Towards a Performance Benchmark for Object-Centric Process Mining Engines

Aaron Küsters[1], Dirk Fahland[2], and Wil M.P. van der Aalst[1]

[1] Chair of Process and Data Science (PADS), RWTH Aachen University
{kuesters,wvdaalst}@pads.rwth-aachen.de
[2] Process Analytics, Eindhoven University of Technology
d.fahland@tue.nl

Abstract. For Object-Centric Process Mining (OCPM) to be adopted in real-life settings, the engines underneath OCPM tools must provide immediate responses to allow for interaction, even when datasets get larger. Whether they do, on which workloads, and at what cost has not been systematically measured for the engines used by current publicly available OCPM tools. We present a first, preliminary empirical benchmark that measures per-object-type control-flow extraction, object-centric queries, object-centric performance analysis, and BI/KPI workloads on seven engines across four classes drawn from OCPM research. Our results on a real-life dataset show that engine choice alone can move the same OCPM workload from an interactive, sub-second answer to one taking tens of seconds. An OCED-native engine (Rust4PM) reaches interactive response times on all eight workloads. Three general-purpose engines reach comparable performance on 4 (Polars), 3 (DuckDB), and 2 (Kuzu) of the eight workloads, and stay at interactive speeds also on the remaining ones, while Pandas, Neo4j, and SQLite respond noticeably slower. In a dedicated sub-study, we observe that a strongly typed data schema can improve performance by 3.4× but may also degrade performance compared to a weakly typed schema. The harness, per-engine implementations, and raw measurements are public and built to extend with further workloads, engines, and datasets.

Keywords: Object-Centric Process Mining · Performance Benchmark · Process Mining Engines · Object-Centric Event Data.

1 Introduction

Object-Centric Process Mining (OCPM) techniques operate on Object-Centric Event Data (OCED). OCED provides a richer structure compared to sequential event logs, modeling events in relation to multiple related objects. Operations and queries over this richer data avoid representational and computational errors in process mining analyses [1] compared to the same data flattened into sequential logs, resulting in more reliable analysis [13] and new use cases [2].

Adoption of OCPM at industrial scale depends, among others, on time-efficient access to OCED. Industrial implementations of process mining over

sequential event logs rely on optimized columnar query engines to realize low running times across various query workloads [19,11]. Response times decide how such analyses can be used, as interactive exploration requires answers within seconds [16]. However, the richer structure of OCED prevents reusing existing implementations for sequential logs for OCPM. To support OCPM workloads over OCED, research explored various *data engines* (comprising data structures, stores, and access methods), such as relational databases [10, Sect. 5], graph databases [9], Python dataframes [6], and custom in-memory data models [14,15]. Various experiments have shown that query execution times can differ substantially between data engines and due to details of the data model [15,12].

There is currently no systematic comparative analysis of runtime performance of different OCED implementations in different data engines for various OCPM workloads. This paper presents a first benchmark to address this gap. In Section 2, we identify representative data engines of different data management paradigms (relations, graph, general and custom in-memory data structures) for implementing OCED, as well as a first collection of different OCPM workloads from literature. We synthesize these into a first proposal for a benchmark setup in Section 3. There, we additionally identify as an important parameter in implementing OCED whether the data schema is weakly typed (i.e., generic stores for objects and relations between objects) or strongly typed (i.e., separate stores per object type and relation type). As a first step, we apply this preliminary benchmark setup to a single real-life dataset, BPIC2017, for which dataset-specific query workloads are documented in existing literature (Section 5.2).

Our results in Section 4 show that only the custom engine reaches the fastest runtime category on every workload, yet general-purpose engines like `Polars` and `DuckDB` often land in the same category, especially for querying and performance analysis. Performance within a single engine family also differs widely, often by one or two orders of magnitude. Beyond the measurements, a practical contribution of this work is in the harness itself: Each of the eight workloads is formulated natively in every engine's paradigm (SQL, Cypher, dataframe operations, and Rust), yielding a public collection of reference implementations for common OCPM problems that can be reused and extended independently of the benchmark. We discuss the results, practical recommendations, and future directions for this benchmark in Sections 5 and 6.

2 Background

In this section, we provide background on OCPM, motivating the benchmark design and setup described in Section 3. Adoption of the object-centric paradigm in tooling depends not only on the conceptual fit of object-centric modeling, but on whether the underlying engines and data models can carry the workloads OCPM analysts actually run on real-life logs. OCPM workloads tend to be more demanding than case-centric ones because events relate to many objects of multiple types, removing the single-case key and forcing analyses to traverse cross-type relations [1]. OCED has emerged as the common concep-

tual model underlying OCPM [10,18]. Concrete OCED implementations include the OCEL 2.0 specification [4]. A separate line of work models OCED as event knowledge graphs (EKGs) [9]. On top of this common model, however, OCPM tools and approaches diverge along two largely independent axes: The *engine* they pick to query the data (from relational and graph databases to dataframes and custom in-memory structures), and the *workload class* that the downstream analysis imposes on it (from control-flow extraction to object-centric queries, performance analysis, and BI/KPI aggregation).

