Window Function Expression: Let the Self-join Enter

Radim Bača
VSB - Technical Universitty of Ostrava
Ostrava, Czech Republic
radim.baca@vsb.cz

ABSTRACT

Window function expressions (WFEs) became part of the SQL:2003 standard, and since then, they have often been implemented in database systems (DBS). They are especially essential to OLAP DBSs, and people use them daily. Even though WFEs are a heavily used part of the SQL language, the amount of research done on their optimization in the last two decades is not significant.

WFE does not extend the expressive power of the SQL language, but it makes writing SQL queries easier and more transparent. DBSs always compile SQL queries with WFE using a sequence of partition-sort-compute operators, which we call a linear strategy. Plans resulting from the linear strategy are robust and, in many cases, efficient.

This article introduces an alternative strategy using a self-join, which is not considered in the current DBSs. We call it the self-join strategy, and it is based on an SQL query transformation where the result query uses a self-join query plan to compute WFE. One output of this work is a tool that can automatically perform such SQL query transformations.

We created a microbenchmark showing that the self-join strategy is more effective than the linear strategy in many cases. We also performed a cost-based experiment to evaluate the query optimizers' ability to select an appropriate strategy. The article's main aim is to show that usage of the self-join strategy for queries with WFE is beneficial if selected in a cost-based manner.

PVLDB Reference Format:
Radim Bača. Window Function Expression: Let the Self-join Enter. PVLDB, 17(9): 2162 - 2174, 2024.
doi:10.14778/3665844.3665848

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/RadimBaca/SQL_window_function_rewrite.

1 INTRODUCTION

Today, we find dozens of DBSs implementing window function expressions (WFEs). WFEs are especially popular among OLAP DBSs where they are fully implemented and supported; however, some other classical parts of SQL are not. For example, Impala [4] and Hive [16] do not implement subqueries behind SELECT, but the WFEs are implemented. Similarly, MySQL 8.0.0 [5] and DuckDB [3] implement WFEs; however, fetching first N records with ties is not supported.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 9 ISSN 2150-8097.
doi:10.14778/3665844.3665848

A growing interest in the WFE is also observed on websites like StackOverflow. To gain some insight into this trend, we analyzed SQL queries posted on StackOverflow [1] and counted the percentage of queries with a WFE for each year. Figure 1 shows the results, and we can observe that the percentage of such queries has steadily increased since 2008.

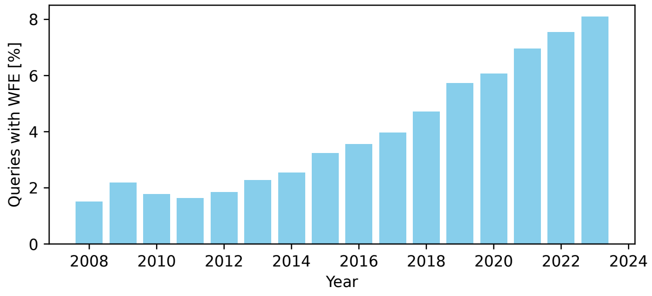


Figure 1: Percentage of SQL queries on StackOverflow that contain a WFE for each year.

Even though WFEs are popular, the amount of research done on their optimization in the last two decades is not significant. Existing papers [2, 11, 23, 30, 32] never treat the WFE as a part of a bigger query but focus mainly on algorithms that enable efficient computation of the WFE. However, our research shows that the surrounding query context in which a WFE is used can be significant. We propose transformations of a query with WFE whose convenience depends on other parts of the query rather than focusing on the efficiency of computing the WFE itself.

We see a motivational example in Listing 1. Let us have a large table of temperature measurements from different places. Our query retrieves the measurements for each place within the last day and includes an additional column showing the temperature deviation from the average place temperature.

A typical query plan for such a query [23] is a sequence of the following operators: partition, sort, compute, and filter. Note that filtering is performed as the last operation. We call this approach linear strategy. It is impossible to perform the predicate pushdown [24] in such a query plan, which has two main negative effects if the selectivity of the outer query predicate is high: 1) DBS sorts all the data and computes deviation from the mean for a large number of records that are of no interest, 2) The DBS fails to use a suitable index on the `mtime` attribute.

The issue with the linear strategy is that it fails to consider the larger context of the SQL query where the WFE is located. It is not

[1]https://github.com/RadimBaca/SO_sqlselect_extraction

```
SELECT *
FROM (
    SELECT temperature -
        AVG(temperature) over (
            PARTITION BY placeid
        ),
        placeid,
        mtime
    FROM measurements
)
WHERE mtime > localtimestamp - INTERVAL '1 day'
```

Listing 1: Motivational query with WFE

difficult to see that we can transform this SQL query into a query with a correlated self-join (see Listing 2). If the predicate condition of the outer query is highly selective, the query plan based on self-join may outperform the query plan based on the linear strategy. In this transformation example, we must be careful with NULL values since the rewrite is equivalent only if `placeid` cannot be NULL.

```
SELECT m1.temperature -
    (
        SELECT AVG(temperature)
        FROM measurements m2
        WHERE m2.placeid = m1.placeid
    ),
    m1.placeid,
    m1.mtime
FROM measurements m1
WHERE m1.mtime > localtimestamp - INTERVAL '1 day'
```

Listing 2: Self-join query corresponding to the motivational query

It is worth noting that modern DBSs do not use the self-join strategy for queries with WFEs, even when there is potential for significant performance improvements. To support this claim, we tested PostgreSQL 14.2, Oracle 19c, SQL Server 2016, MySQL 8.0, FireBird 3.0, SQLite 3, H2 1.4, and Hyper 0.0.18161 using a non-trivial number of queries.

The contributions of this article are the following:

- We describe several transformations of a query with WFEs into a self-join query. We introduce an SQL rewriting tool capable of doing these transformations automatically.
- Introduction of two microbenchmarks that help us better understand the settings under which the self-join strategy is advantageous.
- We also perform a cost-based analysis of two query optimizers with respect to both strategies.

Section 2 describes the basic syntax and semantics of WFEs. Section 3 introduces several transformations of queries with the WFE. Section 4 contains detailed experiments to help us better understand situations where the self-join strategy is advantageous. Section 5 deals with related work in the field.

2 WINDOW FUNCTION EXPRESSIONS

WFEs allow us to formulate complex queries that range from top-k, time series analysis [17], sliding window calculations, gaps-and-islands [22], and many more. It is a powerful tool that simplifies the writing of such queries and allows the optimizer to use stable and efficient query plans that would otherwise be difficult to create.

In this section, we briefly describe their basic syntax and semantics. The WFE defines a new value in the SELECT result and is only allowed in the SELECT or ORDER BY clauses. The value of the WFE for a particular row r is determined by:

- Rowset definition - We define a subset of rows (possibly ordered) that are relevant to r:
  - `PARTITION BY c` - defines a partitioning of rows according to attributes `c`. Therefore, r's partition contains rows with the same `c` value. Optional.
  - `ORDER BY o` - defines ordering of rows in the partition according to attributes `o`. Mandatory for some WFEs (rank and analytical).
  - `RANGE/ROWS` - defines a frame within r's partition. See Section 2.2 for more details. Optional.
- Window function - Defines a function applied on a specified rowset relevant to r.

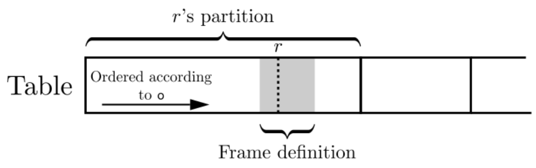


Figure 2: Rowset definition

We can think of a rowset definition as a function that returns a subset of rows for each row, which may also have a defined ordering. Figure 2 visualizes this concept, where the grey area is a rowset for the r row.

