Window Function Optimization: Co-Evaluation and Other Techniques

Daniel Lindner
Hasso Plattner Institute
Potsdam, Germany
daniel.lindner@hpi.de

Felix Naumann
Hasso Plattner Institute
Potsdam, Germany
felix.naumann@hpi.de

Alberto Lerner
Computing Flows
Fribourg, Switzerland
lernera@computingflows.com

ABSTRACT

Window functions are among the most expressive features of modern SQL. Surprisingly, relatively little has been written about their optimization. Some techniques exist, such as pushing predicates through a window under ideal conditions, but known optimizations no longer apply when those conditions are even slightly unmet. We show that these limitations are not fundamental, but persist because a reasoning framework for window function optimization has been missing. We provide such a framework, introducing techniques we call Frame Analysis, Partition Analysis, and a new execution strategy called Co-Evaluation. These clarify when and how optimizations can be applied. Co-Evaluation, in particular, allows early evaluation of predicates even when they depend on the window function's result. We present each technique and organize the results as a table of algebraic equivalences for window functions. We test these optimizations in an open-source engine, where they never hurt performance and make certain common queries up to 40.7× faster, with larger tables yielding larger gains.

PVLDB Reference Format:
Daniel Lindner, Felix Naumann, and Alberto Lerner. Window Function Optimization: Co-Evaluation and Other Techniques. PVLDB, 19(11): 3525-3537, 2026. doi:10.14778/3836663.3836706

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HPI-Information-Systems/wf-optimization.

1 INTRODUCTION

Window functions (WFs) provide explicit support for order-dependent computations to be expressed in SQL [18] and are commonly present in data analytics queries. For instance, they appear in 11 % of Snowflake's customer queries [45], and in Meta's reporting workloads, "most query shapes contain joins, aggregations, or window functions" [38]. In a case study we conducted on traces from two large SAP customers, WF queries accounted for as much as 9.5 % of total execution time. Nearly every modern database engine implements WFs [2, 3, 10–13, 15, 17, 30–35, 40, 42, 44].

For this reason, WFs have been studied in the context of query optimization. Cao et al. report on how to share partitioning and sorting work across multiple windows [9]. Leis et al. present a highly optimized parallel physical window operator for main-memory systems [23]. More recently, Bača shows that rewriting certain windows as self-joins can be beneficial [5].

These works have clearly advanced the field, but some gaps still remain. No table of algebraic equivalences involving WFs exists, nor has a formalization been introduced that is precise enough to derive one. We provide both. Our formalization breaks WFs into basic phases instead of capturing their entire semantics at once. Each phase is a precise yet simple transformation that builds on the output of its predecessor (Section 2). This decomposition enables reasoning about intermediate execution states in a way that a monolith-style formalization could not offer. More importantly, it gives us insights into a number of details that are useful for algebraic optimizations (Section 3).

Some of these equivalences are already used in practice for certain queries, but even slight query changes can render them inapplicable. The challenge is not always to find new transformations, but to recognize when existing ones still apply. We address this challenge through a series of techniques that extend the reach of basic transformations. They comprise two *analyses* that identify optimization opportunities and a new evaluation strategy: *Co-Evaluation*.

The first technique is called *Frame Analysis*. It detects the conditions under which a WF can be replaced by a simpler formulation (Section 4). For instance, a frame spanning a single row, as in the following query, does not need to evaluate the WF.

```
Q1: SELECT *, SUM(a) OVER (
        ROWS BETWEEN CURRENT ROW AND CURRENT ROW)
    FROM r
```

Here, the WF is `SUM(a) OVER(...)` and it specifies the frame, i. e., the set of rows the aggregation considers, as `ROWS BETWEEN ... ROW`. This query applies `SUM(a)` to each row individually, which is the value of attribute `a`. A human would not write such a query, but software-written queries can feature such anomalies. Without the WF, this query runs 4× to 40.7× faster, depending on the table size. Frame Analysis recognizes and replaces such singleton frames.

The second technique, *Partition Analysis*, determines when predicates or joins can be pushed through a WF (Section 5). Some of the techniques here are not new; predicates that match the WF's partition expression exactly can benefit from early execution. However, there are very common queries for which this alignment is not perfect, as in the following query.

```
Q2: SELECT * FROM (
        SELECT *, AVG(price) OVER (
            PARTITION BY EXTRACT(YEAR FROM sold_date)
        ) AS yearly_avg
        FROM sales) AS s
    WHERE sold_date >= DATE '2020-02-15'
```


Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836706

The query computes yearly average prices for recent sales. It is written as an outer filtering query over an inner window query because the inner comes from a view while the outer is user-written. Both the predicate and the window operate on `sold_date`, so one may consider swapping them. Filtering early would attenuate the heavy sorting on the window. This predicate, however, alters the year 2020 partition, making the aggregation produce wrong results.

On close inspection, the predicate implies that only the partitions with a year ≥ 2020 are relevant. A predicate that filtered full years prior to 2020 could be correctly pushed. Partition Analysis offers techniques to recognize this scenario and determine such *derived predicates* that align with the partition expression. On a historical sales database, for instance, this optimization can eliminate decades of partitions in a single pass.

In the example above, we have considered predicates that are independent of the window's output. A harder problem arises when a predicate depends on it, as is the case for the following query Q3.

```
Q3: SELECT * FROM (
        SELECT *, RANK() OVER (
            PARTITION BY dept
            ORDER BY salary DESC) AS rnk
        FROM employees) AS s
    WHERE rnk <= 3
```

The query filters out all but the top three earners per department. The predicate `rnk <= 3` cannot be pushed through the WF because the rank of each `salary` must first be calculated within a department. However, we could discard salaries that cannot possibly rank among the top three early in the computation, substantially reducing the data the window processes. We call this technique *Co-Evaluation* (Section 6), and in our experiments, Q3's performance can run up to 5× faster with this optimization.

A standard WF operator has no place to inject this early pruning. A natural, relatively easy place to add it would be alongside the aggregation, but that yields little benefit. By then, the data has already been sorted, and sorting dominates the cost. The real gains come from pruning before or during sorting. We show how a standard WF operator can be extended to prune early; we call this WF$^+$ and describe our implementation in an open-source database (Section 7).

To measure the benefits, we evaluate our proposed techniques on queries that exemplify the optimization gaps we address. We ran them on tables with varying sizes, partition count, and skew to reveal which factors influence performance (Section 8). As mentioned, some experienced speedups between one and two orders of magnitude. We close the paper by comparing our techniques to related work (Section 9) and by stating our final remarks (Section 10). In summary, the main contributions of this paper are:

- A formal algebraic definition of window functions.
- A formal table of algebraic equivalences involving WFs.
- Frame and Partition Analyses that enable new optimizations or apply known optimizations in new scenarios.
- Implementation notes on how to incorporate the equivalences and analyses into existing engines.
- Co-Evaluation, a technique that fuses predicates with window execution to early prune non-qualifying rows.
- An experimental evaluation that applies the optimizations in practice and quantifies the speedups they bring.

2 WINDOW FUNCTION SEMANTICS

This section formally defines the window function operator. We first establish notation, then describe the operator as a sequence of phases, and finally show how SQL's syntax maps to this model. We define the operator over sequences rather than within relational algebra because relational algebra cannot represent several features required to capture the semantics of WFs, most notably order.

2.1 Preliminaries

We use standard notation for relational operations and introduce symbols that are specific to window functions in Table 1.

Relations and tuples. A *relation* r is a bag (multiset) of tuples. We write $\mathcal{A}(r)$ for the set of attributes of r. For a tuple t and attribute set $A \subseteq \mathcal{A}(r)$, $t[A]$ denotes the projection of t onto A. Two tuples *agree* on A if they have the same values for all attributes in A. Tuple concatenation $t \circ u$ appends the attributes of u to t. We write θ for a generic comparison operator ($=, <, \leq, >, \geq$) and $a\,\theta\,c$ for a predicate comparing attribute a to constant c.

Sequences. A *sequence* $\mathcal{S} = \langle t_1, \ldots, t_n \rangle$ is an ordered list of tuples [1]. We write $|\mathcal{S}|$ for its length, $||\mathcal{S}||$ for its number of distinct elements, $\mathcal{S}[k]$ for the k^{th} tuple (1-indexed), and $\mathcal{S}[l:u]$ for the subsequence from l to u, inclusive. $\mathcal{S}[l:u] = \langle\rangle$ if $l > u$. For index sequences $\mathcal{I} = \langle k_1, k_2, \ldots \rangle$, $\mathcal{S} - \mathcal{I}$ removes the positions in $\mathcal{I}$ from $\mathcal{S}$. We write $\langle k : C(k) \rangle$ for the sequence of positions k satisfying condition C. $\mathcal{S}[\mathcal{I}]$ denotes the sequence $\langle t_{k_1}, t_{k_2}, \ldots \rangle$. $\mathcal{S}[\mathcal{I}][A]$ is the projection $\langle t_{k_1}[A], t_{k_2}[A], \ldots \rangle$. Concatenation is written $\mathcal{S}_1 \circ \mathcal{S}_2$.

Dependencies. An attribute set X is a *unique column combination* (UCC) if no tuple pair agrees on X, i. e., X uniquely identifies each tuple. A *functional dependency* (FD) $X \rightarrow Y$ holds on relation r if for all tuple pairs $t_1, t_2 \in r$: $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$. An *order dependency* (OD) $[X] \mapsto [Y]$ holds if ordering by X implies that Y is ordered [46], e. g., $[date] \mapsto [year]$.

