On the Vexing Difficulty of Evaluating IN Predicates

Altan Birler[1] Thomas Neumann[1]

[1]TUM

IN Expressions

In the SQL standard since its inception. Widely used and supported.

```
SELECT *
FROM R
WHERE (x, y) NOT IN (SELECT a, b FROM L)
```

Looks like a simple anti join? (but it's not)

Looks like linear runtime? (but it's not)

IN Expressions

```
SELECT (x,y) IN (VALUES
                  (1, 2),
                  (3, 4),
                  (5, 6))
FROM (VALUES(1, 2),
            (7, NULL),
            (5, NULL)) s(x,y)
```

$$(x = 1 \land y = 2)$$
$$\lor (x = 3 \land y = 4)$$
$$\lor (x = 5 \land y = 6)$$

IN Expressions

```
SELECT (1, 2) IN (VALUES
                  (1, 2),
                  (3, 4),
                  (5, 6))
-- Result: TRUE
-- Perfect match
```

$$
\begin{aligned}
&(1 = 1 \wedge 2 = 2) && (\text{TRUE}) \\
&\vee (1 = 3 \wedge 2 = 4) && \vee (\text{FALSE}) \\
&\vee (1 = 5 \wedge 2 = 6) && \vee (\text{FALSE})
\end{aligned}
$$

IN Expressions

```
SELECT (7, NULL) IN (VALUES
                       (1, 2),
                       (3, 4),
                       (5, 6))
-- Result: FALSE
-- No match
```

$$
\begin{aligned}
&(7 = 1 \wedge \mathrm{NULL} = 2) && (\mathrm{FALSE}) \\
\vee\ &(7 = 3 \wedge \mathrm{NULL} = 4) && \vee\ (\mathrm{FALSE}) \\
\vee\ &(7 = 5 \wedge \mathrm{NULL} = 6) && \vee\ (\mathrm{FALSE})
\end{aligned}
$$

IN Expressions

```
SELECT (5, NULL) IN (VALUES
                     (1, 2),
                     (3, 4),
                     (5, 6))
-- Result: NULL
-- Partial match
```

$$(5 = 1 \wedge \text{NULL} = 2) \quad (\text{FALSE})$$
$$\vee\, (5 = 3 \wedge \text{NULL} = 4) \quad \vee\, (\text{FALSE})$$
$$\vee\, \mathbf{(5 = 5 \wedge \text{NULL} = 6)} \quad \vee\, \mathbf{(\text{NULL})}$$

To distinguish FALSE and NULL, we conceptually need to iterate over all values in the VALUES list to make sure we have no partial match.

Semantics of IN Expressions

IN expressions with NULLs return 3 values:

- **TRUE:** Exact match. ⇒ where clause with *IN predicate*
- **NULL:** Partial match, either same value or NULL
- **FALSE:** No match. ⇒ where clause with *NOT IN predicate*

Semantics of IN Expressions

So suprising that even database implementors do not get it right.

Wrong results: ClickHouse 25.8, Hyper 9.1.0, DuckDB 1.3.0, Snowflake 9.21.0, Redshift 1.0.118447, DB2 11.1, and Firebolt Core 4.23.5

Partial support (single attribute): SQL Server 2022 and BigQuery 2025-07-22

Correct results: PostgreSQL 17, Oracle 23c, Databricks 2025.16, Umbra 25.07, … (see the paper)

All correct systems can exhibit quadratic runtime.

Runtime of IN Expressions

All correct systems can be slow. Unfortunately, that is fundamental:

Quadratic runtime in general for queries with IN expressions.

We showed: If IN expressions can be computed in $n^{2-\varepsilon}$ time,
SAT can be solved faster than 2^n time.

(We prove this by reducing from orthogonal vectors to IN expressions)

Implementation: Right Mark Join

$$L \ltimes^{M}_{m:x=a \wedge y=b} R$$

- computes IN result as new column m for each tuple from R
- assumes $|L| < |R|$
- if not: swap L and R, use left mark join (see paper)

Quadratic worst case, but fast in common cases with few NULLs.

Implementation: Right Mark Join (Build)

- Keep a hash table on all attributes for checking perfect matches.
- Keep a second hash table on the not-nullable attributes for partial matches. (Keep tuples with nulls separate.)

```
for l in L:
  htFull.insert(l)
  if not isAnyNull(l):
    htProj[notNullable(l)].nonNull.insert(l)
  else:
    htProj[notNullable(l)].null.insert(l)
```

Implementation: Right Mark Join (Probe)

- Check for perfect matches

```
def probe(r):
  if htFull.contains(r):
    return True
  ...
```

Implementation: Right Mark Join (Probe)

- Check for partial matches. We first have to check the tuples on the other side that contain nulls

```
def probe(r):
  ...
  for l in htProj[notNullable(r)].null:
    if p(l, r) is NULL:
      return NULL
  ...
```

Implementation: Right Mark Join (Probe)

- If we still have not found a perfect or partial match, we might still have a chance if our tuple contains nulls.

```
def probe(r):
  ...
  if isAnyNull(r):
    for l in htProj[notNullable(r)].nonNull:
      if p(l, r) is NULL:
        return NULL
  ...
```

Implementation: Right Mark Join (Probe)

- If all else fails, we have no match.

```
def probe(r):
  ...
  return False
```