2.1 Window Functions

We can recognize several basic types of window functions. The basic division can be based on whether or not a frame is considered. Let us start with functions where the frame is not considered:

- Rank - The value of r's WFE equals the order within the rowset. There are several variations depending on duplicate handling:
  - `RANK()` - Rank of r with gaps and duplicates
  - `DENSE_RANK()` - Rank of r without gaps and with duplicates
  - `ROW_NUMBER()` - Order of the r without duplicates
- Analytical functions:
  - `NTILE(N)` - N-tile is assigned to row r.
  - `PERCENT_RANK()` - Percentile is assigned to row r.
  - `LEAD(expr, offset, default)` - Evaluate *expr* on a row preceding r
  - `LAG(expr, offset, default)` - Evaluate *expr* on a row following r

There are also window functions that consider the frame:

- Aggregate - The aggregate function (MIN, MAX, AVG, SUM, COUNT) is computed on a frame.
- Analytical functions:
  - FIRST_VALUE(expr) - Evaluate *expr* on the first row of r's rowset.
  - LAST_VALUE(expr) - Evaluate *expr* on the last row of r's rowset.

EXAMPLE 1. *Let's show the differences between the individual rank window functions. SQL query in Listing 3 computes the rank functions on a* `measurements` *table with a* `temp` *column, and Table 1 shows the result for sample data. Obviously, there is a difference between the rank functions, especially in how they handle duplicates.*

```
SELECT temp,
  RANK() over (ORDER BY temp) rank,
  DENSE_RANK() over (ORDER BY temp) dense,
  ROW_NUMBER() over (ORDER BY temp) rn,
FROM measurements
```

Listing 3: Rank window functions

Table 1: The result of Listing 3 query

temp	*rank*	*dense*	*rn*
1	1	1	1
1	1	1	2
2	3	2	3
null	4	3	4

With this example, we can depict one more aspect that needs to be considered in our paper. The ordering of records when they contain a NULL value. PostgreSQL, Hyper, and Oracle sort the values as we see them in the Table 1. However, some DBSs put NULL values at the beginning of the sorted set, which may affect the final result in our examples. Therefore, we have to be careful when proposing rewrites if attributes may contain NULL values.

The occurrence of the most used window functions in StackOverflow SQL queries can be observed in Figure 3. Rank and aggregate window functions represent the majority of window functions used in those SQL queries (i.e., red and blue bar charts). That is why we focus on these window functions in our article.

Some window functions mentioned in this section are not in Figure 3 because the number of occurrences was close to zero.

2.2 Frame Definition

The frame defines a subset of rows in r's partition. The r's partition is a sequence of rows, where the ordering is defined by the `ORDER BY o` clause. The order within the sequence will be denoted as i, and the value of r's `o` attribute will be denoted as vr. A frame subset may be defined in two different ways:

- `ROWS BETWEEN` K `AND` L - we select the rows where the ordering of the row is in the interval $(i - K, i + L)$
- `RANGE BETWEEN` M `AND` N - we select the rows where value of `o` attribute is in the interval $(vr - M, vr + N)$

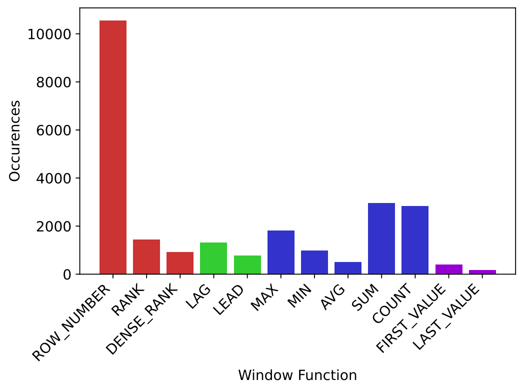


Figure 3: Occurence of different window functions in StackOverflow SQL queries.

The K, L, M, N interval values may be positive and negative, and their sign is specified using `FOLLOWING` or `PRECEDING` keyword, which is mandatory. The interval values must be a constant value, `UNBOUNDED`, or `CURRENT ROW`. However, the frame definition is optional, and the implicit definition `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` is used in the case that the `ORDER BY o` is specified in the WFE.

2.3 Normalized Query with WFEs

WFE often appear in uncorrelated subqueries. In other words, the result of a subquery is not dependent on the outer query and can be computed independently (see Listing 1). Due to this fact, we will only consider this situation in our definitions and examples. However, it is also possible to consider rewriting dependent subqueries with WFE.

Definition 2. Normalized Query with WFE (NQWF) Let us have a SELECT statement `T` with a WFE `W` and an ordering expression `ORDER BY`, where α is any SELECT statement or table definition.

```
T := SELECT iattr, W
     FROM α
     ORDER BY expr
```

Let $\beta_{\texttt{search(T.iattr,T.W)}}$(`T`) be a SELECT statement that uses `T` as an input and may perform some `search(T.iattr,T.W)` selection operation. The `search(T.iattr,T.W)` is a predicate that uses values returned by `T`.

Then we call `T` a *normalized query with WFE* (NQWF) and $\beta_{\texttt{search(T.iattr,T.W)}}$(`T`) a *query with nested NQWF.*

It is not difficult to see that if a query with WFE is in an independent (sub)query without `LIMIT` clause (or without `TOP/FETCH NEXT` clause depending on DBS), then it can be rewritten into NQWF. We show an example of such rewriting in Example 3. The main aim of the NQWF definition is to show that we can isolate the WFE computation from other parts of the query during a transformation. It can be clearly seen in Figure 4, which shows an abstract syntax tree corresponding to a query with nested NQWF.

The NQWF can be anywhere in the query. In fact, in this paper, we are particularly interested in situations where NQWF is a subquery in a larger query β with a highly selective predicate (e.g., see Example 1).

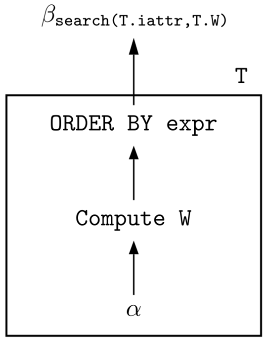


Figure 4: Abstract syntax tree for a query with nested NQWF.

Example 3. *In Listing 4, we see an example of an SQL query with a WFE. In Listing 5, we may see an equivalent NQWF where the α is a subquery behind FROM.*

```
SELECT MIN(m.temp * m.chillfactor) OVER (
      PARTITION BY pl.placeid
   ) min_temp,
   pl.placeid,
   m.mtime
FROM measurements m
JOIN place pl ON m.placeid = pl.placeid
JOIN person pe ON pl.placeid = pe.placeid
WHERE pl.state = 'NY' AND
   pe.fname = 'JOHN'
ORDER BY pl.placeid, m.mtime
```

Listing 4: Lowest temperature per placeid

```
SELECT MIN(alpha.temp) OVER (
      PARTITION BY alpha.placeid
   ) min_temp,
   alpha.placeid,
   alpha.mtime
FROM (
   SELECT m.temp * m.chillfactor temp,
      pl.placeid,
      m.mtime
   FROM measurements m
   JOIN place pl ON m.placeid = pl.placeid
   JOIN person pe ON pl.placeid = pe.placeid
   WHERE pl.state = 'NY' AND
      pe.fname = 'JOHN'
) alpha
ORDER BY alpha.placeid, alpha.mtime
```

Listing 5: NQWF corresponding to the Listing 4 query

In this work, we recognize two different strategies that handle queries with nested NQWF:

(1) Linear - a straightforward partition-sort-compute query plan always used in modern DBSs. See the introduction for a list of DBSs we tested to support this claim.
(2) Self-join - we transform the query into a self-join query. This transformation is the main topic of the following Section 3.

3 SELF-JOIN STRATEGY

This section presents transformations of a query with nested NQWF into a self-join query. In particular, we propose the relational algebra logical tree after each transformation, and we also mention how a rewritten self-join query should look to achieve the logical tree. We describe in more detail a tool that automatically transforms the SQL query for the desired self-join query in Section 4.1.

There are two reasons why we present a logical tree after transformation instead of SQL: (1) the logical tree is much more concise and clear, and (2) we believe that the WFE transformation should be integrated into a cost-based query optimizer at the query algebra level as it is typical for other transformation rules [1, 12, 13, 26, 27, 29]. Therefore, we use the terms "logical tree" and "query" interchangeably in the following text.