Relational operators. We use standard notation. Selection $\sigma_C(r) = \{t \in r \mid C(t)\}$ returns all tuples of r satisfying predicate C [47, p. 56]. Projection $\pi_E(r)$ computes expressions E for each tuple, not limited to attribute names [14, p. 217f]. Sort $\tau_O(r)$ orders tuples by attribute list O [14, p. 219]. Limit $\lambda_k(r)$ returns the first k tuples [24]. Group-by $\gamma_{G,f}(r)$ partitions r by attributes G and computes aggregate f over each group, returning one tuple per group.

For join variations, we adopt [14, pp. 45–58, 220f] as follows: inner join is $r \bowtie_C s = \sigma_C(r \times s) = \{t \circ u \mid t \in r \wedge u \in s \wedge C(t \circ u)\}$. Semi-join is $r \ltimes_C s = \{t \in r \mid \exists u \in s : C(t \circ u)\}$, returning tuples from r that have at least one match in s. Anti-join is $r \triangleright_C s = \{t \in r \mid \nexists u \in s : C(t \circ u)\}$, returning tuples from r with no match in s; tuples with NULL values in the join key are preserved. Left outer join is $r \mathbin{⟕}_C s = (r \bowtie_C s) \cup \{t \circ [\bot, \ldots] \mid t \in r \triangleright_C s\}$, which preserves all tuples from r, padding with NULLs when no match exists. Right outer ($⟖$) and full outer ($⟗$) joins are defined analogously.

Comparison semantics. Lexicographical comparisons $\preccurlyeq$ in the context of partitioning, grouping, and frame peer determination follow SQL's *syntactic equality* semantics, under which NULL values are treated as equal to each other [18, 25]. Thus, NULL = NULL, NULL ≤ NULL, and NULL ≥ NULL all evaluate to true in these contexts, matching the SQL standard. Furthermore, it can be specified whether NULLs are sorted first or last in the result.

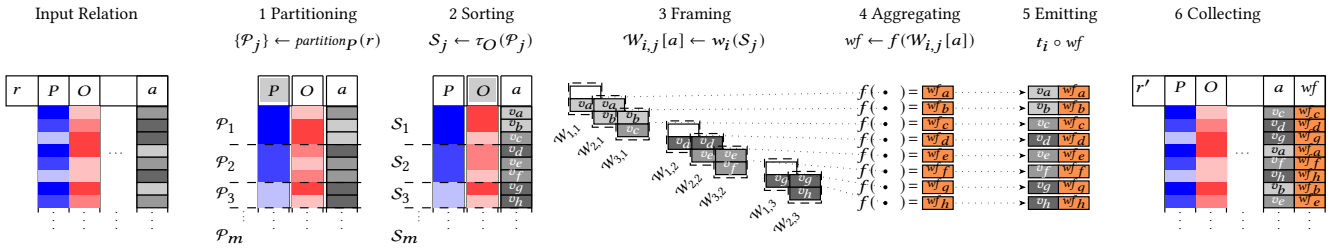

Figure 1: Phases of a window function operator: from input (left) to outputting a WF column (right).

2.2 The Window Function Operator

We denote the window function operator by $\boxplus_{P,O,w,f}(r)$, where P is a set of partition attributes, O is a list of order attributes, w is a frame function, and f is an aggregate function.[1] The window function operator is defined as:

$$\boxplus_{P,O,w,f}(r) \;:=\; \bigcup_{j=1}^{m} \left\{ t_i \circ f(\mathcal{W}_{i,j}[a]) \;\middle|\; i \in \{1,\ldots,|\mathcal{S}_j|\} \right\}$$

where

$$\begin{aligned} \mathcal{W}_{i,j} &:= \mathcal{S}_j[w_i(\mathcal{S}_j)] && \text{(framing)} \\ \mathcal{S}_j &:= \tau_O(\mathcal{P}_j) && \text{(sorting)} \\ \{\mathcal{P}_1,\ldots,\mathcal{P}_m\} &:= partition_P(r) && \text{(partitioning)} \end{aligned}$$

Informally, the operator transforms relation r through a sequence of phases, as depicted by Figure 1. First, tuples are grouped into m partitions $\mathcal{P}_1,\ldots,\mathcal{P}_m$ based on P. Each partition $\mathcal{P}_j$ is sorted by O, producing an ordered sequence $\mathcal{S}_j$. For each tuple t_i in sequence $\mathcal{S}_j$, a frame function w builds the window frame for that tuple, $\mathcal{W}_{i,j}$, and an aggregate function f computes the window function result, wf, over that frame. As a result, the original tuple is extended with the computed value. Finally, results from all partitions are collected back into a (bag) relation. We now define each phase formally.

Phase 1: Partitioning. $partition_P(r) = \{\mathcal{P}_1,\ldots,\mathcal{P}_m\}$, where m is the number of distinct values of P in r, $\mathcal{P}_j \cap \mathcal{P}_k = \emptyset$ for $j \neq k$, every tuple $t \in r$ belongs to exactly one $\mathcal{P}_j$, and $\forall t, t' \in \mathcal{P}_j : t[P] = t'[P]$. The standard treats P as a list of attributes, but in practice, engines allow expressions over these attributes. We assume the latter.

Phase 2: Sorting. For each $\mathcal{P}_j$, $\tau_O(\mathcal{P}_j) = \mathcal{S}_j = \langle t_1,\ldots,t_n\rangle$, where $\{t_1,\ldots,t_n\} = \mathcal{P}_j$ and $\forall i \in \{1,\ldots,n-1\} : t_i[O] \preccurlyeq t_{i+1}[O]$. When O does not uniquely determine order, multiple valid sequences exist; all equivalences in this paper hold for every valid ordering.

Phase 3: Framing. For each position i in $\mathcal{S}_j$, $w_i(\mathcal{S}_j)$ is a sequence of indexes $\langle k_1, k_2, \ldots\rangle$ into $\mathcal{S}_j$ that reduces the partition to the elements that should be considered for aggregation. The frame contents for t_i are $\mathcal{W}_{i,j} = \mathcal{S}_j[w_i(\mathcal{S}_j)] = \langle t_{k_1}, t_{k_2}, \ldots\rangle$. If an exclusion modifier is specified, it filters the positions accordingly.

Phase 4: Aggregating. $f : \langle t_{k_1}, t_{k_2}, \ldots\rangle \to wf$, a function that computes a value wf from $\mathcal{W}_{i,j}[a]$.

Phase 5: Emitting. For each tuple t_i in $\mathcal{S}_j$, emit $t_i \circ f(\mathcal{W}_{i,j}[a])$, the original tuple extended with the computed value.

[1]The aggregate operates on attribute a, which we keep implicit in the same way that aggregation $\gamma_{G,f}(r)$ does. The $\boxplus$ symbol is produced by \boxplus in LaTeX.

Table 1: Selected notation for window function components.

Symbol	Description
$X \to Y$	Functional dependency over attribute lists X and Y
$[X] \mapsto [Y]$	Order dependency: ordering by X implies ordering by Y
$\mathcal{S}$	Sequence of tuples
$\lvert\mathcal{S}\rvert$	Length of sequence $\mathcal{S}$
$\lvert\lvert\mathcal{S}\rvert\rvert$	Number of distinct elements in $\mathcal{S}$
$\mathcal{S}[k]$	Element at position k (1-indexed)
$\mathcal{S}[l:u]$	Subsequence from position l to u (inclusive)
$\langle k_1,\ldots,k_n\rangle$	Elements of index sequence $\mathcal{I}$
$\langle k : C(k)\rangle$	Sequence of indexes k satisfying condition C
$\mathcal{S}[\mathcal{I}]$	Subsequence at positions in index sequence $\mathcal{I}$
$\mathcal{S} - \mathcal{I}$	Sequence $\mathcal{S}$ with positions in $\mathcal{I}$ removed
$\mathcal{S}_1 \circ \mathcal{S}_2$	Concatenation of sequences
$\boxplus_{P,O,w,f}(r)$	Window function operator
P	Partition attributes
O	Order attributes (with direction, NULL ordering)
w	Frame function
f	Aggregate or ranking function over implicit attribute a
$\{\mathcal{P}_1,\ldots,\mathcal{P}_m\}$	Set of partitions of a relation
$\mathcal{S}_j$	Sorted sequence for partition $\mathcal{P}_j$
$\mathcal{W}_{i,j}$	Frame contents for tuple i in partition j

Phase 6: Collecting. The result is $\bigcup_{j=1}^{m}\{t_i \circ f(\mathcal{W}_{i,j}[a]) \mid i \in \{1,\ldots,|\mathcal{S}_j|\}\}$. The ordering imposed by phase 2 is not preserved; the output is a bag with no ordering guarantees.

2.3 From SQL to the Window Function Operator

SQL's `OVER()` clause [18] translates directly into the operator's four parameters:

```
f(a) OVER (              → f, a
  PARTITION BY ...       → P
  ORDER BY ...           → O
  frame_clause           → w
)
```

Obtaining P, f, and a from SQL is straightforward. The order list O, however, encodes not just attributes but also sort direction and `NULL` ordering (`ASC`/`DESC`, `NULLS FIRST`/`LAST`). The frame function w captures the richest semantics: frame mode (`ROWS`, `RANGE`, `GROUPS`), boundaries, and exclusion. Translating SQL's various frame clauses into our sequence notation requires care. Table 2 illustrates the mapping that generalizes to all cases. Note that the SQL standard allows only a single order attribute for `RANGE` mode.

Table 2: Frame clauses to frame functions examples.