OCPM has no agreed-on benchmark workloads or datasets yet, and prior comparative-performance evidence is sparse: Khayatbashi et al. [12] and Küsters et al. [15] compare graph and relational representations on selected query workloads, and we are not aware of work that systematically varies both axes.

Section 2.1 recalls OCED, Section 2.2 surveys the four engine families used in OCPM, and Section 2.3 introduces four workload classes drawn from existing OCPM tooling and literature.

2.1 Object-Centric Event Data

Object-centric event data consists of events and objects of different types linked through relationships. Each event has an event type (its *activity*) and a timestamp, and each object has an object type. Unlike case-centric event logs, an event can refer to multiple objects of multiple types, and objects can themselves be related to other objects. The OCED core model [10] captures this with four parts: events, objects, an event-to-object (E2O) relation, and an object-to-object (O2O) relation, where both relations carry a qualifier per reference.

In our evaluation, we use the BPI Challenge 2017 log [8] in an OCEL 2.0 encoding [4]. Its immediate source is the OCEL 1.0 log of Khayatbashi et al. [12], who transformed an event knowledge graph [9] of the case-centric BPIC2017 XES log [8] into OCEL. We convert this log to OCEL 2.0 and enrich it with object-to-object relations, linking each `Application` to the `Offers` and `Workflow` of its case. To make all workload results unique, we order events that share a timestamp by their event identifier and offset each by successive microseconds. The full transformation scripts and resulting dataset are available in the harness repository. The log captures a loan application process with four object types: `Application`, `Offer`, `Workflow`, and `Case_R`. For example, event `A_Accepted` refers to one `Application` and one `Case_R` object.

2.2 Engines for Object-Centric Event Data

Publicly available implementations using OCED differ widely in how they store and access the data. We group the choices into four families and name the representatives that anchor each family in Section 3.2.

Relational Relational databases are both the source systems holding organizational process data and the backend of several SQL-based process mining

approaches, and the OCEL 2.0 specification defines a relational `SQLite` reference format [4]. Representatives are `SQLite` as a row-oriented and `DuckDB` as a columnar embedded engine (e.g., used in [7] and the ToTeM tool `https://github.com/LukasLiss/totem-tool`).

Dataframe Many published Python OCPM scripts and notebooks consume OCED as dataframes via PM4Py [6] (e.g., the Ocelescope tool `https://github.com/promi4s/ocelescope` or ToTeM). Representatives are `Pandas` as the reference dataframe library and `Polars` as a columnar, query-planned alternative with SQL support.

Graph Esser and Fahland [9] model OCED as event knowledge graphs (EKGs) in `Neo4j`, and Khayatbashi et al. [12] compare EKGs and OCELs as alternative representations. Representatives are `Neo4j` as a server-based property graph and `Kuzu` as an embedded property graph, both queried with Cypher.

Custom in-memory Some OCPM tools use a linked in-memory structure tuned for object-centric traversal. We use `Rust4PM` [14] as a representative, accessed via Python bindings (`https://rust4pm.aarkue.eu/docs`), which stores OCEL in a linked structure with relational indices. As PM4Py's in-memory `OCEL` object is a thin wrapper over dataframes, we count it as part of the dataframe family.

The four families are not claimed to be exhaustive, but account for many of the engines used by current publicly available OCPM tooling and literature. Within each family, we select the engines these tools and this literature build on. Engines deployed in industrial applications, for example distributed database systems, are out of scope for this work.

2.3 Workload Classes on Object-Centric Event Data

The cost of an OCPM analysis depends on the access pattern its workload imposes on OCED. No systematic overview of these workloads exists, so we group them by access pattern, drawing the classes from workloads common in published OCPM tooling and process mining practice [5,18]. We distinguish four classes. The concrete benchmark instances follow in Section 3.1.

(P1) Per-object-type control flow For a fixed object type, the algorithm gathers all relevant events per object, orders them by timestamp, and aggregates the resulting traces into a directly-follows multiset (DFG) or a multiset of trace variants. This is the workload behind most OCPM extensions of classical control-flow discovery [5,3,17].

(P2) Object-centric queries A query enumerates instantiations of event-and-object combinations and filters them by boolean predicates over the bindings. OCED encoded as EKG [9] allows Cypher queries for multi-hop traversal, including cross-object joins between events sharing an entity. OCPQ [15] evaluates a tree of nested query nodes that combine cardinality, temporal, and cross-type

predicates. The two surface languages differ (Cypher vs. the OCPQ tree language), but the access pattern is the same: enumerate bindings, evaluate predicates, return the remaining instantiations.