Of course, given the declarative nature of SQL, one cannot be sure that the compiler will start the optimization with the logical tree under consideration for a given SQL query. However, the proposed SQL transformations are so fundamental that the DBS always produces a query plan corresponding to the query's logical tree in testing. It was verified manually by checking the query plans on the eight DBS mentioned in the introduction for many queries. This claim supports many documentation and blog posts [7, 31, 33] suggesting rewriting the SQL query with a WFE into a self-join query will lead to significantly different query plans.

3.1 Aggregate Window Functions

Let us consider a query with nested NQWF, aggregate window function and without the explicit frame definition:

```
AGG(A) OVER (PARTITION BY B ORDER BY C) w
```

More precisely, it is a WFE with the implicit frame definition. Figure 5 shows a logical tree named LateralAgg using the self-join strategy to transform the query.

Please note that the logical tree contains d-join operator [24] where d stands for dependent, reflecting that the right branch is correlated to the left branch. This operator corresponds to the so-called `JOIN LATERAL` construct, which imposes a given logical tree on DBS. Since we want to keep every tuple from α_1, we use `LEFT JOIN LATERAL` in the corresponding SQL query.

Whereas the σ' corresponds to the `PARTITON BY` clause of WFE, the σ'' corresponds to the `ORDER BY` and implicit frame of WFE. Both `PARTITON BY` and `ORDER BY` clauses are optional in the WFE, so if any is missing in the WFE, we need to remove the corresponding σ operator from the LateralAgg. The top-most operator in the right branch is the $\Gamma_{AGG(A)}$, which computes the aggregate function `AGG(A)` and assigns `w` alias to the column. Grouping is unnecessary in Γ since σ' and σ'' already select a set for aggregation.

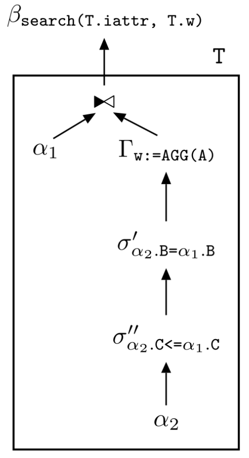

$\beta_{\texttt{search(T.iattr, T.w)}}$
T
$\bowtie$
α_1
$\Gamma_{\texttt{w:=AGG(A)}}$
$\sigma'_{\alpha_2.\texttt{B}=\alpha_1.\texttt{B}}$
$\sigma''_{\alpha_2.\texttt{C}<=\alpha_1.\texttt{C}}$
α_2

Figure 5: Logical tree named LateralAgg using the self-join strategy.

Just as a reminder, there are two types of frame definition (i.e., RANGE and ROWS) when considering an explicit framing definition. Accommodating RANGE in the self-join strategy is quite simple since it just means adding another σ operator with corresponding boundary specification into the LateralAgg logical tree. The situation is more complicated with ROWS. However, the self-join strategy is still possible using the appropriate TOP/LIMIT N operator and, in some cases, UNION operator. Our SQL rewriting tool implements rewriting support for such a feature (see Section 4.1).

The LateralAgg may look expensive at first. Remember, however, that there may be predicates in the β expression that propagate to α_1. One example of such predicate is the `mtime > localtimestamp - INTERVAL '1 day'` that we can see in Listing 1. If the predicate is highly selective, the LateralAgg may significantly outperform the linear strategy since it can lead to a nested-loop join query plan with a few iterations.

3.2 Rank Window Functions

3.2.1 *General Case.*

Let us consider a query with nested NQWF and rank window function:

`RANK() OVER (PARTITION BY B ORDER BY C) rn`

In such case, we can use the LateralAgg logical tree where AGG is COUNT(*) with a slight modification of σ'' having a condition α_2.C < α_1.C. The DENSE_RANK can be expressed using COUNT(DISTINCT C). The only problem is the ROW_NUMBER window function.

Let a unique column combination (UCC) be a set of attributes in a relation (it can also be a relation created as an intermediate result) where there are no two records with the same values in all these attributes. The critical aspect that influences the ability to use LateralAgg in the case of ROW_NUMBER window function is whether the combination of B and C is UCC or not:

(1) The combination of B and C is UCC - it is not difficult to see that in this case, the ROW_NUMBER is equivalent to RANK and DENSE_RANK.
(2) The combination of B and C is *not* UCC - then the result of ROW_NUMBER is non-deterministic. In such case, the self-join strategy is possible only if we add some unique attribute into C set that specifies the ordering (e.g., physical row location).

This influence of UCC knowledge is another example [18, 21] of a situation where the knowledge of advanced metadata can be helpful during query optimization.

3.2.2 *Greatest per Group Case.*

Now let us consider a special case of rank WFE: a query with nested NQWF $\beta_{\texttt{search(T.rn)}}$(T) where `search(T.rn)` is a predicate that compares T.rn to a positive constant. In our article, we will discuss three basic types of predicates:

(1) `T.rn = 1`
(2) `T.rn = N`
(3) `T.rn < N`

This query is commonly encountered when we need to find the greatest per group. Many occurrences of ROW_NUMBER in Figure 3 fall into this query type.

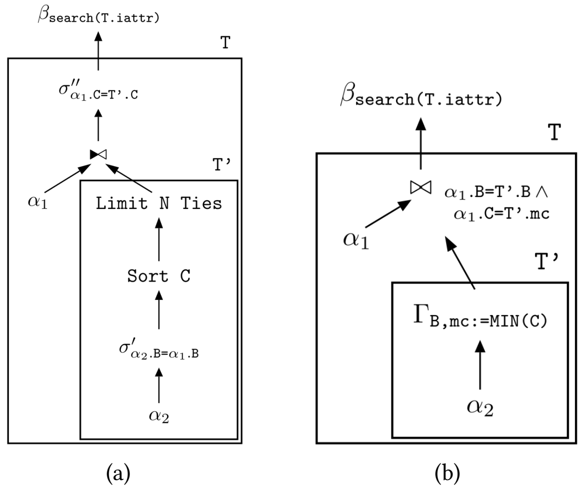

$\beta_{\texttt{search(T.iattr)}}$
T
$\sigma''_{\alpha_1.\texttt{C=T'.C}}$
$\bowtie$
T'
α_1
Limit N Ties
Sort C
$\sigma'_{\alpha_2.\texttt{B}=\alpha_1.\texttt{B}}$
α_2
(a)
$\beta_{\texttt{search(T.iattr)}}$
T
$\bowtie$
α_1.B=T'.B $\wedge$ α_1.C=T'.mc
α_1
T'
$\Gamma_{\texttt{B,mc:=MIN(C)}}$
α_2
(b)

Figure 6: (a) LateralLimitTies (b) JoinMin

In Figure 6, we introduce two logical trees that can be used for the greatest per group case. While the LateralLimitTies can be used for all three predicate types, the JoinMin logical tree is applicable if we have the `T.rn = 1` predicate and if C is a single not-nullable attribute.

The T' in the JoinMin uses $\Gamma_{\texttt{B,MIN(C)}}$ operator that corresponds to the GROUP BY B construct with MIN(C) aggregation. The LateralLimitTies uses a djoin, expressed in SQL syntax by LEFT JOIN LATERAL.

In addition to the type of rank function used, two other aspects govern the suitability of each logical tree:

- The combination of B and C is UCC - again, it is an important aspect since, in such case, all three rank functions are equivalent and ROW_NUMBER is deterministic. If B and C are not UCC, the JoinMin cannot be used with ROW_NUMBER.

- The C attributes can be NULL - in some DBSs, NULL is the lowest value in an ordered set; however, MIN() ignores NULL values if there are other values in the ordered set. That is a problem, and the JoinMin cannot be used. Similarly, with MAX(), if NULL is the highest in an ordered set. Therefore, the JoinMin is not applicable if C can be NULL. This behavior is solvable using a special purpose MIN/MAX aggregate function that returns a NULL value (see Section 3.3.2 for details).