Frame clause $\rightarrow w_i(\mathcal{S})$
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` $\rightarrow \langle 1, \ldots, i \rangle$
`ROWS BETWEEN` n `PRECEDING AND` m `FOLLOWING` $\rightarrow \langle \max(1, i-n), \ldots, \min(\lvert\mathcal{S}\rvert, i+m) \rangle$
`GROUPS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` $\rightarrow \langle k : k \leq i \vee \mathcal{S}[k][O] = \mathcal{S}[i][O] \rangle$
`GROUPS BETWEEN` n `PRECEDING AND` m `FOLLOWING` $\rightarrow \langle k : 0 < \lvert\lvert\mathcal{S}[k:i][O]\rvert\rvert \leq n+1 \vee 0 < \lvert\lvert\mathcal{S}[i:k][O]\rvert\rvert \leq m+1 \rangle$
`RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` $\rightarrow \langle k : \mathcal{S}[k][O] \leq \mathcal{S}[i][O] \rangle$
`RANGE BETWEEN` u `PRECEDING AND` v `FOLLOWING` $\rightarrow \langle k : 0 \leq \mathcal{S}[i][O] - \mathcal{S}[k][O] \leq u \vee 0 \leq \mathcal{S}[k][O] - \mathcal{S}[i][O] \leq v \rangle$
Exclusion $\rightarrow$ **filter on** w
`EXCLUDE CURRENT ROW` $\rightarrow w_i(\mathcal{S}) - \langle i \rangle$
`EXCLUDE GROUP` $\rightarrow w_i(\mathcal{S}) - \langle k : \mathcal{S}[k][O] = \mathcal{S}[i][O] \rangle$
`EXCLUDE TIES` $\rightarrow w_i(\mathcal{S}) - \langle k : k \neq i \wedge \mathcal{S}[k][O] = \mathcal{S}[i][O] \rangle$

3 ALGEBRAIC EQUIVALENCES

Now that we have established a formal definition of the WF operator, we can begin developing algebraic equivalences for it. The idea is familiar from other operators: transform a relational expression into one that (hopefully) executes faster while producing the same results. We make no claim to completeness but believe that the equivalences we present cover enough diverse cases to show applicability, and that the examples are representative. We hope this framework will invite further research.

3.1 Equivalence Classes

We divide equivalences into four classes based on the techniques they use, as Table 3 shows. We summarize them here; later sections develop each in detail.

- *Reductions* target the partition and order clauses. When functional dependencies hold, some attributes in these clauses contribute nothing, because they are constant within the scope where they would be compared. Removing them shrinks the partition or sort keys, with no change to the result.
- *Substitutions* apply when the window falls into some special cases, e. g., a frame has one or zero tuples or it takes the entire partition. If the frame contains at most one tuple, the result depends only on the current row, and a projection suffices. If the frame spans the entire partition, a `GROUP BY` and a join can replace the window.
- *Push-downs* swap selections or joins with the window. The key observation here is that a window requires complete partitions to compute correctly. An operator can be pushed only if it removes entire (non-relevant) partitions or leaves them intact. We call this property partition integrity. Some functional dependencies and uniqueness constraints can help determine when it holds.
- *Limit-like optimizations* address predicates that depend on the window's output. Such predicates cannot be pushed, but some exhibit a useful structure: once false, they stay false for subsequent tuples in partition order. This monotonicity implies a bound on rows that can qualify, enabling aggressive early pruning.

The classes differ in how they relate to cost. Reductions (Equivalences 1 to 3), projection substitution (Equivalence 6), full-relation substitution (Equivalence 4), and limit elimination (Equivalence 13) never produce a slower plan, so an optimizer can apply them as unconditional rules. Predicate and reducing-join push-downs (Equivalences 7, 8, 11, and 12) only shrink the window's input, so they are also safe to apply directly. An optimizer can apply these rules during normalization instead of treating them as alternatives that require enumeration and costing, thus avoiding any increase of the cost-based search space. The remaining equivalences (Equivalences 5, 9, 10, and 14) are cost-dependent, and we identify their cost drivers where each is developed. We treat reductions and some substitutions in the next section, as they follow directly from the definitions in Section 2. The remaining equivalences require the analysis techniques we present in more detail in Sections 4 to 6.

3.2 Reductions

The partition and order clause lengths have some impact on the work done by a window operator. Fewer partition attributes mean a narrower key to hash and compare. The same applies to sorting. FDs offer an opportunity here: when they make a partition or sort attribute redundant, we can remove it without affecting the result.

Consider Equivalence 1. If the remaining partition attributes $P \setminus \{p\}$ functionally determine p, then tuples that agree on $P \setminus \{p\}$ must also agree on p. The partition boundaries are identical with or without p, so any such attributes can be dropped. This reduction affects only phase 1 in Figure 1. As it delivers the same results for that phase, it can be safely applied to the WF.

A similar line of reasoning supports Equivalences 2 and 3. Recall that sorting is lexicographic for compound attributes: attribute o_k is compared only among tuples that already agree on $o_1, \ldots, o_{k-1}$. If an FD makes o_k constant within such groups, it contributes nothing to the comparison and can be removed (Equivalence 2). Similarly, when the partition attributes determine the first ordering attribute, that attribute is constant within every partition and can be dropped (Equivalence 3). These optimizations affect only phase 2 in Figure 1 without altering the results of the sort or affecting other phases. As we hinted above, we expect this class to contain many more equivalences and dependency types than we show here.

Implementation note. Uncovering *FD-based redundancies* can be done at optimization time. For example, if attributes a and b are used together in a partitioning or sorting clause, but the query applies an equality filter to b (turning it into a constant), the FD $\{a, b\} \setminus \{b\} \rightarrow b$ holds [20].

3.3 General Substitutions

Some windows reduce to trivial cases and can be replaced by simpler operators. This can occur when the frame covers entire partitions, a case we address now, or when it covers one row or none, which we defer to Frame Analysis (Section 4).

Table 3: Algebraic equivalences for window function operators.

Class	Subclass/Equivalences	Preconditions and Notes
Attribute reductions (Section 3.2)	Reduce partition by attributes (1) $\boxplus_{P,\ldots}(r) \equiv \boxplus_{P\setminus\{p\},\ldots}$	FD $P \setminus \{p\} \to p$
	Reduce order by attributes (2) $\boxplus_{P,O,\ldots}(r) \equiv \boxplus_{P,O[1:k-1]\circ O[k+1:n],\ldots}(r)$ (3) $\boxplus_{P,O,\ldots}(r) \equiv \boxplus_{P,O[2:n],\ldots}(r)$	FD $\{o_1,\ldots,o_{k-1}\} \to o_k$ FD $P \to o_1$
Operator substitutions (Sections 3.3 and 4)	Replace with group by (4) $\boxplus_{\varnothing,\ldots,f}(r) \equiv r \times \gamma_f(r)$ (5) $\boxplus_{P,\ldots,f}(r) \equiv r \bowtie_{P=P} \gamma_{P,f}(r)$	Equivalence per definition (Section 2.1) if f is standard aggregate function and frame covers entire partition.
	Replace with projection (Frame Analysis) (6) $\boxplus_{\ldots}(r) \equiv \pi_{\mathcal{A}(r)\circ e}(r)$	All windows contain either only the current or no tuple.
Push-downs (Partition Analysis, Section 5)	Predicate push-down (7) $\sigma_{b\theta c}(\boxplus_{P,\ldots}(r)) \equiv \boxplus_{P,\ldots}(\sigma_{b\theta c}(r))$ (8) $\sigma_{b\theta c}(\boxplus_{P,\ldots}(r)) \equiv \sigma_{b\theta c}(\boxplus_{P,\ldots}(\sigma_{p\theta c'}(r)))$	FD $P \to b$ FD $\{b\} \to p \in P, c' \in \pi_p(r), b\ \theta\ c \Rightarrow p\ \theta\ c'$
	Join push-down (9) $\boxplus_{P,\ldots}(r) \bowtie_{r.b=s.x} s \equiv \boxplus_{P,\ldots}(r \bowtie_{r.b=s.x} s)$ (10) $\boxplus_{P,\ldots}(r) \rtimes_{r.b=s.x} s \equiv \boxplus_{P,\ldots}(r \rtimes_{r.b=s.x} s)$ (11) $\boxplus_{P,\ldots}(r) \ltimes_{r.b=s.x} s \equiv \boxplus_{P,\ldots}(r \ltimes_{r.b=s.x} s)$ (12) $\boxplus_{P,\ldots}(r) \triangleright_{r.b=s.x} s \equiv \boxplus_{P,\ldots}(r \triangleright_{r.b=s.x} s)$	P, O, w, f reference only $\mathcal{A}(r)$ and: FD $P \to b$ and x unique x unique FD $P \to b$ FD $P \to b$
Limit-like optimizations (Section 6)	Predicate elimination (13) $\sigma_{f\le c}(\boxplus_{\varnothing,O,w,f}(r)) \equiv \lambda_c(\tau_O(\boxplus_{\varnothing,O,w,f}(r)))$	(i) f = `row_number()` or (ii) $f \in \{$`rank()`, `dense_rank()`$\}$ and O is UCC. For a pure sort-based WF operator, no extra sort τ_O required.
	Predicate co-evaluation (14) $\sigma_C(\boxplus_{P,\ldots}(r)) \equiv \boxplus^{+}_{P,\ldots,C}(r)$	C is any predicate; only MOD predicates benefit from early pruning.

In Equivalence 4, the partition clause is missing, which is equivalent to eliminating phase 1 in Figure 1. More simply put, $m = 1$. When f is an aggregate function (as opposed to a ranking function), and the frame covers the entire partition, the WF computes a single aggregate over the entire relation and attaches it to every row. This is exactly what a Cartesian product with a scalar aggregation produces, including producing results for NULL values.

In Equivalence 5, a partition clause is present, so each partition produces its own aggregation result. A similar optimization still applies, but the result of the aggregation must be joined back to the original tuples on the partition key. Once again, the join uses syntactic equality (see Section 2.1), matching SQL's semantics where NULL values are placed into the same partition. This optimization is a bit nuanced, so we offer reasoning about its correctness next.

