

Towards a Performance Benchmark for OCPM Engines

OBJECTS Workshop @ BPM 2026

Aaron Küsters, Dirk Fahland, Wil M.P. van der Aalst

2026-09-28

OC-DFG on BPIC2017

An OC-DFG has one directly-follows graph per object type.
On BPIC2017 (1.2M events), one such DFG takes on average:

3.4 ms Rust4PM	59.5 ms DuckDB	441 ms Pandas	888 ms Neo4j	23.9 s SQLite
--------------------------	--------------------------	-------------------------	------------------------	-------------------------

Every engine returns the same DFG.
The slowest takes about 7,000× as long as the fastest.

Engines and adoption

-  Adoption of OCPM in practice depends, among other things, on fast access to object-centric event data.
-  Interactive exploration needs answers within seconds. We count sub-second answers as interactive.
-  Without breaking events down to cases, case-centric optimizations do not carry over directly.
-  OCPM tools each pick an engine: relational, graph, dataframe or custom.

 There is **no systematic comparison** of these engines on OCPM workloads.

Seven engines, four families

Family	Engines	Used in OCPM by
 Relational	SQLite, DuckDB	OCEL 2.0 reference format, PyStack't, ToTeM
 Dataframe	Pandas, Polars	PM4Py, Ocelelescope, ToTeM (Pandas); r4pm (Polars)
 Graph	Kuzu, Neo4j	EKGs (Esser and Fahland, Neo4j); OCPQ (Kuzu)
 Custom	Rust4PM	OCPQ, r4pm

Storage layouts

Two consecutive events of Application a1, each with its own Case_R object:

Tables

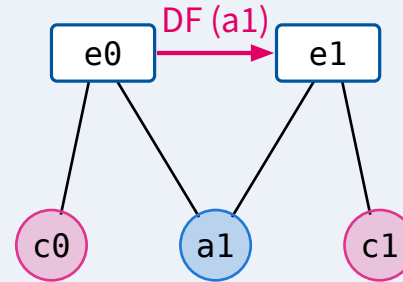
Pandas, Polars, SQLite, DuckDB

id	type	time
e0	A_Concept	09:58
e1	A_Accepted	10:02

id	type	event	object
a1	Application	e0	a1
c0	Case_R	e0	c0
c1	Case_R	e1	a1
		e1	c1

Graph

Kuzu, Neo4j



Linked in memory

Rust4PM

events to objects:

e0 -> [a1, c0]

e1 -> [a1, c1]

objects to events:

a1 -> [e0, e1, ...]

c0 -> [e0, ...]

c1 -> [e1, ...]

Further design decisions, e.g., one table per type in SQLite and DuckDB.

Four workload classes

P1 Control flow

DFG, Variants

Per object type: collect, sort and aggregate traces into a DFG or variants.

P2 Queries

OCPQ, EKG

Match event-object bindings, then filter: OCPQ trees and EKG queries.

P3 Performance

Sync, Sojourn

Per event: find predecessors via shared objects, compute sync and sojourn times.

P4 BI / KPI

Types, Conv

Aggregate by activity or type, or count objects that satisfy a predicate: a heatmap and a conversion rate.

Setup

Data: BPIC2017 (OCEL 2.0)

- 1,202,267 events, 26 activities
- 106,162 objects of 4 types:
Application, Offer, Workflow, Case_R
- object-to-object relations added:
Application to its Offers and Workflow

Protocol

- same output from every engine (verified)
- mean of 10 warm runs, cold run discarded
- isolated processes, randomized order,
60 min timeout
- one fixed configuration per engine, no per-
query tuning

 Rust4PM runs hand-written Rust, not a query language: part of its lead is native code, not only its data model. Exception: P2 runs the dynamic OCPQ engine.

Results

		P1		P2		P3		P4	
		DFG	Variants	OCPQ	EKG	Sync	Sojourn	Types	Conv
Relational	SQLite								
	DuckDB								
Dataframe	Pandas								
	Polars								
Graph	Kuzu								
	Neo4j								
Custom	Rust4PM								

Every cell has a native implementation in SQL, Cypher, dataframes or Rust.

