Do we really understand SQL?

Leonid Libkin
University of Edinburgh

Joint work with Paolo Guagliardo, also from Edinburgh

Basic questions

- We are taught that the core of SQL is essentially syntax for relational calculus (first-order logic). Is it true?
- We are taught that core SQL can be translated into relational algebra. Is it true?
- We are taught that SQL needs 3-valued logic to deal with missing information (nulls). Is it true?

Motivation

- Why even ask such questions? It's the stuff from the 1980s (or earlier). It's all in database textbooks and taught in all database courses.
- This is exactly what we thought until we got into some specific problems related to real-life SQL
  - So we start with a bit of history

Old days (before 1969)

Various ad-hoc database modes:

- network
- hierarchical

writing queries: a very elaborate task

All changed in 1969: Codd's relational model;
now dominates the world

Relational Model

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

Relational Model

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

Language: **Relational Algebra (RA)**

- projection **π** (find book titles)
- selection **σ** (find books that cost at least £40)
- Cartesian product **×**
- union **U**
- difference **-**

Queries

Find ids of customers who buy all books:

$$\pi_{cust\_id}(Pay) - \pi_{cust\_id}\Big((\pi_{cust\_id}(Pay) \times \pi_{title}(Order)) - \pi_{cust\_id,title}\big(\sigma_{order\_id=order}(Order \times Pay)\big)\Big)$$

Queries

Find ids of customers who buy all books:

$$\pi_{cust\_id}(Pay) -$$
$$\pi_{cust\_id}\big((\pi_{cust\_id}(Pay) \times \pi_{title}(Order)\big) -$$
$$\pi_{cust\_id,title}\big(\sigma_{order\_id=order}(Order \times Pay)\big)\big)$$

That's not pretty. But here is a better idea (1971):
express queries in **logic**

Queries

Find ids of customers who buy all books:

$$\pi_{cust\_id}(Pay) - \pi_{cust\_id}\left((\pi_{cust\_id}(Pay) \times \pi_{title}(Order)) - \pi_{cust\_id,title}\left(\sigma_{order\_id=order}(Order \times Pay)\right)\right)$$

That's not pretty. But here is a better idea (1971): express queries in **logic**

$$\{c \mid \forall(o,t,p) \in Order \;\; \exists\, (o',t,p') \in Order: (c,o') \in Pay\}$$

Queries

Find ids of customers who buy all books:

$$\pi_{cust\_id}(Pay) - \pi_{cust\_id}\big((\pi_{cust\_id}(Pay) \times \pi_{title}(Order)) - \pi_{cust\_id,title}(\sigma_{order\_id=order}(Order \times Pay))\big)$$

That's not pretty. But here is a better idea (1971): express queries in **logic**

$$\{c \mid \forall (o,t,p) \in Order \;\; \exists (o',t,p') \in Order: (c,o') \in Pay\}$$

This is *first-order logic* (FO).
Codd 1971: ***RA = FO***.

History continued

Of course programmers don't write logical sentences, they need a programming syntax. Enters **SQL**:

```
SELECT P.cust_id FROM P
WHERE NOT EXISTS
    (SELECT * FROM Order O
     WHERE NOT EXISTS
        (SELECT * FROM Order O1
         WHERE O1.title=O.title AND O1.order_id=P.order))
```

History continued

Of course programmers don't write logical sentences, they need a programming syntax. Enters **SQL**:

```
SELECT P.cust_id FROM P
WHERE NOT EXISTS
    (SELECT * FROM Order O
     WHERE NOT EXISTS
        (SELECT * FROM Order O1
         WHERE O1.title=O.title AND O1.order_id=P.order))
```

$$\forall x F(x) = \neg \exists x \neg F(x)$$

History continued

Of course programmers don't write logical sentences, they need a programming syntax. Enters **SQL**:

```
SELECT P.cust_id FROM P
WHERE NOT EXISTS
    (SELECT * FROM Order O
     WHERE NOT EXISTS
        (SELECT * FROM Order O1
         WHERE O1.title=O.title AND O1.order_id=P.order))
```

