

Homomorphisms

For two databases

$$D = \langle U, R_1^D, \ldots, R_n^D \rangle \quad \text{and} \quad D' = \langle U', R_1^{D'}, \ldots, R_n^{D'} \rangle$$

a homomorphism is a map $h : U \to U'$ such that:

- $h(c) = c$ if $c \in \mathbf{C}$
- if $\bar{a} \in R_i^D$, then $h(\bar{a}) \in R_i^{D'}$.

A homomorphism is

- onto if $h(U) = U'$
- strong onto if $h(D) = D'$

A homomorphism is a valuation (of nulls) if $h(U) \subset \mathbf{C}$.

Two common semantics

Semantics via valuation of nulls

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

Two common semantics

Semantics via valuation of nulls

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

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

Two common semantics

Semantics via valuation of nulls

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

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

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

Two common semantics

Semantics via valuation of nulls

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

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

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

Closed-World-Assumption semantics (CWA semantics):

$$[\![D]\!]_{\text{cwa}} = \left\{ v(D) \;\middle|\; v \text{ is a valuation} \right\}$$

Two common semantics

Semantics via valuation of nulls

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

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

A	B	C
1	2	4
3	4	3
5	5	1
2	5	3
3	7	8
4	2	1

Two common semantics

Semantics via valuation of nulls

A	B	C
1	2	$\perp_1$
$\perp_2$	$\perp_1$	3
$\perp_3$	5	1
2	$\perp_3$	3

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

A	B	C
1	2	4
3	4	3
5	5	1
2	5	3
3	7	8
4	2	1

Open-World-Assumption semantics (OWA semantics):

$$[\![D]\!]_{\text{owa}} = \left\{ \text{complete } D' \;\middle|\; v(D) \subseteq D' \text{ for some valuation } v \right\}$$

Formulae ψ_D defining semantics

Complete models of ψ_D $=$ $[\![D]\!]$

Formulae ψ_D defining semantics

Complete models of ψ_D $=$ $[\![D]\!]$

- D $=$

1	2
3	$\bot$

Formulae ψ_D defining semantics

Complete models of $\psi_D \;=\; [\![D]\!]$

- $D \;=$

1	2
3	$\bot$

- under OWA: $\exists x\; D(1,2) \;\wedge\; D(3,x)$ – a conjunctive query

Formulae ψ_D defining semantics

Complete models of $\psi_D \;=\; [\![D]\!]$

- $D \;=$

1	2
3	$\bot$

- under OWA: $\exists x\ D(1,2) \;\wedge\; D(3,x)$ – a conjunctive query
- under CWA:

$$\exists x \left(\begin{array}{ll} & D(1,2) \;\wedge\; D(3,x) \\ \wedge & \forall y,z\ D(y,z) \rightarrow \left(\begin{array}{ll} & (y,z) = (1,2) \\ \vee & (y,z) = (3,x) \end{array} \right) \end{array} \right)$$

- We'll see this class later: positive formulae with universal guards

Query evaluation: certain answers

Given:

- an incomplete database D
- a query φ, for now a sentence

we want to find certain answers:

$$\text{certain}(\varphi, D) = \bigwedge_{D' \in [\![D]\!]} \varphi(D')$$

i.e., the answer that does not depend on the interpretation of nulls.

Certain answers = validity of $\psi_D \to \varphi$.

Complexity of certain answers

Language: first-order logic. Known results (Abiteboul, Kanellakis, Grahne, 1991; Gheerbrant, L., Tan, 2012):

For OWA: undecidable, even in data complexity.

- Trakhtenbrot's theorem for combined complexity (for data complexity too, but some work is needed).

For CWA: coNP-complete.

- Just guess a valuation so that $v(D) \models \neg\varphi$.

When validity = model checking/query evaluation

We have good answers for both OWA and CWA.

Naïve evaluation for conjunctive queries

Suppose φ is a conjunctive query. Then

$$\begin{array}{c} \text{certain}(\varphi, D) = \text{ true} \\ \Updownarrow \\ \psi_D \to \varphi \text{ is valid} \\ \Updownarrow \\ D \models \varphi \end{array}$$

When φ, ψ are conjunctive queries, the following are equivalent:

- $\models \psi \to \varphi$
- there is a homomorphism from CD_φ to CD_ψ
  - $CD_{\exists \bar{x}\ R_1(\bar{u}_1) \wedge \ldots, R_n(\bar{u}_n)} = \{R_1(\bar{u}_1), \ldots, R_n(\bar{u}_n)\}$
- $CD_\psi \models \varphi$

Since $CD_{\psi_D} = D$ we have $\models \psi_D \to \varphi \ \Leftrightarrow \ D \models \varphi$

Naïve evaluation for conjunctive queries cont'd

Database people call this naïve evaluation of queries over incomplete databases:

- evaluate φ as if nulls were values (e.g., $\perp_1 = \perp_1$, $\perp_1 \neq \perp_2$, $\perp_1 \neq 5$ etc)
- Continues to work for unions of conjunctive queries
- At the core of many database applications (especially in data integration and exchange)

First discovered in 1984 by Imielinski and Lipski.

Works for unions of conjunctive queries (existential positive queries):

- essentially the same argument

Limits of naive evaluation

How far can we push this?

Certain answers to φ can be found by naïve evaluation if for all S:

$$\mathsf{certain}(D, \varphi) = \mathsf{true} \quad \Leftrightarrow \quad D \models \varphi$$

Limits of naive evaluation

How far can we push this?

Certain answers to φ can be found by naïve evaluation if for all S:

$$\text{certain}(D, \varphi) = \text{true} \quad \Leftrightarrow \quad D \models \varphi$$

Within FO, cannot extend this at all:

Theorem

(L., 2011, an application of Rossman's theorem) Let φ be an FO sentence such that certain answers to φ can be found by naïve evaluation under OWA. Then φ is equivalent to a union of conjunctive queries (i.e., an existential positive formula).

Can this be extended beyond OWA?