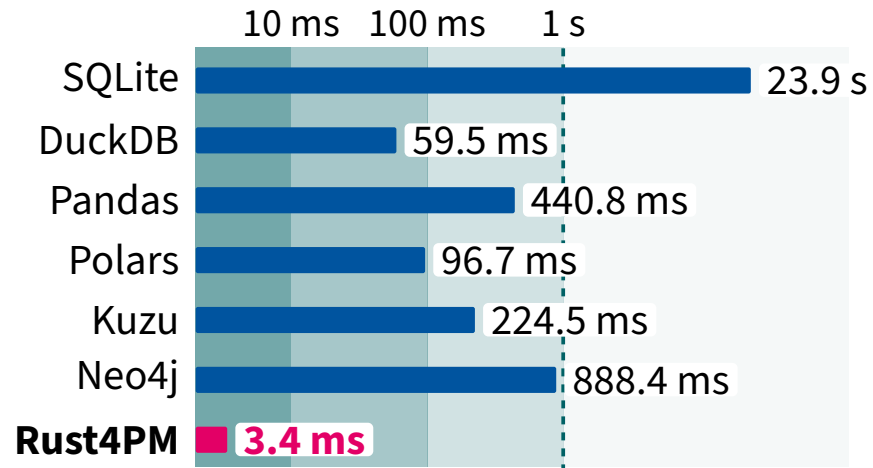
Results

		P1		P2		P3		P4	
		DFG	Variants	OCPQ	EKG	Sync	Sojourn	Types	Conv
Relational	SQLite	23.9 s	4.5 s	28.4 s	106.6 s	38.6 s	36.1 s	2.5 s	317.0
	DuckDB	59.5	114.5	47.3	38.5	251.5	99.0	47.0	26.9
Dataframe	Pandas	440.8	1.1 s	148.3	172.4	3.3 s	1.8 s	99.9	51.4
	Polars	96.7	74.1	22.0	33.7	569.3	372.2	13.4	5.0
Graph	Kuzu	224.5	644.1	62.2	38.5	669.8	461.6	155.6	20.7
	Neo4j	888.4	1.6 s	550.2	445.2	5.7 s	3.8 s	2.0 s	34.9
Custom	Rust4PM	3.4	53.0	26.7	29.0	65.7	49.4	7.5	0.4

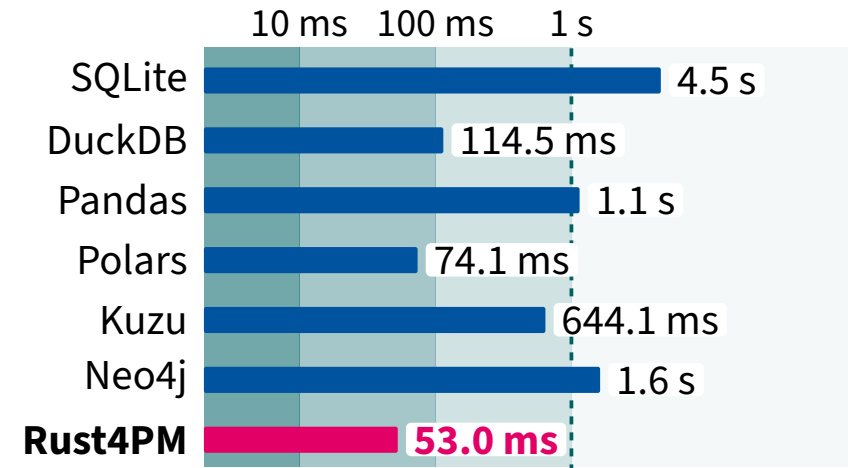
Mean runtime in ms (s where marked), shaded by band: < 10 ms, < 100 ms, < 1 s, ≥ 1 s. **Bold:** fastest band.

