Introduction to Databases

(INFR10080)

(The NULL value)

Instructor: Yang Cao

(Fall 2025)

THE UNIVERSITY *of* EDINBURGH

Changelog

v25.0 Initial version

NULL: all-purpose marker to represent incomplete information
Main source of problems and inconsistencies

> *"…this topic cannot be described in a manner that is simultaneously both comprehensive and comprehensible"*
>
> *"Those SQL features are …fundamentally at odds with the way the world behaves"*
>
> — *C. Date & H. Darwen,* A Guide to SQL Standard

What does NULL mean?

Depending on the context:

- Missing value – there is a value, but it is currently **unknown**
- Non-applicable – there is no value (**undefined**)

But it also behaves as:

- Constant – like any other value
- Unknown – a truth-value in addition to True and False

Meta-incompleteness

We never really know what NULL means
because the meaning is ultimately defined by the application

But we must know how NULL **behaves** according to the Standard
and this behavior depends on the context in which it is used

Missing value vs. Non-applicable

Person

ID	Name	Phone	
1	Jane	NULL	Does Jane have a phone?
2	John	NULL	Does John have a phone?

There is no way of knowing whether a NULL here means that

- there is a currently unknown value for phone (**missing value**)

  or
- there is no value for phone (**non-applicable value**)

NULL and schema design

Person

ID	Name	HasPhone	Phone	
1	Jane	Yes	NULL	← missing
2	John	No	NULL	← non-applicable

What if we want **Phone** to be NOT NULL
when **HasPhone** is Yes?

- we cannot just declare **Phone** as NOT NULL
- we need to use a CHECK constraint

We also want to check that **Phone** is NULL when **HasPhone** is No

NULL and schema design

Person

ID	Name	HasPhone	Phone
1	Jane	NULL	NULL
2	John	NULL	12341

We don't know whether John and Jane have a Phone

- NULL in column **HasPhone** represents a **missing value**
- What does NULL in column **Phone** mean?

What about the value 12341 for John's Phone?

- Rule out these cases with a CHECK constraint
- Declare **HasPhone** to be NOT NULL
  (we cannot say we don't know whether a person has a Phone)

NULL and schema design

Person

ID	Name
2	John

PersonWithPhone

ID	Name	Phone
1	Jane	NULL

Pros:

- NULLs in **Phone** represent missing values
- **Phone** can be declared NOT NULL if needs be

Cons:

- Need to make sure there is **no overlap** (assertion? trigger?)
- How do we say "I don't know whether John has a Phone"?
  (we could add a column **HasPhone** to **Person**...)

NULL and schema design

Person

ID	Name
1	Jane
2	John

PersonWithPhone

ID	Phone
1	NULL

PersonWithPhone(ID) REFERENCES Person(ID)

Pros:

- NULLs in **Phone** represent missing values
- **Phone** can be declared NOT NULL if needs be

Cons:

- How do we say "I don't know whether John has a Phone"?
  (we could add a column **HasPhone** to **Person**...)

Limitations of SQL's NULL as missing values

Person

Name	Age
Jane	NULL
John	NULL
Mary	27
Carl	NULL

Age of Jane, John and Carl is unknown

We know Jane and John have **the same** age

Marked nulls (not part of SQL)

- Each missing value has an **identifier**
- Allow **cross-referencing** of missing values

NULL and constraints

Nulls are not allowed in primary keys

```
CREATE TABLE R ( A INT PRIMARY KEY );
INSERT INTO R VALUES (NULL);
```

ERROR: null value in column "a" violates not-null constraint

Nulls seem to behave as (distinct) missing values with UNIQUE

```
CREATE TABLE R ( A INT UNIQUE );
INSERT INTO R VALUES (NULL);
INSERT INTO R VALUES (NULL);
```

R

A
NULL
NULL

but in fact this is simply because NULLs are **ignored**

NULL and constraints

R

A	B
1	1

S

A	B
1	NULL
2	NULL

S(A,B) REFERENCES R(A,B)

Is NULL treated as a missing value here? **Not really!**

The above instance is legal w.r.t. the FK constraint

NULL and arithmetic operations

Every arithmetic operation that involves a NULL results in NULL

```
SELECT 1+NULL AS sum , 1-NULL AS diff ,
       1*NULL AS mult , 1/NULL AS div
```

```
 sum  | diff | mult | div
------+------+------+------
 NULL | NULL | NULL | NULL
(1 row)
```

Observe that SELECT NULL/0 also returns NULL
instead of throwing a DIVISION BY ZERO error!

