Can We Trust SQL as a Data Analytics Tool?

Leonid Libkin
University of Edinburgh

Joint work with Paolo Guagliardo, also from Edinburgh
SQL

- **The** query language for relational databases
- **International Standard** since 1987
- Implemented in **all systems** (free and commercial)
- **$30B/year** business
- Most common tool used by data scientists
Two exceptions were "Natural Language/Text Processing" and "Networks/Social Graph Processing," which are less tools than they are types of data analysis.

One hundred and fourteen tools were present on the list, but over 200 more were manually entered in the "other" fields.

Figure 1-10. Most commonly used tools (used by at least 10% of sample)
Correlations were tested using a Pearson's chi square test with p=.05. That SQL/RDB is the top bar is no surprise: accessing data is the meat and potatoes of data analysis, and has not been displaced by other tools. The preponderance of R and Python usage is more surprising — operating systems aside, these were the two most commonly used individual tools, even above Excel, which for years has been the go-to option for spreadsheets and surface-level analysis. R and Python are likely popular because they are easily accessible and effective open source tools for analysis. More traditional statistical programs such as SAS and SPSS were far less common than R and Python.

By counting tool usage, we are only scratching the surface: who exactly uses these tools? In comparing usage of R/Python and Excel, we had hypothesized that it would be possible to categorize respondents as users of one or the other: those who use a wider variety of tools, largely open source, including R, Python, and some Hadoop, and those who use Excel but few tools beside it.

Python and R correlate with each other—a respondent who uses one is more likely to use the other—but neither correlates with Excel (negatively or positively): their usage (joint or separate) does not predict whether a respondent would also use Excel. However, if we look at all correlations between all pairs of tools, we can see a pattern that, to an extent, divides respondents. The significant positive correlations can be drawn as edges between tools as nodes, producing a graph with two main clusters.
Future data scientists’ favorite tools
(50 - 80%) wrangling
Main Questions

• Do we understand SQL queries, even simple ones?
• And if we think we do, do query results make sense?
Asking these questions now?

A bit of history: 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.

Query writing made easy: SQL
## Relational Model

### Orders

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

### Pay

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

### Customer

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>
Relational Model

<table>
<thead>
<tr>
<th>ORDERS</th>
<th>TITLES</th>
<th>PRICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Language: Relational Algebra (RA)

- **projection** $\pi$ (find book titles)
- **selection** $\sigma$ (find books that cost at least £40)
- **Cartesian product** $\times$
- **union** $\cup$
- **difference** $-$
Queries

Find ids of customers who buy all books:

\[ \pi_{\text{cust_id}} (\text{Pay}) - \pi_{\text{cust_id}} \left( (\pi_{\text{cust_id}}(\text{Pay}) \times \pi_{\text{title}}(\text{Order})) - \pi_{\text{cust_id},\text{title}} (\sigma_{\text{order_id}=\text{order}} (\text{Order} \times \text{Pay})) \right) \]
Queries

Find ids of customers who buy all books:

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

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

**Find ids of customers who buy all books:**

\[
\pi_{\text{cust_id}}(\text{Pay}) - \\
\pi_{\text{cust_id}}((\pi_{\text{cust_id}}(\text{Pay}) \times \pi_{\text{title}}(\text{Order})) - \\
\pi_{\text{cust_id,title}}(\sigma_{\text{order_id}=\text{order}}(\text{Order} \times \text{Pay})))
\]

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

\[
\{c \mid \forall (o,t,p) \in \text{Order} \ \exists (o',t,p') \in \text{Order}: (c,o') \in \text{Pay}\}
\]
Queries

Find ids of customers who buy all books:

\[ \pi_{\text{cust_id}} \text{ (Pay)} - \]
\[ \pi_{\text{cust_id}} \left( \left( \pi_{\text{cust_id}} \text{ (Pay)} \times \pi_{\text{title}} \text{ (Order)} \right) - \right. \]
\[ \left. \pi_{\text{cust_id}, \text{title}} \left( \sigma_{\text{order_id}=\text{order}} \left( \text{Order} \times \text{Pay} \right) \right) \right) \]

That’s not pretty. But here is a better idea (1971):

express queries in **logic**

\[ \{ c \mid \forall (o,t,p) \in \text{Order} \ \exists (o’,t,p’) \in \text{Order}: (c,o’) \in \text{Pay} \} \]

This is **first-order logic** (FO).