Approach of Gheerbrant, L., Sirangelo, 2013:

- Order databases by $D \preceq D'$ iff $[\![D']\!] \subseteq [\![D]\!]$
  - Idea: D' is more informative
- If $D \models \varphi$ and $D \preceq D'$ imply $D' \models \varphi$ then

$$\mathsf{certain}(\varphi, D) = \mathsf{true} \quad = \quad D \models \varphi$$

- Condition $D \models \varphi, \quad D \preceq D' \quad \Rightarrow \quad D' \models \varphi$:

preservation under $\preceq$

Preservation and naïve evaluation

- What are the orderings $\preceq$ for OWA and CWA?
- $D \preceq D'$ iff
  - For OWA: there is a homomorphism from D to D'
  - For CWA: there is a strong onto homomorphism from D to D'

Hence reduction of certain answers/validity to model checking = preservation under homomorphisms

Preservation and naïve evaluation

- What are the orderings $\preceq$ for OWA and CWA?
- $D \preceq D'$ iff
  - For OWA: there is a homomorphism from D to D'
  - For CWA: there is a strong onto homomorphism from D to D'

Hence reduction of certain answers/validity to model checking = preservation under homomorphisms

Remark: there is a semantics that corresponds to preservation under onto homomorphisms, called *weak closed-world*. Won't deal with it here.

Preservation under homomorphisms

Classical results in model theory for FO formulae:

- Preservation under homomorphisms = existential positive formulae
  - built using $\wedge, \vee, \exists$
  - Rossman 2005: both finite and infinite cases
- Preservation under onto homomorphisms = positive formulae
  - built using $\wedge, \vee, \exists, \forall$
  - Lyndon 1959, for all models; fails in the finite (Stolboushkin 1995)
- Preservation under strong onto homomorphism: a more complicated story
  - A result by Keisler 1965, only for a vocabulary of graphs
  - A rather unpleasant syntax
  - May well be true but a crucial lemma is false
  - But another fragment, discovered by Compton 1983, works

Fragment $\mathsf{Pos}^{\forall \mathsf{G}}$ (positive + universal guards)

Rules for positive formulae:

- Atoms $R(\bar{x})$ and $x = y$ are in $\mathsf{Pos}^{\forall \mathsf{G}}$
- If φ, ψ are in $\mathsf{Pos}^{\forall \mathsf{G}}$ then so are $\varphi \vee \psi$ and $\varphi \wedge \psi$
- If φ is in $\mathsf{Pos}^{\forall \mathsf{G}}$ then so are $\exists x \varphi$ and $\forall x \varphi$
- If φ is in $\mathsf{Pos}^{\forall \mathsf{G}}$ then so are $\exists x \varphi$ and $\forall x \varphi$

New guard rule:

- If φ is in $\mathsf{Pos}^{\forall \mathsf{G}}$, and $\psi(\bar{x})$ is an atom, then

$$\forall \bar{x} \, (\psi(\bar{x}) \; \rightarrow \; \varphi)$$

is in $\mathsf{Pos}^{\forall \mathsf{G}}$.

$\mathrm{Pos}^{\forall \mathrm{G}}$

- $\mathrm{Pos}^{\forall \mathrm{G}}$ formulae are preserved under strong onto homomorphisms.
- Consequently, under CWA,

$$\mathsf{certain}(\varphi, D) = \mathsf{true} \quad \Leftrightarrow \quad D \models \varphi$$

Can this class be further extended? Probably not, or not much.

$\mathrm{Pos}^{\forall \mathrm{G}}$ describes the fragment of FO preserved under strong onto homomorphisms on all structures.

$\mathrm{Pos}^{\forall \mathrm{G}}$

- $\mathrm{Pos}^{\forall \mathrm{G}}$ formulae are preserved under strong onto homomorphisms.
- Consequently, under CWA,

$$\mathrm{certain}(\varphi, D) = \mathrm{true} \quad \Leftrightarrow \quad D \models \varphi$$

Can this class be further extended? Probably not, or not much.

$\mathrm{Pos}^{\forall \mathrm{G}}$ describes the fragment of FO preserved under strong onto homomorphisms on all structures.

Another look at it

Reminder – relational algebra: a procedural language on relations equivalent to FO.

That's – in theory – how FO queries are implemented in databases in practice.

Operations:

- Projection π
- Selection σ
- Cartesian product $\times$
- Union $\cup$
- Difference $-$

- Derived operations: intersection $\cap$, join $\bowtie$, division $\div$

Existential positive $=$ $\sigma, \pi, \times, \cup$

Going beyond existential positive

Reminder – relational algebra division

A	B
a	1
a	2
b	1

$\div$

B
1
2

$=$

A
a

Very common in queries with universal conditions/negation ("find suppliers who supply all parts")

Going beyond existential positive

Reminder – relational algebra division

A	B
a	1
a	2
b	1

$\div$

B
1
2

$=$

A
a

Very common in queries with universal conditions/negation ("find suppliers who supply all parts")

For a query Q expressed with

- $\sigma, \pi, \bowtie, \cup$, and
- $R \div S$, where S is a relation in the database,

certain answers are computed by naive evaluation of Q

What if $D \models \varphi$ does not check validity?

Next thing:

$$\text{certain}(\varphi, D) \;=\; \text{true} \quad \Leftrightarrow \quad \text{algorithm}(\varphi, D) \text{ says yes}$$

The algorithm better be tractable.

When can this work?

- We need to add some form of negation to existential positive formulae: for them, everything works
- Possibility 1: add inequalities to conjunctive queries
  - E.g., $\exists x, y\ S(x, y) \wedge x \neq y$
- Possibility 2: add Boolean combinations of conjunctive queries
  - E.g., $\varphi_1 \wedge (\neg\varphi_2 \vee \varphi_3) \wedge \neg\varphi_4$ where the φ_is are conjunctive queries.
- Good news: in both cases the combined complexity of certain answers is in Π_2^p
  - follows from Sagiv/Yannakakis 1980 and Klug 1988

Beyond unions of conjunctive queries cont'd

- We want tractable data complexity: computing certain(φ, S) polynomial in S
- Bad news: for conjunctive queries with inequalities, data complexity is coNP-complete (Klug; Fagin et al)

A simple proof for unions of conjunctive queries with inequalities

- Given a graph $G = \langle V, E \rangle$ with $V = \{\perp_1, \ldots, \perp_n\}$.
- Consider

$$\varphi \;=\; \exists x\, E(x,x) \quad \vee \quad \exists x_1, x_2, x_3, x_4 \bigwedge_{i \neq j} x_i \neq x_j$$

- certain$(\varphi, G) =$ true iff G is not 3-colorable.