Proof sketch for Equivalence 5. When $w_i(\mathcal{S}_j) = \langle 1,\ldots,|\mathcal{S}_j|\rangle$ for all i, the frame spans the entire partition. By phase 4, aggregate f receives all tuples sharing the same P-value and produces a single value for each partition. The aggregation $\gamma_{P,f}(r)$ computes the same aggregate per P-value and returns one tuple per group. Joining r with $\gamma_{P,f}(r)$ on P replicates each aggregate to the original tuples. For correctness, the join must treat NULLs in P as equal, matching the partitioning semantics. □

Unlike reductions, this substitution is cost-dependent. The aggregation and the join on the partition key can cost more or less than the replaced window, depending on the number of partitions and their cardinalities. An engine should compare both versions.

4 FRAME ANALYSIS

We begin with a motivating example. Recall Q1 from the introduction. The frame clause is `ROWS BETWEEN CURRENT ROW AND CURRENT ROW`, which contains exactly one row. The window function `SUM(a)` applied to a single row simply returns `a`'s value. Frame Analysis recognizes this and rewrites the query with Equivalence 6:

Q1′: `SELECT *, a FROM r`

To use this equivalence, we must determine (a) how to recognize that a frame is singleton or empty, and (b) what expression replaces the window function. The first case is clear from syntax, but others require careful Frame Analysis. We first address (a), as it is a precondition to applying Equivalence 6. Then, we discuss (b), also needed in that equivalence.

4.1 Detecting Singleton or Empty Frames

A frame is guaranteed to contain only the current row under at least one of four alternative conditions:

(i) Frame is `ROWS BETWEEN CURRENT ROW AND CURRENT ROW` or `ROWS BETWEEN 0 PRECEDING AND 0 FOLLOWING`. By Table 2, $w_i(\mathcal{S}_j) = \langle \max(1, i-0),\ldots,\min(|\mathcal{S}_j|, i+0)\rangle = \langle i\rangle$.

(ii) Frame is `RANGE/GROUPS BETWEEN CURRENT ROW AND CURRENT ROW` with `EXCLUDE TIES`. By Table 2, $w_i(\mathcal{S}_j) = \langle k : \mathcal{S}_j[k][O] = \mathcal{S}_j[i][O]\rangle$, the set of peers. `EXCLUDE TIES` removes all peers except the current row, leaving $\langle i\rangle$.

(iii) Frame is `RANGE/GROUPS BETWEEN CURRENT ROW AND CURRENT ROW` and O is a unique column combination. The peer sequence is $\langle k : \mathcal{S}_j[k][O] = \mathcal{S}_j[i][O] \rangle$. Since O is unique, only $k = i$ satisfies this, so $w_i(\mathcal{S}_j) = \langle i \rangle$.

(iv) P is a unique column combination and the frame does not exclude the current row. Since P uniquely identifies each tuple, each partition contains exactly one tuple. Therefore, $|\mathcal{S}_j| = 1$ and $w_i(\mathcal{S}_j) = \langle 1 \rangle$.

Empty frames occur when any of (i)–(iv) holds but the frame additionally excludes the current row, yielding $w_i(\mathcal{S}_j) = \langle\rangle$.

4.2 Eliminating the Window Function

Once the optimizer establishes that $|w_i(\mathcal{S}_j)| \leq 1$ for all i, j, i. e., every frame contains at most one tuple, it must derive the scalar expression e that replaces the window function f. This derivation depends on whether f operates on the frame or the partition.

Aggregate functions operate on the frame, but ranking and distribution functions, such as `row_number()`, `rank()`, `dense_rank()`, `percent_rank()`, and `cume_dist()`, operate on the entire partition. In our example query Q1, if a partition $\mathcal{P}_j$ had 10 rows, `row_number()` still sees all 10 rows and returns values 1 through 10. Thus, ranking and distribution functions reduce to constants only when the partition itself contains a single row, i. e., when condition (iv) holds. Table 4 summarizes the replacement expressions and their applicability conditions. The cases with at most one tuple follow ordinary aggregation semantics, e. g., `count` of an empty frame is 0. The only exception is `count(a)` on a singleton frame and a nullable column, which needs `CASE WHEN a IS NULL THEN 0 ELSE 1 END`.

Table 4: Expression e replacing WF-specific functions f for collapsing frames.

$\lvert\mathcal{W}\rvert$	Function f	Expression e	Conditions
0	`count(a), count(*)`	0	any of (i)–(iv)
0	any other	NULL	any of (i)–(iv)
1	`first_value(a), last_value(a)`	a	any of (i)–(iv)
1	`nth_value(a, 1)`	a	any of (i)–(iv)
1	`nth_value(a, n)`, $n \neq 1$	NULL	any of (i)–(iv)
1	`count(a)`, a nullable	case expr.	any of (i)–(iv)
1	`rank(), dense_rank(), row_number()`	1	(iv)
1	`cume_dist()`	1	(iv)
1	`percent_rank()`	0	(iv)

Recall that condition (iv) implies single-row partitions. Ranking functions assign rank 1 to the sole row. The cumulative distribution function `cume_dist()` returns $1/1 = 1$. Function `percent_rank()` is defined as $(rank - 1)/(partition\ size - 1)$; with one row, this is $0/0$, which the SQL standard defines as 0.

Implementation note. Recognizing collapsing frames can already occur at parse time because the conditions follow directly from translating the SQL frame clause to the internal representation. Determining the replacement expression is a switch on f's type or a lookup in a table like Table 4.

5 PARTITION ANALYSIS

This section presents another reasoning technique that allows common optimizations to apply more broadly than previously assumed. We start again with a motivating example. Recall Q2 from the introduction. The query computes a yearly average and filters to recent sales. The predicate `sold_date >= DATE'2020-02-15'` operates on the same column used in the partition expression `EXTRACT(YEAR FROM sold_date)`. One might hope to push the predicate through the window, but doing so directly would remove rows needed for the yearly average computation. What can be done instead is pushing a *derived predicate*, shown in gray in the following query:

```
Q2′: SELECT * FROM (
       SELECT *, AVG(price) OVER (
           PARTITION BY EXTRACT(YEAR FROM sold_date)
       ) AS yearly_avg
       FROM sales
         WHERE EXTRACT(YEAR FROM sold_date) >= 2020) AS s
     WHERE sold_date >= DATE '2020-02-15'
```

This transformation uses Equivalence 8. Before describing when and how it derives such a predicate, we introduce the concept of partition integrity and work through simpler push-downs.

5.1 Partition Integrity

A window operator requires complete partitions to compute correctly. Recall from Section 2.2 that partitions form in phase 1 and results collect in phase 6. If an operator applied before phase 1 removes only some tuples from a partition, the window function computes over incomplete data, producing incorrect results.

The solution idea is simple: partitions must remain intact. Functional dependencies can help determine when a predicate or join respects this constraint. If an FD holds between the partition attributes P and the filter or join attribute b, then all tuples in a partition share the same value for b. A predicate on b therefore either accepts every tuple in the partition or rejects all of them. The same reasoning applies to join attributes. Such FDs are within reach of a modern optimizer: they follow from declared keys and uniqueness constraints, and can also be derived at optimization time, for instance, when an equality filter makes an attribute constant. Several optimizers already maintain functional dependencies for this kind of reasoning, as we discuss in Section 9.

The FD can run in two directions. When $P \to b$, the partition determines the filter attribute, and we can push the predicate or join directly. When $\{b\} \to p$ for any $p \in P$, the filter attribute determines a partition attribute. We discuss each case in turn.

5.2 Partition Determines Filter Attribute

Equivalences 7 and 9 to 12 require the FD $P \to b$. When it holds, pushing the predicate directly preserves partition integrity. When P and b are arbitrary expressions, checking whether this FD holds requires reasoning about expression semantics. In the general case, this is undecidable [41]. However, we can identify a reduced subset of patterns for which the check is tractable:

(i) $b \in P$: we expect most systems to use this variant already.

(ii) Deterministic functions $e(p), p \in P$: examples are arithmetic expressions with a constant, $(p + c; p \bmod c; \ldots)$ or functions on p, such as substring or date component extraction.

(iii) Hierarchical data types, such as dates and years, if the hierarchy is known: equi-joined attributes also apply because primary keys induce transitive FDs.

An example of (ii) above would be if we inverted the filtering and partitioning expressions in Q2′, i. e., we partition by `sold_date` but filter by year. With the FD {date} → {year}, we could push down a predicate over year because it eliminates entire partitions.

The join push-downs differ in cost. Semi- and anti-joins (Equivalences 11 and 12) only remove tuples, so pushing them never increases the window's input. Inner and outer joins (Equivalences 9 and 10) can expand it, so an engine should push them only when the join does not grow the input it feeds the window.

5.3 Filter Attribute Determines Partition

Equivalence 8 uses the FD in the other direction: $\{b\} \rightarrow p$ for some $p \in P$. Unlike the previous case, we cannot push the original predicate directly. Tuples in the same partition may have different values for b, so pushing would break partition integrity.

The key observation from Q2′ is that a bound on dates implies a bound on years. Any date satisfying `sold_date >= DATE '2020-02-15'` must fall in year 2020 or later. We can therefore derive a weaker predicate, `EXTRACT(YEAR FROM sold_date) >= 2020`, and push it instead. We call the new constant in the derived predicate c'. This eliminates entire partitions—years 2019, 2018, and earlier—before the window operator runs. The original predicate remains above to filter tuples within the year-2020 partition.

Deriving c' depends on the partition expression. These three approaches cover common cases:

(i) For equality predicates and deterministic functions, such as `EXTRACT`, apply the partition function e to c directly, as in our example.
(ii) For equality predicates and hierarchical data types (dates, times, etc.), use the hierarchy to derive the coarser bound.
(iii) Use an order dependency $[b] \mapsto [e(b)]$ when the mapping preserves order for range predicates. Examples of such dependencies are, again, date and time component extractions or monotonic arithmetic expressions with a constant (addition, subtraction, multiplication, division, but not modulo). Exclusive range predicates ($<, >$) must be transformed to their inclusive equivalent to preserve partition integrity.