Codd 1971: \( \text{RA} = \text{FO} \).
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))
```
Of course programmers don’t write logical sentences, they need a programming syntax. Enters SQL:

```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))
```

\[ \forall x F(x) = \neg \exists x \neg F(x) \]
Of course programmers don’t write logical sentences, they need a programming syntax. Enters SQL:

\[ \forall x \; F(x) = \neg \exists x \; \neg F(x) \]

- Take FO and turn into programming syntax:
  - Committee design!
- Then use RA to implement queries.
SQL development

  - The latest standard will make you $1000 poorer
- The core remains the same.
- And yet things are not as obvious as they should be.
- Now a few quiz-type slides....
**TASK:** Relations $R(A), S(A)$
Compute $R - S$. 
**TASK:** Relations $R(A), S(A)$
Compute $R - S$.

Every student will write:

```sql
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:

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

And they are taught it is equivalent to:

```sql
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 $R - S$.

Every student will write:

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

And they are taught it is equivalent to:

```sql
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:

```sql
select * from r except select * 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)
``` 

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.

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 $R - S$.

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 $R - S$.

Every student will write:

$$\text{select R.A from R where R.A not in (select S.A from S)}$$

And they are taught it is equivalent to:

$$\text{select R.A from R where not exists (select S.A from S where S.A=R.A)}$$

and that they can do it directly in SQL:

$$\text{select * from r except select * from s}$$
SELECT * should be simple, no?

\[ Q = \text{SELECT } R.A, R.A \text{ FROM } R \text{ on } \begin{array}{c} \text{R} \\ \text{A} \\ \text{1} \end{array} \text{ gives } \begin{array}{cc} \text{A} & \text{A} \\ \text{1} & \text{1} \end{array} \]
SELECT * should be simple, no?

\[ Q = \text{SELECT } R.A, R.A \text{ FROM } R \text{ on } \]

\[ \begin{array}{c}
R \\
A \\
1
\end{array} \]
gives

\[ \begin{array}{cc}
A & A \\
1 & 1
\end{array} \]

Let’s use it as a subquery:

\[ Q' = \text{SELECT } * \text{ FROM } (Q) \text{ AS } T \]
SELECT * should be simple, no?

\[ Q = \text{SELECT } R.A, R.A \text{ FROM } R \text{ on } \]

\[ \begin{array}{c}
R \\
| A | \downarrow A \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
\end{array} \]

gives

\[ \begin{array}{c}
Q' = \text{SELECT } * \text{ FROM } (Q) \text{ AS } T \\
\end{array} \]

Let’s use it as a subquery:

\[ Q' = \text{SELECT } * \text{ FROM } (Q) \text{ AS } T \]

Output:

- **Postgres**: as above
- **Oracle, MS SQL Server**: compile-time error
SELECT * should be simple, no?

Q = SELECT R.A, R.A FROM R on \[ \begin{array}{c}
R \\
A \\
1 \\
\end{array} \]
gives \[ \begin{array}{cc}
A & A \\
1 & 1 \\
\end{array} \]

Let’s use it as a subquery:
Q’ = SELECT * FROM (Q) AS T

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

SELECT R.A FROM R WHERE EXISTS (Q’)

\[ \begin{array}{c}
\]
SELECT * should be simple, no?

\[ Q = \text{SELECT } R.A, R.A \text{ FROM } R \]

gives

Let's use it as a subquery:

\[ Q' = \text{SELECT } * \text{ FROM } (Q) \text{ AS } T \]

Output:

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

\[ \text{SELECT } R.A \text{ FROM } R \text{ WHERE EXISTS } (Q') \]

Answer:

\[ \begin{array}{c}
A \\
1
\end{array} \]
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, no repeated attributes.

