NULL VALUE

→ The meaning of NULL depends 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**
- **Unknown** — a truth-value in addition to True/False

→ This gives us some meta-incompleteness abt the NULL value:

- We never rlly know what NULL means
  bc. the meaning is defined by the application
- BUT we must know how NULL behaves according to the standard

Person

ID	Name	Phone
1	Jane	NULL
2	John	NULL

→ There is no way of knowing the meaning of NULL here (from the table alone)

missing value? non-applicable value?

Can overcome this by...

Person

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

Use CHECK constraint

- We want Phone to be NOT NULL when HasPhone is Yes
- We want Phone is NULL when HasPhone is No.

NULL & CONSTRAINTS

→ Nulls are not allowed in primary keys

→ Nulls seem to behave as distinct missing values w/ UNIQUE

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

R
A
NULL
NULL

but in fact, it's simply bc. NULLs are ignored!

→ In checking the foreign key constraint, you only take the rows that do not have NULLs in the cols. tht. are affected by the constraint

e.g.

R1	
A	B
1	1

R2	
A	B

S	
A	B
1	NULL
2	NULL

S(A,B) REFERENCES R1(A,B)
S(A,B) REFERENCES R2(A,B)

The above instances are legal w.r.t. the FK constraint

NULL & ARITHMETIC OP.

→ Every arithmetic operation tht. involves a NULL results in NULL

→ This includes: NULL/0 returns NULL instead of throwing division by zero error

→ Here, NULL is treated as an undefined value

NULL & AGGREGATION

→ Aggregate fn ignores NULLs

i.e. Consider R = {0, NULL, 1, 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

The semantics of these NULLs is that of undefined values

→ EXCEPTION:

```
SELECT COUNT(*) FROM R; → 4
```

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

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

→

min	count
NULL	0

NULL & SET OP.

→ Suppose $R = \{1, NULL, NULL\}$ and $S = \{NULL\}$

Q_1 = SELECT * FROM R UNION [ALL] SELECT * FROM S

Q_2 = —— " —— INTERSECT [ALL]

Q_3 = —— " —— EXCEPT [ALL]

BAG SEMANTICS ANS.

↳ Answer to Q_1 : $\{1, NULL\}$ — $\{1, NULL, NULL, NULL\}$

→ Here, NULL is treated as a constant

↳ Answer to Q_2 : $\{NULL\}$ — $\{NULL\}$

↳ Answer to Q_3 : $\{1\}$ — $\{1, NULL\}$

NULL IN SELECT CONDITIONS

→ Suppose $R = \{1, NULL\}$ and $S = \{NULL\}$

Q_1 = SELECT * FROM R, S WHERE R.A = S.A (θ_1)

Q_2 = —— " —— R.A <> S.A (θ_2)

Q_3 = —— " —— R.A = S.A OR R.A <> S.A (θ_3)

tautology in boolean logic

R.A
1
NULL

×

S.A
NULL

=

R.A	S.A
1	NULL
NULL	NULL

↳ In comparisons, NULL is treated as a truth-value: unknown!

e.g. SELECT 1 = NULL AS result; → result = NULL

SELECT 1 = NULL OR TRUE AS result; → result = True

↳ Type-checking in NULL:

SELECT NULL/1 OR TRUE AS result; → ERROR!

↳ returns NULL but it is not a truth-value

↳ Answer to all 3 queries: $\{\}$

θ_1	θ_2	θ_3
u	u	u
u	u	u

→ Condition a) at next page

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

a. Every comparison (except IS [NOT] NULL, and EXISTS) where one of the args. is NULL evaluates to unknown

b. The truth values assigned to each comparison are propagated using the ff. 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

c. The rows for which the condition evaluates to true are returned
↳ false & unknown are discarded

NULL & QUERY EQUIVALENCE

e.g.

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 dbases w/o NULLs, Q_1 and Q_2 give same answers

→ On dbases w/ NULLs, Q_1 returns {NULL} → NULL treated as constant
Q_2 returns {} → NULL treated as truth-values

e.g.

Q_1
```
SELECT R.A FROM R
EXCEPT
SELECT S.A FROM S;
```

Q_2
```
SELECT DISTINCT R.A
FROM R
WHERE NOT EXISTS (
    SELECT *
    FROM S
    WHERE S.A = R.A);
```
= {} if R.A = 1
{} if R.A = NULL

Q_3
```
SELECT DISTINCT R.A
FROM R
WHERE R.A NOT IN (
    SELECT S.A
    FROM S);
```
= 1 <> NULL → u
NULL <> NULL → u

Consider R = {1, NULL} and S = {NULL}

→ $Q_1 = \{1\}$

→ In Q_2, EXISTS never evaluates to unknown (see a) above). $Q_2 = \{1, NULL\}$

→ In Q_3, NOT IN ≡ <> ALL. $Q_3 = \{\}$

JOINS

→ Using JOIN in SQL is by default doing an INNER JOIN

→ There are 3 types of OUTER JOIN:

1) **LEFT JOIN**

R	
A	B
1	3
2	2

S	
C	D
4	1
3	2

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

↓

A	B	C	D
1	3	3	2
2	2	NULL	NULL

2) **RIGHT JOIN**

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

→

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

3) **FULL JOIN**

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

→

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

COALESCING NULL VALUES

```
COALESCE(expr1, expr2) ≡ CASE WHEN expr1 IS NULL
                            THEN expr2
                            ELSE expr1
                         END
```

e.g.

R.A
1
NULL
3

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

→

R.A
1
0
3