On the Vexing Difficulty of Evaluating IN Predicates

Altan Birler
altan.birler@tum.de
Technische Universität München
Garching, Germany

Thomas Neumann
neumann@in.tum.de
Technische Universität München
Garching, Germany

ABSTRACT

In SQL, IN predicates — and their syntactic cousin, quantified comparison predicates — are widely used and are part of the standard since its inception. Perhaps somewhat surprisingly, no efficient evaluation strategy is known. At first glance, these constructs behave like a semi join, but that is only true as long as no NULL values are involved. In the general case, the SQL standard mandates a behavior that is very difficult to implement efficiently. To the best of our knowledge, all existing systems either exhibit quadratic runtime in some cases or produce wrong results. This is rather troubling for such a common operation.

In this paper, we study this problem and make both practical and theoretical contributions: On the theoretical side, we show that the problem is inherently hard, and that there is no hope of finding an algorithm with subquadratic runtime in general, unless SAT can be solved more efficiently. On the practical side we show that while the problem is hard in general, common cases can be solved in linear time, and that even the bad cases can usually be solved more efficiently. Compared to existing solutions, this leads to nearly arbitrary gains in runtime.

CCS CONCEPTS

• **Information systems** → **Query operators**.

KEYWORDS

SQL, query processing, IN expressions, mark join

1 INTRODUCTION

The SQL IN predicate is a common way to check whether a tuple is contained in a bag of tuples. For example, the query

```
SELECT (3, 4) IN (VALUES (1, 2), (3, 4));
```

computes the following disjunction:

```
((3 = 1) AND (4 = 2)) OR ((3 = 3) AND (4 = 4))
```

As the second conjunct is TRUE, the result of the query is TRUE. Given that an index can be built on the bag, this query can be computed in linear time relative to the input size. However, if the bag of tuples contains NULL values, the result of the query must comply with SQL's three-valued logic. For example, the query

```
SELECT (3, 4) IN (VALUES (1, 2), (3, NULL));
```

computes the following disjunction:

```
((3 = 1) AND (4 = 2)) OR ((3 = 3) AND (4 = NULL))
```

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

```
1 -- IN predicate
2 SELECT *
3 FROM R
4 WHERE (x,y) NOT IN (SELECT a,b FROM S)
5 -- EXISTS predicate
6 SELECT *
7 FROM R
8 WHERE NOT EXISTS
9   (SELECT FROM S WHERE (a,b) = (R.x,R.y))
```

Figure 1: NOT IN and NOT EXISTS queries for checking containment. NOT EXISTS requires a correlated subquery while NOT IN does not. Which query runs faster? It depends.

As the first conjunct is FALSE and the second conjunct is NULL, the result of the query is NULL. As we will show, the presence of NULLs makes the problem significantly harder.

IN predicates are widely used and are a fundamental part of the SQL standard since its inception. One might expect that they allow for a linear time algorithm, which surprisingly is not the case. In this work, we show that this SQL-standard-mandated behavior for IN predicates with NULL values does not allow for subquadratic time evaluation in general, assuming the strong exponential time hypothesis. Nonetheless, we show that many common cases can be solved in linear time, and that even the hard cases can be solved relatively efficiently when the number of nullable attributes is small. Unfortunately, most existing database systems are either slow or not standard-compliant. ClickHouse 25.8, Hyper 9.1.0, DuckDB 1.3.0, Snowflake 9.21.0, Redshift 1.0.118447, and Firebolt Core 4.23.5 all give wrong results for queries with IN or NOT IN predicates and nullable attributes. SQL Server 2022 and BigQuery 2025-07-22 do not support IN predicates with multiple attributes, but are otherwise correct. Materialize v0.151.0, Databricks 2025.16, CedarDB v2025-07-23, SQLite 3.46.1, DB2 Developer-C 11.1, MariaDB 11.4, Oracle 23c, PostgreSQL 17, TimescaleDB 2.14, and YugabyteDB 2.18 produce correct results but can exhibit quadratic runtime even in simple cases. Our system Umbra, which implements the techniques presented in this paper, produces correct results and achieves significant speedups over naive approaches.

Many users prefer IN predicates or quantified comparison predicates over EXISTS predicates [3, 6], which also permeates into AI-generated queries [17]. However, the semantics and subpar performance of IN predicates with nullable attributes lead to a lot of confusion for users [12, 20]. There is a wide range of conflicting folklore on when to use IN predicates and when to use the similar EXISTS predicates as seen in Figure 1. IN predicates are sometimes recommended as they avoid correlated subqueries. Two of the top 3 search results for "exists vs in" on Google at the time of writing mentioned this advice [7, 9]. In other sources, including official Oracle documentation [16], EXISTS predicates are claimed

to perform better for larger (less selective) subqueries, likely as certain database implementations optimize IN predicates for small inline tables. Digging further down, some claim that EXISTS and IN perform equally well [18], but NOT EXISTS performs "radically" better than NOT IN if nullable values are involved [19]. This is a very confusing situation for users, and it is unclear whether the advice is applicable to a wide range of database systems, let alone any specific one. In this work, we aim to clarify the situation and provide clear recommendations for system implementers: Namely, that the combination of query decorrelation (cf. Section 4), mark join operators, and our proposed algorithm for evaluating nullable mark join operators (cf. Section 6) provides a robust and efficient solution for evaluating the widest range of queries.

In Section 2, we discuss the limited related work in efficiently evaluating IN predicates. In Section 3, we introduce the semantics of IN and quantified comparison predicates, and discuss the associated difficulties. In Section 4, we discuss how such predicates can be decorrelated and optimized in algebra using mark join operators. In Section 5, we give a sketch of why the problem is hard in general, and why it is *highly* unlikely that a general subquadratic algorithm exists. Given that the problem is hard in general, we focus on solving common cases efficiently in Section 6. In Section 7, we evaluate our approach to show that it is possible to achieve significant speedups over naive approaches. Finally, we conclude in Section 8, summarizing our recommendations.

2 RELATED WORK

While general issues with NULL values in SQL have been widely recognized [12, 20], the specific problem of evaluating NOT IN predicates efficiently has received little attention in the literature, despite being a fundamental part of the SQL standard since its inception. Many systems still use dependent nested loops to evaluate NOT IN predicates, which often leads to poor performance. A few database systems, such as PostgreSQL, SQLite, and MariaDB, have specific implementations for NOT IN predicates involving NULL values[1], and among these only PostgreSQL seems to reliably exhibit linear runtime when only one nullable attribute is involved (cf. Section 7); we are not aware of formal analyses of their algorithms. Although the source code is not thoroughly documented, the PostgreSQL implementation seems to be similar to the right mark join approach we present in Section 6.2.