Implementation Note. Using (i) above, an engine would need to evaluate an expression $c' = e(c)$ dynamically during optimization time. We expect, however, that e would already be parsed, and therefore, the evaluation to be relatively fast because no table data needs to be accessed. Additional machinery would be required to inspect expression e and recognize, from its shape, that it contains a function covered by case (i) above.

6 CO-EVALUATION

Partition Analysis covered predicates that can be pushed through the window operator. Some predicates, however, depend on the window's output and cannot be moved as above. Rather than evaluating these predicates after the window completes, we can push aspects of them into the window's internal execution, reducing data early. As we mentioned above, we call this technique Co-Evaluation.

6.1 Motivation

Equivalence 13 shows that a predicate on a ranking function can become a `LIMIT` when the window has no partitions and the sort list is a unique column combination. Here is an example of a query that can benefit, and its restructured version:

Q4:
```
SELECT * FROM (
    SELECT *, RANK() OVER (
        ORDER BY <UCC expr> DESC) AS rnk
    FROM r) AS s
WHERE rnk <= 3
```

Q4′:
```
SELECT * FROM (
    SELECT *, RANK() OVER (
            ORDER BY <UCC expr> DESC)
    FROM r)
~~ORDER BY <UCC expr> DESC~~
LIMIT 3
```
push into the window

The query ranks rows by a unique expression and keeps only the top three. This pattern, ranking followed by a threshold filter, is common in analytics, for example when finding best-performing items or most recent entries. The restructured version replaces the rank filter with a `LIMIT`. This works because no ties can occur without partitioning and a unique ordering. The first three rows in sort order are exactly those with rank at most three. The proof sketch makes this precise.

Proof sketch for Equivalence 13. With $P = \emptyset$, all tuples belong to a single partition ordered by O (phase 2). Function `row_number()` assigns increasing integers along this order. Functions `rank()` and `dense_rank()` do so as well when O is a unique column combination. Filtering to ranks at most c therefore selects exactly the first c tuples in sort order, which is equivalent to $\lambda_c(\tau_O(\cdot))$. □

Since the sort is duplicated in the restructured version, an optimizer could conceivably push the LIMIT from the outer sort into the window's internal sort, pruning data early in the query and eliminating the outer sort entirely.

This is a powerful optimization. However, it requires the WF to not have partitioning. Our motivation query Q3 partitions over `dept`, so Equivalence 13 does not apply. This is where the notion of Co-Evaluation may help.

6.2 MOD Predicates

Any valid predicate over a window function can be co-evaluated while the function result is computed. Rather than adding another pass on the data to calculate the predicate, an engine can do so in phase 4 of the WF definition (Section 2.2). This will save the work of going over the data once more after the WF calculation finishes.

Some predicates, however, present a *stopping property*. Recall that phase 2 of the WF definition (Section 2.2) sorts each partition into a sequence $\mathcal{S}_j$. With respect to this order, a predicate C over a window function may, once it becomes false, stay false throughout the rest of the sequence. More formally:

$$\forall i \in \{1, \ldots, |\mathcal{S}_j| - 1\} : \neg C(t_i) \Rightarrow \neg C(t_{i+1})$$

We call a predicate that presents this property *monotonic order-dependent* (MOD). In Q3, the `rnk <= 3` predicate is MOD. A running sum over non-negative values with a cumulative frame (e. g., `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`) is also MOD. Once the predicate `SUM(a) OVER (...) < threshold` becomes

false, it remains false for the rest of S_j. The same intuition holds for the time interval elapsed since a fixed point. While a between predicate on the WF result is not strictly MOD, the predicate's upper bound still implies a stop criterion that can be used for pruning. In the case study we mentioned in Section 1, 8 of the 10 most expensive WF queries featured a MOD predicate.

6.3 Pruning in Early Phases of a WF

Equivalence 14 enables two optimizations based on MOD predicates. The first is *early stop*, where we can iterate through the partition in sort order only until the predicate turns false. For predicates that tend to turn false after just a few elements, such as rank with low k, this can save many iterations. This raises a question: why build and sort a full partition if we will not traverse it completely?

The alternative is to prune rows that cannot qualify even before sorting. We call this *early pruning*. In our example, rows that cannot rank in the top k can be discarded during the partitioning process, as we show shortly.

Early pruning can deliver substantial performance improvements by reducing the data that reaches the sort. From a complexity standpoint, if n tuples are distributed across m partitions, sorting costs $\mathcal{O}(\frac{n}{m}\log\frac{n}{m})$ per partition. Conceptually, maintaining a priority queue of size k for each partition suffices to answer the query without sorting when k is small. This costs $\mathcal{O}(\frac{n}{m}\log k)$ per partition, or $\mathcal{O}(n\log k)$ overall. When $k \ll \frac{n}{m}$ (as is typical for top-k queries), the savings are substantial. For `rank()` and `dense_rank()`, the queues must handle duplicates and have a size $k' \geq k$, unless a UCC on the ordering attributes ensures no ties occur.

Applying these ideas to Q3 changes its WF execution as shown below, with the clauses involved in pruning highlighted:

```
Q3′:  SELECT * FROM (
          SELECT *, RANK() OVER (
                  PARTITION BY dept
                  ORDER BY salary DESC) AS rnk
          FROM employees) AS s
      WHERE rnk <= 3
```

prune to top-k per partition

MOD predicate, $k = 3$

Implementation note. Recognizing MOD predicates can occur at optimization time by inspecting the predicate structure and the window function type. Applying the optimizations, however, requires operator modifications. We call such operators WF$^+$ and describe one implementation next.

7 WF$^+$ IMPLEMENTATION

The phases in Section 2 reflect how window functions may be implemented in commercial systems. Changing some of these phases to perform early pruning can, therefore, transform an existing operator into a WF$^+$ one. To show how, we integrated our techniques into DuckDB, a modern engine geared towards analytical queries [36] and contrast its native execution and our changes in Figure 2.

As the top of the figure shows, DuckDB's native WF is a parallel implementation that follows Leis et al. [23]: (A) local partitioning distributes tuples from morsels (*chunks*) across local radix buckets by hashing each tuple's partition key, so a single window partition may span several buckets and a bucket may hold several partitions, (B) local sort orders tuples within each worker's buckets, (C) global multi-way merge combines sorted runs [21, 22] so that workers

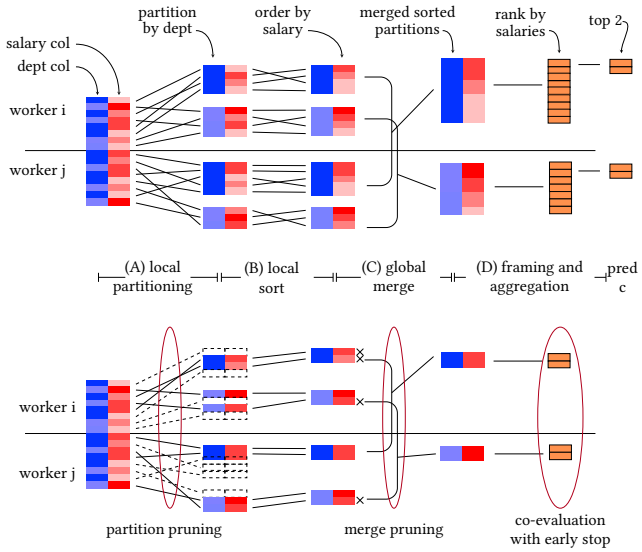

Figure 2: DuckDB's WF pipeline for Q3. We use `rnk<=2` for the purpose of this illustration. Top: standard execution. Bottom: WF$^+$ with pruning at different phases. The benefits of WF$^+$'s pruning points are cumulative, as early pruning reduces the work of all subsequent phases.

scale without sorting the entire relation at once, and (D) framing/aggregation computes the results, using a segment tree to share aggregation work across overlapping frames. The bottom part of the figure depicts three pruning points where WF$^+$ techniques can reduce data volume. Note that WF$^+$ does not exit or skip any of these phases. As Figure 2 shows, every phase still runs, and the pruning points only reduce the data volume that reaches subsequent phases.

7.1 Pruning Points

We next describe these optimizations, going from late to early steps.

Co-Evaluation with Early Stop. We implement Co-Evaluation on the framing and aggregation step. It decides if a tuple matches a predicate while iterating the materialized chunks generated by the merge phase. This implementation can be used for any predicate, but it is most beneficial when the predicate involves the WF result. If the co-evaluated predicate is MOD, we skip the remaining partition elements once the predicate turns false. As mentioned above, using MOD predicates to prune on or before the sort yields better results.

Merge Pruning. It is possible to take MOD predicates into consideration during the merge step depicted in Figure 2. In our implementation, we extend the sorted-batch merger to do so. It stops reading tuples from (initial and merged) sorted runs beyond their rank-k element for each partition. We only consider ranking, but this strategy can conceptually generalize to further MOD predicates at higher evaluation cost. Our merger takes k as input, but that is the extent of state added at this pruning point. Merge pruning does not add any overhead other than additional key comparisons. As expected, materialization, framing, and the WF evaluation benefit from less data read from the result of this step.

Partition Pruning. This is the earliest pruning point. Note that a bucket at this point does not observe all of the query's partition key values. A partition may be scattered across buckets of several workers (as Figure 2 shows), or a bucket may contain several partitions (not shown). In our implementation, we dynamically create heaps of size k for each partition per worker. When a tuple gets assigned to a bucket, we identify its partition. If the ordering value is larger than the respective heap's maximum value, we disregard the tuple. Otherwise, we update the heap. For Q3, elements outside the top-3 salaries per department can get discarded here.