Here, NULL is treated as an **undefined** value

NULL and aggregation (1)

Aggregate functions ignore nulls

Consider $R = \{0, \text{NULL}, 1, \text{NULL}\}$ on attribute A

```
SELECT MIN(A), MAX(A), COUNT(A), SUM(A),
       CAST( AVG(A) AS numeric(2,1) )
FROM   R ;
```

```
 min | max | count | sum | avg
-----+-----+-------+-----+-----
   0 |   1 |     2 |   1 | 0.5
(1 row)
```

NULL and aggregation (2)

Aggregate functions ignore nulls

Consider $R = \{0, \text{NULL}, 1, \text{NULL}\}$ on attribute A

Exception:

```
SELECT COUNT(*) FROM R ;

 count
-------
     4
(1 row)
```


NULL and aggregation

Aggregation (except COUNT) on an empty bag results in NULL

Consider $R = \{0, 1, \text{NULL}\}$ on attribute A

```
SELECT MIN(A), MAX(A), SUM(A), AVG(A), COUNT(A)
FROM   R
WHERE  A = 2 ;

 min  | max  | sum  | avg  | count
------+------+------+------+-------
 NULL | NULL | NULL | NULL |     0
(1 row)
```

The semantics of these nulls is that of **undefined** values

NULL and set operations

What is the answer to

Q_1: SELECT * FROM R UNION SELECT * FROM S

Q_2: SELECT * FROM R INTERSECT SELECT * FROM S

Q_3: SELECT * FROM R EXCEPT SELECT * FROM S

when $R = \{1, \text{NULL}, \text{NULL}\}$ and $S = \{\text{NULL}\}$?

- Answer to Q_1: $\{1, \text{NULL}\}$
- Answer to Q_2: $\{\text{NULL}\}$
- Answer to Q_3: $\{1\}$

In set operations NULL is treated like any other value

NULL and set operations

What is the answer to

Q_1: SELECT * FROM R UNION ALL SELECT * FROM S

Q_2: SELECT * FROM R INTERSECT ALL SELECT * FROM S

Q_3: SELECT * FROM R EXCEPT ALL SELECT * FROM S

when $R = \{1, \text{NULL}, \text{NULL}\}$ and $S = \{\text{NULL}\}$?

- Answer to Q_1: $\{1, \text{NULL}, \text{NULL}, \text{NULL}\}$
- Answer to Q_2: $\{\text{NULL}\}$
- Answer to Q_3: $\{1, \text{NULL}\}$

NULL in selection conditions (1)

What is the answer to

Q_1: SELECT * FROM R, S WHERE R.A = S.A

Q_2: SELECT * FROM R, S WHERE R.A <> S.A

Q_3: SELECT * FROM R, S WHERE R.A = S.A OR R.A <> S.A

when $R = \{1, \text{NULL}\}$ and $S = \{\text{NULL}\}$?

R.A
1
NULL

$\times$

S.A
NULL

$=$

R.A	S.A
1	NULL
NULL	NULL

Answer to all three queries: {}

NULL and comparisons

```
SELECT 1=NULL AS result;
 result
--------
 NULL
(1 row)
```

This is not an undefined value – it is a truth-value: **unknown**

```
SELECT 1=NULL OR TRUE AS result;
 result
--------
 t
(1 row)
```

Try: SELECT NULL/1 OR TRUE AS result;

Evaluation of selection conditions

SQL uses three truth values: **true** (t), **false** (f), **unknown** (u)

1. Every comparison (except IS [NOT] NULL and EXISTS) where one of the arguments is NULL evaluates to unknown
2. The truth values assigned to each comparison are propagated using the following tables:

AND	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

OR	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

	NOT
t	f
f	t
u	u

3. The rows for which the condition evaluates to true are returned

NULL in selection conditions (2)

What is the answer to

Q_1: SELECT * FROM R, S WHERE R.A = S.A

Q_2: SELECT * FROM R, S WHERE R.A <> S.A

Q_3: SELECT * FROM R, S WHERE R.A = S.A OR R.A <> S.A

when $R = \{1, \text{NULL}\}$ and $S = \{\text{NULL}\}$?