- 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**.
From spherical to real cows

- We do it for the basic fragment of SQL:
  - `SELECT-FROM-WHERE` without aggregation
  - but with pretty much everything else

[G.,L. A Formal Semantics of SQL Queries, its Validation and Applications. PVLDB 2017]
Syntax

\[
\tau : \beta := T_1 \text{ AS } N_1, \ldots, T_k \text{ AS } N_k \quad \text{for } \tau = (T_1, \ldots, T_k), \beta = (N_1, \ldots, N_k), \quad k > 0
\]

\[
\alpha : \beta' := t_1 \text{ AS } N'_1, \ldots, t_m \text{ AS } N'_k \quad \text{for } \alpha = (t_1, \ldots, t_m), \beta' = (N'_1, \ldots, N'_m), \quad m > 0
\]

Queries:

\[
Q := \text{SELECT [DISTINCT] } \alpha : \beta' \text{ FROM } \tau : \beta \text{ WHERE } \theta
\]

| SELECT [DISTINCT] * FROM \tau : \beta WHERE \theta |
| Q (UNION | INTERSECT | EXCEPT) [ALL] Q |

Conditions:

\[
\theta := \text{TRUE | FALSE | } P(t_1, \ldots, t_k), \quad P \in \mathcal{P}
\]

| t IS [NOT] NULL |
| \bar{t} [NOT] IN Q | EXISTS Q |
| \theta AND \theta | \theta OR \theta | \text{NOT } \theta |

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

Terms t: constants, nulls, or composite names

Predicates: anything you want on constants
Semantics: labels

\[ \ell(R) = \text{tuple of names provided by the schema} \]
\[ \ell(\tau) = \ell(T_1) \cdots \ell(T_k) \quad \text{for } \tau = (T_1, \ldots, T_k) \]
\[ \ell(\text{SELECT [DISTINCT] } \alpha : \beta' \text{ FROM } \tau : \beta \text{ WHERE } \theta) = \beta' \]
\[ \ell(\text{SELECT [DISTINCT] } * \text{ FROM } \tau : \beta \text{ WHERE } \theta) = \ell(\tau) \]
\[ \ell(Q_1 (\text{UNION | INTERSECT | EXCEPT}) [\text{ALL}] Q_2) = \ell(Q_1) \]
Semantics

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

\(Q\): query

\(D\): database

\(\eta\): 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

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

\[
[(t_1, \ldots, t_n)]_\eta = ([t_1]_\eta, \ldots, [t_n]_\eta)
\]
Semantics: queries

\[
\begin{align*}
[R]_{D,\eta,x} &= R^D \\
[\tau:\beta]_{D,\eta,x} &= [T_1]_{D,\eta,0} \times \cdots \times [T_k]_{D,\eta,0} \quad \text{for } \tau = (T_1, \ldots, T_k) \\
\left[ \begin{array}{c} \text{FROM} \\ \text{WHERE} \end{array} \right] \tau:\beta_{D,\eta,x} &= \left\{ \left[ \begin{array}{c} \bar{r}, \ldots, \bar{r} \\ \text{\textcircled{k times}} \end{array} \right] \mid \bar{r} \in_k [\tau:\beta]_{D,\eta,0}, \left[\theta\right]_{D,\eta^\prime} = t, \eta^\prime = \eta \oplus \ell(\tau:\beta) \right\} \\
\left[ \begin{array}{c} \text{SELECT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] \alpha:\beta'_{D,\eta,x} &= \left\{ \left[ \begin{array}{c} [\alpha]_{\eta'}, \ldots, [\alpha]_{\eta'} \\ \text{\textcircled{k times}} \end{array} \right] \mid \eta' = \eta \oplus \ell(\tau:\beta), \bar{r} \in_k \left[ \begin{array}{c} \text{FROM} \\ \text{WHERE} \end{array} \right] \tau:\beta_{D,\eta,x} \right\} \\
\left[ \begin{array}{c} \text{SELECT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] \tau:\beta_{D,\eta,0} &= \left[ \begin{array}{c} \text{SELECT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] \ell(\tau:\beta):\ell(\tau)_{D,\eta,0} \\
\left[ \begin{array}{c} \text{SELECT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] \tau:\beta_{D,\eta,1} &= \left[ \begin{array}{c} \text{SELECT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] c \text{ AS } N_{D,\eta,1} \quad \text{for arbitrary } c \in \mathbb{C} \text{ and } N \in \mathbb{N} \\
\left[ \begin{array}{c} \text{SELECT DISTINCT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] \alpha:\beta'_{D,\eta,x} &= \varepsilon \left( \left[ \begin{array}{c} \text{SELECT} \\ \text{FROM} \\ \text{WHERE} \end{array} \right] \alpha:\beta'_{D,\eta,x} \right)
\end{align*}
\]
Semantics: conditions