$\forall x F(x) = \neg \exists x \neg F(x)$

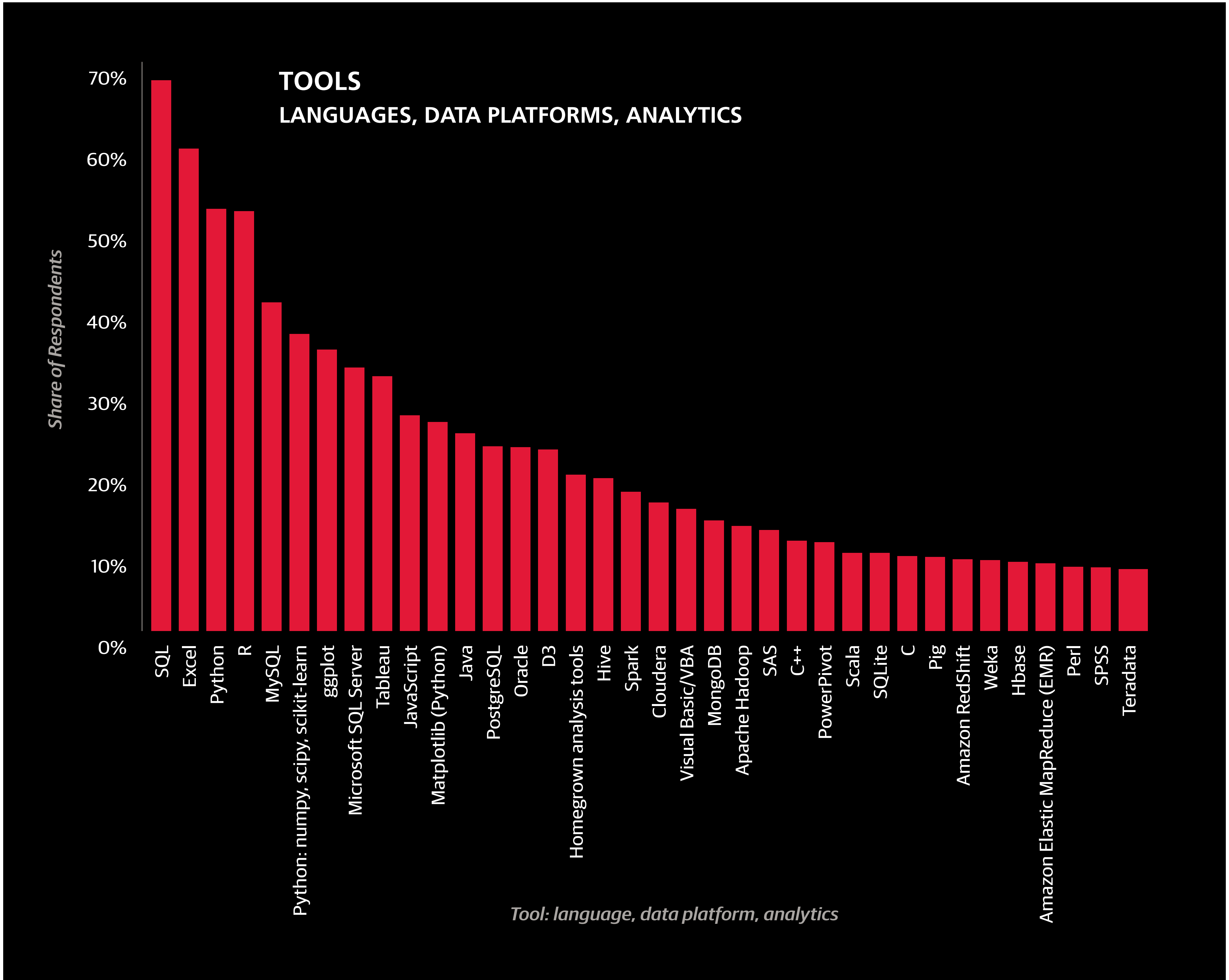
Idea:

- Take FO and turn into into programming syntax.
- Then use RA to implement queries.