P1: Control flow

DFG directly-follows graph



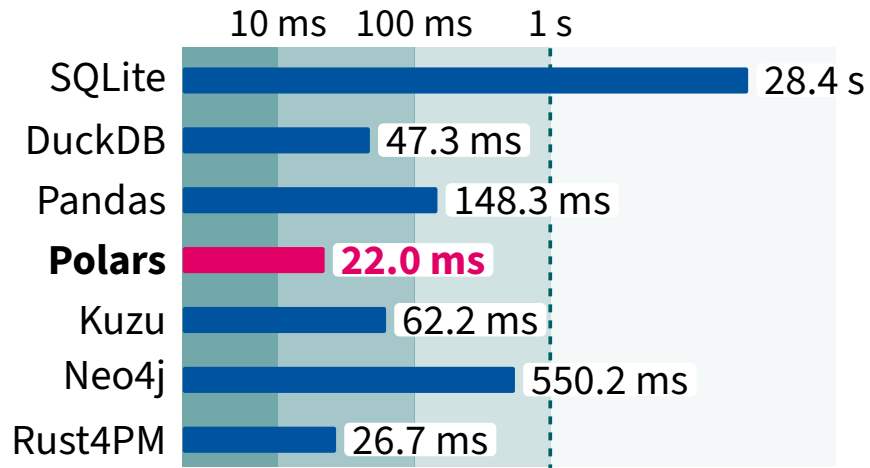
Variants trace variants



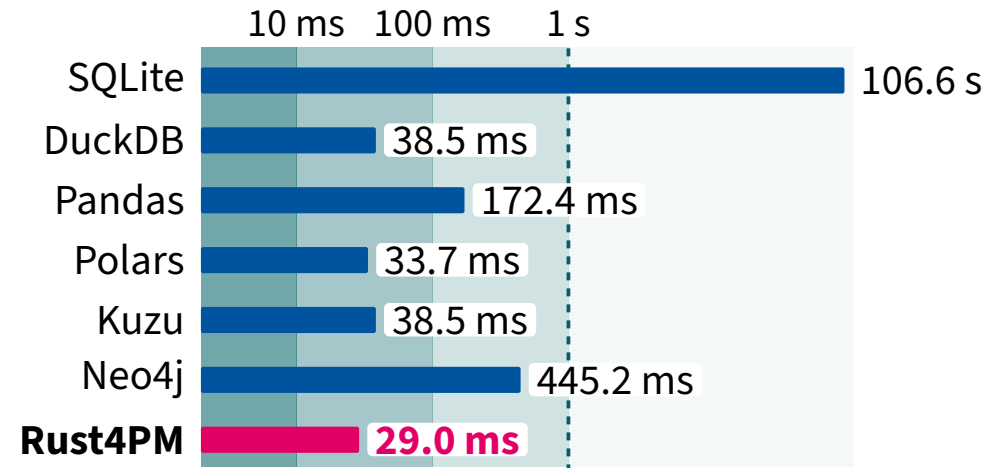
Rust4PM is fastest (3.4 ms and 53.0 ms). DuckDB and Polars stay under 115 ms. SQLite needs 23.9 s and 4.5 s.