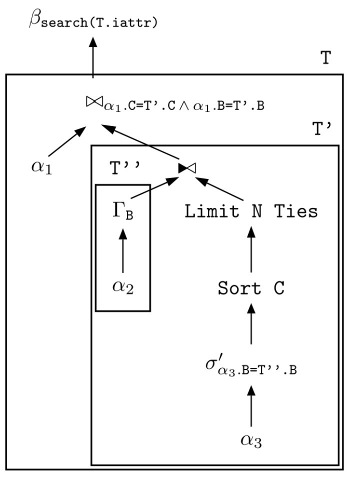
The JoinMin translates into a more efficient query plan than the LateralAgg and the LateralLimitTies. It has just an equi join with grouping, and some DBS even implement a special purpose GroupJoin operator to speed up such plans [25].

The LateralLimitTies usually leads to query plans similar to the LateralAgg, but the LateralLimitTies can be faster if an appropriate index exists. The problem with both is that it repeats the computation (the T' subplan) for the same α_1.B values. Some DBS solve this problem by subplan result caching. We address this problem in Section 3.3.

3.3 LateralLimitTies Optimization

This section introduces two possible solutions that avoid repeated costly computations of the LateralLimitTies. The first solution adds computation of distinct *B* values into the T' subtree. The second solution is the introduction of a custom NMIN/NMAX aggregate operator.

3.3.1 LateralDistinctLimitTies. Figure 7 shows the LateralDistinctLimitTies logical tree that introduces a distinct B values computation (see the T" subplan). The tree is universal, meaning it does not have the restrictions on predicate use that the JoinMin has. As we will see in the experimental section, this logical tree consistently outperforms (or equals) the LateralLimitTies in all DBS used in our experiments. That is valid despite the α being three times in the logical tree.



$\beta_{\text{search(T.iattr)}}$
T
$\bowtie_{\alpha_1.\text{C=T'.C} \wedge \alpha_1.\text{B=T'.B}}$
T'
α_1
T''
$\rtimes$
Γ_B
Limit N Ties
α_2
Sort C
$\sigma'_{\alpha_3.\text{B=T''.B}}$
α_3

Figure 7: LateralDistinctLimitTies

3.3.2 Using New NMIN/NMAX Operator. Let us consider the existence of a special purpose operator NMIN and NMAX that returns Nth lowest and Nth highest values, respectively. An algorithm implementing such an operator on an unsorted set finds the result in a single set scan. The algorithm keeps a set of N lowest/highest values and updates it during every cursor advance. It is easy to handle distinct values (if we compute DENSE_RANK) in NMIN and NMAX operator.

The JoinNMin logical tree is a small modification of the JoinMin where NMin is used instead of MIN. The JoinNMin can be used for all three types of predicates. The only limitation is that we cannot use the JoinNMin if we compute ROW_NUMBER and B and C is not UCC.

The NMIN and NMAX functions are not in the arsenal of standard DBS, but PostgreSQL allows a user to define its custom aggregate function. Unfortunately, the NULL handling would require some special symbol for "input set is empty" (see Section 3.4). Therefore, we implement the NMIN in PostgreSQL, but we do not consider the attributes with NULL values for simplicity.

3.4 NULL Values and Logical Tree Equivalence

If the B and C attributes contain NULL values, then we must be careful about the linear and self-join strategy equivalence. The linear strategy keeps rows with any such NULL value. On the other hand, the self-join removes them if they are not adequately handled in the σ or $\bowtie$ operators. If B can be NULL, then the σ' condition has to be extended $\alpha_2.\text{B} = \alpha_1.\text{B} \vee (\alpha_2.\text{B is null} \wedge \alpha_1.\text{B is null})$ in every logical tree to achieve the equivalence. Unfortunately, some DBSs have difficulty using indexes on the B attribute in the case of such an extended predicate. Therefore, we omit NULL values in our experiments. Handling extended predicate and, therefore, NULL values is a solvable problem at the query plan level since most DBS do not have issues with it.

Ordering of sets with NULL values and their effect on possible transformation was mentioned several times in Sections 3.2, and 3.3. It affects the JoinMin since MIN and MAX return NULL values only if the input set is empty. In other words, NULL is not returned when it is the minimum/maximum value according to the DBS set ordering. The most elegant solution is to use some custom NMIN/NMAX aggregate functions (operators) that do not ignore NULL values in such sets. However, as mentioned, we need another special symbol for "input set is empty" if we do not ignore NULL.

Attentive readers may observe that we miss the precise rank values in the greatest per group case result if we apply the self-join strategy. We presume the SQL user no longer finds the rank value significant after the row selection. Nevertheless, if required, the rank value can be calculated after the row selection in most scenarios, and the expense will be minimal since we already have a much smaller set of relevant rows.

4 EXPERIMENTS

This section presents the structure and results of two microbenchmarks, one focusing on the greatest per group problem and the second on the aggregate WFEs with a selective predicate. They allow in-depth testing of the profitability of the self-join strategy. We also show an experiment with selected TPC-H benchmark queries that could be expressed using WFE.

We run our experiments on four different DBS. Specifically, on Hyper [20]. PostgreSQL [6] and two commercial ones denoted as DBMS1 and DBMS2. We ran the experiment in main memory to avoid the non-deterministic loading of blocks from secondary storage. Each query is executed three times during each test, and we select the best time. We set a five-minute timeout for one query.

All experiments have been executed on a computer with two Intel Xeon processors X6136@3.00GHz, 2.0 MB L2 cache per core, and with 12 physical cores and 24 logical ones; Windows Server 2016 x64. The server has several hundreds of gigabytes of RAM available, and we ensure all tests are held in main memory.

The microbenchmarks use our SQL rewriting tool (see Section 4.1). The tool allows us to generate required self-join variants from a query with nested NQWF and enables testing thousands of different queries. Each microbenchmark is based on a few query templates and SQL scripts, simplifying its future extension or DBS addition.

The microbenchmarks operate on two artificial tables `RTab(A,B,C)` and `PTab(A,B,C,Padding)` containing one million rows in the case of PostgreSQL, DBMS1 and DBMS2 and ten million rows in the case of Hyper:

- The `A` attribute is a unique integer.
- The `B` attribute is an integer, where the number of unique values varies in the experiments and takes values of 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, and 30 % of the total number of records (this set is called `BDistinct`).
- The `C` attribute is a random integer from 0 to 10000.
- The `Padding` is a string 100 bytes long.

We avoid NULL values for reasons described in Section 3.4. In the case of Hyper, we use ten million rows tables because the query times were on the edge of measurability for one million rows.

The code used in this article, together with experimental results and code analyzing the results and generating the graphs, is available on github [2].

4.1 SQL Rewriting Tool

One of the results of this work is a SQL rewriting tool that takes a query with nested NQWF and rewrites it into an equivalent SQL using a self-join strategy. The tool is an essential element of the microbenchmarks and allows us to perform many tests with different settings.

The tool allows us to select a logical tree and even handles NQWF containing more than one WFE or aggregate window functions with explicit framing. Explicit framing sometimes requires a UNION operator, so the result query may be quite slow. We do not perform explicit framing testing in our benchmarks.

4.2 Greatest Per Group Microbenchmark

We introduced four self-join logical trees in Section 3 that address the greatest per group problem. The main aim of this microbenchmark is to compare their performance.