Elhemali et al. [6] propose an approach that handles NULL values but is restricted to a single attribute by decomposing the operation into a sequence of anti joins. Bellamkonda et al. [3] generalize this approach with an algorithm that takes $O(2^d \cdot n)$ time where d is the number of nullable attributes and n is the size of the input, by materializing the right side of the predicate. Ahmed & Amor [2] propose a method that materializes the left side, but details are sparse. In Section 6, we present two new algorithms that handle arbitrary NOT IN predicates and guarantee linear runtime when only one nullable attribute is involved: one algorithm that materializes the left side and one that materializes the right side, allowing the appropriate choice depending on the query.

[1]Both PostgreSQL and MariaDB refer to this as a "subselect" operation.

3 BACKGROUND

In this section, we give a review of NULL semantics in SQL and formally define IN, quantified comparison, and EXISTS expressions.

3.1 NULL Semantics

SQL uses three-valued logic where the results of logical expressions can be TRUE, FALSE, or NULL. The NULL value is supposed to represent an unknown value, but its semantics can be unintuitive.

In Figure 2, we show the Three-valued logic tables for the comparison operators = and IS, as well as the logical operators AND and OR. The = operator returns NULL if one of its operands is NULL. The IS operator, which we use to represent IS NOT DISTINCT FROM expressions in SQL, checks whether both operands are exactly the same, and only returns TRUE or FALSE. The IS operator is useful as it often appears after decorrelation of subqueries in SQL and is equivalent to = for non-NULL values. Since it does not return NULL, it is easier to evaluate than =. The logical operators AND and OR are defined such that if the result can be determined from the non-NULL operands, the result is that determined value. For example, if one operand of AND is FALSE, the result is always FALSE, even if the other operand is NULL.

3.2 IN, SOME, ALL, and EXISTS

Let $\mathbf{x} = (x_1, \ldots, x_d)$ be a row value expression. The IN predicate

$$(x_1, \ldots, x_d) \texttt{ IN (SELECT } r_1, \ldots, r_d \texttt{ FROM } \mathcal{R})$$

is defined as a three-valued membership test of $\mathbf{x}$ in the relation $\mathcal{R} := \{r^{(1)}, \ldots, r^{(n)}\}$:

$$\texttt{IN}(\mathbf{x}, \mathcal{R}) := \bigvee_{i=1}^{n} \bigwedge_{j=1}^{d} \left(x_j = r_j^{(i)}\right)$$

If $\mathbf{x}$ has no NULL values and there is an exact match in $\mathcal{R}$, the result is TRUE. If there is no exact match, but there is a row $\mathbf{y} \in \mathcal{R}$ such that all x_j are equal to y_j or are NULL, the result is NULL. If there is not even a partial match, the result is FALSE. Note that not all databases support IN predicates with an arbitrary number of attributes, but only with a single attribute. Our approach in Section 6 will be particularly optimized for this case.

It is easy to compute whether the result of an IN predicate is TRUE for a given row $\mathbf{x}$. One can simply build a hash table of the rows in $\mathcal{R}$ and check whether $\mathbf{x}$ is contained in it. However, it is difficult to distinguish between NULL and FALSE results. Consequently, for a NOT IN expression, FALSE is easy to determine, but NULL and TRUE are hard to distinguish. Thus, in contrast to IN predicates, NOT IN remains hard to evaluate even when used within WHERE clauses.

Quantified comparison predicates are similar, but extend to different comparison operators. We define the SOME (ANY) predicate:

$$\mathbf{x} \circ \texttt{SOME(SELECT r FROM } \mathcal{R})$$

where $\circ$ is one of the comparison operators $\{=, <>, <, \leq, >, \geq\}$. The semantics of the SOME predicate are defined as follows:

$$\texttt{SOME}(\circ, \mathbf{x}, \mathcal{R}) := \bigvee_{i=1}^{n} \left(\mathbf{x} \circ \mathbf{r}^{(i)}\right)$$

=	a	b	NULL
a	TRUE	FALSE	NULL
b	FALSE	TRUE	NULL
NULL	NULL	NULL	NULL

IS	a	b	NULL
a	TRUE	FALSE	FALSE
b	FALSE	TRUE	FALSE
NULL	FALSE	FALSE	TRUE

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Figure 2: Three-valued logic tables for the SQL comparison operators = and IS, and the logical operators AND and OR. a and b are distinct non-NULL values.

Note that $\texttt{IN}(\mathbf{x}, \mathcal{R})$ is equivalent to $\texttt{SOME}(=, \mathbf{x}, \mathcal{R})$. However, its negation NOT $\texttt{IN}(\mathbf{x}, \mathcal{R})$ is not equivalent to $\texttt{SOME}(<>, \mathbf{x}, \mathcal{R})$, but actually equivalent to $\neg\texttt{SOME}(=, \mathbf{x}, \mathcal{R})$ or $\texttt{ALL}(<>, \mathbf{x}, \mathcal{R})$.

The EXISTS predicate for membership checking has subtly different semantics from IN predicates. The EXISTS predicate

$$\texttt{EXISTS}\,(\texttt{SELECT FROM}\ \mathcal{R}\ \texttt{WHERE}\ (r_1, \ldots, r_d) = (x_1, \ldots, x_d))$$

is defined as a two-valued membership test of $\mathbf{x}$ in the relation $\mathcal{R}$:

$$\texttt{EXISTS}(\mathbf{x}, \mathcal{R}) := \bigvee_{i=1}^{n} \bigwedge_{j=1}^{d} \texttt{COALESCE}\left(x_j = r_j^{(i)}, \texttt{FALSE}\right)$$

The result is TRUE if $\mathbf{x}$ has no NULL values and there is an exact match and FALSE otherwise. The EXISTS predicate is easier to evaluate than the IN predicate, because it does not require distinguishing between the aforementioned NULL and FALSE cases. Thus, if a database optimizer is able to recognize such a query, it can be executed efficiently, usually by using a semi join operator.

4 DECORRELATION & OPTIMIZATION

Neumann & Kemper [14] give a way to decorrelate arbitrary correlated subqueries. Neumann et al. [15] further extend this work to handle IN or EXISTS predicates in arbitrary expressions by introducing the mark join operator. In this section, we summarize the relevant parts of their work and give new definitions that are consistent with null semantics and three-valued logic.