P2: Queries

OCPQ 7 query trees



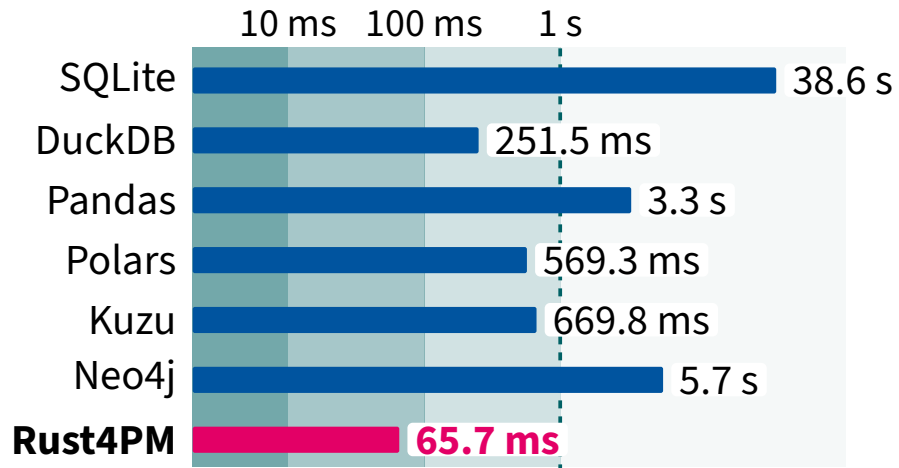
EKG 3 Cypher queries



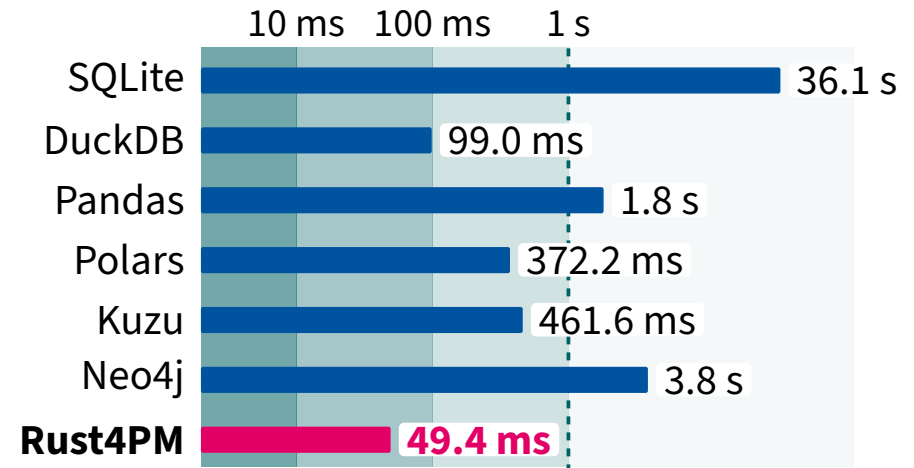
Rust4PM, Polars, DuckDB and Kuzu all answer in under 100 ms. On OCPQ, Polars (22.0 ms) is faster than the OCPQ engine on Rust4PM (26.7 ms).