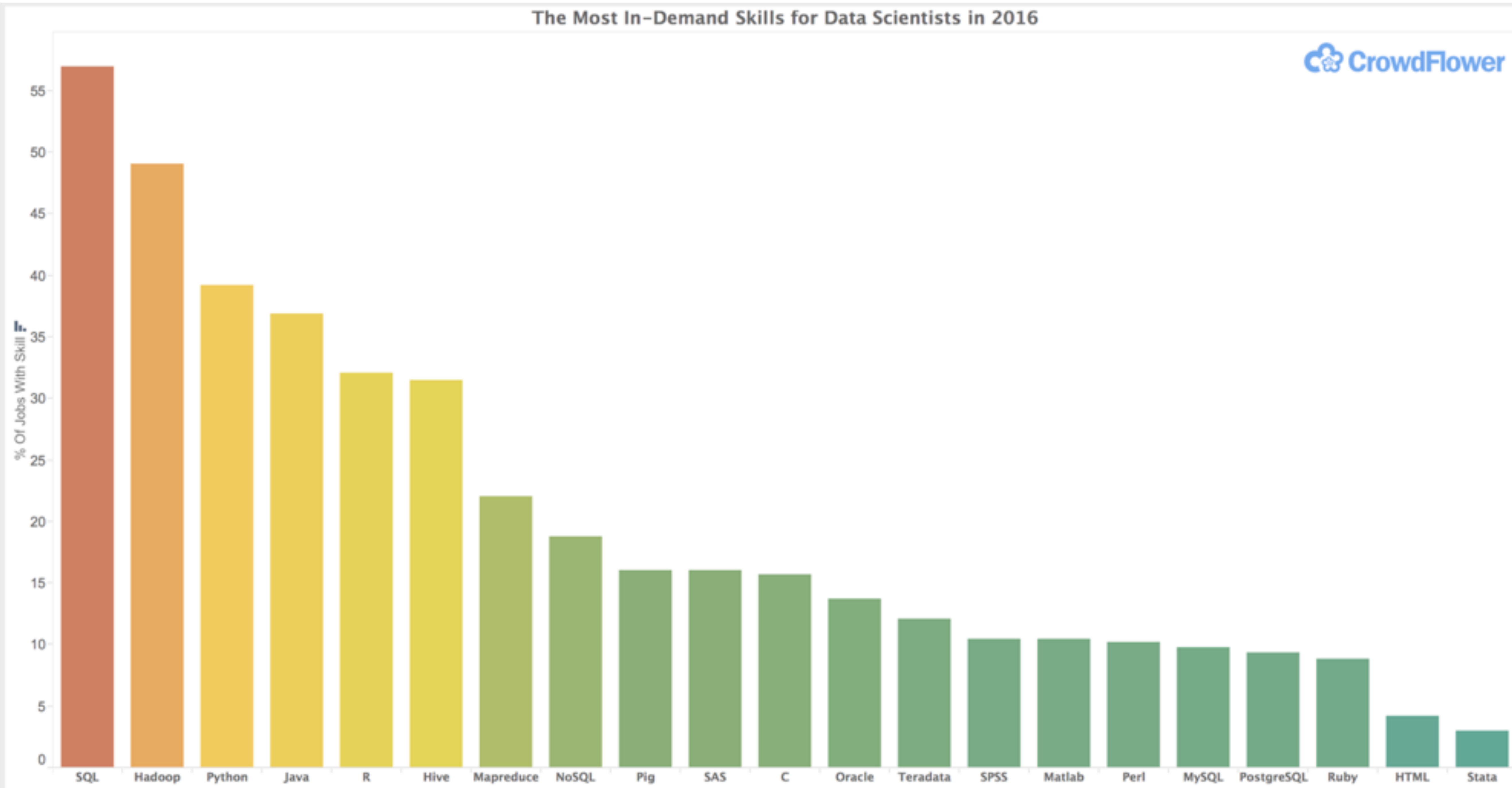
SQL development

- SQL has since become the dominant language for relational databases
- Standards: SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016
- The latest standard is in 9 parts, will make you $1000 poorer if you buy them all.
- But the core remains the same, essentially FO.
- And it is **the main big data tool**!

Data scientists' favorite tools



Future data scientists' favorite tools



But do we understand it?

- Even the basic fragment, that stays the same in all the Standards:
    - does it have the power of RA? Does it have the power of FO?
    - Is there a formal semantics of it?
- Let’s do a little quiz and see how well we know the basics.

TASK: Relations R(A), S(A)

Compute R - S.

TASK: Relations R(A), S(A)

Compute R - S.

Every student will write:

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

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

Every student will write:

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

And they are taught it is equivalent to :

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

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

Every student will write:

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

And they are taught it is equivalent to :

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

and that they can do it directly in SQL:

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

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

R

A
1
null

S

A
null

Every student will write:

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

And they are taught it is equivalent to :

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

and that they can do it directly in SQL:

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

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

R

A
1
null

S

A
null

Outputs:

Every student will write:

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

And they are taught it is equivalent to :

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

and that they can do it directly in SQL:

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

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

R

A
1
null

S

A
null

Outputs:

A

Every student will write:

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

And they are taught it is equivalent to :

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

and that they can do it directly in SQL:

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

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

R

A
1
null

S

A
null

Outputs:

Every student will write:

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

A

And they are taught it is equivalent to :

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

A
1
null

and that they can do it directly in SQL:

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

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

R

A
1
null

S

A
null

Outputs:

Every student will write:

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

A

And they are taught it is equivalent to :

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

A
1
null

and that they can do it directly in SQL:

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

A
1

An exam question that nicely brings down the average grade

What is the output of these queries?

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

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

An exam question that nicely brings down the average grade

What is the output of these queries?

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

1

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

∅

SQL vs Relational Algebra: attributes may repeat

Q = SELECT R.A, R.A FROM R on

A
1
null

gives

A	A
1	1
null	null

SQL vs Relational Algebra: attributes may repeat

Q = SELECT R.A, R.A FROM R on

A
1
null

gives

A	A
1	1
null	null

Let's use it as a subquery:

Q' = SELECT * FROM (Q) AS T

SQL vs Relational Algebra: attributes may repeat

Q = `SELECT R.A, R.A FROM R` on

A
1
null

gives

A	A
1	1
null	null

Let's use it as a subquery:

Q' = `SELECT * FROM (Q) AS T`

<u>Output:</u>

- Postgres: as above
- Oracle, MS SQL Server: compile-time error

SQL vs Relational Algebra: attributes may repeat

Q = SELECT R.A, R.A FROM R on

A
1
null

gives

A	A
1	1
null	null

Let's use it as a subquery:

Q' = SELECT * FROM (Q) AS T

<u>Output:</u>

- Postgres: as above
- Oracle, MS SQL Server: compile-time error

SELECT R.A FROM R WHERE EXISTS (Q')

SQL vs Relational Algebra: attributes may repeat

Q = SELECT R.A, R.A FROM R on

A
1
null

gives

A	A
1	1
null	null

Let's use it as a subquery:

Q' = SELECT * FROM (Q) AS T

<u>Output:</u>

- Postgres: as above
- Oracle, MS SQL Server: compile-time error

SELECT R.A FROM R WHERE EXISTS (Q')

Answer:

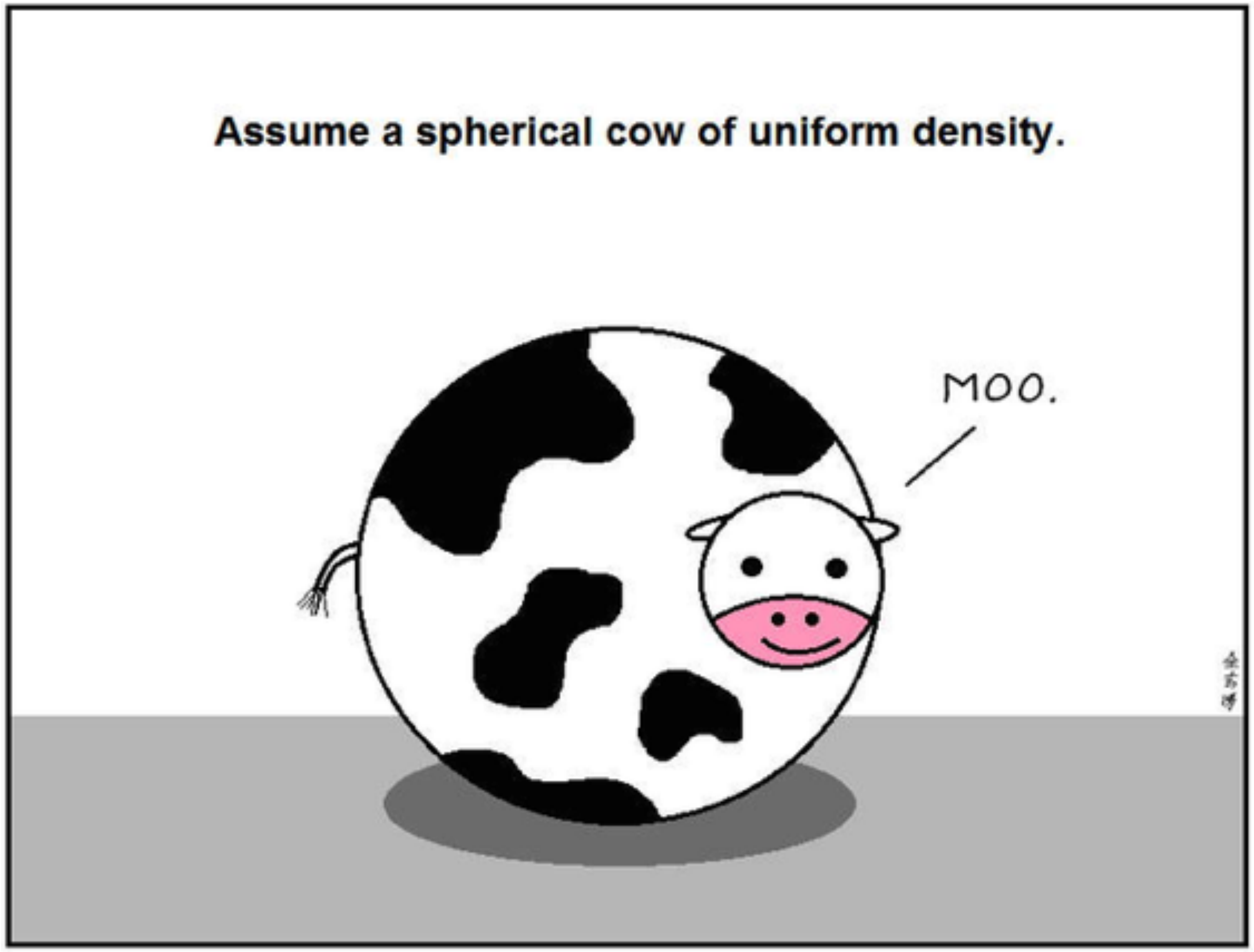
A
1
null

Why do we find these questions difficult?