A mark join between a relation R and a relation S is:

$$R \rtimes^{M}_{m:p} S := \left\{ \mathbf{r} \circ \left(m : \bigvee_{s \in S} p(\mathbf{r} \circ \mathbf{s}) \right) \middle| \mathbf{r} \in R \right\}$$

In other words, a mark join computes the disjunction of the predicate p on all rows of S for each row of R. This disjunction is then added as a new attribute m to the result.

Mark joins can be used to express decorrelated queries with IN and EXISTS predicates. For example, the query

```
SELECT *
FROM R
WHERE R.x IN (SELECT S.a FROM S WHERE R.y = S.b)
```

can be expressed using a mark join:

$$\sigma_m \left(R \rtimes^{M}_{m:R.x=S.a \wedge R.y \text{ IS } S.b} \sigma_{S.b \text{ IS NOT NULL}}(S) \right)$$

The IS predicate is a shorthand for IS NOT DISTINCT FROM in SQL (cf. Section 3.1). It checks equivalence and returns either TRUE or FALSE. Mark joins can also be used to express quantified comparison predicates similarly. The query:

```
SELECT *
FROM R
WHERE R.x<>SOME(SELECT S.a FROM S WHERE R.y = S.b)
```

can also be expressed using a mark join:

$$\sigma_m \left(R \rtimes^{M}_{m:R.x<>S.a \wedge R.y \text{ IS } S.b} \sigma_{S.b \text{ IS NOT NULL}}(S) \right)$$

After decorrelation, a mark join is often followed by a selection on the marker. Such plans can be transformed to semi and anti joins:

$$R \ltimes_p S \equiv \sigma_m \left(R \rtimes^{M}_{m:p} S \right)$$

$$R \triangleright_{\texttt{COALESCE}(p,\texttt{TRUE})} S \equiv \sigma_{\neg m} \left(R \rtimes^{M}_{m:p} S \right)$$

The semi join checks for each row of R whether the predicate p is TRUE for at least one row of S, while the anti join checks whether the predicate p is FALSE for all rows of S. The resulting semi join and anti join (if the COALESCE expression can be optimized away) can be evaluated efficiently for many predicates p.

While most IN and quantified comparison expressions result in mark joins with equality or basic inequality predicates, SQL actually allows for arbitrary predicates. We can express such predicates in the following general form:

```
SELECT (TRUE IN (SELECT p(R, S) FROM S)) AS m
FROM R
```

Here, $p(R, S)$ is an arbitrary predicate that can involve multiple attributes from both R and S. This query is equivalent to the mark join on predicate p:

$$R \rtimes^{M}_{m:p} S$$

5 COMPLEXITY OF EVALUATING IN

We would like to evaluate all possible IN predicates efficiently. Unfortunately, we show that the behavior mandated by the SQL standard is inherently hard to implement efficiently, given commonly made assumptions in complexity theory.

To show that IN predicates are hard, we reduce from the orthogonal vectors problem [10, 23]. The orthogonal vectors problem is defined as follows: Given a set of n binary vectors in $\{0, 1\}^d$, is there a pair of vectors that are orthogonal, i.e., their dot product is zero? This problem is conjectured to require $n^{2-o(1)}$ time to solve [1, 21]. This is a consequence of the strong exponential time hypothesis (SETH) [5, 8], which states that k-SAT with n variables and m clauses cannot be solved in $2^{n(1-\epsilon)}\text{poly}(m)$ time (where $\text{poly}(m)$ denotes a polynomial in m) for any $\epsilon > 0$ given large enough k. Put simply, if SAT requires time with an exponential base of 2, then orthogonal vectors require quadratic time, and thus IN predicates require quadratic time as well.

We reduce the orthogonal vectors problem to IN predicates by writing a SQL query that computes orthogonal vectors using IN, as shown in Figure 3. The SQL query illustrates the orthogonal vectors problem for 3-dimensional vectors, where $\mathcal{R}$ is the set of vectors,

```
WITH
R(v0,v1,v2) AS (VALUES (0,1,0),(1,1,0),(1,1,1)),
S(v0,v1,v2) AS
(SELECT NULLIF(1-v0,1),
        NULLIF(1-v1,1),
        NULLIF(1-v2,1)
        -- can be extended to more dimensions
 FROM R),
SELECT
(SELECT count(*)
 FROM R
 WHERE (v0,v1,v2) NOT IN (SELECT v0,v1,v2 FROM S))
<
(SELECT count(*) FROM R);
```

Figure 3: IN predicate query for computing the orthogonal vectors problem. The input is given in $\mathcal{R}$, and the output indicates whether there is a pair of orthogonal vectors.

though it can be extended to an arbitrary number of dimensions by increasing the number of attributes of $\mathcal{R}$ and extending the NULLIF expressions accordingly. Two vectors $\mathbf{x}, \mathbf{y} \in \mathcal{R}$ are orthogonal if and only if, for all indices i such that $y_i = 1$, we have $x_i = 0$. This is encoded in SQL three-valued logic as:

$$\bigwedge_i (x_i = \texttt{NULLIF}(1 - y_i, 1))$$

Note that equality in SQL evaluates to NULL if one of the operands is NULL (cf. Figure 2).

(1) If $y_i = 1$, then $\texttt{NULLIF}(1 - y_i, 1) \equiv 0$, so the equality reduces to $x_i = 0$.
(2) If $y_i = 0$, then $\texttt{NULLIF}(1 - y_i, 1) \equiv \texttt{NULL}$, so the equality reduces to $x_i = \texttt{NULL}$. The equality evaluates to NULL regardless of the value of x_i.

Thus, the conjunction over all indices i evaluates to FALSE if and only if $\mathbf{x}$ and $\mathbf{y}$ are *not* orthogonal and to TRUE or NULL otherwise. For a given vector $\mathbf{x}$, an IN expression gives the disjunction over all vectors $\mathbf{y} \in \mathcal{R}$ of the above expression. Thus, the IN expression evaluates to FALSE if and only if there is no vector $\mathbf{y} \in \mathcal{R}$ that is orthogonal to $\mathbf{x}$, and to TRUE or NULL otherwise. So, to determine whether there is a pair of orthogonal vectors in $\mathcal{R}$, we can count how many vectors $\mathbf{x} \in \mathcal{R}$ have no orthogonal vector in $\mathcal{R}$ using a NOT IN predicate, and check whether this number is less than n (the total number of vectors). The scanning of R, the computation of S from R, and the final aggregations can all be done in linear time. The bottleneck of the query is the evaluation of the NOT IN predicate. Thus, the problem of evaluating NOT IN predicates (or IN expressions) is at least as hard as the orthogonal vectors problem, which is conjectured to require quadratic time.