P3: Performance

Sync synchronization time



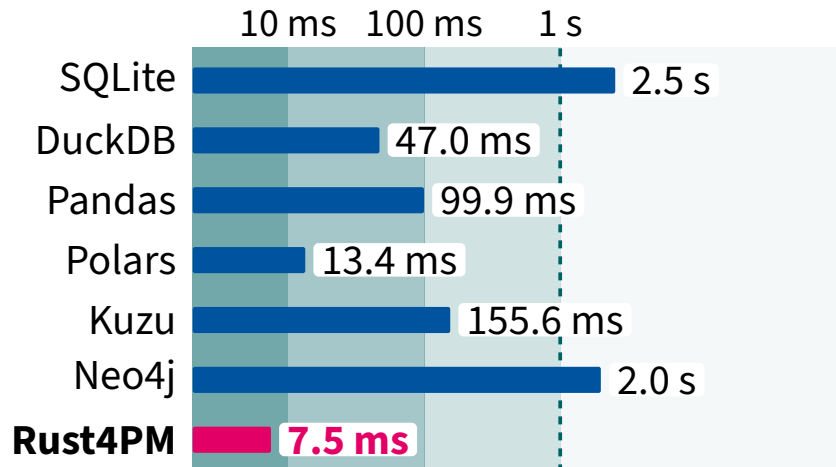
Sojourn sojourn time



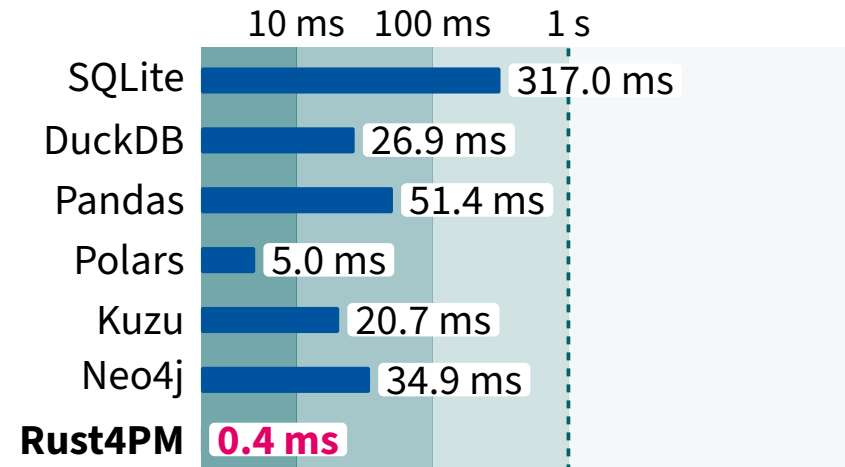
Only Rust4PM stays clearly under 100 ms. DuckDB comes next (251.5 ms and 99.0 ms). Pandas, Neo4j and SQLite need seconds.

P4: BI / KPI

Types activity × type heatmap



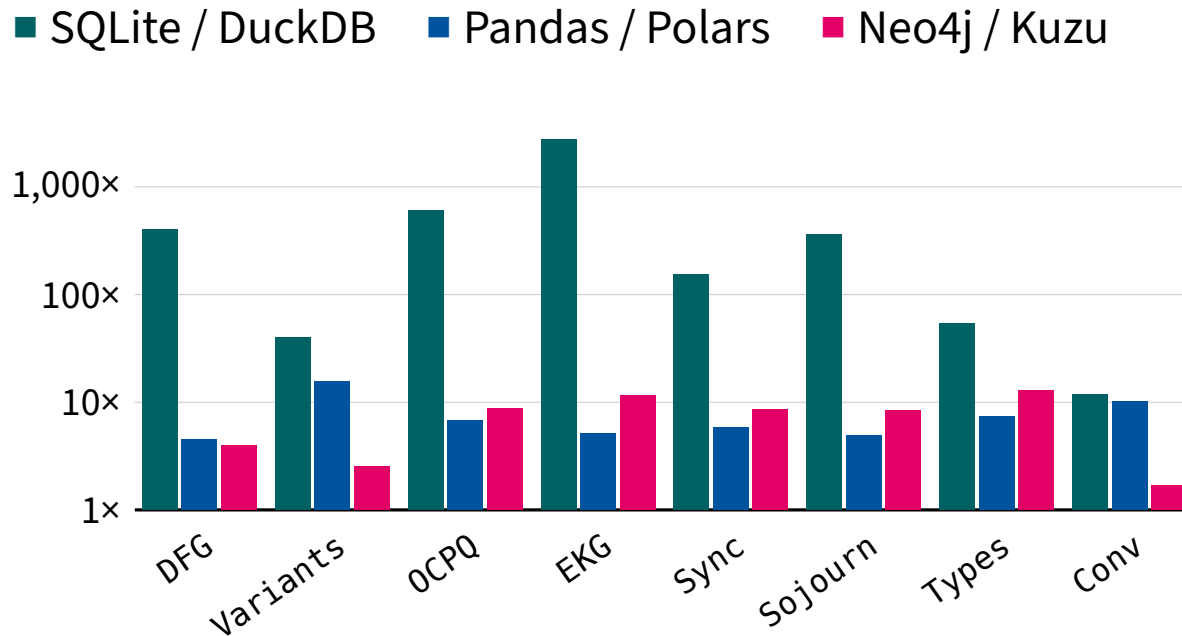
Conv conversion rate



On Types, Neo4j (2.0 s) and SQLite (2.5 s) are over 1 s. Conv returns one number: all engines except SQLite need less than 100 ms.

Engine families

Slowdown of the slower engine in each family, per workload (log scale):



Same SQL (up to dialect details) on the same schema: SQLite is **12× to 2,800×** slower than DuckDB (EKG: 106.6 s vs. 38.5 ms).

