Dependency Shape Predicts Execution Behavior Across Independent Data Processing Systems

Thomas Dionysopoulos, CFA

Abstract

When two independently-developed data processing systems implement the same operation, do they make the same execution decisions? We test this by constructing a frozen 8-valued taxonomy that classifies operations solely by data-dependency shape—which rows an operation must access to produce each output row. Without inspecting either target system, we assign 30 operations per system to taxonomy categories and predict three execution behaviors: streaming eligibility, buffering requirements, and warmup. We evaluate against two systems with different architectures: Polars (Rust, morsel-driven parallelism) and DuckDB (C++, push-based pipelines). The taxonomy predicts buffering requirements at 96.7% accuracy in both systems, with the single shared error (filter) reflecting a classification boundary. Streaming and warmup predictions show system-dependent accuracy (76.7–100%), with all errors traceable to architectural choices (DuckDB classifies global aggregates as pipeline breakers; Polars classifies them as streaming) and API conventions (SQL computes partial windows; Polars emits null). Adversarial analysis reveals three layers of prediction strength: mathematical constraints on data access (unfalsifiable), rational engineering at the mathematical lower bound (empirically robust), and system-specific convention (falsifiable and partially falsified). Across 180 predictions, combined accuracy is 91.1%, with zero errors from incorrect dependency-shape assignments.

1 Introduction

Data processing systems—query engines, DataFrame libraries, time-series compilers—must decide how to execute each operation in a pipeline. Should it stream through the data or materialize first? How much state must it buffer? Can it produce output before seeing all input? These decisions are embedded in optimizer logic, pipeline schedulers, and physical operator implementations.

Different systems make these decisions independently. Polars implements a morsel-driven streaming engine in Rust. DuckDB implements a push-based pipeline executor in C++. They share no code and were developed by different teams for different use cases. Yet both must answer the same questions about the same operations.

This paper asks: **do the answers depend on the operation's mathematical structure, or on the system's architecture?**

We approach this question empirically. We construct a taxonomy of operations based on a single property—the *dependency shape* of the operation, meaning which input rows it must access to produce each output row. We freeze the taxonomy before examining either target system, then use it to predict execution behaviors in both.

The key finding is that the answer is *both*, but in a structured way. Buffering requirements are determined by dependency shape (96.7% accuracy, identical in both systems). Streaming eligibility and warmup behavior are partially determined by dependency shape and partially by system architecture. The errors decompose into three layers: mathematical constraints that no implementation can circumvent, rational engineering choices that Pareto-optimal implementations converge on, and system-specific conventions that vary across architectures.

Contributions.

1. A frozen-predictor methodology for testing whether operation taxonomies transfer across systems without overfitting.

2. Empirical evidence that dependency shape predicts buffering at 96.7% across two independently-developed systems, with streaming and warmup predictions showing architecture-dependent divergence.
3. A three-layer decomposition of prediction strength (mathematical, engineering, conventional) that explains where and why cross-system accuracy varies.
4. Adversarial analysis demonstrating that a pathological System C can break predictions, but only through Pareto-disimprovements that no rational implementation would make.

2 The Dependency-Shape Taxonomy

2.1 Categories

We classify operations into eight categories based solely on which input rows are accessed to produce each output row. The taxonomy was developed inside a time-series execution system (Blisp) and is frozen for this study—no modifications are made based on Polars or DuckDB behavior.

Table 1: The 8-valued dependency-shape taxonomy.

Category	Mnemonic	Access Pattern	Example
ELM	Elementwise	$y_i = f(x_i)$	abs, log, sqrt
PTW	Pointwise cross-row	$y_i = f(x_i, x_{i-k})$	lag, diff
WIN	Windowed	$y_i = f(x_{i-w+1}, \ldots, x_i)$	rolling mean
PFX	Prefix	$y_i = \text{fold}(x_0, \ldots, x_i)$	cumulative sum
REC	Recursive	$y_i = f(x_i, y_{i-1})$	EWM mean
GBL	Global	$y = \text{reduce}(x_0, \ldots, x_n)$	sum, count
MSK	Mask	$y_i = \text{predicate}(x_i)$	is_null, is_nan
SEL	Selection	$y = \text{subset/reorder}(x)$	sort, filter

Assignment rule. Each operation is assigned to the category whose data access pattern matches its mathematical definition. The assignment considers only input dependencies, not implementation strategy. An operation that computes $y_i = |x_i|$ is ELM regardless of how the implementation buffers, streams, or materializes.