While our reduction is only in one direction, meaning that an efficient solution for the orthogonal vectors problem does not imply an efficient solution for IN predicates, progress on the orthogonal vectors problem might be of interest. The naive algorithm for the orthogonal vectors problem is to compute the dot product of all pairs of vectors, which takes $O(n^2 \cdot d)$ time, similar to the naive algorithm for IN predicates. An alternative approach, that can be mapped to IN predicates, is to enumerate all subsets of 0 indices in a vector and check whether the corresponding vector with 1s at that subset exists in the set, which takes $O(2^d \cdot n \cdot d)$ time. The orthogonal vectors problem also has $\tilde{O}(n + 2^d)$ algorithms [10] (where $\tilde{O}$ hides a log factor), though these exploit the binary nature of the vectors and may not extend to IN over arbitrary domains. Recently, Williams [22] gave an $\tilde{O}(n \cdot 1.35^d)$ algorithm for binary vectors, exponentially faster in d than the naive method.

6 APPROACH

In this section, we will present our approach to efficiently evaluate mark joins. We discussed mark joins in Section 4, where we showed that they can be used to express IN, EXISTS, and quantified comparison predicates. We have previously shown in Section 5 that the problem is hard in general, which means that we cannot hope to find a subquadratic algorithm for all cases. However, we can still find efficient algorithms for common cases.

The mark join has two sides, the side whose attributes are visible in the output (the produced side), and the side that is only used to determine the marker (the hidden side). Joins are often asymmetric [4], meaning that one side is often much smaller than the other side. Thus, for both performance and memory consumption reasons, we want to materialize only one side of the join, and pipeline the other side without materialization. We implement two variants of the mark join, the right mark join $\mathcal{L} \bowtie^M_p \mathcal{R}$, which materializes the hidden side, and the left mark join $\mathcal{L} \bowtie^M_p \mathcal{R}$, which materializes the produced side. The right mark join is somewhat more intuitive as the produced side is pipelined, and the marker is computed directly. The left mark join is more complex, as the produced side is materialized, and the marker must be computed in multiple passes. Both are required to cover all use cases efficiently, and our approach to both guarantees linear time complexity in the common case when there is only a single nullable attribute in the join predicate.

We assume that the predicate is of the form:

$$p(\mathbf{l}, \mathbf{r}) \equiv (\mathbf{l}.\mathbf{x} = \mathbf{r}.\mathbf{x}) \wedge (\mathbf{l}.\mathbf{y} \text{ IS } \mathbf{r}.\mathbf{y}) \wedge \text{residual}(\mathbf{l}, \mathbf{r})$$

where

(1) $\mathbf{x}$ and $\mathbf{y}$ are sets of attributes
(2) $\mathbf{l}$ is a tuple from the left side
(3) $\mathbf{r}$ is a tuple from the right side
(4) $=$ is the standard equality comparison operator. It returns NULL if either side is NULL (cf. Figure 2).
(5) IS is the SQL IS NOT DISTINCT FROM comparison operator. It checks for exact equivalence either returning TRUE or FALSE, never NULL (cf. Figure 2). We use IS to also represent $=$ predicates where both sides are not nullable.

We refer to the combination of $\mathbf{x}$ and $\mathbf{y}$ as the *keys* of the predicate. We refer to $\mathbf{y}$ as the *not-nullable* key attributes, as they cannot result in NULL predicates due to the use of the IS operator. Note that $a = b$ is equivalent to a IS b when both a and b are not nullable, thus they are treated the same way. We refer to $\mathbf{x}$ as the *nullable* key attributes, as they can result in NULL predicates. The residual predicate is the part of the predicate that is not covered by the keys, and needs to be evaluated separately. It may be an arbitrary boolean expression supported by SQL, but is most commonly inequality predicates as they are directly supported by quantified expressions. In Section 4, we describe how mark joins with arbitrary predicates can be expressed in SQL.

```
# Build
for l in L:
  htFull[keys(l)].append(l)
  if not isAnyNull(keys(l)):
    htProj[notNullable(l)].nonNull.append(l)
  else:
    htProj[notNullable(l)].null.append(l)
  # Initialize marker and right tuple storage (
      left mark join only)
  l.m = False
  htProj[notNullable(l)].rightTuple = None
```

Figure 4: Pseudocode for the build logic shared by both mark joins. Two hash tables are built, one on the full key and one on the not-nullable key.

Both right mark join and left mark join are designed to be as simple as possible while reducing to the optimal linear-time algorithms for the case when there is only a single nullable comparison ($|\mathbf{x}| = 1$). This is the special case that most often results from IN or quantified comparison predicates, as the predicate itself often compares a single attribute while the rest of the predicate results from pulling up correlated predicates from the subquery.

6.1 Shared Logic

Firstly, we define several helper functions and data structures that are used in both algorithms. The function `keys(t)` returns the entire key of tuple `t`, i.e., the combination of the nullable and not-nullable key attributes. The function `notNullable(t)` returns only the not-nullable key attributes of tuple `t`, i.e., the attributes involved in the IS comparisons. The function `isAnyNull(k)` returns TRUE if any attribute among those involved in = comparisons in the key `k` is NULL, and FALSE otherwise.

Both the right mark join and left mark join build hash tables on their left (build) side, which the query optimizer picks to be the smaller side of the join. We build one hash table `htFull` on the entire key, to check for perfect matches, i.e., tuples where the predicate evaluates to TRUE. Additionally, we build a second hash table `htProj` on the not-nullable key attributes, which is used to find potential partial matches, i.e., tuples where the predicate may evaluate to NULL. `htProj` contains the following information from the build side: (1) Tuples that have no NULL values in the key attributes; (2) Tuples that have NULL values in the key attributes. (3) Storage space for a single tuple from the right (probing) side (only for left mark join).

The build logic is shown in Figure 4. For every tuple in the build side, we first place it in `htFull`. Then, we check whether there are any NULL values in the nullable key attributes and place the tuple in the appropriate list in `htProj`.