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

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

\[ [t \text{ IS NOT NULL}]_{D, \eta} = \neg [t \text{ IS NULL}]_{D, \eta} \]

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

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

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

\[ \exists Q_{D, \eta} = \begin{cases} 
  t & \text{if } [Q]_{D, \eta, 1} \neq \emptyset \\
  f & \text{if } [Q]_{D, \eta, 1} = \emptyset 
\end{cases} \]

\[ [\text{TRUE}]_{D, \eta} = t \]

\[ [\theta_1 \text{ AND } \theta_2]_{D, \eta} = [\theta_1]_{D, \eta} \land [\theta_2]_{D, \eta} \]

\[ [\text{FALSE}]_{D, \eta} = f \]

\[ [\theta_1 \text{ OR } \theta_2]_{D, \eta} = [\theta_1]_{D, \eta} \lor [\theta_2]_{D, \eta} \]

\[ [\text{NOT } \theta]_{D, \eta} = \neg [\theta]_{D, \eta} \]

Truth Tables:

\[
\begin{array}{c|ccc}
\wedge & t & f & u \\
\hline
  t & t & f & u \\
  f & f & f & f \\
u & u & f & u \\
\end{array}
\]

\[
\begin{array}{c|ccc}
\lor & t & f & u \\
\hline
  t & t & t & t \\
  f & t & f & u \\
u & u & t & u \\
\end{array}
\]

\[
\begin{array}{c|ccc}
\neg & t & f \\
\hline
  t & f \\
  f & t \\
u & u \\
\end{array}
\]

Figure 4: Semantics of basic SQL: Queries.

Figure 5: Semantics of basic SQL: Conditions.
Semantics: operations

\[
\begin{align*}
[Q_1 \ \text{UNION ALL} \ Q_2]_{D,\eta,x} &= [Q_1]_{D,\eta,0} \cup [Q_2]_{D,\eta,0} \\
[Q_1 \ \text{INTERSECT ALL} \ Q_2]_{D,\eta,x} &= [Q_1]_{D,\eta,0} \cap [Q_2]_{D,\eta,0} \\
[Q_1 \ \text{EXCEPT ALL} \ Q_2]_{D,\eta,x} &= [Q_1]_{D,\eta,0} - [Q_2]_{D,\eta,0} \\
[Q_1 \ \text{UNION} \ Q_2]_{D,\eta,x} &= \varepsilon([Q_1 \ \text{UNION ALL} \ Q_2]_{D,\eta,x}) \\
[Q_1 \ \text{INTERSECT} \ Q_2]_{D,\eta,x} &= \varepsilon([Q_1 \ \text{INTERSECT ALL} \ Q_2]_{D,\eta,x}) \\
[Q_1 \ \text{EXCEPT} \ Q_2]_{D,\eta,x} &= \varepsilon([Q_1]_{D,\eta,0}) - [Q_2]_{D,\eta,0}
\end{align*}
\]

Bag interpretation of operations; \(\varepsilon\) 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:

\[
\left[ \begin{array}{l}
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D, \eta, x} \begin{array}{l}
\tau : \beta \\
\theta
\end{array} = \left\{ \begin{array}{l}
\bar{\tau}, \ldots, \bar{\tau} \\
\text{unbind every name that occurs among labels of the FROM clause}
\end{array} \right| \bar{\tau} \in_k [\tau : \beta]_{D, \eta, 0}, \ [\theta]_{D, \eta'} = t, \ \eta' = \eta \oplus \ell(\tau : \beta)
\end{array}
\]

- in \( \eta \), 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
  - Previous attempts (Ceri and Gottlob, Van den Bussche and Vansummeren restricted the language severely: no nulls, for example).