2.2 Target Behaviors

We predict three execution behaviors for each operation:

Streaming eligibility. Can the operation produce output incrementally as input arrives, without blocking until all input is consumed?

Buffering requirement. Must the operation retain data beyond $O(1)$ running state before producing output? An $O(1)$ accumulator (running sum) does not constitute buffering. An $O(w)$ sliding window or $O(n)$ materialization does.

Warmup requirement. Does the operation produce null or undefined output for initial rows before it can emit meaningful values?

2.3 The Frozen Predictor

The predictor maps each category to predicted behaviors. It was written once—before examining Polars or DuckDB—and is not modified between experiments.

3 Experimental Design

3.1 Frozen-Predictor Methodology

The experiment proceeds in four steps:

Table 2: Frozen predictor: dependency shape → execution behavior.

Category	Stream	Buffer	Warmup	Reasoning
ELM	yes	no	no	Per-element. Stateless.
PTW	yes	yes	yes	Fixed offset: $O(k)$ buffer. Leading nulls.
WIN	no	yes	yes	Sliding window: $O(w)$ state. Partial warmup.
PFX	yes	no	no	$O(1)$ accumulator. Emits from row 0.
REC	yes	no	no	$O(1)$ recurrence state. Emits from row 0.
GBL	yes	yes	no	Partial aggregation can stream. Must see all data.
MSK	yes	no	no	Per-element property check. Stateless.
SEL	no	yes	no	Reordering/indexing needs full dataset.

1. **Assign.** Select ∼30 operations per system. Assign each to a taxonomy category based on its mathematical definition, before inspecting the system's implementation.
2. **Predict.** Apply the frozen predictor (table 2) to generate 3 predicted behaviors per operation (90 predictions per system).
3. **Measure.** Determine actual execution behavior by examining source code, physical plan operators, and streaming eligibility gates.
4. **Evaluate.** Compare predictions to measurements. Analyze errors.

The predictor is frozen across both experiments. We do not recalibrate after the Polars experiment. This prevents overfitting and makes the DuckDB experiment a genuine out-of-sample test.

3.2 System Selection

We selected Polars and DuckDB because they satisfy three criteria:

Independent development. Polars (`pola-rs/polars`) is a Rust DataFrame library developed by Ritchie Vink and contributors. DuckDB (`duckdb/duckdb`) is a C++ analytical database developed by Mark Raasveldt and Hannes Mühleisen. They share no code, no common codebase ancestry, and no organizational relationship.

Different architectures. Polars uses morsel-driven parallelism with a streaming engine that routes operations through dedicated node types (`PhysNodeKind` variants) or falls back to in-memory execution (`InMemoryMap`). DuckDB uses a push-based pipeline model where operators are classified as Operators (streaming), Sources, or Sinks (pipeline breakers), with pipelines connected through Sink–Source boundaries.

Shared operation vocabulary. Both systems implement common operations (abs, rolling mean, cumulative sum, global aggregation, sort, filter) whose mathematical definitions are identical.

3.3 Operation Selection

We selected 30 operations per system covering all represented taxonomy categories.

Polars (30 operations): ELM=5, PTW=2, WIN=5, PFX=4, REC=2, GBL=6, MSK=3, SEL=3. All 8 categories represented.

DuckDB (30 operations): ELM=6, PTW=2, WIN=5, PFX=5, GBL=6, MSK=3, SEL=3. Seven categories represented. REC is absent: DuckDB SQL has no native exponentially-weighted moving average operations.

3.4 Behavior Measurement

Polars streaming. An operation is streaming-eligible if it has a dedicated `PhysNodeKind` variant in the streaming engine. Operations that fall back to `InMemoryMap` or `InMemoryMapNode` are not streaming-eligible.