6.2 Right Mark Join $\mathcal{L} \ltimes^M_p \mathcal{R}$

In the right mark join, the probing side is the produced side with the marker. The probing side must return TRUE if there is a perfect match, a tuple from the left side for which the predicate evaluates to TRUE. If there is no perfect match, there might still be a partial match, which is a tuple from the left side for which the predicate evaluates to NULL. The predicate can only evaluate to NULL if the attributes in $\mathbf{x}$ are either equivalent or one of them is NULL. The probing algorithm is shown in Figure 5. The right (probing) side checks `htFull` with its entire key. If it already finds a perfect or partial match (the match may end up being partial due to residual predicates), it returns. Otherwise, with its not-nullable key, it probes the tuples that share the same not-nullable key, as the predicate would return FALSE for the other tuples. If our key does not contain any nulls, it suffices to check the tuples in `htProj`.null, i.e., the tuples that contain a NULL value in the key, as the perfect matches have been checked via `htFull` already. If our key contains a NULL value, we must also check the corresponding tuples in `htProj`.nonNull. Among the tuples in `htProj` that match the partial not-nullable key, we check whether the predicate evaluates to NULL, in which case we return a NULL marker. If no match is found, we return FALSE.

```
# Probe
for r in R:
  m = FALSE
  # Check for perfect matches
  for l in htFull[keys(r)]:
    # Still have to check residual predicates
    m = m OR p(l, r)
    if m is TRUE:
      break

  if m is FALSE:
    # No matches yet, look for partial matches
    # At best we can return a NULL marker
    for l in htProj[notNullable(r)].null:
      if p(l, r) is NULL:
        m = NULL
        break
    # If we have NULLs, we might have partial
    #  matches with not NULLs
    if m is FALSE and isAnyNull(keys(r)):
      for l in htProj[notNullable(r)].nonNull:
        if p(l, r) is NULL:
          m = NULL
          break
  emit(r, m = m)
```

Figure 5: Pseudocode for the right mark join

If there is only a single nullable key attribute and there are no residual predicates, the algorithm needs constant time per tuple, and linear time overall. To show this, we argue that the iterations over `htProj`.nonNull and `htProj`.null will only iterate over a single tuple at most. If there is only a single key attribute which is nullable, `htProj`.null reduces to a single entry, which is either empty or contains a single NULL value. Thus, if the probing side is not NULL, the iteration over `htProj`.null will break immediately, as the list is either empty or has a single tuple where the predicate will evaluate to null. Otherwise, if the probing side is NULL, then the iteration over `htProj`.nonNull will break immediately in the same way.

6.3 Left Mark Join $\mathcal{L} \bowtie^M_p \mathcal{R}$

In the left mark join, the materialized build side is the produced side with the marker. Thus, we need to materialize the left side first, set up the markers correctly when iterating over the right side, and finally, iterate over the materialized tuples to produce the

```
# Update markers
for r in R:
  # Check for perfect matches, update markers
  for l in htFull[keys(r)]:
    # Check residual predicates
    l.m = l.m OR p(l, r)

  # Handle partial matches
  # Store up to 1 tuple per notNullable key
  # These are later checked with the build tuples
  # If limit is exceeded, do not store the tuple
  # Go and update the markers manually
  if notNullable(r) in htProj:
    if isAnyNull(keys(r)):
      # No tuples yet or equivalent values, reuse
      if htProj[notNullable(r)].rightTuple in (
          None, r):
        htProj[notNullable(r)].rightTuple = r
        continue
      # Tuple already exists, manually update
          markers
      for l in htProj[notNullable(r)].nonNull:
        l.m = l.m OR p(l, r)
    # Manually update markers for NULLs
    for l in htProj[notNullable(r)].null:
      l.m = l.m OR p(l, r)

# Produce result
for l in htFull.values():
  if l.m is FALSE:
    # Check for a partial match
    rightTuple = htProj[notNullable(l)].rightTuple
    if rightTuple is not None:
      l.m = p(l, rightTuple)
  emit(l, m = l.m)
```

Figure 6: Pseudocode for the left mark join

result. For every tuple in the right side, we could potentially visit all tuples on the left side, updating the markers in case there are partial or full matches. However, we can optimize this by materializing some tuples from the right side, and probing those materialized tuples from the left side, similar to how we did in the right mark join in Section 6.2. Still, we do not want to end up materializing all tuples from both sides. So we limit the number of tuples that we materialize from the right side to one per not-nullable key. For simplicity, this limit is always one, but it could be increased to a small constant if desired. If we exceed that limit, we iterate over the materialized tuples from the left side and update their markers.

The algorithm is shown in Figure 6. We first mark all perfect matches in `htFull` that share the entire key. Then we try to store this tuple in the second hash table `htProj`, if there is a tuple from the left side that has the same not-nullable key. If `htProj` has no tuple stored for this not-nullable key, we store the tuple from the right side. If `htProj` already has a tuple with the same values (`IS` comparison), we can skip storing this tuple. Otherwise, `htProj` already has a tuple with a different key, and we iterate over the matching tuples on the left side and update their markers by evaluating the predicate. Finally, we iterate over the materialized tuples from the left side. If the marker is still `FALSE`, we check the predicate against the tuple containing `NULL`s from `htProj.rightTuple` that shares the same not-nullable key and update the marker. We then produce the tuple with the final marker.

The storage requirement of this algorithm is linear in the size of the smaller side of the join, as we only materialize one side fully and store a constant number of tuples from the other side per left-side not-nullable key. Additionally, if there is only a single nullable key attribute and there are no residual predicates, the algorithm needs constant time per tuple. This is because `htProj.rightTuple` will either be empty or contain a single tuple, which consists of a single `NULL` value. This means that we would never need to iterate over `htProj.nonNull` as we are processing the right side. Also, in case we manually update markers if `isAnyNull(keys(r))` is `FALSE`, the iteration over `htProj`'s null list will only contain a single unique tuple with a `NULL` value. This means that we only need to mark at most one left-side tuple per right-side tuple if the left side contains no duplicates. Our implementation contains an additional optimization for the case when the left side may contain duplicates and there is only one nullable key attribute. In this case, if the right side does not contain any `NULL` values in the key, we set a flag in `htProj` to indicate that all left-side tuples with `NULL` in their key have been marked. Then, when finally iterating over the left side to produce the result, we check this flag in case the left-side tuple has a `NULL` value.

6.4 Analysis

The right mark join and the left mark join use memory linear in the size of the smaller side of the join, as they only materialize one side fully. The left mark join additionally stores at most a single tuple from the other side per left-side not-nullable key, which is also linear in the size of the smaller side. We have also shown that both the right mark join and left mark join algorithms have linear time complexity in the common case when there is only a single nullable attribute in the join predicate and no residual predicates.