We use the query from Listing 6 as a query processed by linear strategy. We call it a linear strategy query in the following text. The

[2]https://github.com/RadimBaca/SQL_window_function_rewrite

```
SELECT A, B
FROM (
    SELECT A, B,
        ROW_NUMBER() OVER (
            PARTITION BY B
            ORDER BY A
        ) RN
    FROM RTAB
) T1
WHERE RN = 1
```

Listing 6: Greatest per group query used to compare different logical trees

linear strategy query is then rewritten into four different self-join queries, allowing us to compare them. We run comparisons with various test settings:

1. `B` uniqueness - using `BDistinct` values ranging from 0.001 to 3%.
2. Padding - we slightly modify the query to work with `PTab` and also return the `padding` attribute.
3. Parallelization - we test the queries with parallel processing enabled (8 threads) and disabled. Hyper is always parallelized.
4. Indexes - the data table is always a heap, and we apply different physical designs:
    - None - there is no index in the database. The only option for Hyper.
    - `I(A)` - secondary index on `A`.
    - `I(B)` - secondary index on `B`.
    - `I(A);(B)` - two secondary indexes on `A` and `B`.
    - `I(AB)` - secondary indexes on `A,B`.
    - `I(BA)` - secondary indexes on `B,A`.

Note that `I(AB)` and `I(BA)` are covering indexes in the case of each query in this benchmark.

A test is a comparison of a self-join strategy query with a linear strategy query using one combination of settings. Since we apply each combination of settings, the final number of tests for one self-join strategy on one DBS is 192, except Hyper, which does not support indexes and single-threaded processing. We are mainly interested in a ratio between linear strategy processing time (denoted as T_{lin}) and self-join strategy processing time (denoted as T_{sj}).

4.2.1 Ratio Results. In Figure 8, we can observe the ratio between T_{lin} and T_{sj} for different DBSs and test settings. We use boxplots that summarize the ratio for all tests with selected settings. Value 1 (red line) represents a situation where T_{lin} and T_{sj} are equal, and the ratios above represent a situation where the self-join strategy is faster.

Let us first summarize the results of the greatest per group microbenchmark from the logical tree type perspective since it has a major effect on the T_{lin} and T_{sj} ratio. In this paragraph, we answer the question: into which self-join logical tree should we rewrite an NQWF? The JoinMin is always the fastest option and should be used if possible. The LateralDistinctLimitTies outperforms the

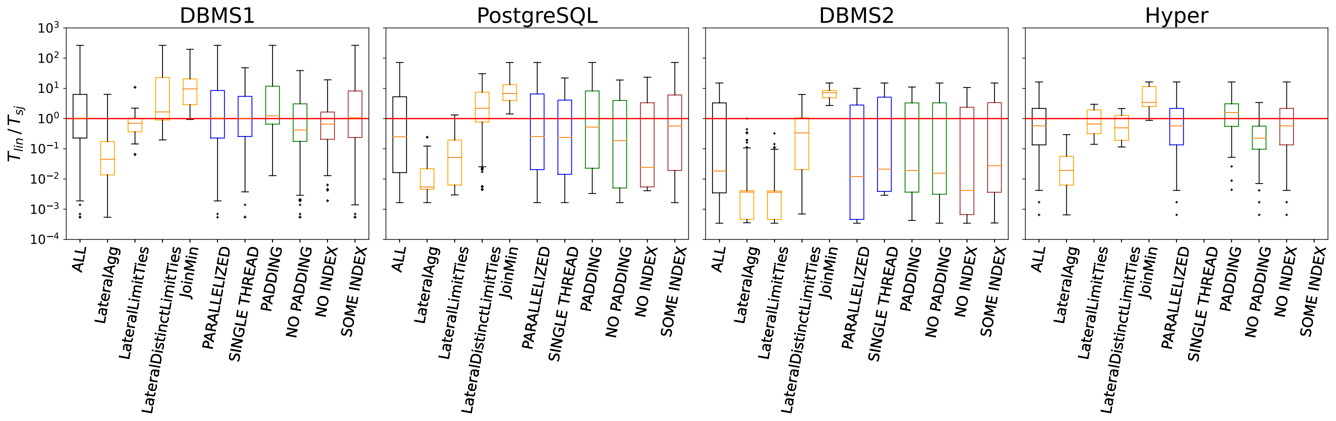


Figure 8: Ratio between T_{lin} and T_{sj} for different DBSs and settings in greatest per group microbenchmark.

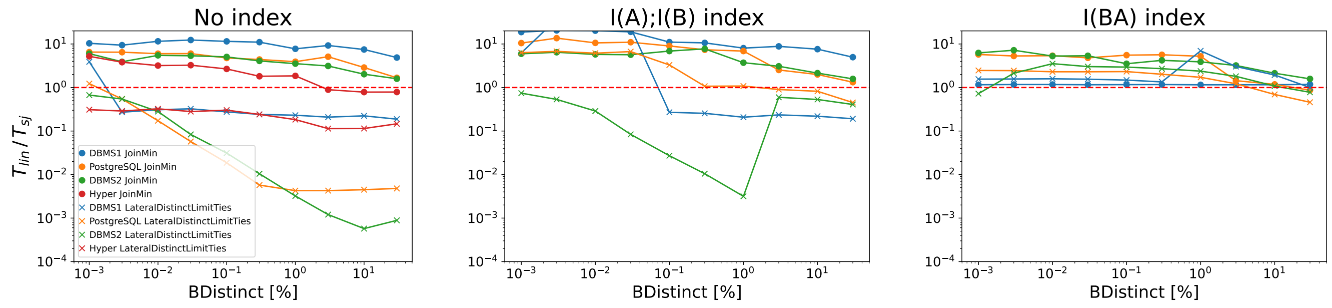


Figure 9: Ratio between T_{lin} and T_{sj} depending on BDistinct value for parallel tests without the padding attribute. The graph focuses just on JoinMin and LaterDistinctLimitTies.

LateralLimitTies; therefore, the LateralDistinctLimitTies should always be used instead of the LateralLimitTies. Sometimes, we cannot use the JoinMin or the LateralDistinctLimitTies, so only the LateralAgg remains. The LateralAgg is worth considering only if a high selectivity predicate is in the β that can be pushed down to α_1 by a query optimizer.

As mentioned, the selection of the logical tree type mainly influences the ratio. However, the ratio spans several orders of magnitude for individual logic plans since other aspects may also have an effect, although not as significantly. Let us try to analyze this in more detail:

(1) Parallelization - Multi-threaded processing is advantageous for linear strategy in both commercial DBSs. That is influenced by the fact that the sorting is more straightforward to parallelize than a complex plan created by the self-join strategy.
(2) Padding - When dealing with larger rows, the self-join strategy is more beneficial than the linear strategy because sorting can become challenging with the latter. This result supports our hypothesis that sorting is a major issue in the linear strategy.
(3) Indexes - The existence of indices increases the profitability of the self-join strategy in most cases. In this case, DBS can effectively use an indexed nested-loop join if selectivity allows it.
(4) BDistinct - The self-join strategy tends to be faster at low BDistinct values. We discuss this effect in more detail here.

Figure 9 shows the dependence of the ratio of T_{lin} and T_{sj} on BDistinct where several parameters are fixed. We fixed the parallel processing and the padding attribute parameters. We use `BDistinct` ranging from 0.001 to 30% for this experiment. The plots show the JoinMin and the LateralDistinctLimitTies for several index choices. Lower BDistinct values mean a smaller result size and, therefore, an advantage for the self-join strategy.

In Figure 9 with `I(A);I(B)` indices, we observe an unusual jump in the ratio of T_{lin} and T_{sj} for commercial DBSs. This is due to the optimizer's decision to (not) use indices for certain values of BDistinct. The self-join strategy benefits from an index if the DBS optimizer decides it is worth using. We can also notice that in the case of the covering index `I(BA)`, such jumps no longer occur because the optimizer always uses the index.

More generally, the self-join strategy is advantageous if it leads to a plan with an indexed nested-loop join with few repetitions. The larger the intermediate result of the linear strategy compared to the actual result, the more worthwhile it is to use the self-join strategy. This is because the main drawback of the linear strategy is the expensive sorting.