Beyond unions of conjunctive queries cont'd

Good news: for Boolean combinations of conjunctive queries, data complexity stays in PTIME (Gheerbrant, L., 2012)

- of course we cannot use naïve evaluation
- but we can effectively search for a counter-model instead
- works for both OWA and CWA, but algorithms are different
- We sketch them now, by means of simple examples.

The OWA algorithm by example

The basic case: $q = \psi \to \varphi$, where

- ψ is a conjunctive query,
- φ is a union of conjunctive queries.

It has only one falsifying valuation

ψ	φ	$\psi \to \varphi$
false	false	true
true	true	true
true	false	false
false	true	true

The algorithm by example

Relation D:

1	3	$\perp_1$
2	3	$\perp_1$

Queries:

- ψ: there is a tuple (1,_,4) is in the relation.
- φ: there is a tuple (2,_,4) is in the relation.
- $q = \psi \to \varphi$

The OWA algorithm by example cont'd

1. Convert ψ into its tableau and glue it into D:

1	3	$\perp_1$
2	3	$\perp_1$
1	$\perp_2$	4

2. Evaluate φ naïvely on the new relation.
3. Verdict: it's not true.
4. We found our counter model.
5. Hence certain$(q, D) =$ false.

Generalization to arbitrary Boolean combinations

1. list every falsifying valuations for the Boolean combination;
2. apply the previous procedure for each one of them;
3. the number of valuations is fixed since the query is fixed.

It still works under CWA

Idea: We want to extend D (in a "closed-world" way) so that it satisfies ψ when we deal with queries $\psi \to \varphi$.

1	3	$\bot_1$
2	3	$\bot_1$

There is only one way to do it for the query
$\exists x\ D(1, x, 4)\ \to\ \exists y\ D(2, y, 4)$

1	3	4
2	3	4

The counter model search aborted: $\mathsf{certain}_{\mathsf{CWA}}(\psi \to \varphi, D) = \mathit{true}$.

Will this work?

By work we mean: in real databases.

Not a chance, this is not how query evaluation algorithms are implemented.

Only suitable for "small data".

Let's see now what happens in real databases and what we can do.

Incomplete information

We accumulate a lot of incomplete data, but handling incomplete information by relational database products (SQL) is very problematic:

- *"Those SQL features are . . . fundamentally at odds with the way the world behaves"* (Date & Darwen, 'A Guide to SQL Standard')
- *"If you have any nulls in your database, you're getting wrong answers to some of your queries . . . you have no way of knowing which queries you're getting wrong answers to"*
- *"You can never trust the answers you get from a database with nulls"* (Date, 'Database in Depth')