R.A	S.A
1	NULL
NULL	NULL

θ_1
u
u

θ_2
u
u

θ_3
u
u

NULL and query equivalence (1)

Q_1

```
SELECT R.A FROM R
INTERSECT
SELECT S.A FROM S
```

Q_2

```
SELECT DISTINCT R.A
FROM   R, S
WHERE  R.A = S.A
```

On databases without nulls, Q_1 and Q_2 give the same answers

On databases **with nulls**, they do not

For example, when $R = S = \{\text{NULL}\}$

- Q_1 returns $\{\text{NULL}\}$
- Q_2 returns $\{\}$

NULL and query equivalence (2)

Consider $R = \{1, \text{NULL}\}$ and $S = \{\text{NULL}\}$

Q_1:

```
SELECT R.A FROM R
EXCEPT
SELECT S.A FROM S ;
```

Answer: $\{ 1 \}$

Q_2:

```
SELECT DISTINCT R.A
FROM   R
WHERE  NOT EXISTS (
       SELECT *
       FROM   S
       WHERE  S.A=R.A );
```

Answer: $\{ 1, \text{NULL} \}$

Q_3:

```
SELECT DISTINCT R.A
FROM   R
WHERE  R.A NOT IN (
       SELECT S.A
       FROM   S );
```

Answer: $\{\}$

Inner joins

R	
A	B
1	3
2	2

S	
C	D
4	1
3	2

```
SELECT * FROM R [INNER] JOIN S ON R.B = S.C ;
```

A	B	C	D
1	3	3	2

Outer joins (1)

R	
A	B
1	3
2	2

S	
C	D
4	1
3	2

```
SELECT * FROM R LEFT [OUTER] JOIN S ON R.B = S.C ;
```

A	B	C	D
1	3	3	2
2	2		

Same as

```
SELECT * FROM R JOIN S ON R.B = S.C
UNION ALL
SELECT R.*, NULL, NULL FROM R
WHERE NOT EXISTS (
    SELECT * FROM S WHERE R.B = S.C )
```

Outer joins (2)

R	
A	B
1	3
2	2

S	
C	D
4	1
3	2

```
SELECT * FROM R RIGHT [OUTER] JOIN S ON R.B = S.C ;
```

A	B	C	D
		4	1
1	3	3	2

Same as

```
SELECT * FROM R JOIN S ON R.B = S.C
UNION ALL
SELECT NULL, NULL, S.* FROM S
WHERE NOT EXISTS (
    SELECT * FROM R WHERE R.B = S.C )
```


Outer joins (3)

R	
A	B
1	3
2	2

S	
C	D
4	1
3	2

```
SELECT * FROM R FULL [OUTER] JOIN S ON R.B = S.C ;
```

A	B	C	D
1	3	3	2
2	2		
		4	1

Same as

```
SELECT * FROM R JOIN S ON R.B = S.C
UNION ALL
SELECT R.*, NULL, NULL FROM R
WHERE NOT EXISTS (
    SELECT * FROM S WHERE R.B = S.C )
UNION ALL
SELECT NULL, NULL, S.* FROM S
WHERE NOT EXISTS (
    SELECT * FROM R WHERE R.B = S.C )
```

Coalescing null values

Syntax: COALESCE(expr1, expr2)

Same as

```
CASE WHEN expr1 IS NULL
     THEN expr2
     ELSE expr1
END
```

Example

R:

A
1
3

```
SELECT COALESCE(R.A,0) AS A FROM R
```

gives

A
1
0
3