Neo4j is 1.7× to 12.8× slower than Kuzu, about 10× on queries.

Kuzu beats Neo4j and Polars beats Pandas on **all 8** workloads.

Schema typing

The same two events and objects, in the two layouts we compare:

Strongly typed (OCEL 2.0 layout)

one table per event and object type

event_A_Concept		event_A_Accepted	
id	time	id	time
e0	09:58	e1	10:02
object_Application		object_Case_R	
id		id	
a1		c0	
		c1	

Weakly typed

one table for all types, with a type column

event_attr			object_attr	
id	type	time	id	type
e0	A_Concept	09:58	a1	Application
e1	A_Accepted	10:02	c0	Case_R
			c1	Case_R

Both keep the event_object table and OCEL 2.0's event and object id-to-type lists. Kuzu: one node table per type vs. one Event and one Object node table.

Typing results

Speedup of the strongly typed layout over the weakly typed one:

Engine	Strong typing helps	Strong typing hurts
Kuzu	P2, Conv: up to 3.4× faster	DFG, P3, Types: up to 2.7× slower
DuckDB	P1: 1.4× faster, P2: 1.2× faster	largely unaffected elsewhere (0.94× to 1.06×)
SQLite	P2: timeout → completes	P1, P3: up to 7× slower

Typed tables help queries that filter by type. They can hurt workloads that combine all types (Kuzu, SQLite).

Picking an engine

On BPIC2017, for someone building an OCPM tool:

SQL

DuckDB over SQLite: same queries, 12× to 2,800× faster, under 1 s on every workload

Python dataframes

Polars over Pandas: faster on all 8 workloads

Graph

Kuzu over Neo4j: faster on all 8 workloads

Native structures

Rust4PM saves most on DFG (59.5 vs. 3.4 ms) and Sync (251.5 vs. 65.7 ms)

For query-heavy tools, typed tables help (Kuzu up to 3.1× on P2).

Harness

- Reference implementations** of 8 OCPM workloads: SQL, Cypher, Pandas, Polars and r4pm (Rust)
-  **Extend it:** add your engine, workload or dataset
-  Raw measurements, container definitions (Apptainer, Docker) and pinned dependencies



[github.com/aarkue/
oc-performance-code](https://github.com/aarkue/oc-performance-code)

Limits and open questions

! Limits

- one log (BPIC2017), no scaling behavior
- one machine, no distributed systems
- no per-query tuning
- Rust4PM's lead is partly native code, not only its data model

? Open questions

- Which log characteristics drive the cost? Synthetic logs with controlled object types, events per object, loops.
- Should OCEL 2.0 / OCED / ... recommend a layout for analysis, next to the exchange format?
- Which workloads are missing, e.g., discovery or conformance checking?
- Beyond runtime: memory, implementation effort, deployment.

Takeaways



A first, preliminary benchmark of OCPM engines: 7 engines, 8 workloads in 4 classes, one real-life log.



Whether object-centric analysis is interactive depends on the engine. DuckDB, Polars and Kuzu stay under 1 s on every workload.



Schema typing has a smaller effect: it helps type-filtered queries and can slow down cross-type workloads.



The harness is open: add your engine or workload and compare.

Thank you

Thanks!

🔗 github.com/aarkue/oc-performance-code

✉️ kuesters@pads.rwth-aachen.de





Bonus

Setup details

-  16 physical cores (Intel Xeon Platinum 8468), 32 GB memory, isolated container.
-  SQLite in memory. DuckDB and Kuzu up to 80% of memory. Neo4j: 16 GB heap and 16 GB page cache (store: 3.6 GB).
-  Graph engines get materialized directly-follows edges, as in the EKG literature. A Neo4j index on these edges would speed up DFG by over 3×. We omit it because Kuzu has no equivalent.
-  Python 3.14, duckdb 1.5.3, polars 1.41.2, pandas 3.0.3, r4pm 0.5.5a5 (0.5.5a6 for the Neo4j re-run), kuzu 0.11.3, SQLite 3.53.1, Neo4j 5.26.27.