- Reason 1: there is no formal semantics of SQL.
  - The Standard is rather vague, not written formally, and different vendors interpret it differently.
- Reason 2: theory works with a simplified model, no nulls, no duplicates.
  - Under these assumptions several semantics exist (1985 - 2017) but they do not model the real language.

It is much harder to deal with the real thing than with theoretical abstractions

It is much harder to deal with the real thing than with theoretical abstractions



Another example: Query equivalences

Q1(x) :- T(x,y)
Q2(x) :- T(x,y), T(u,v)

Another example: Query equivalences

Q1(x) :- T(x,y)
Q2(x) :- T(x,y), T(u,v)

In theory: equivalent; on

A	B
1	2
3	4

return

A
1
3

Another example: Query equivalences

Q1(x) :- T(x,y)
Q2(x) :- T(x,y), T(u,v)

In theory: equivalent; on

A	B
1	2
3	4

return

A
1
3

Now the same in SQL:

Another example: Query equivalences

Q1(x) :- T(x,y)
Q2(x) :- T(x,y), T(u,v)

In theory: equivalent; on

A	B
1	2
3	4

return

A
1
3

Now the same in SQL:

Q1 = `SELECT R.A FROM R` returns

A
1
3

Another example: Query equivalences

Q1(x) :- T(x,y)
Q2(x) :- T(x,y), T(u,v)

In theory: equivalent; on

A	B
1	2
3	4

return

A
1
3

Now the same in SQL:

Q1 = `SELECT R.A FROM R` returns

A
1
3

Q2 = `SELECT R1.A FROM R R1, R R2` returns

A
1
1
3
3

The infamous NULL

- Comparisons with nulls, like 2 = NULL, result in truth value *unknown*
- It then propagates: *true* ∧ *unknown* = *unknown*, *true* ∨ *unknown* = *true*
    - rules of propositional 3-valued logic of Kleene
- When condition is evaluated, only tuples for which it is *true* are returned
    - *false* and *unknown* are treated the same
- It's a weird logic and it is **not** the 3-valued predicate calculus!

The bottom line

- Many spherical cows out there but no real one.
- There are lots and lots of issues to address to give proper semantics of SQL
- None of the simplified semantics came even close.
- We do it for the basic fragment of SQL:
  - `SELECT-FROM-WHERE` without aggregation
  - but with pretty much everything else

Syntax

$$\tau : \beta \;\coloneqq\; T_1 \;\texttt{AS}\; N_1,\; \ldots,\; T_k \;\texttt{AS}\; N_k \quad \text{for } \tau = (T_1, \ldots, T_k),\, \beta = (N_1, \ldots, N_k), \quad k > 0$$

$$\alpha : \beta' \;\coloneqq\; t_1 \;\texttt{AS}\; N_1',\; \ldots,\; t_m \;\texttt{AS}\; N_k' \quad \text{for } \alpha = (t_1, \ldots, t_m),\; \beta' = (N_1', \ldots, N_m'),\, m > 0$$

QUERIES:

$$\begin{aligned} Q \;\coloneqq\;& \texttt{SELECT} \,[\texttt{DISTINCT}]\, \alpha : \beta' \;\texttt{FROM}\; \tau : \beta \;\texttt{WHERE}\; \theta \\ \mid\;& \texttt{SELECT} \,[\texttt{DISTINCT}]\, * \;\texttt{FROM}\; \tau : \beta \;\texttt{WHERE}\; \theta \\ \mid\;& Q \;(\texttt{UNION} \mid \texttt{INTERSECT} \mid \texttt{EXCEPT})\; [\texttt{ALL}]\; Q \end{aligned}$$

CONDITIONS:

$$\begin{aligned} \theta \;\coloneqq\;& \texttt{TRUE} \mid \texttt{FALSE} \mid P(t_1, \ldots, t_k),\; P \in \mathcal{P} \\ \mid\;& t \;\texttt{IS}\; [\texttt{NOT}]\; \texttt{NULL} \\ \mid\;& \bar{t}\; [\texttt{NOT}]\; \texttt{IN}\; Q \mid \texttt{EXISTS}\; Q \\ \mid\;& \theta \;\texttt{AND}\; \theta \mid \theta \;\texttt{OR}\; \theta \mid \texttt{NOT}\; \theta \end{aligned}$$

Names: either simple (R, A) or composite (R.A)

Terms t: constants, nulls, or composite names

Predicates: anything you want on constants

Semantics: labels

$$\ell(R) = \text{tuple of names provided by the schema}$$

$$\ell(\tau) = \ell(T_1)\cdots\ell(T_k) \quad \text{for } \tau = (T_1,\ldots,T_k)$$

$$\ell\begin{pmatrix}\texttt{SELECT}\ [\texttt{DISTINCT}]\ \alpha : \beta' \\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta\end{pmatrix} = \beta'$$

$$\ell\big(\texttt{SELECT}\ [\texttt{DISTINCT}]\ *\ \texttt{FROM}\ \tau : \beta\ \texttt{WHERE}\ \theta\big) = \ell(\tau)$$

$$\ell\big(Q_1\ (\texttt{UNION} \mid \texttt{INTERSECT} \mid \texttt{EXCEPT})\ [\texttt{ALL}]\ Q_2\big) = \ell(Q_1)$$