The benefits of pruning data this early can be significant. With m WF partitions, the sort phase reads only $m \cdot k$ instead of all n tuples in the best case, reducing sort complexity from $\mathcal{O}(n \cdot \log n)$ to $\mathcal{O}(m \cdot k \cdot \log(m \cdot k))$. This improvement is immense if there are only a few but large partitions and k is small. However, in the worst case, all partitions have too few tuples, and each worker reads fewer than k tuples per WF partition, but still inserts them into the heaps. The overhead of iterating over the tuples to filter them and maintaining k-sized heaps can grow to dominate the cost (see Section 8.2).

7.2 Adaptive Partition Pruning

To mitigate this case during execution, we introduce an adaptive partition pruning strategy. Its main idea is driven by the following question: *how many tuples are required to answer the query?* A query that filters for the top-k tuples of m partitions and processes more than $m \cdot k$ tuples can potentially discard excess ones. Put differently, if $m \cdot k \cdot \delta <$ |read tuples|, pruning could theoretically disregard tuples for $\delta \geq 1$. In practice, the overhead of the heap maintenance may still dominate the cost if δ is only marginally over 1. We have determined experimentally that δ should be 5 (see Section 8.4). This value seems high at first, but it is still a linear factor compared to linearithmic sorting, which Adaptive Partition Pruning can shorten.

The threshold, however, is only part of the problem. The workers we see in Figure 2 do not know m, the number of partitions, in advance. In fact, the precise m will not be known until the end of the query's execution. Thus, we estimate m *on a per-worker basis*. For each chunk, a worker scans all tuples and calculates a hash for the partition value before partitioning the data. Our WF$^+$ implementation uses that step to also update an instance of a HyperLogLog sketch [16]. The sketch can provide a good approximation m' with little overhead and no additional statistics or previous optimizer decision. With this up-to-date estimation, we decide whether to prune for each chunk individually and upfront by calculating $m' \cdot k \cdot \delta$ and comparing that value to the number of tuples the worker read so far.

This makes co-evaluation's cost a runtime decision rather than an optimizer one. Each worker decides per chunk from its HyperLogLog estimate and the threshold δ, so the operator captures the cost-dependent case at execution time without a static cost model.

8 EVALUATION

The experimental evaluation of the presented techniques is divided as follows: Section 8.1 evaluates the equivalences; Section 8.2 presents an ablation study of our WF$^+$ operator's pruning points; Section 8.3 expands that study to skew; Section 8.4 experimentally determines δ; and Section 8.5 analyses early pruning in detail.

Setup. We execute the microbenchmarks on one Intel Xeon Platinum 8180 socket with 378 GiB of NUMA-local memory. For stable results, we limited execution to the 28 physical cores. The machine is running Ubuntu 24.04, and we compile DuckDB and our WF$^+$ implementation over it with GCC-14.2 and `-O3`.

Methodology and Datasets. All our experiments execute actual queries in DuckDB unless stated otherwise. In some experiments, we have baseline and optimized queries to compare. In other experiments, we compare the official distribution of DuckDB with our modified one. The data we used in the experiments consists of synthetic relations varying in size, number of partitions, and skew. To eliminate any variation in response time, we execute each experiment from 10 to 1000 times and report the median runtime.

8.1 Equivalence Microbenchmarks

To evaluate the benefits of the analyses discussed in Sections 4 and 5, and to cover the full equivalence table, we execute a before-and-after query pair for each equivalence class in Table 3. The left-hand side of each pair presents the pattern on the left-hand side of an algebraic equivalence of the table, and the right-hand side applies that equivalence's rewrite, just as in Q1 vs. Q1′. We study three facets per equivalence, varying the number of rows in the base table, varying the number of partitions at a fixed table size, and varying the skew (α in a Zipf distribution) at fixed size and partition count. The results appear in Figure 3. Each equivalence employs a table specifically designed for its scenario, resulting in different sweep ranges for each column in the figure.

We observe that our variations match or outperform the original queries in every tested case. The frame-collapse equivalence (Equivalence 6) yields the largest gains, up to 40.7× on the largest table, because it removes the window operator entirely. The push-down equivalences improve as the number of partitions or the degree of skew grows because aligning the predicate with the partition expression discards entire partitions before the window runs. The remaining equivalences deliver consistent speedups by shrinking the sort and partition keys. In the few cases where an equivalence offers no benefit, its rewrite stays on par with the original query rather than degrading it.

8.2 Co-Evaluation and its Optimizations

To evaluate the benefits of the pruning techniques from Section 7, we measure the performance of our WF$^+$ implementation in isolation. We test each pruning technique individually and contrast it with DuckDB's unaltered WF implementation. We also experiment with combining all the techniques at once. Figure 4 shows multi-threaded (MT) and single-threaded (ST) runtimes for a query resembling Q3 on uniformly distributed partitions.

Co-Evaluation with Early Stop does not show significant improvements over the baseline. Though this technique reduces the tuples emitted by the operator, all tuples have been partitioned, sorted, merged, and materialized into chunks before. Thus, the optimization potential is small for DuckDB's execution model, highlighting the necessity for optimizations at earlier pipeline stages.

Merge Pruning shows consistent improvements as long as it can effectively reduce the number of tuples: it is around 1.2× to 1.6× faster than the baseline for large relations and few partitions. For

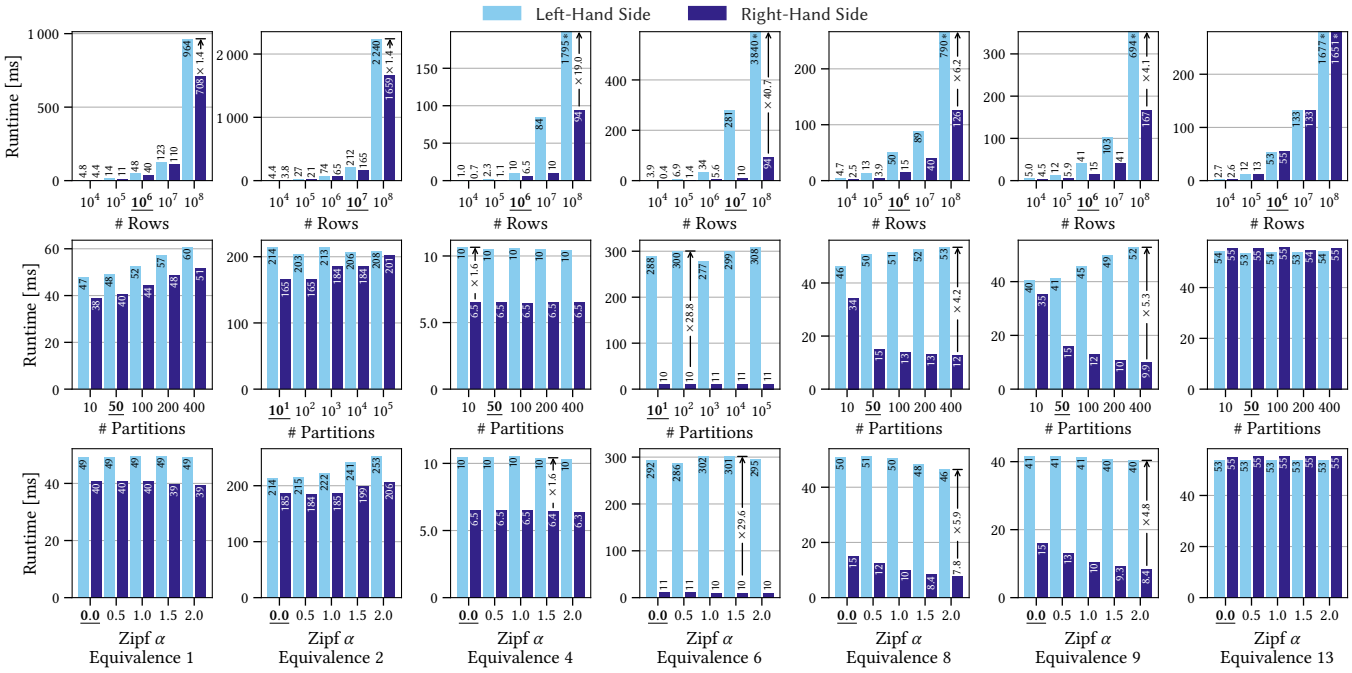

Figure 3: Runtime of each equivalence's left-hand side (native execution) vs. right-hand side (with our optimization), across varying table size, partition count, and skew. Each column corresponds to one equivalence class from Table 3. Within each column, parameter values that are fixed in the other charts are marked. Speedup factors are annotated where applicable.

the numerous small partitions case, the local sorted runs do not contain enough tuples per WF partition for pruning, and merge pruning can only reduce the tuples that are materialized, framed, and aggregated. This is the case for the right-most data point in the MT plots above 100 000 rows: DuckDB uses multiple workers that only see few (if any) of the ten tuples per partition. Here, the performance improvement diminishes.

Partition Pruning is extremely beneficial for large partitions, where many tuples can be aggressively discarded early. We observe a maximum 9× speedup for 100 million rows and 10 partitions. However, the bookkeeping overhead grows and dominates the benefit, at least when few or no tuples can be discarded from local partitions. In the worst cases, the performance completely degrades and hits the timeout of 3× the baseline runtime. For the datasets with 10 and 100 million tuples, DuckDB utilizes all 28 workers, so this tipping point is already reached with a partition size of 100 tuples. This behavior shows that partition pruning requires our adaptive approach to be feasible in practice.

Adaptive Partition Pruning detects unfavorable cases, switching to regular execution so performance never drastically drops when partition pruning produces considerable overhead. There is only one case (10 million rows, 10 000 partitions) where it switches too early; in the remaining cases, it closely matches regular partition pruning with 2×–5× speedups. The difference compared to the regular version arises from HyperLogLog maintenance for every chunk, which could be reduced by fewer updates. Overall, this heuristic makes partition pruning viable for different dataset characteristics.