- 3-valued logic of SQL vs the usual Boolean logic: 3-valued logic **does not add expressiveness**.
  - Although it does not mean we should get rid of it now…
Does it matter which DBMS we use?

• We already saw it does. In fact in our experiments we adjusted things a bit for Postgres and Oracle.

• But how much of a difference does it make?

• We have a random query generator, so let’s experiment:
  • generate lots of queries (over 150K)
  • send to standard DBMSs (Oracle, MySQL, MS SQL Server, PostgreSQL, IBM DB2)
  • and see what happens…
Discrepancies between RDBMSs

- About 2% of queries do not behave the same way on different DBMSs
  - and they come from the most basic fragment
- Lots of issues are minor and syntactic
  - different syntax for set operations (eg `EXCEPT` vs `MINUS`) or functions (eg `%` vs `MOD`, or `substring` vs `substr`)
- But some are serious - and surprise even people with good SQL knowledge. Four of the most surprising examples to follow…
Is empty string equal to itself?

```
SELECT *
FROM R
WHERE ''=''
```
Is empty string equal to itself?

```
SELECT *
FROM   R
WHERE  ''=''
```

• Usually it is, but not in Oracle: the above query always returns the empty table.

• Because Oracle implements NULL as ‘’

• Madness? Yes. With a string operation that produces ‘’ you deal with 3-valued logic before you realize it!
Can you divide by zero?

```
SELECT R.A/S.B
FROM R, S

R={1}, S={0}
```
Can you divide by zero?

- Usually not except in **MySQL 5.6**
- It returns NULL
- OK, they realized it in MySQL 5.7 and now by default it’s a warning. But one can go back to the 5.6 mode if one wishes…

```
SELECT R.A/S.B
FROM    R, S

R={1},   S={0}
```
Is equality transitive?

\[ x = y \text{ and } y = z \text{ imply } x = z, \text{ right?} \]
Is equality transitive?

\[ x = y \text{ and } y = z \text{ imply } x = z, \text{ right?} \]

• Usually yes, but not in MySQL

• \( x = '1a', y = 1, z = '1b' \)

• Why is this a problem? SQL books teach programmers to overspecify join conditions: to \( R.A = S.A \text{ AND } S.A = T.A \) add explicitly \( R.A = T.A \)

• But now it can turn a true condition into false!
Can you compare tuples in IN subqueries?

```sql
SELECT * 
FROM R 
WHERE (R.A, R.B) IN SELECT (S.A, S.B FROM S)
```
Can you compare tuples in IN subqueries?

```sql
SELECT *  
FROM R  
WHERE (R.A, R.B) IN SELECT (S.A, S.B FROM S)
```

• Usually yes, except in MS SQL Server.
• Why? No clue…

• Also SQL Server has UNION but no UNION ALL.
• Please explain this.
A simple tool

• We actually have a tool that lets you:
  • specify parameters of a query workload
  • generate lots of random queries, and
  • run against DBMSs you want to compare

• Have fun with results… at least you know what to expect.
We have the semantics.

We understand differences between RDBMS.

Does everything make sense now?