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)
```

Answer: ~~Ord3~~ none

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~~ 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

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

- Of course now queries are more than sentences: they can return sets of tuples.
- Standard approach, used in all applications: data integration and exchange, inconsistent data, querying with ontologies, data cleaning.
- If we model true as $\{()\}$ and false as $\{\}$, this is exactly what we had before for sentences.

Correctness guarantees: involving nulls

A small problem with the above definition: it eliminates all tuples with nulls.

Consider

R =

1	2
3	$\perp$

and Q is $R(x, y)$ (just return R itself).

certain(Q, R) =

1	2

Correctness guarantees: involving nulls

A natural extension of the standard definition of certain answers (proposed by Lipski in 1984 but quickly forgotten).

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

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

Correctness guarantees: involving nulls

A natural extension of the standard definition of certain answers (proposed by Lipski 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, $\text{certain}_\perp(Q, D)$, if

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

Correctness guarantees: involving nulls

A natural extension of the standard definition of certain answers (proposed by Lipski 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, $\text{certain}_\perp(Q, D)$, if

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

Known: $\text{certain}(Q, D)$ is the set of null free tuples in $\text{certain}_\perp(Q, D)$

$$\text{certain}(Q, D) \subseteq \text{certain}_\perp(Q, D) \subseteq \text{naive evaluation of } Q \text{ on } D$$

Certain answers and conjunctive queries

A very typical picture:

- Certain answers can be computed efficiently for conjunctive queries
  - perhaps extensions like unions of conjunctive queries
  - or perhaps a fragment of conjunctive queries
- Outside conjunctive queries, certain answers are computationally intractable
  - often see coNP, Π_2^p, coNEXP lower bounds, even undecidability
- The behavior of a database theoretician: move on to the next problem, establish more lower bounds outside of conjunctive queries

Certain answers and conjunctive queries

A very typical picture:

- Certain answers can be computed efficiently for conjunctive queries
  - perhaps extensions like unions of conjunctive queries
  - or perhaps a fragment of conjunctive queries
- Outside conjunctive queries, certain answers are computationally intractable
  - often see coNP, Π_2^p, coNEXP lower bounds, even undecidability
- The behavior of a database theoretician: move on to the next problem, establish more lower bounds outside of conjunctive queries

How can SQL differ from certain answers?

- SQL can produce false negatives: miss some of the certain answers
- SQL can produce false positives: return answers that are not certain.
- We view false positives as being much worse: telling a lie as opposed to not telling the whole truth.