Lastly, we combine Co-Evaluation, Merge Pruning, and Adaptive Partition Pruning for the final WF$^+$ implementation. The runtimes are close to or even below the best of the individual techniques and show that this combination is ultimately beneficial: in most of the cases where Merge Pruning or Adaptive Partition Pruning outperform each other, we benefit from the best of both. The performance is drastically better than the baseline for few and large partitions, and only converges to the baseline for really small partitions.

8.3 Skewed Data

Real-world data is never uniformly distributed [48], so we modify the setup of Section 8.2 by varying the degree of skew. We fix the relation size to 1 million tuples and distribute the partitions according to a zeta distribution with a skewness factor α between 1.1 and 5. Higher α corresponds to a larger share of data that concentrates on a few partitions. Figure 5 shows the results of this experiment.

The proposed optimizations perform well also on skewed datasets. As in our previous experiments, Merge Pruning consistently achieves 1.2× to 1.5× speedups compared to the baseline in most cases. Partition Pruning speeds up the execution up to a factor of 5 (MT) or 10 (ST) for high skew factors/few partitions because most of the large partitions' tuples can be discarded. However, there is some overhead for many small partitions. Adaptive Partition Pruning does not directly pick to prune for many partitions, but soon reaches performance close to the regular version. Again, the combination of optimizations almost always achieves the best performance of the individual approaches, with speedups around 3× (MT) and 5× (ST).

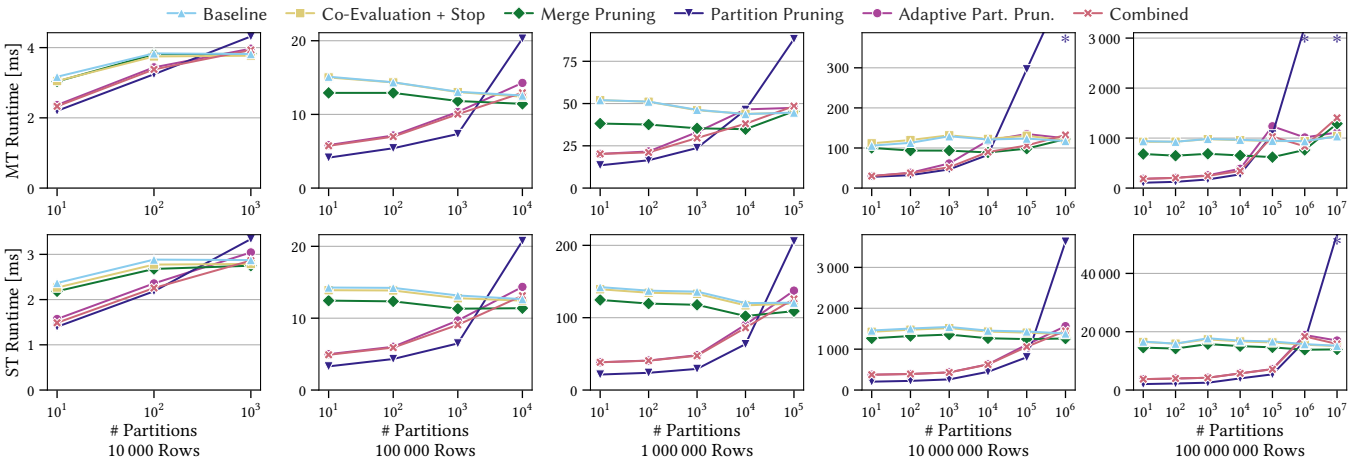

Figure 4: Microbenchmarks for Co-Evaluation (Q3, $k = 3$, $\delta = 5$) using different optimizations for different relation sizes and numbers of WF partitions, multi- (MT) and single-threaded (ST). Asterisk $*$ depicts a time-out within three times of the Baseline. The combination of optimization techniques is almost always considerably faster than the baseline.

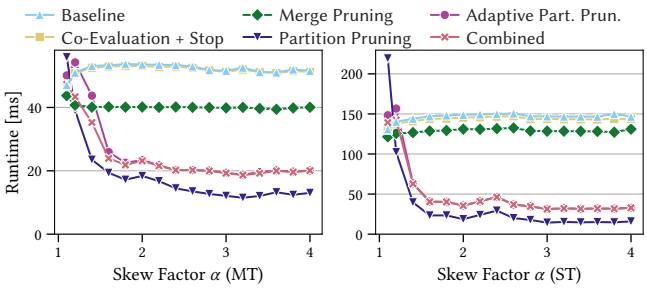

Figure 5: Microbenchmarks for Co-Evaluation (Q3, $k = 3$, $\delta = 5$) using different optimizations for 1 000 000 tuples and different skew factors of a zeta distribution, multi- (MT) and single-threaded (ST).

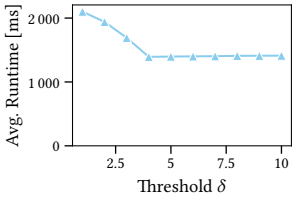

Figure 6: Average runtimes for adaptive partition pruning with different δ across all configurations seen in Figure 4.

8.4 Adaptive Partition Pruning Threshold

Section 7 described the Adaptive Partition Pruning strategy and introduced the threshold δ as a tuning parameter. To assess its influence and find a value that performs well for different scenarios, we execute all configurations from Section 8.2 with Adaptive Partition Pruning and δ between 1 and 10. Then, we calculate the average runtime across all configurations, shown in Figure 6.

While the concrete value for δ may vary for individual WF$^+$ implementations, this experiment demonstrates that such a threshold exists. In general, higher values perform better, and the aggregated runtimes vary by a factor of 1.5 between the best and the worst value for δ. Small thresholds cannot detect cases where the overhead of partition pruning deteriorates. Too large a value stops pruning too early when it is still beneficial. In our experiments, $\delta = 5$ is the optimal value, and the more detailed evaluation in Section 8.2 shows that it performs well in most of the cases.

8.5 Detailed Pipeline Analysis

In this last experiment, we investigate the impact of Co-Evaluation with combined optimizations on individual pipeline steps of the WF evaluation for Q3. We use a dataset of 1 million rows and two partition counts $m_1 = 100$ and $m_2 = 10\,000$. For all configurations, DuckDB uses 9 workers for partitioning and local sort. These workers radix-partition the tuples into 16 local radix buckets, and all related local radix buckets are merged to a single sorted bucket and materialized by one worker per global bucket. Finally, 25 workers perform framing and the window function computation. Figure 7 shows the timeline for each phase per worker.

At a high level, we observe that Adaptive Partition Pruning improves performance by a factor of almost 3 for 100 partitions, and that Merge Pruning is beneficial for 10 000 partitions. For 100 partitions, local partitioning takes longer than the baseline due to Partition Pruning. Workers maintain a HyperLogLog and partition-local heaps to reject tuples outside the top-3. As a result, radix buckets are much smaller, and local sorting is much faster. Only $m_1 \cdot k = 300$ tuples must be read per worker and from the merged buckets. Thus, merging, materialization, and framing/aggregation happen almost instantly compared to the baseline. The additional overhead in partitioning pays off in all subsequent phases.

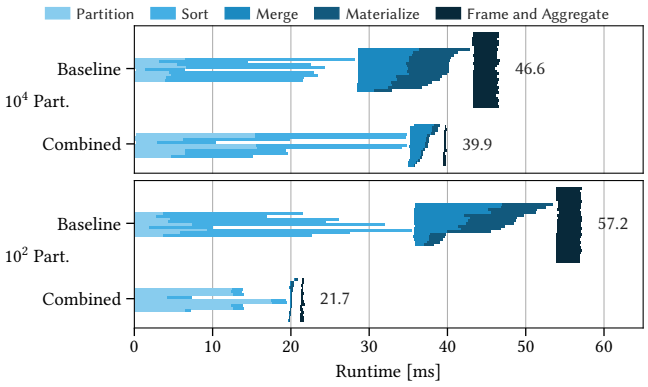

Figure 7: Per-worker execution timeline for different WF processing phases (cf. Figure 2) for Q3 and $k = 3$ on 1 million rows. Each line represents a worker and each phase may use a different number of workers. Combined pruning reduces total time by eliminating later work in early phases.

The performance benefit of optimized Co-Evaluation is not as large for 10 000 partitions. Here, Adaptive Partition Pruning introduces overhead with the HyperLogLog, as the workers decide against partition pruning. Thus, the local sort times are comparable to the baseline. With Merge Pruning, we need to read $m_2 \cdot k = 30\,000$ tuples from each local run, and merge 270 000 tuples overall. This number is higher than for 100 partitions, but still considerably lower than in the unmodified operator, which reflects in the merge runtime. Merge Pruning continues to reduce the number of tuples for materialization and framing/aggregation: it is a good fallback strategy when Adaptive Partition Pruning cannot eliminate tuples.

9 RELATED WORK

To the best of our knowledge, ours is the first work to provide a rigorous table of equivalences involving WFs. The work, however, intersects with several efforts in database query optimization. We present these intersections in groups.

Dependency-aware optimization. The reductions and push-downs in Table 3 rely on functional and order dependencies, which modern optimizers increasingly maintain. Apache Calcite [6] tracks functional dependencies for query rewriting, and Orca [43], the optimizer behind Greenplum, represents them natively. Simmen et al. [39] showed how order properties are derived and propagated during optimization. Kossmann et al. [20] surveyed about 60 optimization techniques that use data dependencies. Obtaining and exploiting further dependencies for window optimization is an active direction, including work by some of the authors [26].