DuckDB streaming. An operation is streaming-eligible if it executes as an Operator in the push-based pipeline (not a Sink+Source pipeline breaker). Scalar functions, `PhysicalFilter`, `PhysicalStreamingWindow`, and `PhysicalStreamingLimit` are Operators. `PhysicalWindow`, `PhysicalOrder`, `PhysicalUngroupedAggregate`, and `PhysicalHashAggregate` are pipeline breakers.

Buffering and warmup are measured by examining kernel implementations, state data structures, and default output behavior for initial rows.

4 Transfer Experiment 1: Polars

4.1 Ground Truth

Table 3: Polars: 30 operations, measured execution behaviors.

#	Operation	Cat	Stream	Buffer	Warmup
1	abs	ELM	yes	no	no
2	log	ELM	yes	no	no
3	exp	ELM	yes	no	no
4	sqrt	ELM	yes	no	no
5	sign	ELM	yes	no	no
6	shift	PTW	yes	yes	yes
7	diff	PTW	yes	yes	yes
8	rolling_mean	WIN	no	yes	yes
9	rolling_sum	WIN	no	yes	yes
10	rolling_std	WIN	no	yes	yes
11	rolling_min	WIN	no	yes	yes
12	rolling_max	WIN	no	yes	yes
13	cum_sum	PFX	yes	no	no
14	cum_prod	PFX	yes	no	no
15	cum_min	PFX	yes	no	no
16	cum_max	PFX	yes	no	no
17	ewm_mean	REC	yes	no	no
18	ewm_std	REC	yes	no	no
19	sum	GBL	yes	yes	no
20	mean	GBL	yes	yes	no
21	std	GBL	yes	yes	no
22	min	GBL	yes	yes	no
23	max	GBL	yes	yes	no
24	count	GBL	yes	yes	no
25	is_null	MSK	yes	no	no
26	is_not_null	MSK	yes	no	no
27	is_nan	MSK	yes	no	no
28	filter	SEL	yes	no	no
29	gather	SEL	no	yes	no
30	sort	SEL	no	yes	no

4.2 Results

Overall accuracy: 88/90 = 97.8%.

Table 4: Polars accuracy by behavior and category.

Per category

Category	Score	Accuracy
ELM	15/15	100.0%
PTW	6/6	100.0%
WIN	15/15	100.0%
PFX	12/12	100.0%
REC	6/6	100.0%
GBL	18/18	100.0%
MSK	9/9	100.0%
SEL	7/9	77.8%

Per behavior

Behavior	Score	Accuracy
Streaming	29/30	96.7%
Buffering	29/30	96.7%
Warmup	30/30	100.0%

4.3 Error Analysis

Both errors occur on `filter`. Predicted: streaming=no, buffering=yes (the SEL prediction). Actual: streaming=yes, buffering=no.

Filter is classified as SEL because it changes the row set (output has fewer rows than input). But its *input* dependency is per-row: the predicate for row i depends only on row i. In Polars, filter is implemented as a dedicated streaming node (`PhysNodeKind::Filter`) that evaluates each morsel independently with no buffering.

The SEL category conflates two kinds of operations: those whose input dependency is per-row (filter) and those whose input dependency is global (sort, gather). DependencyClass assigns filter to SEL based on output effect rather than input dependency.

5 Transfer Experiment 2: DuckDB

5.1 Ground Truth

Table 5: DuckDB: 30 operations, measured execution behaviors.

#	Operation	Cat	Stream	Buffer	Warmup
1	ABS(x)	ELM	yes	no	no
2	LN(x)	ELM	yes	no	no
3	EXP(x)	ELM	yes	no	no
4	SQRT(x)	ELM	yes	no	no
5	SIGN(x)	ELM	yes	no	no
6	ROUND(x)	ELM	yes	no	no
7	LAG(x,k)	PTW	yes	yes	yes
8	LEAD(x,k)	PTW	yes	yes	no
9	AVG OVER w	WIN	no	yes	no
10	SUM OVER w	WIN	no	yes	no
11	STDDEV OVER w	WIN	no	yes	no
12	MIN OVER w	WIN	no	yes	no
13	MAX OVER w	WIN	no	yes	no
14	SUM OVER UBP	PFX	yes	no	no
15	COUNT OVER UBP	PFX	yes	no	no
16	MIN OVER UBP	PFX	yes	no	no
17	MAX OVER UBP	PFX	yes	no	no
18	ROW_NUMBER()	PFX	yes	no	no
19	SUM(x)	GBL	no	yes	no