- We want to impose correctness guarantees: no false positives.

Life beyond conjunctive queries

Question 1: Does SQL compute wrong (non-certain) answers for real-life queries?

Question 2: If SQL cannot compute certain answers, can it approximate them?

Do wrong answers really occur?

Wrong answers (false positives) are tuples that

- returned by SQL evaluation, but
- are not certain answers.

What we know (L., 2015; Console, Guagliardo, L. 2016):

- For SQL queries without negation, there are no false positives.
- Otherwise (e.g., with `not exists`), they can occur.
- If we use Boolean logic to evaluate $\wedge, \vee, \neg$, they will occur.

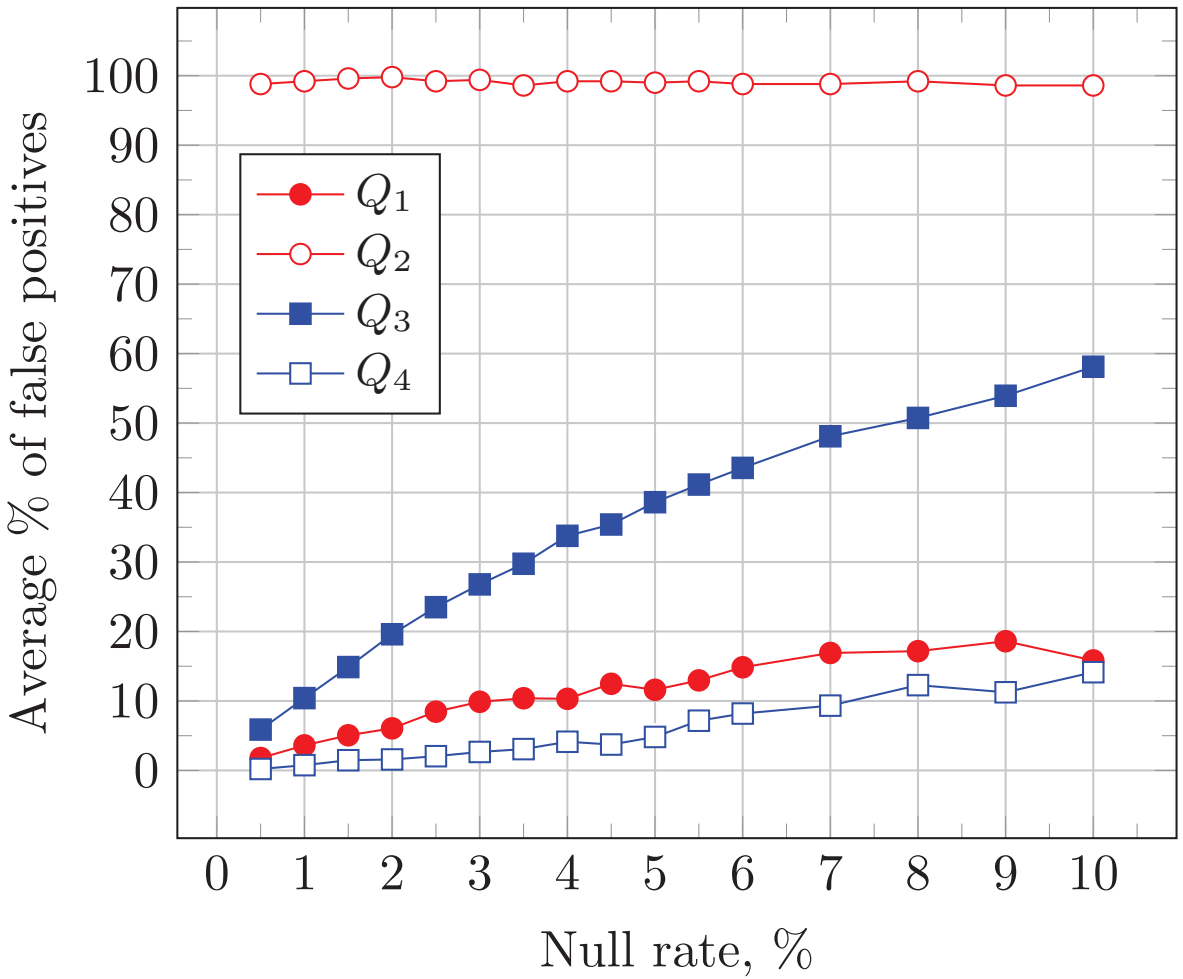
But how big of a problem is it for real-life queries?

Wrong answers and queries with negation

- A good source of queries: TPC benchmarks (especially TPC-H, typical decision support queries)
- Issues:
  1. surprisingly few have negation (Q21 and Q22)
  2. the standard data generator does not produce nulls
  3. checking for certain answers is coNP-hard
- Solutions:
  1. complement with typical textbook queries that involve negation
  2. generate nulls randomly in nullable attributes
  3. For queries in our experiments, design – by hand – algorithms that let us quickly find lower bounds on the number of nulls.

False positives – lots of them

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



SQL and certain answers

Can SQL evaluation be changed to coincide with certain answers?

No!

SQL and certain answers

Can SQL evaluation be changed to coincide with certain answers?

No!

Complexity argument:

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

SQL and certain answers

Can SQL evaluation be changed to coincide with certain answers?

No!

Complexity argument:

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

Can SQL evaluation be changed to produce only certain answers?

Yes!

In more than a single way.

Old solutions

Reiter 1986, Vardi 1986:

- Represented databases as logical theories;
- Queries must be logical formulae in a special shape, so that application of negation is restricted.

Wouldn't work for real databases, but introduced two important ideas:

- approximations for certain answers computed by evaluating queries on a database with nulls;
- the idea of matching/unification for handling queries with negation.

We'll see now how these ideas work in modern solutions.

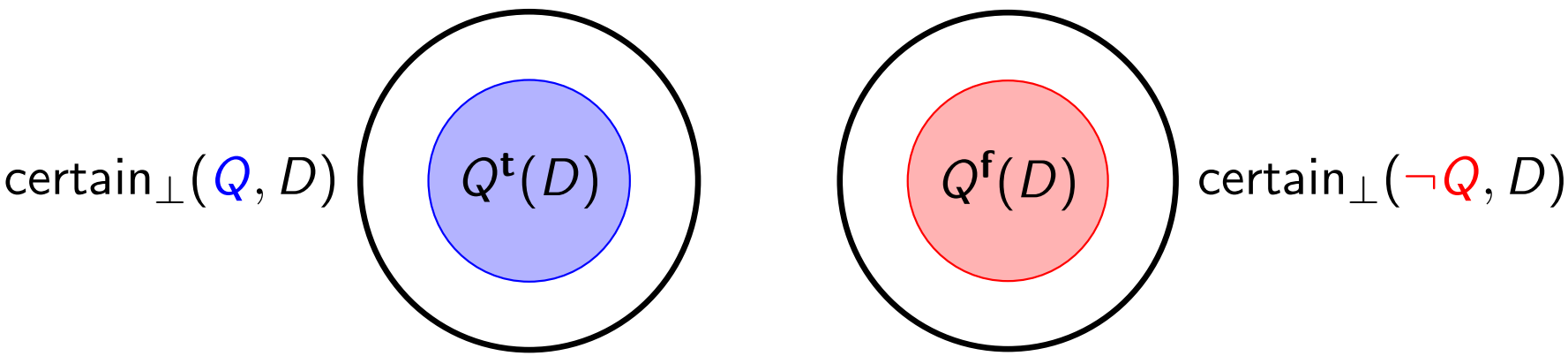
The first solution

There is an effective translation of queries (L., 2015)

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

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

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

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

adom: the universe (set of all elements in the database)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

AC^0 and efficiency are NOT the same!

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

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

What is "match"?

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