(P3) Object-centric performance analysis Object-Centric Performance Analysis, introduced by Park et al. [17], mainly has two steps: 1. For each event e, the workload first *links* e to the set of events it depends on through shared objects (via E2O and O2O, or via typed directly-follows edges materialized in an EKG [9]). 2. It then runs a *temporal computation* over the timestamps of those preceding events to produce a per-event value. Concrete instances of the temporal step are scalar measurements computed from minima, maxima, and differences over those timestamps, such as flow time, synchronization time, sojourn time, and pooling time [17,3], as well as the identification of the "responsible" delaying or enabling event, i.e., the related event whose timestamp is closest to, or furthest from, the timestamp of the event under consideration. Per-event measurements are sometimes also aggregated to the type level. The dominant cost of these kinds of workloads is imposed by many independent E2O/O2O joins and sorting or selection per event. The intermediate results typically have a cardinality in $\mathcal{O}(|\text{events}|)$, i.e., one row per event in the log.

(P4) BI/KPI workloads Reports and dashboards over OCED consume *aggregate or distributional* summaries, either globally or on the level of event or object types. This usage is exemplified by the process-metric operators for common KPI calculations, such as in Celonis PQL [19] and is the OCED counterpart of classical BI aggregation. The workload often groups rows by activity or object type and computes counts, sums, or averages per group, or evaluates an existence or ordering predicate per object and counts the objects that satisfy it. Additionally, one or two relational hops (E2O or O2O) may precede the grouping. KPI-based workloads are dominated by simple per-entity computations with subsequent aggregation to a few or even a single scalar.

While other workloads can also be aggregated to produce KPI-like measures, we still classify KPIs as a separate workload class, focusing on computationally less expensive intermediate computations in contrast to the other classes.

These four workload classes are not exhaustive. However, they specifically cover access patterns that admit comparable implementations across all four engine families and that the OCPM community has published tooling for. Workloads that require engine-specific extensions (e.g., process discovery routines that are not easily translatable to SQL) are out of scope here.

The identified engine and workload classes form the dimensions of our benchmark: For the evaluation, we instantiate each of them with at least one concrete implementation, introduced next.

3 Setup

The proposed benchmark has two axes: The workload classes (Section 3.1), and seven engines in four families (Section 3.2). We measure each engine on ev-

ery workload, and call one such measurement a *cell*. A dedicated sub-study (Section 3.3) investigates whether a strongly-typed database schema influences workload performance. As a dataset, we use the BPI Challenge 2017 log [8] in an OCEL 2.0 encoding as described in Section 2.1, as it combines rich object-centric characteristics, including object-to-object relations, with documented query workloads from existing literature. The dataset contains 1 202 267 events with 26 distinct activities and 106 162 objects across four types.

3.1 Workload Classes and Corpora

We define concrete workloads by instantiating each access-pattern class from Section 2.3 with two concrete tasks or corpora. A corpus groups workloads that share the same access pattern. Instances within a corpus differ only by input parameter (e.g., the object type) and are reported as means across the parameters.

P1 `DFG`: Weighted Directly-Follows-Graph and `Variants`: Unique trace variants with counts, each instantiated per the four BPIC2017 object types.

P2 `OCPQ`: 7 OCPQ trees from [15] and `EKG`: 3 of the EKG Cypher queries from [9].

P3 `Sync`: Per-event synchronization time [17] (span between the earliest and latest events directly preceding it through a shared object) together with the object linking the latest (the delaying object), and `Sojourn`: Per-event sojourn time [17] (time since the latest event preceding it for some shared object). For both workloads, the 100 rows with the highest durations (i.e., synchronization time or sojourn time) are returned, sorted by duration.

P4 `Types`: The joint activity-by-object-type distribution over E2O (a heatmap based on type-level aggregation) and `Conv`: Conversion rate of `Application` objects with a related `Offer` that has an `O_Accepted` event (a KPI based on O2O+E2O traversal).

3.2 Engines

Based on the review of engine families in Section 2.2, we compare seven engines that current OCPM tools or literature use: `SQLite` [4,15] and `DuckDB` [7,15] (relational), `Pandas` [6] and `Polars` [14] (dataframe), `Kuzu` [15] and `Neo4j` [9,15] (graph), and `Rust4PM` [14,15] (custom), accessed via Python bindings with an in-memory linked data structure tuned for OCED relationship traversal. Each engine uses one fixed set of defaults. Per-query tuning is out of scope for this paper, though for many engine-workload pairs we compared several formulations and kept the fastest, e.g., native dataframe code versus using `Polars`' SQL interface. The alternatives, and the procedure used to select among them, are documented in the harness repository.