#	Operation	Cat	Stream	Buffer	Warmup
20	AVG(x)	GBL	no	yes	no
21	STDDEV_SAMP	GBL	no	yes	no
22	MIN(x)	GBL	no	yes	no
23	MAX(x)	GBL	no	yes	no
24	COUNT(x)	GBL	no	yes	no
25	IS NULL	MSK	yes	no	no
26	IS NOT NULL	MSK	yes	no	no
27	isnan(x)	MSK	yes	no	no
28	WHERE	SEL	yes	no	no
29	ORDER BY	SEL	no	yes	no
30	DISTINCT	SEL	no	yes	no

5.2 Results

Overall accuracy: 76/90 = 84.4%.

Table 6: DuckDB accuracy by behavior and category.

Per category

Category	Score	Accuracy
ELM	18/18	100.0%
PTW	5/6	83.3%
WIN	10/15	66.7%
PFX	15/15	100.0%
GBL	12/18	66.7%
MSK	9/9	100.0%
SEL	7/9	77.8%

Per behavior

Behavior	Score	Accuracy
Streaming	23/30	76.7%
Buffering	29/30	96.7%
Warmup	24/30	80.0%

5.3 Error Analysis

14 errors, grouped by mechanism:

GBL streaming (6 errors). Predicted: streaming=yes. Actual: streaming=no. DuckDB implements global aggregates as Sink+Source pipeline breakers (`PhysicalUngroupedAggregate`). All input must be consumed before any output is produced. Polars implements the same operations as streaming reducers (`ReduceNode`). The underlying computation is identical: $O(1)$ accumulators processing data incrementally, output deferred until all input is consumed. The systems classify this pattern differently.

WIN warmup (5 errors). Predicted: warmup=yes. Actual: warmup=no. SQL window functions compute over partial windows for initial rows. At row 0, `AVG(x) OVER (ROWS BETWEEN 24 PRECEDING AND CURRENT ROW)` computes the average of the single available row, producing a non-null result. Polars's rolling operations default to `min_periods=window_size`, producing null for the first $w - 1$ rows.

SEL filter (2 errors). Identical to Polars. Filter's input dependency is per-row; its category assignment is based on output effect.

PTW LEAD warmup (1 error). Predicted: warmup=yes. Actual: warmup=no. LEAD looks forward: the first rows emit valid output (they reference future rows that exist). Nulls appear at the end (last k rows), not the beginning.

6 Cross-System Analysis

6.1 The Buffering Invariant

The central finding: buffering accuracy is identical across systems.

Table 7: Cross-system accuracy comparison.

Behavior	Polars	DuckDB	Invariant?
Buffering	29/30 (96.7%)	29/30 (96.7%)	**Yes**
Streaming	29/30 (96.7%)	23/30 (76.7%)	No
Warmup	30/30 (100.0%)	24/30 (80.0%)	No
Overall	**88/90 (97.8%)**	**76/90 (84.4%)**	—
Combined	**164/180 (91.1%)**		—

The single buffering error is the same operation (filter) in both systems. This is not coincidence—it reflects a real property of filter's data access pattern (per-row) that conflicts with its category assignment (SEL). Every other buffering prediction is correct in both systems.

Buffering is the behavior most directly determined by dependency shape: the data access pattern of an operation determines how much data it must retain. An ELM operation accesses only the current row—it cannot require buffering regardless of implementation choices. A GBL operation must inspect all data—it requires buffering regardless of how the system classifies it. These are mathematical constraints, not engineering preferences.

6.2 Why Streaming Diverges

The streaming dimension shows a 20-point accuracy gap between systems (96.7% vs 76.7%). All six additional errors in DuckDB occur on GBL operations.

The divergence is architectural. Both Polars and DuckDB process global aggregates with $O(1)$ accumulators that update incrementally and emit output only after all input is consumed. The behavioral pattern is identical:

1. Initialize accumulator state.
2. Process input chunks, updating accumulator.
3. After all input is consumed, finalize and emit result.