Left unification antijoin

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

Unifying tuples: an illustration

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

$$\begin{pmatrix} 1 & \bot & 1 & 3 \\ \bot' & 2 & \bot' & 3 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 2 & 1 & 3 \end{pmatrix}$$

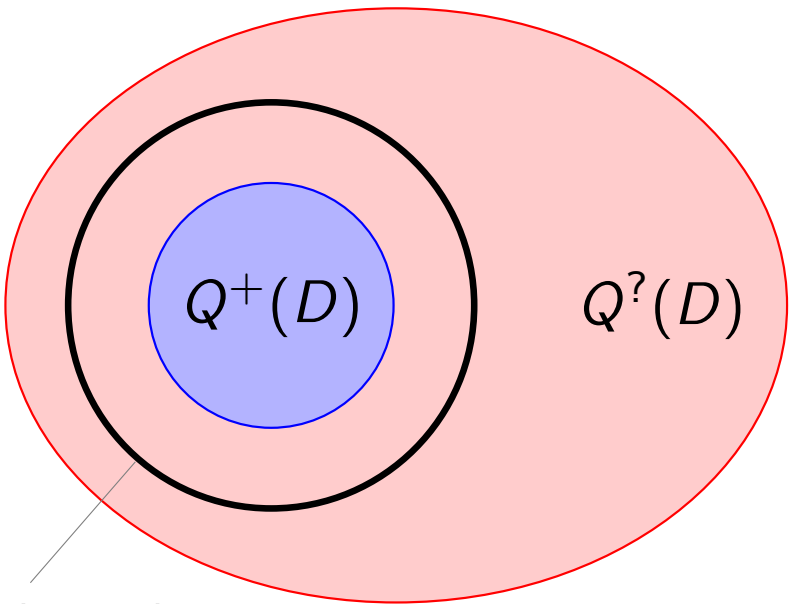
but $\begin{pmatrix} 1 & \bot & 2 & 3 \\ \bot' & 2 & \bot' & 3 \end{pmatrix}$ do not unify.

This can be checked in linear time.

New translation

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

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

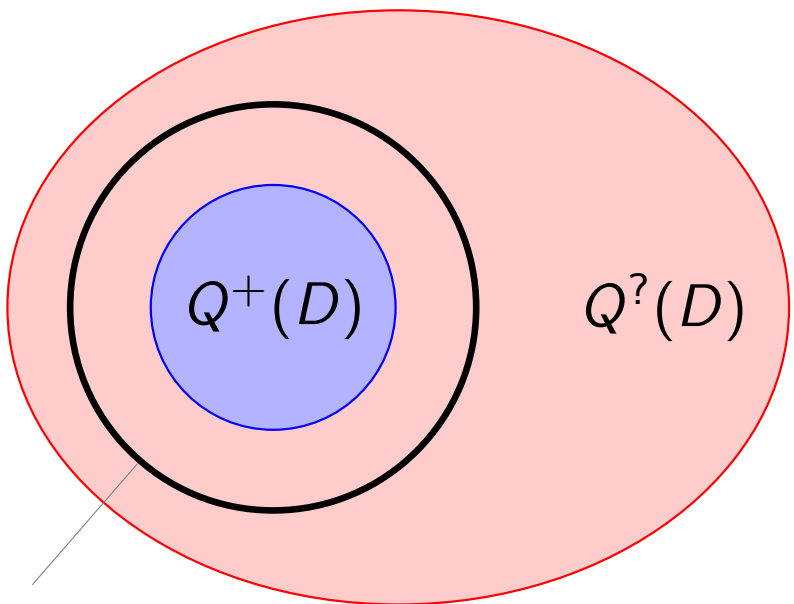


$$(Q_1 - Q_2)^{+} = Q_1^{+} \,\overline{\ltimes}_u\, Q_2^{?}$$
$$R^{?} = R$$
$$(Q_1 \cup Q_2)^{?} = Q_1^{?} \cup Q_2^{?}$$
$$(Q_1 \cap Q_2)^{?} = Q_1^{?} \ltimes_u Q_2^{?}$$
$$(Q_1 - Q_2)^{?} = Q_1^{?} - Q_2^{+}$$
$$\left(\sigma_{\theta}(Q)\right)^{?} = \sigma_{\neg(\neg\theta)^{*}}\left(Q^{?}\right)$$
$$(Q_1 \times Q_2)^{?} = Q_1^{?} \times Q_2^{?}$$
$$\left(\pi_{\alpha}(Q)\right)^{?} = \pi_{\alpha}\left(Q^{?}\right)$$

New translation

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

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



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

$$R^? = R$$

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

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

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

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

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

$$\left(\pi_\alpha(Q)\right)^? = \pi_\alpha\left(Q^?\right)$$

Certain and possible answers

(Guagliardo, L., 2016) For every valuation h of nulls:

$$h(Q^+(D)) \subseteq Q(h(D))$$

$$Q(h(D)) \subseteq h(Q^?(D))$$

- 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:

$$Q^{\boldsymbol{t}} \;=\; R \cap \big((\pi_\alpha(\mathsf{adom}^k \,\overline{\ltimes}_u\, T) - \pi_\alpha(\mathsf{adom}^k \ltimes_u T)) \cup \sigma_{\theta^*}(S)\big)$$

(uncomputable in practice) but

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

(easy to compute)

Does it work in practice?

We ran our queries and translations on TPC-H instances with nulls and measured 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?

We ran our queries and translations on TPC-H instances with nulls and measured 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)

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: $\pi_{\texttt{o\_orderkey}}\left(\texttt{orders} \,\overline{\ltimes}_{\theta}\, \texttt{lineitem}\right)$

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

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: $\pi_{\texttt{o\_orderkey}}\left(\texttt{orders} \,\overline{\ltimes}_{\theta}\, \texttt{lineitem}\right)$

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

The good

Q_3 Find orders supplied entirely by a specific supplier

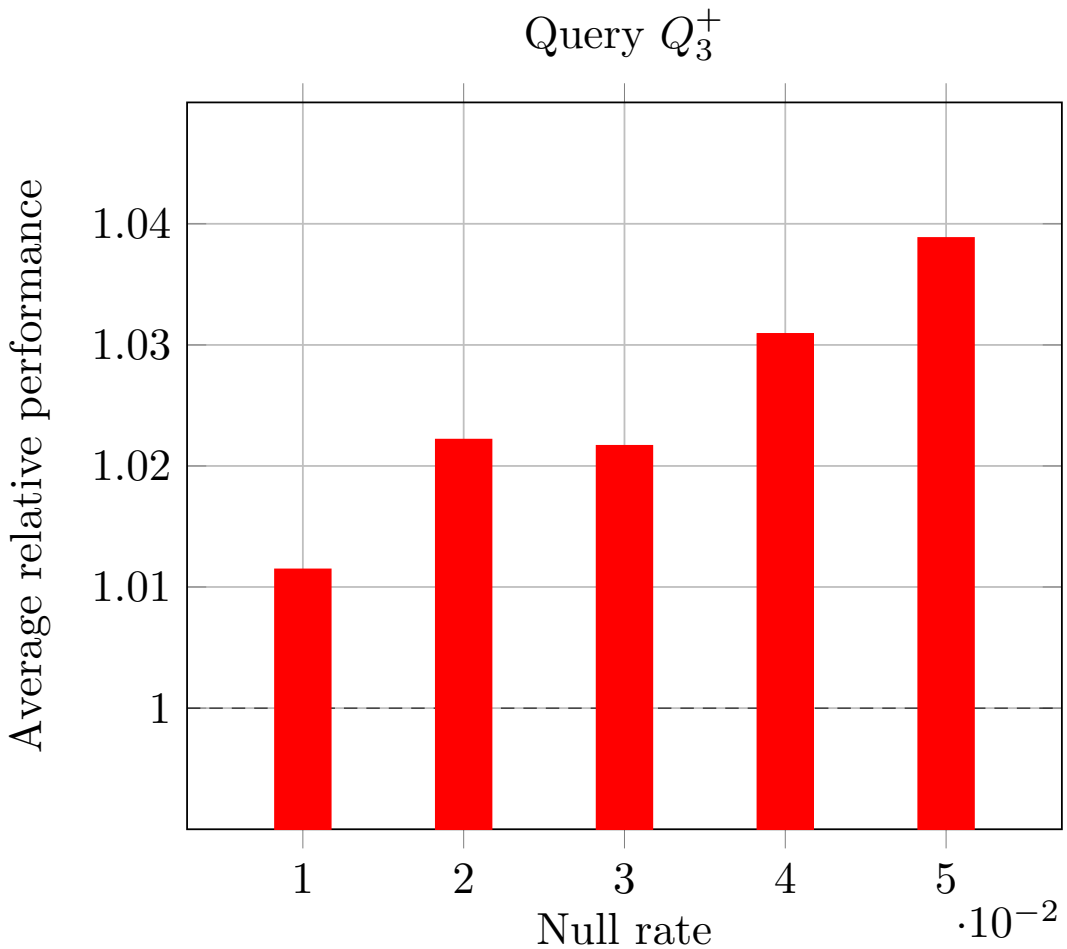
```
SELECT o_orderkey
FROM   orders
WHERE  NOT EXISTS (
       SELECT   *
       FROM     lineitem
       WHERE    l_orderkey = o_orderkey