• Not at all, the fun is just starting.
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:

**Unpaid orders:**

```sql
select O.order_id
from Orders O
where O.order_id not in (select order from Pay P)
```
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:

**Unpaid orders:**

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

**Customers without an order:**

```sql
select C.cust_id
from Customer C
where not exists
  (select * from Orders O, Pay P
   where C.cust_id=P.cust_id
     and P.order=O.order_id)
```
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical 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 P)
```

**Answer:** Ord3.

**Customers without an order:**

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

**Answer:** none.
A company database: orders, customers, payments

Orders

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>&quot;Big Data&quot;</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>&quot;SQL&quot;</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>&quot;Logic&quot;</td>
<td>50</td>
</tr>
</tbody>
</table>

Pay

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

Customer

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical 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 P)
```

**Customers without an order:**

```
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
  where C.cust_id=P.cust_id
  and P.order=O.order_id)
```
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

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

Customers without an order:
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
   where C.cust_id=P.cust_id
   and P.order=O.order_id)
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

**Unpaid orders:**

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

**Customers without an order:**

```sql
select C.cust_id from Customer C
where not exists 
  (select * from Orders O, Pay P 
   where C.cust_id=P.cust_id 
   and P.order=O.order_id)
```
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

**Unpaid orders:**

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

**Customers without an order:**

```sql
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)
```
### Orders, Customers, and Payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

### A company database: orders, customers, payments

### In the real world, information is often missing

### Unpaid orders:

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

### Customers without an order:

```sql
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
  where C.cust_id=P.cust_id
  and P.order=O.order_id)
```
### A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

#### Unpaid orders:

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

**Old Answer:** Ord3  **New:** NONE!

#### Customers without an order:

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

**Old answer:** none  **New:** c2!
Types of errors

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

- Nothing prevents us from finding an efficient query evaluation that *avoids false positives*
- Surprisingly not known until very recently
  - L., “Certain answers and SQL’s 3-valued logic”, ACM TODS 2016
  - theoretical complexity: *excellent*.
  - practical complexity: *poor*
- Hence a new evaluation scheme invented
The \(+/\ ?\) approximation scheme

\[ Q \mapsto (Q^+, Q?) \]
The \( + / ? \) approximation scheme

\[
R^+ = R
\]
\[
(\sigma_\theta(Q))^+ = \sigma_{\theta^*}(Q^+)
\]
\[
(\pi_\alpha(Q))^+ = \pi_\alpha(Q^+)
\]
\[
(Q_1 \times Q_2)^+ = Q_1^+ \times Q_2^+
\]
\[
(Q_1 \cup Q_2)^+ = Q_1^+ \cup Q_2^+
\]
\[
(Q_1 \cap Q_2)^+ = Q_1^+ \cap Q_2^+
\]
\[
(Q_1 - Q_2)^+ = Q_1^+ \setminus Q_2^+
\]

\[
R^? = R
\]
\[
(\sigma_\theta(Q))^? = \sigma_{\neg \theta^*}(Q^?)
\]
\[
(\pi_\alpha(Q))^? = \pi_\alpha(Q^?)
\]
\[
(Q_1 \times Q_2)^? = Q_1^? \times Q_2^?
\]
\[
(Q_1 \cup Q_2)^? = Q_1^? \cup Q_2^?
\]
\[
(Q_1 \cap Q_2)^? = Q_1^? \setminus Q_2^?
\]
\[
(Q_1 - Q_2)^? = Q_1^? - Q_2^+
\]

Idea:
- modify selection conditions \( \theta \rightarrow \theta^* \)
- use (anti)semijoins for \(-\) and \(\cap\)
Testing the scheme

• Standard benchmark: TPC-H.

• Do real-life queries produce wrong answers?
  • Answer: yes, and lots

• Does the scheme behave well?
  • Answer: mainly yes
    • either very small overhead, or much faster
    • but some queries experience slowdown
False positives are a real problem

lower bound on the percentage of wrong answers

probability that a null occurs in an attribute not declared as \textbf{NOT NULL}
Experimental evaluation

Measure relative runtime performance of $Q^+$ w.r.t. $Q$

\[
\frac{\text{average total running time of } Q^+}{\text{average total running time of } Q}
\]

Tells us how much faster/slower $Q^+$ is compared to $Q$
The good

< 4% overhead
The fantastic over 1000 times faster.

The original query spends most of its time looking for wrong answers.
The tolerable

Optimizer does not handle well joins with disjunctions

Figure 4: Average relative performance of queries with correctness guarantees.