Previous WF treatments. The approach most similar to ours appears in Bellamkonda et al. [7], who presented techniques such as merge pruning. That paper, however, missed other earlier-stage pruning opportunities that we present here. Moreover, the absence of a clear definition of WFs makes it difficult to assess the correctness of the proposed query manipulations. Regarding the works we cited in Section 1, as we mentioned there, our intention is to complement them and provide a rigorous base for others to do so as well. In particular, Cao et al. [9] proposed that cover-set search can be improved with the kind of functional dependency usage of Equivalences 2 and 3. Cover-set search is precisely the joint optimization of multiple windows that share a partitioning or ordering while computing different aggregates over the same sorted data. Our equivalences are complementary, as the reductions can shrink the keys such windows share before the sharing takes place.

Nested relational algebra. Since the foundational work of Makinouchi [29], Jaeschke and Schek [19], and Roth et al. [37], different authors have used nested relations and nested algebra to reason about SQL structures. Liu and Ramamohanarao [27] and Liu and Yu [28] defined algebraic equivalences. Cao and Badia [8] used nested algebra to optimize subqueries. Auerbach et al. [4] provided an extension, e. g., to model views. However, at that time, those works did not consider ordered tuple sequences.

10 CONCLUSION

In this work, we introduced a table of algebraic equivalences for window functions that formalizes existing transformations and introduces novel ones. We showed that certain transformations are applicable to a broader range of cases when specific conditions regarding window framing and partitioning are considered. Notably, one transformation enables the early application of a window predicate, which is typically executed at the conclusion of the window function, by pushing it to earlier stages. This optimization, referred to as WF$^+$, allows for data pruning during the initial phases of window function execution. Collectively, these transformations, along with the WF$^+$ optimization, yielded significant performance improvements across a wide variety of tested scenarios.

ACKNOWLEDGMENTS

We thank Martin Boissier and Matthias Herzog for their assistance with the case study and Sarah Kleest-Meißner for her feedback on the manuscript.

Daniel Lindner received funding from SAP SE. This project was partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Serge Abiteboul and Seymour Ginsburg. 1986. Tuple sequences and lexicographic indexes. *J. ACM* 33, 3 (1986), 409–422. https://doi.org/10.1145/5925.5926

[2] Alibaba Cloud. 2025. Window Functions. PolarDB-X Documentation. https://doc.polardbx.com/en/dev-guide/topics/window.html (accessed July 17, 2026).

[3] Amazon Web Services. 2025. Window Functions. Amazon Redshift Documentation. https://docs.aws.amazon.com/redshift/latest/dg/c_Window_functions.html (accessed July 17, 2026).

[4] Joshua S. Auerbach, Martin Hirzel, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2017. Handling Environments in a Nested Relational Algebra with Combinators and an Implementation in a Verified Query Compiler. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1555–1569. https://doi.org/10.1145/3035918.3035961

[5] Radim Bača. 2024. Window Function Expression: Let the Self-Join Enter. *Proceeding of the VLDB Endowment (PVLDB)* 17, 9 (2024), 2162–2174. https://doi.org/10.14778/3665844.3665848

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings*

of the International Conference on Management of Data (SIGMOD). 221–230. https://doi.org/10.1145/3183713.3190662

[7] Srikanth Bellamkonda, Tolga Bozkaya, Biswapriyo Ghosh, Abhinav Gupta, John Haydu, Sankar Subramanian, and Andrew Witkowski. 2000. *Analytic Functions in Oracle 8i.* Technical Report. Oracle Corporation. http://infolab.stanford.edu/infoseminar/archive/SpringY2000/speakers/agupta/paper.pdf

[8] Bin Cao and Antonio Badia. 2007. SQL query optimization through nested relational algebra. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 46 pages. https://doi.org/10.1145/1272743.1272748

[9] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. 2012. Optimization of Analytic Window Functions. *Proceeding of the VLDB Endowment (PVLDB)* 5, 11 (2012), 1244–1255. https://doi.org/10.14778/2350229.2350243

[10] CedarDB GmbH. 2026. Reference: Window Functions. CedarDB Documentation. https://cedardb.com/docs/references/queries/window/ (accessed July 17, 2026).

[11] ClickHouse Inc. 2025. Window Functions. ClickHouse Documentation. https://clickhouse.com/docs/sql-reference/window-functions (accessed July 17, 2026).

[12] Cockroach Labs. 2025. Window Functions. CockroachDB Documentation. https://www.cockroachlabs.com/docs/stable/window-functions (accessed July 17, 2026).

[13] DuckDB Foundation. 2025. Window Functions. DuckDB Documentation. https://duckdb.org/docs/sql/window_functions (accessed July 17, 2026).

[14] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book* (2 ed.). Pearson Education.

[15] Google Cloud. 2025. Window Function Calls. BigQuery Documentation. https://docs.cloud.google.com/bigquery/docs/reference/standard-sql/window-function-calls (accessed July 17, 2026).

[16] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 683–692. https://doi.org/10.1145/2452376.2452456

[17] IBM. 2025. OLAP Specification. Db2 Documentation. https://www.ibm.com/docs/en/db2/11.5?topic=expressions-olap-specification (accessed July 17, 2026).

[18] International Organization for Standardization. 2023. *Information technology – Database languages SQL – Part 2: Foundation (SQL/Foundation).* Standard Specification ISO/IEC 9075-2:2023(E).

[19] Gerhard Jaeschke and Hans-Jörg Schek. 1982. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 124–138. https://doi.org/10.1145/588111.588133

[20] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *The VLDB Journal* 31, 1 (2022), 1–22. https://doi.org/10.1007/S00778-021-00676-3

[21] Laurens Kuiper. 2025. Redesigning DuckDB's Sort, Again. https://duckdb.org/2025/09/24/sorting-again (accessed July 17, 2026).

[22] Laurens Kuiper and Hannes Mühleisen. 2023. These Rows Are Made for Sorting and That's Just What We'll Do. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2050–2062. https://doi.org/10.1109/ICDE55515.2023.00159

[23] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *Proceeding of the VLDB Endowment (PVLDB)* 8, 10 (2015), 1058–1069. https://doi.org/10.14778/2794367.2794375

[24] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. 2005. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 131–142. https://doi.org/10.1145/1066157.1066173

[25] Leonid Libkin and Liat Peterfreund. 2023. SQL Nulls and Two-Valued Logic. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 11–20. https://doi.org/10.1145/3584372.3588661

[26] Daniel Lindner, Daniel Ritter, and Felix Naumann. 2026. Unleashing Data Dependency-based Query Optimization. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 516–529. https://doi.org/10.48786/EDBT.2026.41

[27] Hong-Cheu Liu and Kotagiri Ramamohanarao. 1994. Algebraic Equivalences Among Nested Relational Expressions. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 234–243. https://doi.org/10.1145/191246.191287

[28] Hong-Cheu Liu and Jeffrey Xu Yu. 2005. Algebraic equivalences of nested relational operators. *Information Systems (IS)* 30, 3 (2005), 167–204. https://doi.org/10.1016/J.IS.2003.12.001

[29] Akifumi Makinouchi. 1977. A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 447–453.

[30] Microsoft Corporation. 2025. OVER Clause (Transact-SQL). Microsoft SQL Server Documentation. https://learn.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql (accessed July 17, 2026).

[31] OceanBase. 2025. Window Function. OceanBase Database Documentation. https://en.oceanbase.com/docs/common-oceanbase-database-10000000001718113 (accessed July 17, 2026).

[32] Oracle Corporation. 2025. Analytic Functions. Oracle Database SQL Language Reference. https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/Analytic-Functions.html (accessed July 17, 2026).

[33] Oracle Corporation. 2025. Window Function Concepts and Syntax. MySQL Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/window-functions.html (accessed July 17, 2026).

[34] PingCAP. 2025. Window Functions. TiDB Documentation. https://docs.pingcap.com/tidb/stable/window-functions/ (accessed July 17, 2026).

[35] PostgreSQL Global Development Group. 2025. Window Functions. PostgreSQL Documentation. https://www.postgresql.org/docs/current/tutorial-window.html (accessed July 17, 2026).

[36] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1981–1984. https://doi.org/10.1145/3299869.3320212

[37] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. 1988. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems (TODS)* 13, 4 (1988), 389–417. https://doi.org/10.1145/49346.49347

[38] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[39] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. 1996. Fundamental Techniques for Order Optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 57–67. https://doi.org/10.1145/233269.233320

[40] SingleStore Inc. 2025. Window Functions. SingleStore Documentation. https://docs.singlestore.com/cloud/reference/sql-reference/window-functions/ (accessed July 17, 2026).

[41] Michael Sipser. 1997. *Introduction to the theory of computation.* International Thomson Publishing.

[42] Snowflake Inc. 2025. Window Functions. Snowflake Documentation. https://docs.snowflake.com/en/sql-reference/functions-window (accessed July 17, 2026).

[43] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 337–348. https://doi.org/10.1145/2588555.2595637

[44] SQLite Consortium. 2025. Built-In Window Functions. SQLite Documentation. https://www.sqlite.org/windowfunctions.html (accessed July 17, 2026).

[45] Jan Vincent Szlang, Sebastian Breß, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights From the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proceeding of the VLDB Endowment (PVLDB)* 18, 12 (2025), 5126–5138. https://doi.org/10.14778/3750601.3750632

[46] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2013. Expressiveness and Complexity of Order Dependencies. *Proceeding of the VLDB Endowment (PVLDB)* 6, 14 (2013), 1858–1869. https://doi.org/10.14778/2556549.2556568

[47] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Volume I.* Principles of computer science series, Vol. 14. Computer Science Press.

[48] Wangda Zhang and Kenneth A. Ross. 2022. Exploiting Data Skew for Improved Query Performance. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 34, 5 (2022), 2176–2189. https://doi.org/10.1109/TKDE.2020.3006446