Polars classifies steps 1–2 as "streaming" because they execute within the streaming engine (`ReduceNode`). DuckDB classifies the entire pattern as "pipeline breaking" because the operator is a Sink (steps 1–2) that becomes a Source (step 3) only after finalization.

The predictor was written before examining either system. Its GBL→streaming=yes prediction matches Polars's classification and disagrees with DuckDB's. A DuckDB-calibrated predictor would reverse this—matching DuckDB at the cost of Polars accuracy. This symmetry confirms that GBL streaming classification is a system convention, not a dependency-shape property.

6.3 Why Warmup Diverges

Warmup accuracy drops from 100% (Polars) to 80% (DuckDB) due to two mechanisms.

SQL partial-window semantics (5 errors). In DuckDB's SQL, window functions with bounded frames produce non-null results for partial windows. The first row of `AVG(x) OVER (ROWS BETWEEN 24 PRECEDING AND CURRENT ROW)` computes the average of 1 available row, not null. In Polars, `rolling_mean(window_size=25)` defaults to `min_periods=25`, producing null for the first 24 rows. The mathematical fact—that the first $w-1$ rows have incomplete windows—is the same. Whether incomplete windows produce null is a convention.

Directional asymmetry (1 error). LEAD(x,k) produces trailing nulls (last k rows), not leading nulls (first k rows). The PTW warmup prediction is calibrated to backward-looking operations.

6.4 DuckDB's Independent Taxonomy

DuckDB independently developed a multi-level operation classification that we did not design and did not know about before beginning this study.

Level 1: Pipeline role. DuckDB classifies physical operators as Operators (streaming) or Sink+Source (pipeline breakers). This mirrors the dependency-shape taxonomy's streaming prediction for 6 of 7 tested categories. The sole divergence is GBL.

Level 2: Window executors. DuckDB routes window functions to specialized executors: `WindowAggregateExecutor`, `WindowValueExecutor`, `WindowLeadLagState`, `WindowRowNumberExecutor`. The routing between `WindowLeadLagState` (PTW) and `WindowAggregateExecutor` (WIN/PFX) mirrors the PTW vs WIN/PFX distinction.

Level 3: Aggregate taxonomy. DuckDB organizes aggregate function source code into directories named `distributive/`, `algebraic/`, and `holistic/`—an independently-developed classification embedded in file structure. It captures a dimension orthogonal to dependency shape: distributive aggregates (sum, count) have trivially combinable accumulators, algebraic aggregates (avg, stddev) have formula-based combination, and holistic aggregates (median, mode) must buffer all input. All three are GBL in the dependency-shape taxonomy.

The dependency-shape taxonomy is strictly coarser than DuckDB's Level 3. It captures "needs all data" without distinguishing accumulator strategies. This is by design—the taxonomy predicts pipeline-level behavior, not algorithm selection.

7 Adversarial Analysis

7.1 The Strongest Attack

The strongest criticism of these results is:

> DependencyClass predicts implementation conventions, not mathematical properties. Polars and DuckDB agree because their developers read the same papers, use the same textbook algorithms, and face the same hardware constraints. The taxonomy describes what competent engineers do, not what operations require.

We take this attack seriously and examine it per category and per behavior.

7.2 Mathematical Constraints vs Engineering Choices

For each category and behavior, we determine whether the prediction follows from mathematical necessity (no correct implementation can violate it) or from engineering convention (correct implementations exist that violate it). Table 8 summarizes the analysis.

7.3 Adversarial Implementations

To test the "convention" attack, we construct implementations that preserve the mathematical operation while violating predictions.

1. **abs(x): materialize, then compute.** Correct but wastes $O(n)$ memory for an $O(1)$ task. No rational system.
2. **rolling_mean: stream with $O(1)$ ring buffer.** Possible and correct. Requires cross-chunk state management.
3. **EMA: buffer all values, compute forward pass.** Correct but uses $O(n)$ for an $O(1)$ task. No rational system.
4. **cum_sum: materialize, then prefix scan.** Correct but $O(n)$ for $O(1)$. No rational system.
5. **Global sum: streaming accumulator (Polars-style).** Correct. Polars does this.
6. **lag(x,k): re-read source at offset.** Correct. Trades memory for I/O. Possible on memory-constrained hardware.