Unfortunately, if there are multiple nullable attributes in the join predicate, both algorithms may need to iterate over multiple tuples in `htProj.nonNull` and `htProj.null`, resulting in potentially quadratic runtime. Consider the case when the predicate has two nullable key attributes and no not-nullable key attributes. Then, `htProj` will have only a single entry where `htProj.nonNull` and `htProj.null` combined contain the entire build side. Thus, for every tuple in the probing side, we would need to iterate over the entire build side, resulting in quadratic runtime. Thus, these algorithms are only guaranteed to perform well if there are not-nullable key attributes that partition the build side into sufficiently smaller groups, or if there is only a single nullable key attribute. In other cases, the algorithm due to Bellamkonda et al. [3] with $O(2^d \cdot n)$ time complexity where d is the number of nullable attributes and n is the size of the input may be more appropriate. However, this will only remain the case for a very small number of nullable attributes, as the exponential factor quickly dominates. Note that the memory requirement of that algorithm also scales exponentially with the number of nullable attributes. Thus, we find our approach to be a good practical compromise between complexity, performance, and memory consumption for the common cases.

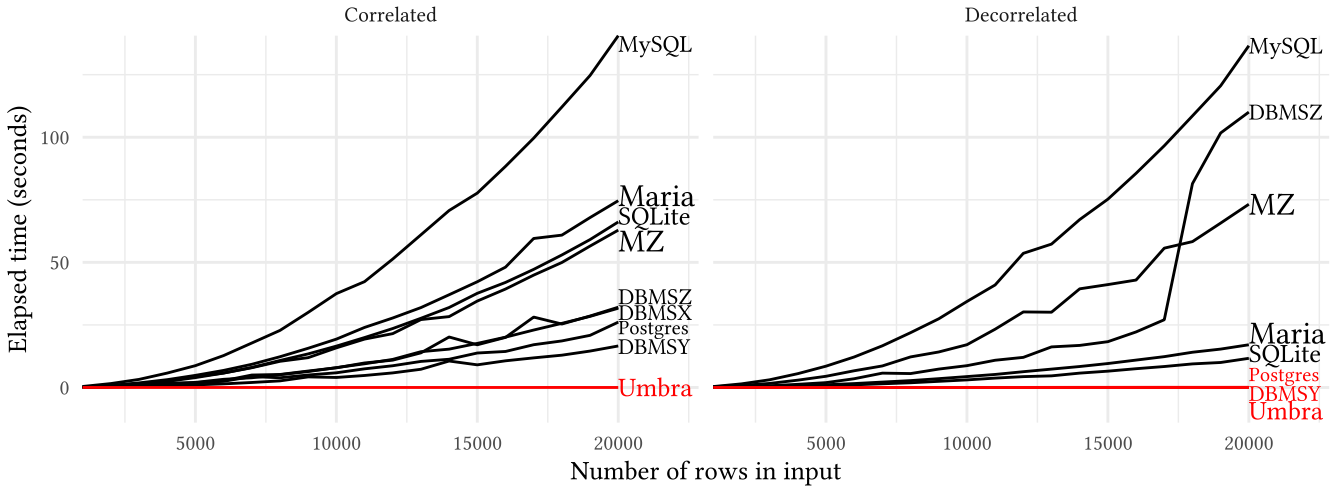

Figure 7: Runtime of `NOT IN` predicates for correlated vs. decorrelated queries with varying input sizes. Many systems exhibit quadratic runtime even in simple cases.

6.5 Optimizations

While the described algorithms are correct and exhibit linear time with a single nullable attribute, there are several optimizations that can be applied to further improve performance. For example, a list of nullable tuples does not need to be stored, if we know that no attributes from that input side are nullable. Additionally, if there is only a single nullable attribute (and no residuals but an arbitrary number of not-nullable attributes), we often do not need to store entire lists of tuples.

For the right mark join, `htProj.null` can be replaced by a flag indicating whether any tuple with `NULL`s has been seen. If so, the iteration over `htProj.null` can be replaced with a check if such a tuple exists, and if so, the marker could be set to `NULL`. The same can be done for `htProj.nonNull`. The list can be replaced by a single flag which is checked to set the marker to `NULL`.

For the left mark join, `htProj.nonNull` is not required and can be removed. `htProj.null` can be replaced by a flag indicating whether a right side tuple intended to mark all left side tuples with `NULL`s. In the final iteration over left tuples, this flag can be checked to set the marker to `NULL` if it is `FALSE`. Additionally, `htProj.rightTuple` can also be replaced by a flag, indicating whether a `NULL` value was seen. This is because there is only one possible distinct right-side tuple that can be placed there, and that is the tuple where the single nullable attribute is `NULL`.

In the left mark join, for each right-side tuple, we may need to iterate over multiple left-side tuples to update their markers to `NULL`. Whenever we set a marker to `NULL`, we can remove that tuple from its corresponding list. This way, we avoid iterating over the same tuple multiple times.

6.6 Multi-threading

To process large data sets, scalable multi-threaded execution is required. We can achieve this by utilizing multi-threaded scans on the input relations, using morsel-driven parallelism [11]. For parallel construction of `htFull`, we use unchained hash tables [4]. We model the construction of `htProj` as a parallel aggregation operation [11], where merging two entries involves appending the lists of tuples. In a right mark join, different probes can run concurrently without any interference. In a left mark join, multiple threads may try to update the markers of the same stored tuple concurrently, or try to store a tuple from the right side into `htProj` concurrently. For such operations, we use compare-and-swap instructions[2]. Markers have three states, so they fit into a single byte which supports compare-and-swap on most architectures. The `htProj` hash table is built within the build phase and does not receive any modifications afterwards, so it does not need to concurrently grow or shrink.

We maintain the lists of tuples in `htProj` as linked lists. To concurrently remove tuples from the lists in `htProj` in the left mark join (cf. Section 6.5), we optimistically update their links. To remove a tuple from the list, we read its next pointer, and then atomically store this value into the previous tuple's next pointer. If the next tuple has been concurrently removed, this removal may be lost. However, this is not a problem, as its marker has already been set to `NULL`. We will try removing it again in the next iteration.

7 EVALUATION

We integrated our approach into Umbra [13], our relational database system. We conduct our measurements on the database systems Materialize v0.151.0 (MZ), MySQL 9.4.0, PostgreSQL 17, MariaDB 11.4, SQLite 3.46.1, Umbra 25.12, and three commercial systems DBMS{X,Y,Z} on a Ryzen 9 5950X with 64GB RAM, 16 cores, and 32 threads. In these initial measurements, the databases are run within Docker containers restricted to 1 core each. We have omitted measurements for ClickHouse 25.8, Hyper 9.1.0, DuckDB 1.3.0, and Firebolt Core 4.23.5 as they produce incorrect results. Our goal is not to show performance differences between the systems, but rather how the algorithms behave on different inputs.

[2]When updating values in compare-and-swap loops, we make sure to read the value and check it, before actually running the compare-and-swap operation, in the test and test-and-set style. This is to avoid unnecessary contention on the values in question.

