Do we really understand SQL?
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**

<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>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>cl</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>

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:

\[ \pi_{\text{cust_id}} (\text{Pay}) - \]

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

Find ids of customers who buy all books:

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

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

Find ids of customers who buy all books:

\[ \pi_{\text{cust}_\text{id}} (\text{Pay}) - \]

\[ \pi_{\text{cust}_\text{id}} \left( (\pi_{\text{cust}_\text{id}}(\text{Pay}) \times \pi_{\text{title}}(\text{Order})) - \right. \]

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

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

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

Find ids of customers who buy all books:

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

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: \( RA = FO \).
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:

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

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**: 

```
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))
```

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


- 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!
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 $R - S$.

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:

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

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:

\[
\text{select * from r except select * from s}
\]
**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:

```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:

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

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:

$$\text{select * from r except select * from s}$$
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
```

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

1

∅
SQL vs Relational Algebra: attributes may repeat

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

\[
\begin{array}{|c|}
\hline
A \\
\hline
1 \\
null \\
\hline
\end{array}
\]

gives

\[
\begin{array}{|c|c|}
\hline
A & A \\
\hline
1 & 1 \\
null & null \\
\hline
\end{array}
\]
SQL vs Relational Algebra: attributes may repeat

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

\[ \begin{array}{c}
A \\
1 \\
null
\end{array} \]

\text{gives}

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

Let’s use it as a subquery:

\[ Q' = \text{SELECT } * \text{ FROM } (Q) \text{ AS } T \]
SQL vs Relational Algebra: attributes may repeat

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

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

Output:
- **Postgres**: as above
- **Oracle, MS SQL Server**: compile-time error
SQL vs Relational Algebra: attributes may repeat

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

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

\[ \text{SELECT } R.A \text{ FROM } R \text{ WHERE EXISTS } (Q’) \]
SQL vs Relational Algebra: attributes may repeat

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

\[
\begin{array}{c|c|c}
A & A \\
\hline
1 & 1 \\
null & null
\end{array}
\]
gives

Let’s use it as a subquery:

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

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

**SELECT R.A FROM R WHERE EXISTS (Q’)**

Answer:

\[
\begin{array}{c|c}
A & \\
\hline
1 & \\
null &
\end{array}
\]
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.
  - Under these assumptions several semantics exist (1985 - 2017) but they do not model the real language.
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

\[
\begin{array}{cc}
A & B \\
1 & 2 \\
3 & 4 \\
\end{array}
\]

return

\[
\begin{array}{c}
A \\
1 \\
3 \\
\end{array}
\]
Another example: Query equivalences

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

In theory: equivalent; on

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

return

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

Now the same in SQL:

Q1 = SELECT R.A FROM R

returns
Another example: Query equivalences

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

Now the same in SQL:

Q1 = SELECT R.A FROM R
returns

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

In theory: equivalent; on

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

return

A

1
3
The infamous NULL

• Comparisons with nulls, like 2 = NULL, result in truth value unknown

• It then propagates: true \land unknown = unknown, true \lor 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

\[ \tau : \beta := T_1 \text{ as } N_1, \ldots, T_k \text{ as } N_k \quad \text{for } \tau = (T_1, \ldots, T_k), \beta = (N_1, \ldots, N_k), \ k > 0 \]
\[ \alpha : \beta' := t_1 \text{ as } N'_1, \ldots, t_m \text{ as } N'_m \quad \text{for } \alpha = (t_1, \ldots, t_m), \beta' = (N'_1, \ldots, N'_m), m > 0 \]

Queries:
\[ Q := \text{SELECT } [\text{DISTINCT}] \alpha : \beta' \text{ FROM } \tau : \beta \text{ WHERE } \theta \]
\[ \quad \mid \text{SELECT } [\text{DISTINCT}] * \text{ FROM } \tau : \beta \text{ WHERE } \theta \]
\[ \quad \mid Q \text{ (UNION | INTERSECT | EXCEPT) [ALL] } Q \]

Conditions:
\[ \theta := \text{TRUE} | \text{FALSE} | P(t_1, \ldots, t_k), \ P \in \mathcal{P} \]
\[ \quad \mid t \text{ is [NOT] NULL} \]
\[ \quad \mid \bar{t} \text{ [NOT] IN } Q \mid \text{EXISTS } Q \]
\[ \quad \mid \theta \text{ AND } \theta \mid \theta \text{ OR } \theta \mid \text{NOT } \theta \]

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

Terms t: constants, nulls, or composite names

Predicates: anything you want on constants
Semantics: labels

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

\[ \text{Semantics} \]

\[ \alpha \]