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

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

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

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 )
```

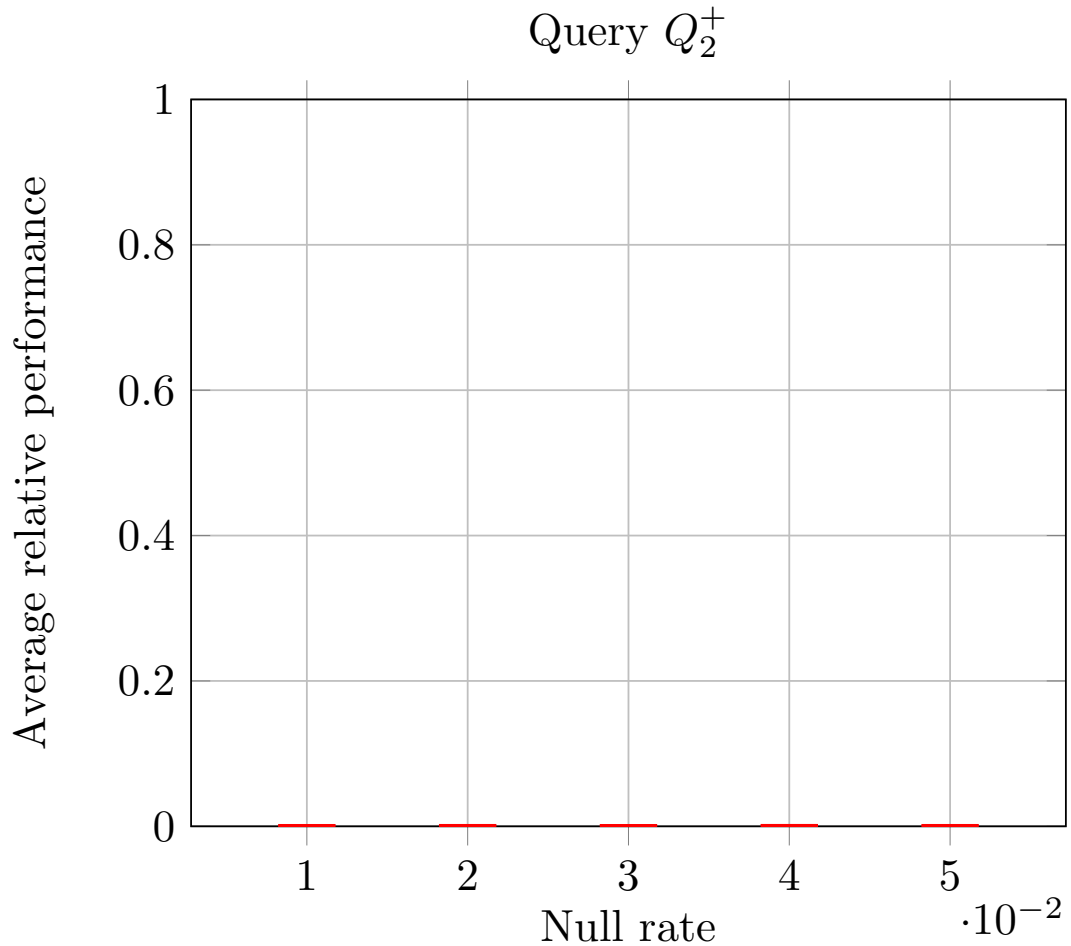
The fantastic

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

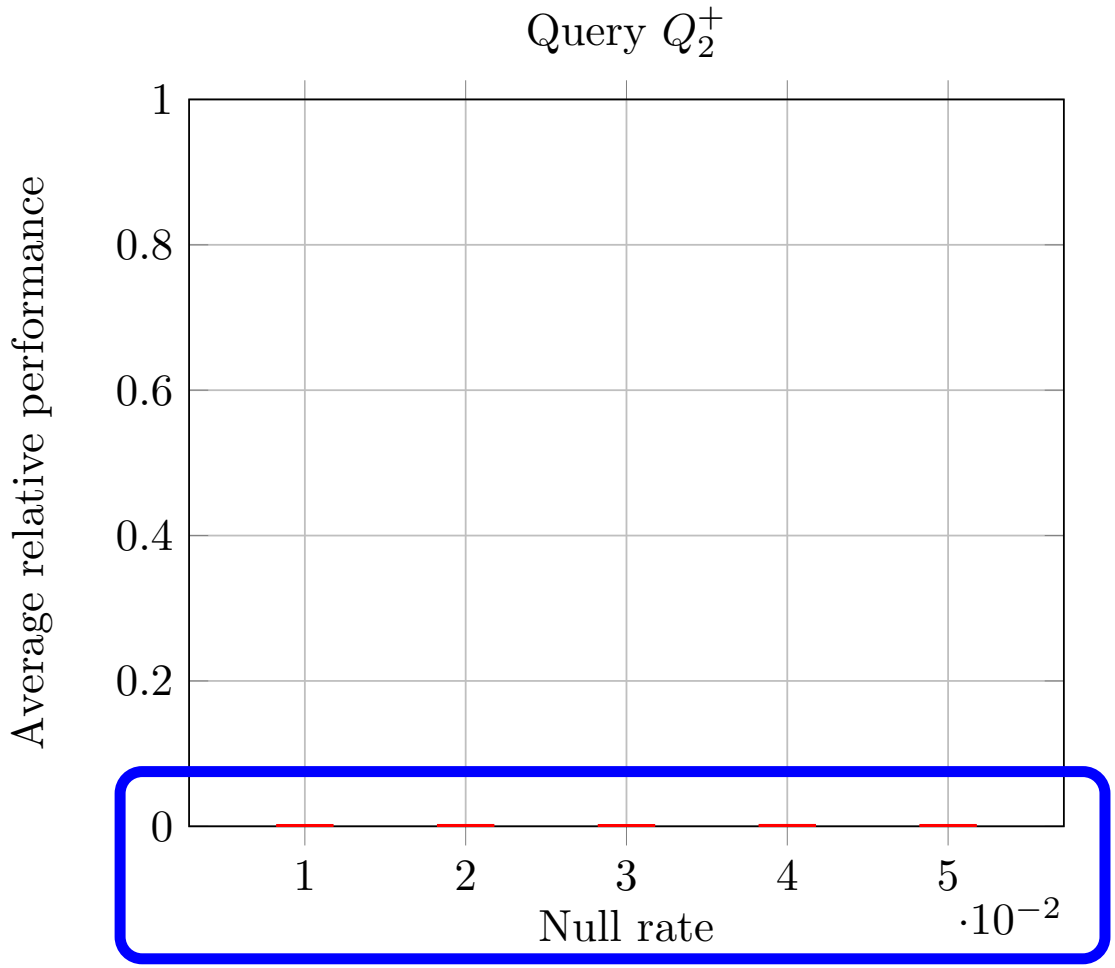
Over 10^3 times faster (same or better up to 10GB)



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

The fantastic: Results

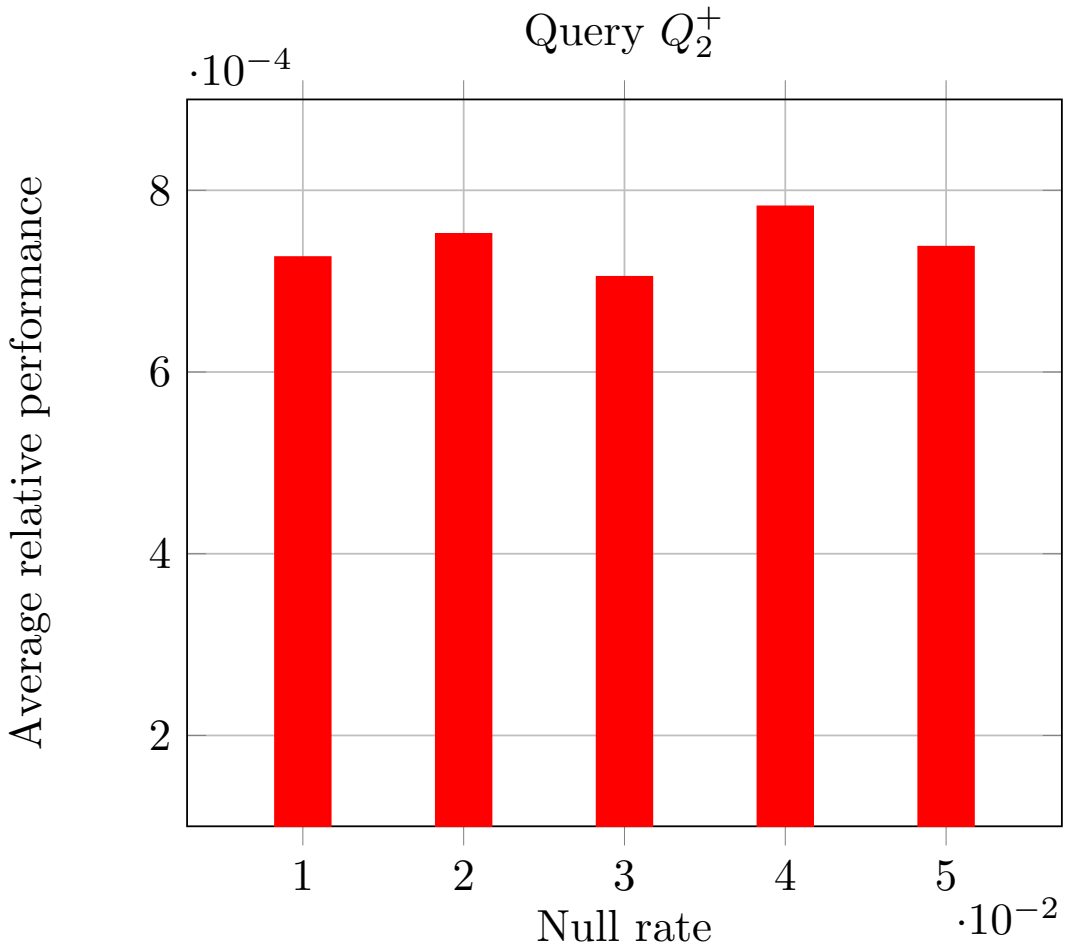
Over 10^3 times faster (same or better up to 10GB)



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

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

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

- 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

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

As bad as computing a Cartesian product

We can do better

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

The tolerable: Problems with the optimizer

Joins with disjunctions in correlated subqueries

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

As bad as computing a Cartesian product

We can do better

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

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

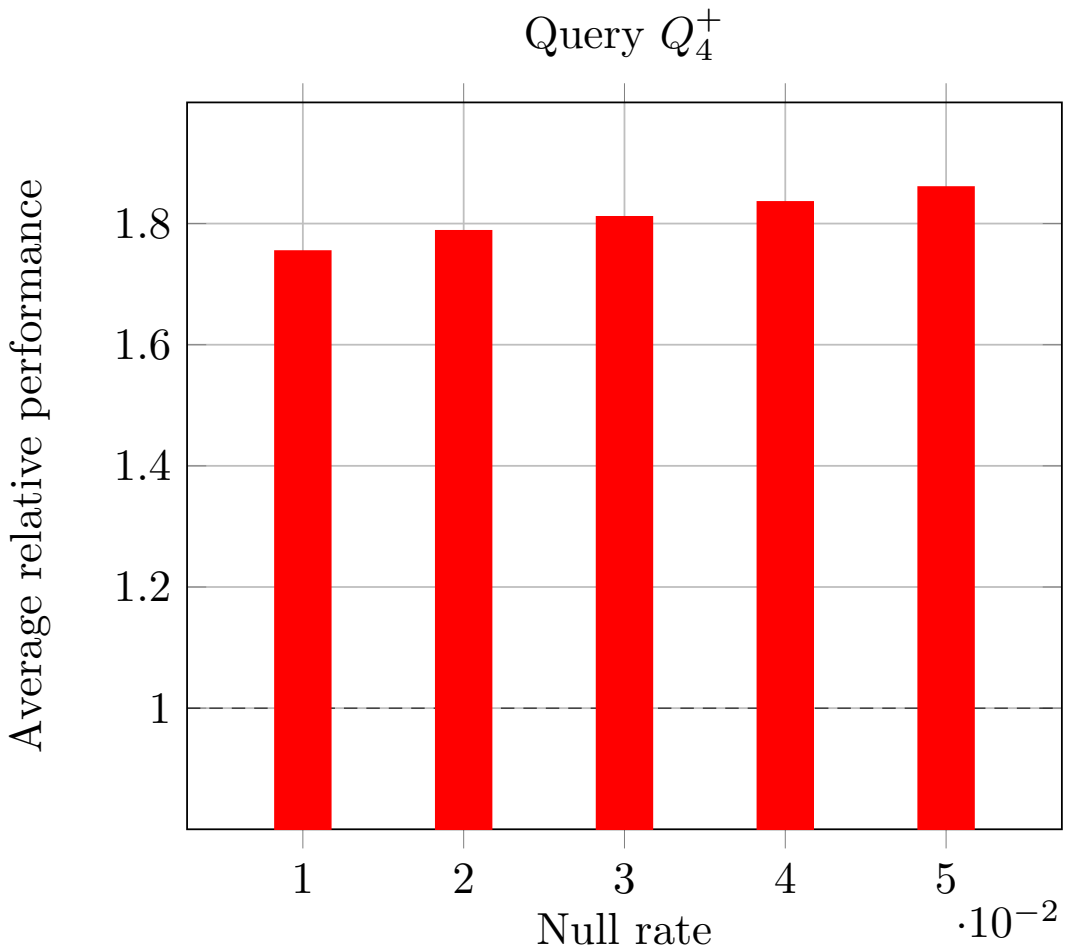
- Exponential blowup of the query.
- Complexity went from $|D|^{O(|Q|)}$ to $|D|^{2^{O(|Q|)}}$.
  - Double-exponential query complexity!
  - Theory teaches us that this is impossible to evaluate.
- Split one nested subquery into several ones.
  - Practice teaches us that this is much harder to evaluate.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

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

- 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!
- From Postgres' optimizer source code:
  `/* We stop as soon as we hit a non-AND item */`

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	$\perp_1$
$\perp_2$	$\perp_3$	3
$\perp_4$	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 | $\perp_1$ | it returns 1

What's next

- If one wants to live with wrong answers, who are we to tell them that they cannot?
  - Very true in the UK now; perhaps in the US in November?
- But for whose who care, two new modes of evaluation:

  SELECT CERTAIN and SELECT POSSIBLE

  to under- and over-approximate certain answers.

What's needed for SELECT CERTAIN/POSSIBLE?

- Well under way or done:
  - Implementation of marked nulls in Postgres for better translations
  - Direct SQL-to-SQL translation
  - Incorporating integrity constraints (keys, foreign keys).
  - Bag semantics.
- Uncharted territory:
  - Aggregate queries.
  - Other nulls (especially nonapplicable: outerjoins).

References

1. A. Gheerbrant, L., C. Sirangelo. Naïve evaluation of queries over incomplete databases. *ACM TODS* 39(4):(2014). (First in PODS'13.)
2. L. SQL's three-valued logic and certain answers. *ACM TODS* 41(1):(2016). (First in ICDT'15.)
3. M. Console, P. Guagliardo, L. Approximations and refinements of certain answers via many-valued logics. *KR 2016*.
4. P. Guagliardo, L. Making SQL queries correct on incomplete databases: a feasibility study. *PODS 2016*.

Thank you!

Questions?

References

1. A. Gheerbrant, L., C. Sirangelo. Naïve evaluation of queries over incomplete databases. *ACM TODS* 39(4):(2014). (First in PODS'13.)
2. L. SQL's three-valued logic and certain answers. *ACM TODS* 41(1):(2016). (First in ICDT'15.)
3. M. Console, P. Guagliardo, L. Approximations and refinements of certain answers via many-valued logics. *KR 2016*.
4. P. Guagliardo, L. Making SQL queries correct on incomplete databases: a feasibility study. *PODS 2016*.

Thank you!

Questions?