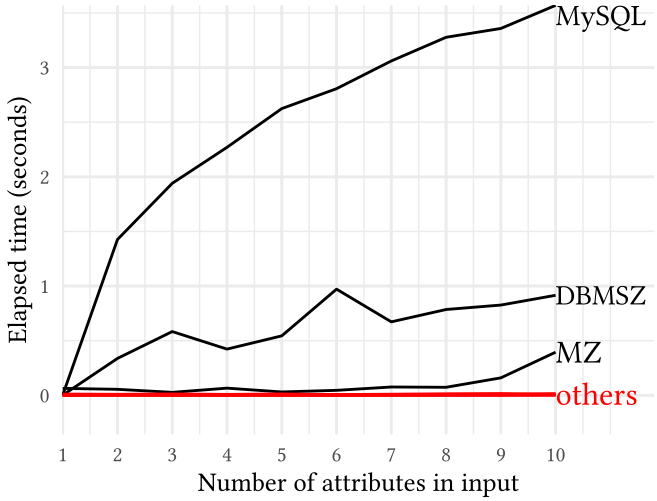

Figure 8: Runtime of the orthogonal vectors query on 2000 vectors with varying number of attributes.

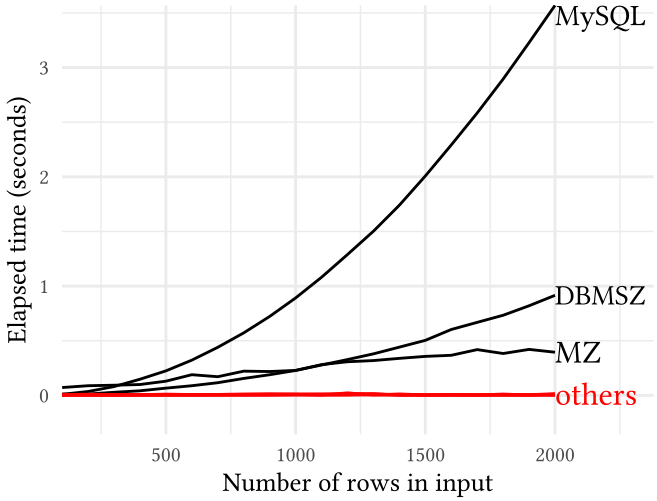

Figure 9: Runtime of the orthogonal vectors query on 10-dimensional vectors with varying numbers of vectors.

We want to evaluate how different database systems scale with increasing input sizes where linear time execution is possible. We evaluate a correlated query `SELECT count(*) FROM R WHERE R.a NOT IN (SELECT S.a FROM S WHERE R.b = S.b)` and its decorrelated version using `NOT IN` with two attributes, where `a` is nullable and `b` is not. The tables are filled with constant $b = 1$ and a values ranging from 0 to 20000. We run the queries, add a single row $(NULL, 1)$ to `S`, run the queries again, and take the maximum elapsed time of both. We show the results in Figure 7.

All tested database systems except for Umbra fail to run the correlated queries in linear time. We believe this is because their algebra lacks a mark join operator, which makes many of the decorrelation transformations simple and the resulting plan efficient. PostgreSQL, DBMSY, and Umbra can run the decorrelated queries in linear time, while the rest of the systems exhibit quadratic runtime. The linear time systems are orders of magnitude faster than the quadratic-time systems in the final run. Of course, if we increase the number of nullable attributes and make the input arbitrarily complex, all systems must resort to quadratic runtime algorithms.

We also find that, when we replace `NOT IN` expressions with `IN` expressions, all systems exhibit linear runtime, except SQLite and MariaDB, which cannot automatically decorrelate the correlated version of the query. While it is understandable that many systems have not yet implemented the more complex optimizations needed for `NOT IN`, the resulting performance gap remains a surprising inconsistency for users.

We also want to evaluate how well systems handle a more complex `NOT IN` query with multiple nullable attributes. For this, we ran the orthogonal vectors query from Figure 3 on various systems on vectors with randomly generated values. Note that this does not represent a real database workload, but rather a synthetic benchmark to illustrate performance characteristics. Although the problem is theoretically quadratic, randomly generated vectors allow for implementations to find partial matches quickly, running much faster than quadratic time. To show the impact of the number of nullable attributes on performance, we fixed the number of rows to 2000 and varied the number of attributes in Figure 8. While the increase in nullable attributes increases the difficulty of the general problem, it remains likely to find partial matches. While MySQL, DBMSZ, and Materialize could not fully exploit the properties of the data, the other databases, MariaDB, Umbra, SQLite, DBMSY, and PostgreSQL, show close to linear runtime, indicating that they effectively exploit that partial matches are likely. DBMSX is not able to execute these queries as it does not support `IN` predicates with multiple attributes. Our approach integrated in Umbra is among the fastest approaches.

In Figure 9, we fix the number of nullable attributes to 10 and vary the number of input rows. We see a similar pattern here as well. MySQL, DBMSZ, and Materialize could not fully exploit the niceness of the data, and MySQL clearly shows quadratic runtime. The rest of the systems exhibit close to linear runtime. What is surprising is that many systems are able to perform better on this very unrealistic query than on the simpler two-attribute case. This shows that while nested loop execution with early-breaking of loops can give good runtime if the input data is nicely structured, it also is not reliable.

We additionally want to compare the two variants of mark join we have presented in Section 6. For this, we run both variants on Umbra with the mark join query `SELECT (L.a, L.b) IN (SELECT a, b FROM R) FROM L`, where the attributes a are nullable. For our measurements, we vary the sizes of the produced and hidden sides (cf. Section 6). The tables are filled with constant $b = 1$ and a values ranging from 0 to 10^6 and an additional $(NULL, 1)$ row. The queries are run with all available cores on the machine. We repeat the queries 10 times and report, for each variant, the median execution time plus the compilation time. The results are shown in Figure 10.