Semantics

$$[\![Q]\!]_{D,\eta,x}$$

Q: query

D: database

η: environment (values for composite names)

x: Boolean switch to account for non-compositional nature of SELECT * (to show where we are in the query)

Semantics of terms

$$[\![t]\!]_\eta = \begin{cases} \eta(A) & \text{if } t = A \\ c & \text{if } t = c \in \mathsf{C} \\ \texttt{NULL} & \text{if } t = \texttt{NULL} \end{cases}$$

$$[\![(t_1, \ldots, t_n)]\!]_\eta = \big([\![t_1]\!]_\eta, \ldots, [\![t_n]\!]_\eta\big)$$

Semantics: queries

$$[\![R]\!]_{D,\eta,x} = R^D$$

$$[\![\tau:\beta]\!]_{D,\eta,x} = [\![T_1]\!]_{D,\eta,0} \times \cdots \times [\![T_k]\!]_{D,\eta,0} \quad \text{for } \tau = (T_1,\ldots,T_k)$$

$$\left[\!\!\left[\begin{array}{rl} \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,x} = \left\{ \underbrace{\bar r,\ldots,\bar r}_{k \text{ times}} \;\middle|\; \bar r \in_k [\![\tau:\beta]\!]_{D,\eta,0},\ \boxed{[\![\theta]\!]_{D,\eta'} = \mathbf{t},}\ \eta' = \eta \overset{\bar r}{\oplus} \ell(\tau:\beta) \right\}$$

$$\left[\!\!\left[\begin{array}{rl} \texttt{SELECT} & \alpha:\beta' \\ \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,x} = \left\{ \underbrace{[\![\alpha]\!]_{\eta'},\ldots,[\![\alpha]\!]_{\eta'}}_{k \text{ times}} \;\middle|\; \eta' = \eta \overset{\bar r}{\oplus} \ell(\tau:\beta),\ \ \bar r \in_k \left[\!\!\left[\begin{array}{rl} \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,x} \right\}$$

$$\left[\!\!\left[\begin{array}{rl} \texttt{SELECT} & * \\ \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,0} = \left[\!\!\left[\begin{array}{rl} \texttt{SELECT} & \ell(\tau:\beta):\ell(\tau) \\ \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,0}$$

$$\left[\!\!\left[\begin{array}{rl} \texttt{SELECT} & * \\ \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,1} = \left[\!\!\left[\begin{array}{rl} \texttt{SELECT} & c \ \texttt{AS}\ N \\ \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\!\right]_{D,\eta,1} \quad \text{for arbitrary } c \in \mathsf{C} \text{ and } N \in \mathsf{N}$$

$$\left[\!\!\left[\begin{array}{l} \texttt{SELECT DISTINCT}\ \alpha:\beta' \mid * \\ \texttt{FROM}\ \tau:\beta\ \texttt{WHERE}\ \theta \end{array} \right]\!\!\right]_{D,\eta,x} = \varepsilon\left(\left[\!\!\left[\begin{array}{l} \texttt{SELECT}\ \alpha:\beta' \mid * \\ \texttt{FROM}\ \tau:\beta\ \texttt{WHERE}\ \theta \end{array} \right]\!\!\right]_{D,\eta,x} \right)$$

Semantics: conditions

$$[\![P(t_1,\ldots,t_k)]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } P\big([\![t_1]\!]_\eta,\ldots,[\![t_k]\!]_\eta\big) \text{ holds and } [\![t_i]\!]_\eta \neq \texttt{NULL} \text{ for all } i \in \{1,\ldots,k\} \\ \mathbf{f} & \text{if } P\big([\![t_1]\!]_\eta,\ldots,[\![t_k]\!]_\eta\big) \text{ does not hold and } [\![t_i]\!]_\eta \neq \texttt{NULL} \text{ for all } i \in \{1,\ldots,k\} \\ \mathbf{u} & \text{if } [\![t_i]\!]_\eta = \texttt{NULL} \text{ for some } i \in \{1,\ldots,k\} \end{cases}$$

$$[\![t \texttt{ IS NULL}]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } [\![t]\!]_\eta = \texttt{NULL} \\ \mathbf{f} & \text{if } [\![t]\!]_\eta \neq \texttt{NULL} \end{cases}$$

$$[\![t \texttt{ IS NOT NULL}]\!]_{D,\eta} = \neg[\![t \texttt{ IS NULL}]\!]_{D,\eta}$$