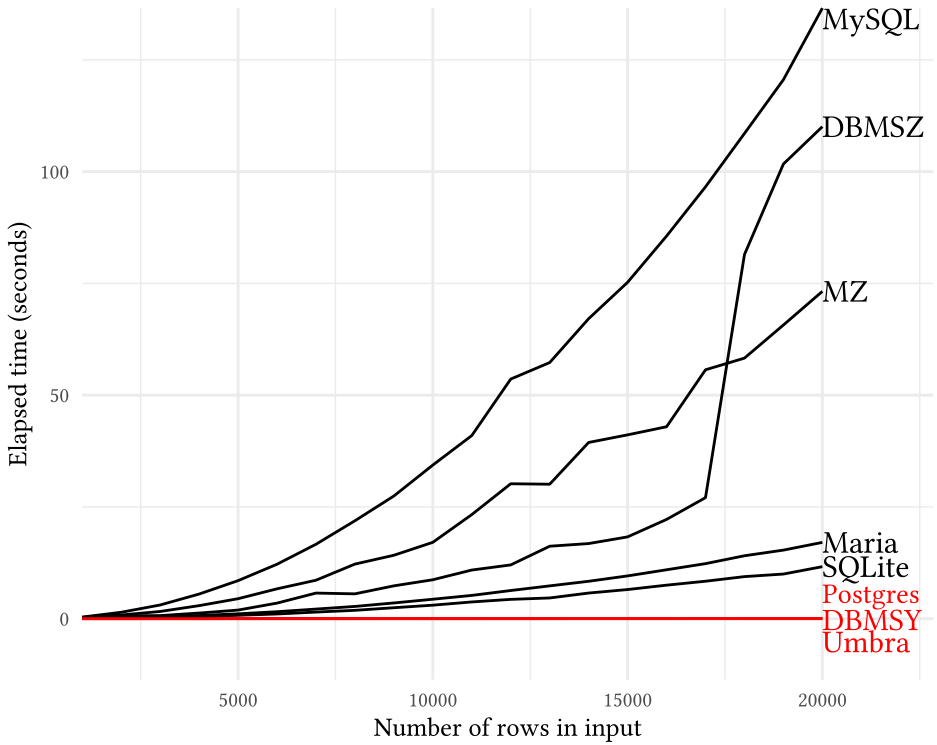
Implementation: Right Mark Join

- Handles all cases correctly.
- Exhibits linear runtime if there is only a single nullable atribute (even with many not nullable attributes).
- Smooth performance degradation for more complex cases.

Evaluation: Single Nullable Attribute

```
SELECT count(*)
FROM R
WHERE
(R.a, R.b) NOT IN (
  SELECT S.a, S.b
  FROM S)
```

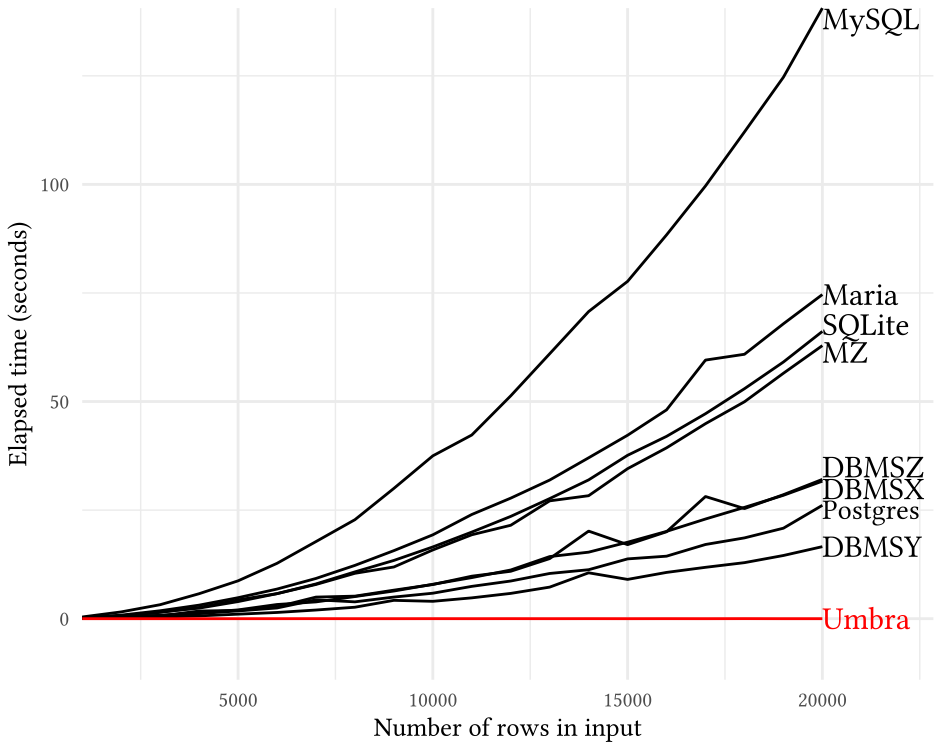
Some systems do not support this query!



Evaluation: Single Nullable Attribute

```
SELECT count(*)
FROM R
WHERE
R.a NOT IN (
  SELECT S.a
  FROM S
  WHERE S.b = R.b)
```

Equivalent to previous query but often slower!



Conclusion

- IN expressions are confusing and difficult to evaluate.
- ideally, we would change the standard
- until then, efficient implementation gracefully handles common cases

More details in the paper:

- Formal definition of the semantics
- Proof of hardness
- Left mark join
- Optimizations

TUM is hiring tenure track professor, til Feb 3rd. urlr.me/Kbdgy8