Adversarial implementations that break predictions fall into two groups. **Pathological** (#1, #3, #4): use strictly more resources than the mathematical minimum for no benefit. **Legitimate alternatives** (#2, #5, #6): trade one resource for another. Real systems sometimes make legitimate-alternative choices.

Table 8: Per-prediction determination: mathematical vs engineering vs convention.

Category	Behavior	Layer	Argument
ELM	buffer=no	Math	$y_i = f(x_i)$ references only x_i. $O(1)$ is the information-theoretic minimum.
ELM	stream=yes	Math	Output[i] can be produced upon receiving input[i].
ELM	warmup=no	Math	$f(x_0)$ is defined for all $f \in$ ELM.
PTW	buffer=yes	Math	$y_i = x_{i-k}$ requires x_{i-k}. Minimum $O(k)$ state.
PTW	warmup (lag)	Math	x_{0-k} is outside the domain.
PTW	warmup (lead)	Math	LEAD's first rows are defined (look ahead).
PTW	stream=yes	Eng	Streaming is achievable with $O(k)$ ring buffer. Materialization is also correct but wastes resources.
WIN	buffer=yes	Math	Function over $[i-w+1, i]$ requires w values.
WIN	stream=no	Conv	Rolling mean *can* stream with $O(1)$ state. Both systems choose materialization.
WIN	warmup	Partial	Incomplete windows are mathematical. Null output for them is convention (SQL: no; POLARS: yes).
PFX	buffer=no	Eng	$O(1)$ accumulator suffices. Pareto-optimal.
PFX	stream=yes	Eng	Can emit per row. Pareto-optimal.
PFX	warmup=no	Math	fold$(x_0) = x_0$ is defined.
GBL	buffer=yes	Math	Exact reduce$(x_0 \dots x_n)$ requires all n values.
GBL	stream	Conv	Accumulation can be streaming. Systems classify differently.
GBL	warmup=no	Math	Single output row. Warmup does not apply.
MSK	all	Math	Identical to ELM. Per-element predicate.
SEL sort	buffer=yes	Math	Total order requires all elements.
SEL filter	stream=yes	Math	Predicate depends only on current row.

7.4 System C: A Falsification Design

We design a hypothetical System C that intentionally maximizes prediction failures: all operations materialize before computing (even abs), window operations stream with cross-chunk state, global aggregates stream, and no warmup (partial results are non-null). Estimated accuracy: approximately 42%.

But System C is pathological. Materializing $|x_i|$ wastes $O(n)$ memory for zero benefit. Every design choice is a Pareto-disimprovement—more resources, identical output. The question is not whether System C can be built, but whether it *would* be. It is strictly dominated by systems that operate near the mathematical lower bound.

7.5 The Three Layers

The adversarial analysis reveals three distinct levels of prediction strength:

Layer 1: Mathematical constraints. Some predictions cannot be violated by any correct implementation. ELM operations access one row per output ($O(1)$ lower bound). PTW lag's first k outputs are undefined (domain constraint). GBL operations must consume all input. Sort must consume all input. These are theorems derivable from operations' mathematical definitions.

Layer 2: Rational engineering. Some predictions follow from Pareto-optimal implementation choices. ELM implementations stream (no rational reason to block). PFX and REC implementations use $O(1)$ state (sufficient and minimal). An adversarial implementation can violate these by using more resources than necessary, but no rational system does.

Layer 3: System conventions. Some predictions encode choices that vary across rational implementations. GBL streaming classification. WIN warmup behavior. WIN streaming eligibility. These are genuine engineering choices where different rational systems disagree.

The decomposition explains the accuracy pattern. Buffering accuracy (96.7% in both systems) draws primarily from Layers 1–2. Streaming and warmup draw from all three layers and diverge where Layer 3 conventions differ.