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

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

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

\[
[R]_{D,\eta,x} = R^D
\]
\[
[\tau:\beta]_{D,\eta,x} = [T_1]_{D,\eta,0} \times \cdots \times [T_k]_{D,\eta,0} \quad \text{for } \tau = (T_1, \ldots, T_k)
\]
\[
\begin{aligned}
\left[ \begin{array}{c}
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,x}^{\tau:\beta} \theta
\end{aligned}
\]
\[
\left\{ \bar{r}, \ldots, \bar{r} \right\}_{k \text{ times}} \quad \bar{r} \in_k [\tau:\beta]_{D,\eta,0}, \quad [\theta]_{D,\eta'} = t, \quad \eta' = \eta \oplus \ell(\tau:\beta)
\]
\[
\begin{aligned}
\left[ \begin{array}{c}
\text{SELECT} \\
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,x}^{\alpha:\beta'} \theta
\end{aligned}
\]
\[
\left\{ \left[\alpha\right]_{\eta'}, \ldots, \left[\alpha\right]_{\eta'} \right\}_{k \text{ times}} \quad \eta' = \eta \oplus \ell(\tau:\beta), \quad \bar{r} \in_k \left[ \begin{array}{c}
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,x}^{\tau:\beta} \theta
\]
\[
\begin{aligned}
\left[ \begin{array}{c}
\text{SELECT} \\
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,0}^{\tau:\beta} \theta
\end{aligned}
\]
\[
\begin{aligned}
\left[ \begin{array}{c}
\text{SELECT} \\
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,0} \quad \ell(\tau:\beta) : \ell(\tau)
\end{aligned}
\]
\[
\left[ \begin{array}{c}
\text{SELECT} \\
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,1}^{\alpha:\beta'} \theta
\]
\[
\begin{aligned}
\left[ \begin{array}{c}
\text{SELECT DISTINCT} \\
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,x}^{\tau:\beta \mid \ast} \theta
\end{aligned}
\]
\[
\varepsilon \left( \left[ \begin{array}{c}
\text{SELECT} \\
\text{FROM} \end{array} \right]_{D,\eta,x}^{\alpha:\beta' \mid \ast} \right)
\]

Figure 5: Semantics of basic SQL: Conditions.

Figure 4: Semantics of basic SQL: Queries.
Semantics: conditions

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

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

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

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

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

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

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

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

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

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

Truth Tables:

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

\[
\left[ \begin{array}{c}
\text{FROM} \\
\text{WHERE}
\end{array} \right]_{D,\eta,x}^{\tau:\beta} = \left\{ \bar{r}, \ldots, \bar{r} \mid \bar{r} \in_k [\tau:\beta]_{D,\eta,0}, \theta \right\}_{D,\eta'} = t, \quad \eta' = \eta \oplus \ell(\tau:\beta)
\]

• 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 \( r \)
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

- 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 ∧, ∨, ¬
  - 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 $\land$, $\lor$, $\neg$

  • 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 \( \land, \lor, \neg \)
  - 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: \( \uparrow p \) 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

$\varphi$ is true $\iff$ $\psi$ is 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

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

Translations work at the level of SQL too!
2-valued SQL

Idea — 3 simultaneous translations:

• conditions $P \rightarrow P^t$ and $P^f$

• Queries $Q \rightarrow Q'$

$P^t$ 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

\[
\begin{align*}
    P(\bar{t})^t &= P(\bar{t}) & P(t_1, \ldots, t_k)^f &= \text{NOT } P(t_1, \ldots, t_k) \text{ AND } \bar{t} \text{ IS NOT NULL} \\
    (\exists Q)^t &= \exists Q' & (\exists Q)^f &= \text{NOT EXISTS } Q' \\
    (\theta_1 \land \theta_2)^t &= \theta_1^t \land \theta_2^t & (\theta_1 \land \theta_2)^f &= \theta_1^f \lor \theta_2^f \\
    (\theta_1 \lor \theta_2)^t &= \theta_1^t \lor \theta_2^t & (\theta_1 \lor \theta_2)^f &= \theta_1^f \land \theta_2^f \\
    (\neg \theta)^t &= \theta^t & (\neg \theta)^f &= \theta^t \\
    (t \text{ IS NULL})^t &= t \text{ IS NULL} & (t \text{ IS NULL})^f &= t \text{ IS NOT NULL} \\
    (\bar{t} \text{ IN } Q)^t &= \bar{t} \text{ IN } Q' & ((t_1, \ldots, t_n) \text{ IN } Q)^f &= \text{NOT EXISTS } (\text{SELECT } * \text{ FROM } Q' \text{ AS } N(A_1, \ldots, A_n) \text{ WHERE} \\
    & & & (t_1 \text{ IS NULL OR } A_1 \text{ IS NULL OR } t_1 = N.A_1) \text{ AND } \ldots \\
    & & & \ldots \text{ AND } (t_n \text{ IS NULL OR } A_n \text{ IS NULL OR } t_n = N.A_n))
\end{align*}
\]

Note: a lot of disjunctions with IS NULL conditions
Shall we switch to 2-valued SQL?
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!

    • we talked about it earlier