Table 1: Ranges of average relative performance – $Q_i$ vs $Q_i^+$ for instances up to 10GB.
Joins and disjunctions in translations
Joins and disjunctions in translations

select …
from R1, R2,…
where not exists
  (select *
   from R, S, T, U
   where R.A=S.C
   and other conditions
select ...
from R1, R2, ...
where not exists
  (select *
   from R, S, T, U
   where R.A=S.C
   and other conditions
Joins and disjunctions in translations

select …
from R1, R2,…
where not exists
  (select *
   from R, S, T, U
   where R.A=S.C
   and other conditions

select …
from R1, R2,…
where not exists
  (select *
   from R, S, T, U
   where
     (R.A=S.C or R.A is null)
   and other conditions
Joins and disjunctions in translations

select ...
from R1, R2, ...
where \text{not exists}
  (select *
   from R, S, T, U
   where R.A=S.C
   and other conditions)

select ...
from R1, R2, ....
where \text{not exists}
  (select *
   from R, S, T, U
   where (R.A=S.C \text{ or } R.A \text{ is null})
   and other conditions)

Good plan: hash join etc
Very fast
Joins and disjunctions in translations

```sql
select ...
from R1, R2,...
where not exists
  (select *
   from R, S, T, U
   where R.A=S.C
   and other conditions)

select ...
from R1, R2,....
where not exists
  (select *
   from R, S, T, U
   where
     (R.A=S.C or R.A is null)
   and other conditions)
```

Good plan: hash join etc
Very fast

Query plan: nested loop join for the subquery
Takes forever
Why?

• One reason: estimation

• DBMSs estimate selectivity of conditions (by sampling the database) for estimating sizes of joins

• It’s 2017; they ought to be good at it, but:

• All major DBMSs underestimate join sizes by several orders of magnitude! (Leis, Gubichev, Boncz, Kemper, Neumann: How Good Are Query Optimizers, Really? VLDB 2015)

• Underestimate join size: nested loop is no big deal
It’s only part of the story…

select …
from …
where not exists
(select *
  from R1, R2, R3…
  where R1.A=R2.B or R1.C is null
  and R1.C=R2.D or R2.D is null
  …..
  and R3.F=R2.G or R3.F is null

Apply:

\[ \neg \exists ((f_1 \lor g_1) \land (f_2 \lor g_2) \land (f_n \lor g_n)) \rightarrow \neg \exists (F_1 \land G_1) \land \ldots \land \neg \exists (F_m \land G_m) \]