$$[\![(t_1,\ldots t_n) = (t'_1,\ldots,t'_n)]\!]_{D,\eta} = \bigwedge_{i=1}^{n} [\![t_i = t'_i]\!]_{D,\eta} \qquad [\![(t_1,\ldots t_n) \neq (t'_1,\ldots,t'_n)]\!]_{D,\eta} = \bigvee_{i=1}^{n} [\![t_i \neq t'_i]\!]_{D,\eta}$$

$$[\![\bar{t} \texttt{ IN } Q]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } \exists \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{t} \\ \mathbf{f} & \text{if } \forall \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{f} \\ \mathbf{u} & \text{if } \nexists \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} = \mathbf{t} \text{ and } \exists \bar{r} \in [\![Q]\!]_{D,\eta,0} \text{ s.t. } [\![\bar{t} = \bar{r}]\!]_{D,\eta} \neq \mathbf{f} \end{cases}$$

$$[\![\bar{t} \texttt{ NOT IN } Q]\!]_{D,\eta} = \neg[\![\bar{t} \texttt{ IN } Q]\!]_{D,\eta}$$

$$[\![\texttt{EXISTS } Q]\!]_{D,\eta} = \begin{cases} \mathbf{t} & \text{if } [\![Q]\!]_{D,\eta,1} \neq \varnothing \\ \mathbf{f} & \text{if } [\![Q]\!]_{D,\eta,1} = \varnothing \end{cases}$$

$$[\![\texttt{TRUE}]\!]_{D,\eta} = \mathbf{t} \qquad [\![\theta_1 \texttt{ AND } \theta_2]\!]_{D,\eta} = [\![\theta_1]\!]_{D,\eta} \wedge [\![\theta_2]\!]_{D,\eta} \qquad [\![\texttt{NOT } \theta]\!]_{D,\eta} = \neg[\![\theta]\!]_{D,\eta}$$

$$[\![\texttt{FALSE}]\!]_{D,\eta} = \mathbf{f} \qquad [\![\theta_1 \texttt{ OR } \theta_2]\!]_{D,\eta} = [\![\theta_1]\!]_{D,\eta} \vee [\![\theta_2]\!]_{D,\eta}$$

TRUTH TABLES:

$\wedge$	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

	$\neg$
t	f
f	t
u	u

Semantics: operations

$$
\begin{aligned}
[\![Q_1 \ \textbf{UNION ALL}\ Q_2]\!]_{D,\eta,x} &= [\![Q_1]\!]_{D,\eta,0} \cup [\![Q_2]\!]_{D,\eta,0} \\
[\![Q_1 \ \textbf{INTERSECT ALL}\ Q_2]\!]_{D,\eta,x} &= [\![Q_1]\!]_{D,\eta,0} \cap [\![Q_2]\!]_{D,\eta,0} \\
[\![Q_1 \ \textbf{EXCEPT ALL}\ Q_2]\!]_{D,\eta,x} &= [\![Q_1]\!]_{D,\eta,0} - [\![Q_2]\!]_{D,\eta,0} \\
[\![Q_1 \ \textbf{UNION}\ Q_2]\!]_{D,\eta,x} &= \varepsilon\big([\![Q_1 \ \textbf{UNION ALL}\ Q_2]\!]_{D,\eta,x}\big) \\
[\![Q_1 \ \textbf{INTERSECT}\ Q_2]\!]_{D,\eta,x} &= \varepsilon\big([\![Q_1 \ \textbf{INTERSECT ALL}\ Q_2]\!]_{D,\eta,x}\big) \\
[\![Q_1 \ \textbf{EXCEPT}\ Q_2]\!]_{D,\eta,x} &= \varepsilon\big([\![Q_1]\!]_{D,\eta,0}\big) - [\![Q_2]\!]_{D,\eta,0}
\end{aligned}
$$

Bag interpretation of operations; ϵ is duplicate elimination

Looks simple, no?

- It does not. Such basic things as variable binding changed several times till we got them right.

- The meaning of the new environment:

$$\left[\!\left[\begin{array}{ll} \texttt{FROM} & \tau:\beta \\ \texttt{WHERE} & \theta \end{array} \right]\!\right]_{D,\eta,x} = \left\{ \underbrace{\bar{r},\ldots,\bar{r}}_{k \text{ times}} \;\middle|\; \bar{r} \in_k [\![\tau:\beta]\!]_{D,\eta,0},\;\; [\![\theta]\!]_{D,\eta'} = \mathbf{t},\;\; \boxed{\eta' = \eta \overset{\bar{r}}{\oplus} \ell(\tau:\beta)} \right\}$$

  - in η, unbind every name that occurs among labels of the FROM clause

  - then bind non-repeated names among those to values taken from record r

How do we know we got it right?

- Since the Standard is rather vague, there is only one way — **experiments.**

- But what kind of benchmark can we use?

- For performance studies there are standard benchmarks like TPC-H. But they won't work for us: not enough queries.