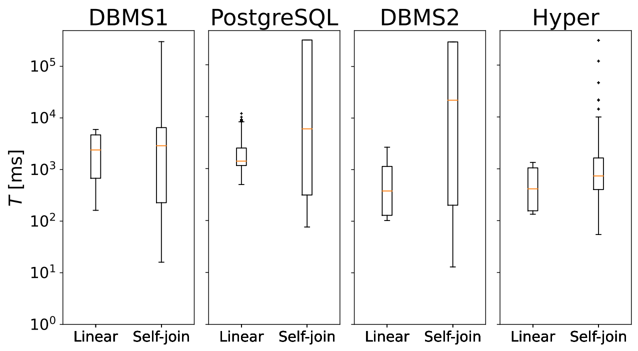


Figure 10: Absolute query processing times for the linear and self-join strategy.

4.2.2 Absolute Query Processing Times Results. It is noticeable from Figure 10 that the self-join queries reach much longer processing times when compared to the linear strategy queries, which is an obvious drawback of the self-join strategy. The main aim of this article is to show that we can achieve significant speed-up using the self-join strategy compared to the linear strategy; however, there is a risk related to wrong cost estimation. We discuss the accuracy of some cost-based optimizers in more detail in Section 4.4.

Table 2 summarizes the percentage of self-join queries reaching the five-minute limit. The percent of such queries in the case of the LateralAgg is significant in all DBS. Reaching the five-minute limit means that the ratios of the LateralLimitTies and the LateralAgg presented in Figure 8 are, in reality, lower in many cases. On the other hand, ratios of the LateralDistinctLimitTies and the JoinMin are accurate. The linear strategy queries never reach the five-minute limit.

Table 2: Percent of self-join queries reaching the five-minute limit

	DBMS1	PostgreSQL	DBMS2	Hyper
JoinMin	0 %	0 %	0 %	0%
LateralDistinctLimitTies	0 %	5.2 %	5.2 %	0%
LateralLimitTies	0 %	31.3 %	80 %	0%
LateralAgg	27.6 %	77 %	76 %	12.5 %

4.3 Aggregate Window Functions Microbenchmark

In this section, we investigate the usefulness of the LateralAgg. As is evident from results presented in Section 4.2, the LateralAgg can be helpful only if there is a high selectivity predicate in β that can be propagated into α_1. There is no such predicate in the greatest per group microbenchmark, which is why LateralAgg performs so poorly there. In this microbenchmark, we evaluate the situation where β contains such a predicate using the query from Listing 7.

```
SELECT A, B, AGG
FROM (
    SELECT A, B,
        AGG_FUN() OVER (
            PARTITION BY B
            ORDER BY A
        ) AGG
    FROM RTAB
) T1
WHERE C < SEL
```

Listing 7: Query template used in the aggregate window function microbenchmark

Similarly to the greatest per group microbenchmark, there are various test settings:

- The selectivity of β - it is set by the SEL constant, which takes values of 1, 2, 4, 8, 16, and 32. A value of 1 for the SEL constant corresponds to a selectivity of 0.1 ‰ relative to the total table size (i.e., 1 million rows).
- AGG_FUN() - we tested the COUNT(*) and MIN(A) aggregation functions.
- Constructs - we tested queries with both PARTITION BY and ORDER BY constructs (denoted as PB_OB) and queries with just PARTITION BY (denoted as PB).
- B uniqueness - different BDistinct values.
- Padding - we slightly modify the query to work with PTab and also return the padding attribute.
- Parallelization - we test the queries with parallel processing enabled (8 threads) and disabled.
- Indexes - we use twelve different index configurations since the C attribute is added compared to the greatest per group microbenchmark. However, in tests without ORDER BY constructs, we only use four index configurations because the A attribute is missing in predicates, making indexes with the A attribute useless.

All these settings give a total of 7,680 different tests.

4.3.1 Ratio Results. Figure 11 summarizes ratios of this microbenchmark for different DBSs. Similarly to the greatest per group microbenchmark, some settings make the self-join strategy more profitable:

(1) Single-threaded processing,
(2) larger rows (with padding),
(3) and usage of indexes.

We can also observe that the WFE constructs have an impact as well. For example, the WFE without ORDER BY (PB) and the window function with MIN make the self-join strategy more advantageous in all DBSs.

The diagrams in Figure 12 show the effect of selectivity on the ratio of T_{lin} and T_{sj} for different indices. We fixed other parameters such as parallel execution, omitted padding, chose the MIN window function, and used the PB_OB constructs. Each graph contains two rows for each DBS: (1) for BDistinct 0.001 and (2) for BDistinct 3.

First, we examine Figure 12 from the BDistinct perspective. BDistinct does not determine the size of the result as in the case of the greatest per group microbenchmark. In these tests, a low BDistinct

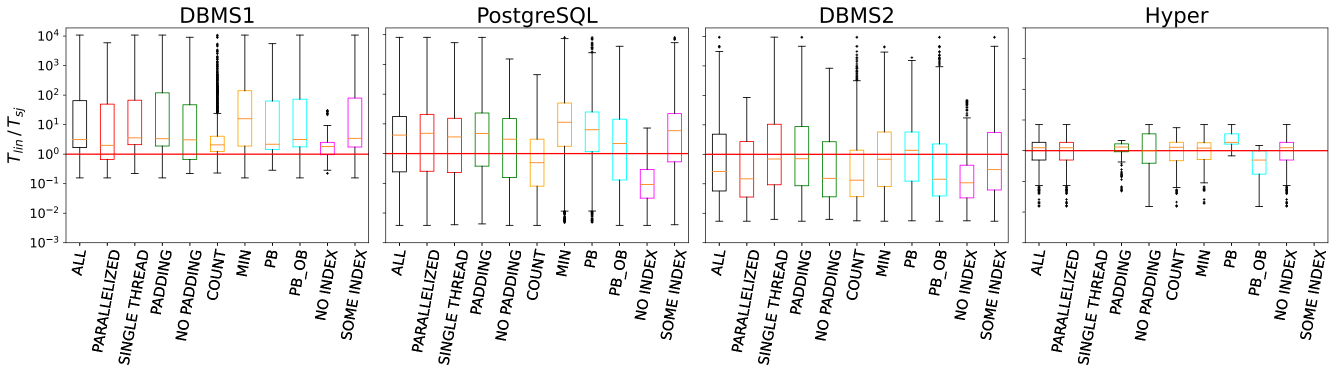


Figure 11: Ratio between T_{lin} and T_{sj} for different DBS and settings in aggregate window functions microbenchmark.

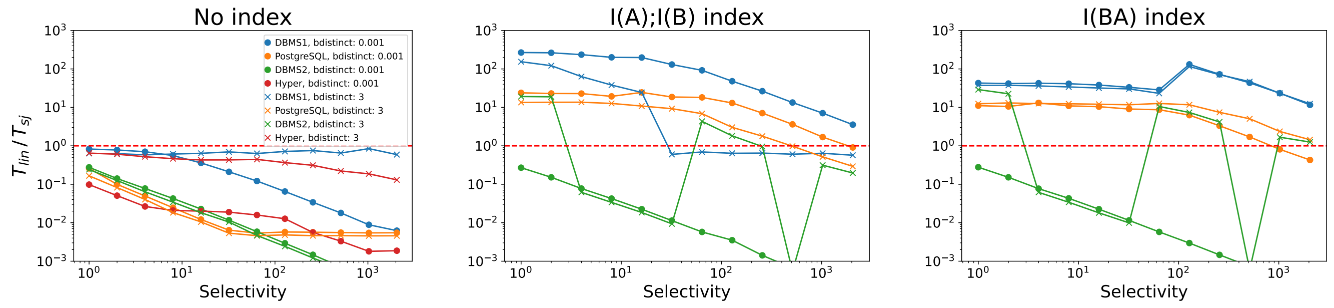


Figure 12: Ratio between T_{lin} and T_{sj} depending on selectivity of β for parallel tests, without the padding attribute, using `PARTITION BY`, `ORDER BY`, and `MIN` function.

has a rather negative effect because it can give us a large intermediate result for each attribute B value. For some DBS, it then leads to execution plans that sort the intermediate result, which subsequently slows down the execution of the self-join query significantly. An important factor is using an appropriate index instead of sorting. Ideally, the self-join strategy again leads to an indexed nested-loop join with a few iterations.