Exponential blow up
Before…

A textbook query: *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_partkey = p_partkey 
        AND l_suppkey = s_suppkey 
        AND p_name LIKE '%'||$color||'%' 
        AND s_nationkey = n_nationkey 
        AND n_name = $nation 
)
```
... and after

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 does theory teach us?

• Query $Q$ on database $D$ is evaluated in $O(D^Q)$. Turning $Q$ into something of size $2^Q$ is BAD!
• What does theory teach us?
  • Query Q on database D is evaluated in $O(D^Q)$. Turning Q into something of size $2^Q$ is BAD!

• What do systems texts teach us?
  • Optimizers work within a single \texttt{select} statement, splitting is BAD!
• What does theory teach us?
  • Query Q on database D is evaluated in $O(D^Q)$. Turning Q into something of size $2^Q$ is BAD!

• What do systems texts teach us?
  • Optimizers work within a single select statement, splitting is BAD!

• What actually happens?
  • Exponential blowup makes the query several orders of magnitude faster!
• What does theory teach us?
  • Query Q on database D is evaluated in $O(D^Q)$. Turning Q into something of size $2^Q$ is BAD!

• What do systems texts teach us?
  • Optimizers work within a single `select` statement, splitting is BAD!

• What actually happens?
  • Exponential blowup makes the query several orders of magnitude faster!
  • Why? It makes enough modifications to teach DBMS to avoid nested loops.
Bottom line

• SQL is the main data science tool

• But we do not fully understand it:
  • We do not have a full semantics of it
  • We do not know how to handle missing data
  • We do not know how to optimize some very common queries
  • We do not know how to estimate
  • We do not know how to handle disjunctions
Bottom line

• SQL is the main data science tool

• But we do not fully understand it:
  • We do not have a full semantics of it
  • We do not know how to handle missing data
  • We do not know how to optimize some very common queries
    • We do not know how to estimate
    • We do not know how to handle disjunctions

This is why data wrangling research is so important
For logicians in the room (if there is time)

• a bit on the logic of SQL.

• We know SQL uses 3-valued logic when NULL is present. Seems like a lot of baggage for just one extra value!

• Committee decision from almost 40 years ago.

• Let’s revisit things (history as it should have been)
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 ∧, ∨, ¬
  - Quantifiers: ∀ is conjunction, ∃ 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 ∧, ∨, ¬
  - Quantifiers: ∀ is conjunction, ∃ 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 \( \land, \lor, \neg \)
  • 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: \( \uparrow \text{p} \) means that p is true.

- Amazingly, 40 years later SQL adopted the same idea.
What did SQL really do?

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

\[
\uparrow \varphi = \begin{cases} 
    \text{true}, & \text{if } \varphi \text{ is true} \\
    \text{false}, & \text{if } \varphi \text{ is false or unknown}
\end{cases}
\]
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: \( \uparrow \) can be expressed in 3-valued FO.

3-valued FO = 3-valued FO with \( \uparrow \)

THEOREM: For every formula \( \varphi \) of 3-valued FO, there is a formula \( \psi \) of the usual 2-valued FO such that

\[ \varphi \text{ is } \text{true} \iff \psi \text{ is } \text{true} \]
THEOREM: \( \uparrow \) can be expressed in 3-valued FO.

3-valued FO = 3-valued FO with \( \uparrow \)

THEOREM: For every formula \( \varphi \) of 3-valued FO, there is a formula \( \psi \) of the usual 2-valued FO such that

\[ \varphi \text{ is } \text{true} \iff \psi \text{ is } \text{true} \]

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 $P^t$
We have produced a formal semantics of a basic fragment of SQL that behaves like two-valued SQL: translation

<table>
<thead>
<tr>
<th>Original Expression</th>
<th>Translated Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(i)^t = P(i)$</td>
<td>$P(t_1, \ldots, t_k)^f = \neg P(t_1, \ldots, t_k) \text{ AND } \bar{t} \text{ IS NOT NULL}$</td>
</tr>
<tr>
<td>$(\exists Q)^t = \exists Q'$</td>
<td>$(\exists Q)^f = \neg \exists Q'$</td>
</tr>
<tr>
<td>$(\theta_1 \land \theta_2)^t = \theta_1^t \land \theta_2^t$</td>
<td>$(\theta_1 \land \theta_2)^f = \theta_1^f \lor \theta_2^f$</td>
</tr>
<tr>
<td>$(\theta_1 \lor \theta_2)^t = \theta_1^t \lor \theta_2^t$</td>
<td>$(\theta_1 \lor \theta_2)^f = \theta_1^f \land \theta_2^f$</td>
</tr>
<tr>
<td>$(-\theta)^t = \theta^f$</td>
<td>$(-\theta)^f = \theta^t$</td>
</tr>
<tr>
<td>$(t \text{ IS NULL})^t = t \text{ IS NULL}$</td>
<td>$(t \text{ IS NULL})^f = t \text{ IS NOT NULL}$</td>
</tr>
<tr>
<td>$(\bar{t} \text{ IN } Q)^t = \bar{t} \text{ IN } Q'$</td>
<td>$((t_1, \ldots, t_n) \text{ IN } Q)^f = \neg \exists \text{ WHERE } (t_1 \text{ IS NULL OR } A_1 \text{ IS NULL OR } t_1 = N.A_1) \text{ AND } \cdots \cdots \text{ AND } (t_n \text{ IS NULL OR } A_n \text{ IS NULL OR } t_n = N.A_n)$</td>
</tr>
</tbody>
</table>

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?