We find that the left mark join variant is faster when the produced side is much smaller than the hidden side, and vice versa. This confirms our expectation that limiting the number of materialized tuples is beneficial. We also find that if the sizes are similar, the right mark join is often faster as the algorithm is simpler and requires fewer atomics. We fitted a linear boundary (dashed line in Figure 10)

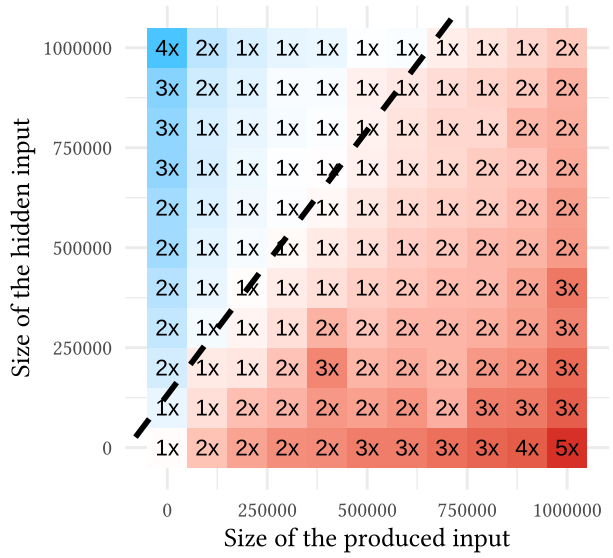

Figure 10: Comparison of left and right mark join variants on a mark join query with a single nullable attribute on both sides. We vary the sizes of the produced and hidden sides and show the fastest variant and the runtime ratio for each configuration. The dashed line represents the best-fitting linear boundary along which the two strategies perform equally. The blue area on the top left indicates where the left mark join is faster, and the red area on the bottom right indicates where the right mark join is faster.

where both variants perform equally well. This boundary can be used by a query optimizer to choose the best variant for a given query. We find that the slope of the boundary is approximately 1.3, indicating that the right mark join is preferable even when the hidden side is 30% larger than the produced side. In absolute numbers, the top right cell of the grid where both input sides contain 10^6 tuples takes 9ms with the right mark join (15ms with the left mark join), around 5ns (7ns) per tuple.

8 CONCLUSION

While we have shown that `IN` predicates are theoretically hard to evaluate efficiently if there are an arbitrary number of nullable attributes, we have also presented a practical algorithm that can handle many common cases efficiently, namely in linear time, if there is only a single nullable attribute. This is enough for many database systems, as they do not support more than one nullable attribute in `IN` predicates. Our conclusions are twofold. Firstly, the SQL standard's mandated behavior for `IN` predicates and quantified comparison predicates is unreasonably hard to understand, use, and implement efficiently. Reporting `FALSE` instead of `NULL` would make these predicates both intuitive and more efficient, by avoiding the complexities of three-valued logic. Secondly, although the mandated behavior is hard, we can still implement efficient algorithms for computing these predicates. Efficient implementations do not just speed up existing queries, but alleviate a large burden from users, who struggle with the idiosyncrasies of both the SQL standard and database implementations.

REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. Tight Hardness Results for LCS and Other Sequence Similarity Measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, Venkatesan Guruswami (Ed.). IEEE Computer Society, 59–78. doi:10.1109/FOCS.2015.14

[2] Rafi Ahmed and Angela Amor. 2010. Null aware anti-join. https://patents.google.com/patent/US7676450B2/en

[3] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zaït, and Chun Chieh Lin. 2009. Enhanced Subquery Optimizations in Oracle. *Proc. VLDB Endow.* 2, 2 (2009), 1366–1377. doi:10.14778/1687553.1687563

[4] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 4:1–4:9. doi:10.1145/3662010.3663442

[5] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. 2009. The Complexity of Satisfiability of Small Depth Circuits. In *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5917)*, Jianer Chen and Fedor V. Fomin (Eds.). Springer, 75–85. doi:10.1007/978-3-642-11269-0_6

[6] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. 2007. Execution strategies for SQL subqueries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 993–1004. doi:10.1145/1247480.1247598

[7] GeeksforGeeks Contributors. 2024. *IN vs EXISTS in SQL.* https://www.geeksforgeeks.org/sql/in-vs-exists-in-sql/ Accessed: 2025-07-15.

[8] Russell Impagliazzo and Ramamohan Paturi. 2001. On the Complexity of k-SAT. *J. Comput. Syst. Sci.* 62, 2 (2001), 367–375. doi:10.1006/JCSS.2000.1727

[9] Haroon Javed. 2024. *Understanding the Distinction Between EXISTS and IN in SQL.* https://www.baeldung.com/sql/in-vs-exists Accessed: 2025-07-15.

[10] Daniel M. Kane and Richard Ryan Williams. 2019. The Orthogonal Vectors Conjecture for Branching Programs and Formulas. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA (LIPIcs, Vol. 124)*, Avrim Blum (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 48:1–48:15. doi:10.4230/LIPICS.ITCS.2019.48

[11] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. doi:10.1145/2588555.2610507

[12] Leonid Libkin and Liat Peterfreund. 2023. SQL Nulls and Two-Valued Logic. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Floris Geerts, Hung Q. Ngo, and Stavros Sintos (Eds.). ACM, 11–20. doi:10.1145/3584372.3588661

[13] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[14] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings (LNI, Vol. P-241)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.). GI, 383–402. https://dl.gi.de/handle/20.500.12116/2418

[15] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The Complete Story of Joins (in HyPer). In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings (LNI, Vol. P-265)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.). GI, 31–50. https://dl.gi.de/handle/20.500.12116/657

[16] Oracle Corporation. 2008. *Use of* `EXISTS` *versus* `IN` *for Subqueries.* Oracle Corporation, Redwood Shores, CA. https://docs.oracle.com/cd/B19306_01/server.102/b14211/sql_1016.htm#i36215 Section 11.5.3.4 of *Oracle Database Performance Tuning Guide*, 10g Release 2 (10.2), document B14211-03.

[17] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (2025), 4144–4157.

[18] Gail Shaw. 2009. *EXISTS vs IN.* https://www.sqlinthewild.co.za/index.php/2009/08/17/exists-vs-in/ Accessed: 2025-07-15.

[19] Gail Shaw. 2010. *NOT EXISTS vs NOT IN.* https://www.sqlinthewild.co.za/index.php/2010/02/18/not-exists-vs-not-in/ Accessed: 2025-07-15.

[20] Etienne Toussaint, Paolo Guagliardo, Leonid Libkin, and Juan Sequeda. 2022. Troubles with Nulls, Views from the Users. *Proc. VLDB Endow.* 15, 11 (2022), 2613–2625. doi:10.14778/3551793.3551818

[21] Ryan Williams. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.* 348, 2-3 (2005), 357–365. doi:10.1016/J.TCS.2005.09.023

[22] Ryan Williams. 2024. The Orthogonal Vectors Conjecture and Non-Uniform Circuit Lower Bounds. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*. IEEE, 1372–1387. doi:10.1109/FOCS61266.2024.00088

[23] Ryan Williams and Huacheng Yu. 2014. Finding orthogonal vectors in discrete structures. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, Chandra Chekuri (Ed.). SIAM, 1867–1877. doi:10.1137/1.9781611973402.135