8 Threats to Validity

Shared intellectual ancestry. Both systems draw from the database systems tradition (push-based execution, morsel-driven parallelism [1]). Agreement might reflect shared ancestry rather than universal structure. Counter: the GBL streaming divergence directly falsifies this. If shared ancestry caused agreement, both systems would classify GBL identically—but they do not. Agreement on buffering persists despite divergent pipeline architectures.

Textbook algorithm convergence. Engineers implement standard algorithms because they are well-known. Partially valid—Layer 2 predictions predict textbook choices. But the adversarial analysis shows these are Pareto-optimal. Predicting that engineers make Pareto-optimal choices is a structural claim.

Polars-calibrated predictor. The taxonomy was developed in Blisp, which shares Polars's GBL streaming convention. A DuckDB-calibrated predictor would change 6 streaming and 5 warmup predictions, improving DuckDB accuracy to ∼96.7% while reducing Polars. The symmetry is the finding: calibration-sensitive dimensions are convention-dependent.

Limited systems tested. Two systems is a small sample. We do not claim universality. Replication on Flink, cuDF, or Spark would strengthen or bound the result.

Selection bias in behaviors. Streaming, buffering, and warmup relate to dependency shape by construction. Other behaviors (cache locality, SIMD utilization, thread scalability) might not be predicted. We claim only that dependency shape predicts the three measured behaviors.

9 Related Work

Aggregate taxonomy. Gray et al. (1997) classified aggregates as distributive, algebraic, or holistic. DuckDB's source tree explicitly uses this taxonomy. The dependency-shape taxonomy is strictly coarser: all three are GBL.

Query execution models. The volcano model (Graefe, 1994) introduced iterator-based pull execution. Push-based execution (Neumann, 2011) reversed control flow. Morsel-driven parallelism (Leis et al., 2014) added work-stealing. These models differ in control flow but agree on pipeline breakers—except for global aggregates (section 6).

Window function classification. SQL window functions are specified by frame type, bounds, and partitioning. DuckDB's `PhysicalStreamingWindow` classifies functions by streaming eligibility. The dependency-shape taxonomy captures a coarser WIN vs PFX distinction aligned with this gate.

Operation properties in optimizers. Production optimizers use per-operation properties (monotonicity, null handling, determinism) for transformation decisions. The dependency-shape taxonomy asks whether a single categorical property predicts multiple behaviors simultaneously.

10 Conclusion

We tested whether a dependency-shape taxonomy developed inside one system predicts execution behavior in two independently-developed systems. The answer is structured: dependency shape universally predicts buffering requirements (96.7%, identical in both systems) and partially predicts streaming eligibility and warmup (76.7–100%, with system-dependent divergence).

All 16 prediction errors across both experiments trace to three mechanisms: system pipeline architecture (GBL streaming), API convention (WIN warmup), and a classification boundary (SEL filter). Zero errors result from incorrect dependency-shape assignments.

Adversarial analysis reveals three layers: mathematical constraints that no implementation can circumvent, rational engineering at the mathematical lower bound, and system-specific conventions. Buffering draws from the first two layers; streaming and warmup draw from all three.

The practical implication is that a single categorical property— dependency shape—can serve as a portable metadata layer across systems. It reliably predicts which operations need buffered state, the property most relevant for memory planning, replay semantics, and pipeline construction. Where it fails to transfer, the failures are systematic and explainable rather than random.

References

[1] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework. In *SIGMOD*, pages 743–754, 2014.

[2] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[3] G. Graefe. Volcano—an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[4] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

A Falsification Cases

17 edge cases tested across both systems. None required a new taxonomy category.

Polars (9 cases): grouped rolling (A), partitioned windows (A), dynamic windows (A), resampling (A), interpolation (B: PTW with bidirectional refinement), asof joins (A), irregular timestamps (N/A), rolling quantile (A), rolling median (A).

DuckDB (8 cases): PARTITION BY windows (A), holistic aggregates (A), QUALIFY clause (A), external aggregation (A), parallel hash aggregate (A), FILTER clause on aggregates (A), aggregate ORDER BY (A), RANGE frame windows (A).

Verdicts: A = existing category sufficient (15), B = existing category with refinement (1), N/A = not an operation (1), C = new category required (0), D = different coordinate required (0).