If we look at Figure 12 from the selectivity of β perspective, higher selectivity results in a greater advantage of the self-join strategy. This result is consistent with expectations. For this experiment, we extended the range of SEL values up to 2048, corresponding to 20% selectivity. As we can see, in the case of a suitable index, some DBSs can construct a favorable self-join plan even for such low selectivity. We observe unusual jumps in the ratio for commercial DBSs, again due to the optimizer's decision to (not) use indices in certain cases.

In contrast to the greatest per group microbenchmark, the five-minute limit was rarely reached in this microbenchmark. That shows that the highly selective condition from β was successfully pushed down and used to achieve more efficient query plans.

4.4 Cost-based Strategy

Cost-based strategy means that for an SQL query Q_{lin} with WFE, we do the following:

(1) We create Q_{sj}, which is a transformed version of Q_{lin} using the self-join strategy.
(2) We let the DBS query optimizer optimize both queries, and we read the estimated cost for each query.
(3) We select the query with the lower cost.

We only present results for DBMS1 and PostgreSQL, from which we could extract estimated plan cost information within our Java code that runs all tests. Additionally, we only present the results for tests that ran within five minutes since we were actually collecting the cost information after a query run.

Table 3 compares query processing average times for both microbenchmarks, two DBSs, and all strategies. Table 3 also includes an optimal strategy as a baseline. The optimal strategy represents a hypothetical case where we always choose the faster strategy. We see the average query times of the linear and self-join strategies in the first two rows. The optimal strategy average query time follows, and the most important is the bolded fourth row, which shows the average query time of the cost-based strategy. *The main result of this test is that the cost-based strategy outperforms the linear strategy in every column.*

Interesting for us are the tests where the self-join strategy is a false positive (SJSFP tests) since we want to know the ratio between the strategies' times when the cost estimation is wrong. Table 3 shows the average and minimum ratios for the SJSFP tests. These

Table 3: Cost-based strategy selection statistics

	Greatest per group Microbenchmark		Aggregate WF Microbenchmark	
	DBMS1	PostgreSQL	DBMS1	PostgreSQL
Linear strategy average time [ms]	2735	2171	5098	2467
Self-join strategy average time [ms]	11525	18812	1909	18297
Optimal strategy average time [ms]	1651	1005	1608	1330
Cost-based strategy average time [ms]	**1884**	**1082**	**2152**	**1596**
SJSFP tests average ratio	0.71	-	0.98	-
SJSFP tests minimum ratio	0.41	-	0.96	-
# tests	715	550	7680	7578
Linear strategy false positives [%]	20.7	6.5	21.4	20
Self-join strategy false positives [%]	2	0	0.35	0

ratios are not significantly bad in the case of these tests. A few tests with the LateralDistinctLimit algorithm have more than twice the T_{sj} time; however, most SJSFP tests have nearly identical times for both strategies, so the ratio is close to one. Moreover, in the case of PostgreSQL, there are no SJSFP tests.

The last two rows in the table show how often the optimizer makes a mistake when estimating the cost. That is, how often the plan selected based on estimated cost is slower. The penultimate and last rows indicate how often a false selection of the linear and self-join strategies occurs, respectively. The optimizer favors the linear strategy (see the higher linear strategy false positives values in Table 3). It is good since the linear strategy is more conservative and does not tend to lead to extreme query plans, as mentioned in Section 4.2.2.

The experiment in this section shows we get better times on average if we use the SQL query transformation combined with the built-in cost mechanism.

4.5 NMIN Experiment

In Section 3.3.2, we introduced the `NMIN` function, which enables usage of the JoinNMin logical tree when the predicate in β is other than `T.rn = 1`. We implement the `NMIN` function in PostgreSQL, which allows the implementation of a custom aggregation function.

We perform the same test as in Section 4.2 but use the `T.rn = 5` predicate this time and compare only the JoinNMin and the LateralDistinctLimitTies. That is because the LateralDistinctLimitTies outperforms the remaining lateral logical trees in the greatest per group microbenchmark.

Summarized results of this test are shown in Figure 13, and they correspond to the previous results presented in Section 4.2. The results show that the JoinNMin outperforms the LateralDistinctLimitTies significantly. We can observe that JoinNMin's results are just as good as JoinMin's. So, the JoinNMin, in particular, increases the number of situations where we can use the very efficient self-join strategy.

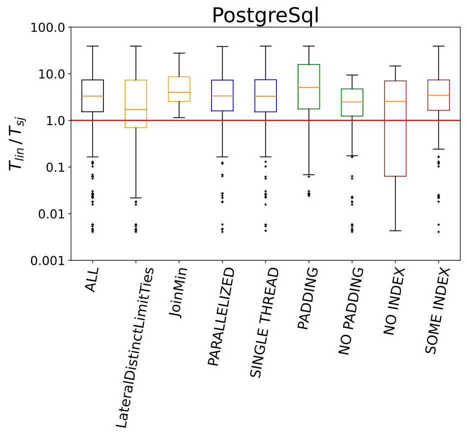


Figure 13: Ratio between T_{lin} and T_{sj} for PostgreSQL using the greatest per group microbenchmark and with the JoinNMin

.

4.6 TPC-H Benchmark

In the list of 22 TPC-H queries, three can be expressed using the WFE: Q2, Q15, and Q17. So, we do not use the original TPC-H SQL query notation in this experiment, but we have rewritten them into equivalent ones using WFE.

TPC-H is still one of the most relevant database benchmarks [14] almost three decades after its introduction. We used a TPC-H with a scale factor of 10, where many tables are an order of magnitude bigger than the table used in our microbenchmark.

We processed the queries with default parallelization (i.e., more threads can be used if the optimizer finds them suitable). We did not create any indexes. We leave the default physical design created by a popular benchmark testing software HammerDB [28], which already contains appropriate indexes. Table 4 briefly summarises TPC-H queries and the logical tree applied to each query.

Table 4: Description of three tested TPC-H queries.

Query	Description	Logical tree
Q2	Greatest per group query with ties. Expressed using `RANK` window function with `PARTITION BY` and `ORDER BY`.	JoinMin
Q15	Query consists of two parts: 1) computation of aggregate for certain groups, and 2) searching for rows with the greatest aggregate. We keep the view expressing the first part and use the `RANK` window function with `ORDER BY` for the second part.	JoinMin
Q22	Aggregate value computation with a highly selective condition (1 per mille) in the outer query. Expressed using `SUM` window function with `PARTITION BY`.	LateralAgg

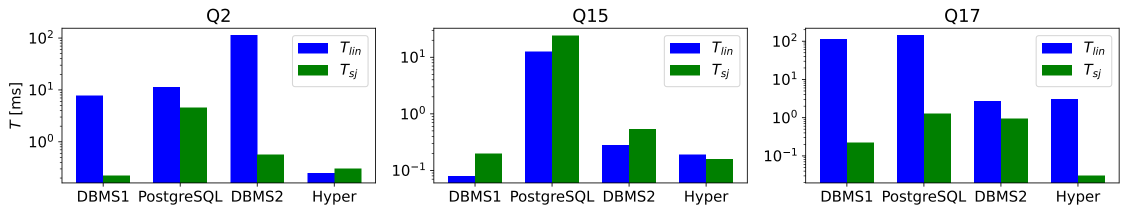


Figure 14: Results of different compilation strategies for the selected TPC-H queries.

Figure 14 summarizes the measured results. Q2 is a model example of a query, as seen in Listing 6. We can use the JoinMin logical tree; therefore, the self-join strategy outperforms (or is approximately equal to) the linear strategy in every DBSs. The Q15 contains `rn = 1` predicate as well; however, the computation of the α dominates the query, and no index or nested-loop join can be used to speed up the query. Therefore, the α (i.e., the first part of the Q15 query) is computed twice, and the self-join strategy is twice as slow. Q17 is another example of a model query, as seen in Listing 7. We can see that the self-join strategy is more efficient in the case of Q17. That is caused mainly because the selectivity of the β condition is very high. In the case of Q17, we can see an improvement of several orders of magnitude even though the LateralAgg logical tree transformation is used.