Experimental Validation

- Benchmarks have rather few queries (22 in TPC-H). Validating on 22 queries is not a good evidence.
- But we can look at benchmarks, and then generate lots of queries that look the same.
- In TPC-H:
  - 8 tables,
  - maximum nesting depth = 3,
  - average number of tables per query = 3.2,
  - at most 8 conditions in WHERE (except two queries)

Validation: results

- Small adjustments of the Standard semantics (for Postgres and Oracle)
- Random query generator
- Naive implementation of the semantics
- Finally: experiments on 100,000 random queries

Validation: results

- Small adjustments of the Standard semantics (for Postgres and Oracle)
- Random query generator
- Naive implementation of the semantics
- Finally: experiments on 100,000 random queries
- **Yes, we got it right!**

What can we do with this?

- Equivalence of basic SQL and Relational Algebra: formally proved for the first time.
- 3-valued logic of SQL vs the usual Boolean logic: is there any difference?

Basic SQL = Relational Algebra

- We formally prove **SQL = Relational Algebra (RA)**
    - with nulls, subqueries, bags, all there is. And RA has to be defined properly too, to use bags and SQL's 3-valued logic.
    - a small caveat: in RA, attributes cannot repeat. So the equality is wrt queries that do not return repeated attributes.

3-valued logic of nulls

- From the early SQL days and database textbooks: **if you have nulls, you need 3-valued logic.**

- But 3-valued logic is not the first thing you think of as a logician.

- And it makes sense to think as a logician: after all, the core of SQL is claimed to be first-order logic in a different syntax.

What would a logician do?

What would a logician do?

- First Order Logic (FO)
  - domain has usual values and NULL
  - Syntactic equality: NULL = NULL but NULL ≠ 5 etc
  - Boolean logic rules for $\wedge$, $\vee$, $\neg$
  - Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction

What did SQL do?

What did SQL do?

- 3-valued FO (a textbook version)
  - domain has usual values and NULL
  - comparisons with NULL result in *unknown*
  - Kleene logic rules for $\wedge$, $\vee$, $\neg$
  - Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction

What did SQL do?

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

What did SQL do?

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

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

- Conditions in WHERE are evaluated under 3-valued logic. But then only those evaluated to true matter.
- Studied before only at the level of propositional logic.
- In 1939, Russian logician Bochvar wanted to give a formal treatment of logical paradoxes. He needed to assert that something is true, and introduced a new connective: $\uparrow$**p** means that p is true.
- Amazingly, 40 years later SQL adopted the same idea.

What did SQL really do?

- 3-valued FO with ↑:
  - domain has usual values and NULL
  - comparisons with NULL result in *unknown*
  - Kleene logic rules for $\wedge$, $\vee$, $\neg$
  - Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction
  - Add ↑ with the semantics

$$\uparrow\varphi = \begin{cases} \textit{true}, & \text{if } \varphi \text{ is } \textit{true} \\ \textit{false}, & \text{if } \varphi \text{ is } \textit{false} \text{ or } \textit{unknown} \end{cases}$$

What IS the logic of SQL?

What IS the logic of SQL?

- We have:
  - logician's 2-valued FO
  - 3-valued FO (Kleene logic)
  - 3-valued FO + Bochvar's assertion (SQL logic)

What IS the logic of SQL?

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

THEOREM: ↑can be expressed in 3-valued FO.

3-valued FO = 3-valued FO with ↑

THEOREM: For every formula φ of 3-valued FO, there is a formula ψ of the usual 2-valued FO such that

φ is *true* ⇔ ψ is *true*

THEOREM: $\uparrow$ can be expressed in 3-valued FO.

3-valued FO = 3-valued FO with $\uparrow$

THEOREM: For every formula φ of 3-valued FO, there is a formula ψ of the usual 2-valued FO such that

φ is *true* $\Leftrightarrow$ ψ is *true*

Translations work at the level of SQL too!

2-valued SQL

Idea — 3 simultaneous translations:

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

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

In Q' we simply replace P by P^t

2-valued SQL: translation

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

Note: a lot of disjunctions with IS NULL conditions

Shall we switch to 2-valued SQL?

- Not so fast perhaps. Two reasons:
  - all the legacy code that uses 3-values
  - using 2 truth values introduces many new disjunctions. And DBMSs don't like disjunctions!

Shall we switch to 2-valued SQL?

- Not so fast perhaps. Two reasons:
  - all the legacy code that uses 3-values
  - using 2 truth values introduces many new disjunctions. And DBMSs don't like disjunctions!
- As to why, this comment line in Postgres optimizer code sheds some light:

Shall we switch to 2-valued SQL?

- Not so fast perhaps. Two reasons:
  - all the legacy code that uses 3-values
  - using 2 truth values introduces many new disjunctions. And DBMSs don't like disjunctions!
- As to why, this comment line in Postgres optimizer code sheds some light:
  - */* we stop as soon as we hit a non-AND item */*

Questions?