The Q15 shows a limitation of our greatest per group microbenchmark, which concluded that the JoinMin should consistently outperform the linear strategy. The α is a single table in the microbenchmark, enabling many query plan optimizations. Even having more joined tables with selections is not a big problem, as we can see in Q2. However, the α in Q15 is a query with aggregations that needs to be fully materialized before it can be further handled in the self-join. Therefore, the advantages of the JoinMin are limited in the case of Q15, and the linear strategy is more efficient.

5 RELATED WORK

Even though WFEs are popular, the amount of research that has been done on their optimization in the last two decades [2, 11, 23, 30, 32] is not significant. Works describe efficient parallelized linear strategy operator [23], resource sharing during computation of many queries with WFE [2], or efficient implementation of WFEs with framing that is not part of the SQL:2003 standard [30]. Some works aim to solve different *range problems* [9, 10, 15, 19], which is a name in the algorithm community for problems represented by WFEs. The range problem works usually focus on efficient computation of a particular WFE (e.g., percentiles or medians).

Some papers focus on transforming the entire SQL query [8, 34]. Zuzarte et al. [34] describe a WinMagic transformation rule that proceeds in the opposite direction to the one we propose. Thus, the rule searches for and replaces self-join using a WFE. A similar approach can be found in the paper [8]. These works are based on the assumption that WFEs are always a better option than a self-join query plan. We show in our paper that this assumption is not correct.

Work [1] summarizes a number of different heuristic and cost-based SQL transformations and shows how to apply them in a query optimizer. Our transformation can be incorporated into a query compiler in the same way.

Surprisingly, as far as we know, no existing work indicates that the self-join strategy can be more effective than the linear strategy under certain conditions.

6 CONCLUSION AND FUTURE WORK

This article proposes a holistic view of a compilation of SQL queries with WFEs. We presented several transformations for SQL queries with rank and aggregate window functions and compared their effectiveness. We created two microbenchmarks showing the influence of different DBSs and query aspects, such as usage of parallelization, data row size, selectivity of important query parts, and the existence of different indexes. Our experiments show that achieving more than one order of magnitude better query times is possible using the self-join strategy in all tested DBSs. We proposed and tested the `NMIN` aggregation function that extends the number of situations where the most efficient JoinMin logical tree can be used. Our experiment with several TPC-H queries shows that the model situations described in our paper are not marginal but can be observed even in such a popular benchmark.

We have added an experiment that shows that existing cost-based optimizers are, in many cases, able to select the faster of the two strategies based only on the built-in plan cost estimation. We show that the cost-based strategy outperforms the linear strategy on average query processing time, and at least one-third of the total time can be saved on average.

In our article, we have focused on only the two most basic and most used categories of WFEs (rank and aggregate). However, the self-join strategy can be applied to other WFEs we have not considered in this article.

Applying our transformation into a query optimizer is not complicated and does not require complex query matching. Transformation of an SQL query using a self-join strategy is straightforward. As our experiments suggest, incorporating such transformation into a DBS query optimizer on a query algebra level is possible and would be beneficial.

ACKNOWLEDGMENTS

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic (SGS, No. SP2024/100). We thank Michal Krátký for reading the article and valuable comments. Finally, we are grateful for the opportunity to use modern AI tools such as chatGPT and Grammarly, as without them, the whole work would have taken incomparably more time.

REFERENCES

[1] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-based Query Transformation in Oracle. In *VLDB*, Vol. 6. 1026–1036.

[2] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-window Aggregates. In *Proceedings of the Thirtieth international conference on Very large data bases*, Vol. 4. 336–347.

[3] DuckDB authors. 2023. DuckDB - Window Functions. Retrieved July 8, 2023. https://duckdb.org/docs/sql/window_functions

[4] Impala authors. 2023. Subqueries in Impala SELECT Statements. Retrieved August 22, 2023. https://impala.apache.org/docs/build/html/topics/impala_subqueries.html

[5] MySQL authors. 2023. MySQL - Window Functions. Retrieved July 8, 2023. https://dev.mysql.com/doc/refman/8.0/en/window-functions.html

[6] PostgreSQL authors. 2023. PostgreSql 14.2. Retrieved August 20, 2022. https://www.postgresql.org/docs/14/release-14.html

[7] Guilherme Banhudo. 2020. SQL Performance Tips 1. Retrieved January 2, 2024. https://towardsdatascience.com/sql-performance-tips-1-50eb318cd0e5

[8] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun-Chieh Lin. 2009. Enhanced Subquery Optimizations in Oracle. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1366–1377.

[9] Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. 2011. Towards Optimal Range Medians. *Theoretical Computer Science* 412, 24 (2011), 2588–2601.

[10] Gerth Stølting Brodal and Allan Grønlund Jørgensen. 2009. Data Structures for Range Median Queries. In *International Symposium on Algorithms and Computation*. Springer, 822–831.

[11] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. 2012. Optimization of Analytic Window Functions. *Proceedings of the VLDB Endowment* 5, 11 (2012).

[12] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 34–43.

[13] Umeshwar Dayal. 1987. Of Nests aud Trees: A Untied Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings of the VLDB Conference*. VLDB Endowment, 197–208.

[14] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1206–1220.

[15] Sariel Har-Peled and S Muthukrishnan. 2008. Range Medians. In *European Symposium on Algorithms*. Springer, 503–514.

[16] 2023 Hive authors. Retrieved August 22. 2023. Language Manual SubQueries. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+SubQueries

[17] Silu Huang, Erkang Zhu, Surajit Chaudhuri, and Leonhard Spiegelberg. 2023. T-Rex: Optimizing Pattern Search on Time Series. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[18] Yannis E Ioannidis. 1996. Query Optimization. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 121–123.

[19] Allan Grønlund Jørgensen and Kasper Green Larsen. 2011. Range Selection and Median: Tight Cell Probe Lower Bounds and Adaptive Data Structures. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 805–813.

[20] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[21] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data Dependencies for Query Optimization: a Survey. *The VLDB Journal* 31, 1 (2022), 1–22.

[22] Alexander Kozak. 2006. Islands and Gaps in Sequential Numbers. Retrieved January 2, 2024. https://learn.microsoft.com/en-us/previous-versions/sql/legacy/aa175780(v=sql.80)

[23] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1058–1069.

[24] Guido Moerkotte. 2023. *Building Query Compilers. Retrieved October 10, 2023.* http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[25] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-by and Join by GroupJoin. *Proceedings of the VLDB Endowment* 4, 11 (2011), 843–851.

[26] M. Muralikrishna. 1992. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proceedings of the 18th International Conference on Very Large Data Bases*, Vol. 92. Morgan Kaufmann Publishers Inc., 91–102.

[27] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. 1996. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. ACM, 435–446.

[28] Steve Shaw. 2012. HammerDB: the Open Source Database Load Test Tool. Retrieved June 1, 2021. https://www.hammerdb.com/

[29] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 307–322.

[30] Adrian Vogelsgesang, Thomas Neumann, Viktor Leis, and Alfons Kemper. 2022. Efficient Evaluation of Arbitrarily-framed Holistic SQL Aggregates and Window Functions. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, 1243–1256.

[31] Richard Wesley. 2021. Windowing in DuckDB. Retrieved January 2, 2024. https://duckdb.org/2021/10/13/windowing.html

[32] Richard Wesley and Fei Xu. 2016. Incremental Computation of Common Windowed Holistic Aggregates. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1221–1232.

[33] Wenjun Zhou. 2014. Window Function vs. Self-join in SAP HANA. Retrieved January 2, 2024. https://blogs.sap.com/2014/10/04/window-function-vs-self-join-in-sap-hana/

[34] Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, and Kwai Wong. 2003. Winmagic: Subquery Elimination Using Window Aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 652–656.