Most workloads on `Rust4PM` are implemented as native Rust code, rather than via a dynamic query language. Thus, our benchmark specifically compares what an OCED-native library (`Rust4PM`) can achieve and measures the specific native-code vs. generic query-language gap (discussed explicitly in Section 5.2). As an exception to this, the P2 workloads (OCPQ and EKG) are evaluated as

dynamic OCPQ trees using the OCPQ engine based on `Rust4PM`, which uses the same underlying OCED data structure. This differentiation is made deliberately, as querying workloads are inherently dynamic, for example, allowing users to create and execute complex queries at runtime.

3.3 Schema Variants Substudy: Weak vs. Strong Typing

During initial experiments, motivated by engines that require explicit schema definitions such as `Kuzu`, we observed that performance also depends on whether OCED is stored weakly typed, e.g., generic event and object tables, or strongly typed, e.g., one table per object type and activity. Our baseline schema uses typed entity tables and weakly typed relationship tables (one for E2O and O2O each). We compare it against weakly typed variants of `DuckDB`, `SQLite`, and `Kuzu`, where all events and objects sit in one generic table with the type as a column. In Section 4.5 we compare the performance impact of entity typing.

3.4 Measurement Protocol

Each engine implements every workload natively (e.g., in SQL, dataframe operations, Cypher, or Rust code combining existing library calls). The timed region ends when the engine has produced a Python value of identical shape across engines. This boundary matches how an analyst actually consumes the result: The translation back into a Python object counts as part of the workload, so engines that return data in their native form pay that conversion cost inside the measurement. Cells are executed sequentially in isolated Python subprocesses with randomized execution order. The first, *cold* run is discarded, and the reported time is the mean of N=10 warm wall-clock runs. Every cell runs under a 60-minute wall-clock budget for all runs. All results are checked for correctness, i.e., it is verified that all engines produce the same output for each workload.

Evaluation ran on a compute node in an isolated container, pinned to 16 physical cores of an Intel Xeon Platinum 8468 CPU and 32 GB of memory. Software versions: Python 3.14.5, duckdb 1.5.3, polars 1.41.2, pandas 3.0.3, r4pm 0.5.5a6, kuzu 0.11.3, SQLite 3.53.1, and Neo4j 5.26.27 (Python driver 6.2.0). `SQLite` runs in memory, `DuckDB` and `Kuzu` by default use up to 80 % of system memory, and `Pandas` and `Polars` run uncapped by default. `Neo4j`'s default page cache does not necessarily keep the whole dataset in memory, so we raise its heap and page cache to 16 GB each, above the 3.6 GB store, so none is memory-bound. `Neo4j` indexes node identifiers and event timestamps, `Kuzu` indexes its primary keys (only supported index type), and the relational engines use the identifier and relationship-endpoint indexes of the OCEL 2.0 export. In contrast to all other engines, we materialize directly-follows edges for the graph database engines `Neo4j` and `Kuzu`, as done in reference work [9,12]. `Neo4j` would additionally support a relationship-property index on the directly-follows edge, which speeds up `DFG` by over 3×, but we omit it, as `Kuzu` cannot define an equivalent and the reference EKG encodings also leave it unindexed. The image and pinned dependencies are published with the harness.

4 Results

Table 1 shows the results of our benchmark. We interpret runtimes by order-of-magnitude band (under 10 ms, under 100 ms, under 1 s, and above), not by exact rank, considering sub-second responses to be *interactive*.

Table 1. Mean warm runtime (ms) on BPIC2017 per engine and access pattern. Cells are shaded by order-of-magnitude band, darker = faster: under 10 ms, under 100 ms, under 1 s, and at or above 1 s. Cells in each column's fastest band are in bold. Repeated runs of the same cell typically differ by about 3 % (median relative standard deviation).

		P1: Control flow		P2: Queries		P3: OC-Perf		P4: BI/KPI	
Family	Model	`DFG`	`Variants`	`OCPQ`	`EKG`	`Sync`	`Sojourn`	`Types`	`Conv`
Relational	SQLite	23886.2	4537.3	28418.0	106623.3	38626.2	36081.9	2535.9	317.0
	DuckDB	59.5	114.5	**47.3**	**38.5**	251.5	**99.0**	47.0	26.9
Dataframe	Pandas	440.8	1148.2	148.3	172.4	3322.8	1813.4	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	1625.0	550.2	445.2	5699.2	3836.5	1988.0	34.9
Custom	Rust4PM	**3.4**	**53.0**	**26.7**	**29.0**	**65.7**	**49.4**	**7.5**	**0.4**

4.1 DFG and Variants (P1)

`DFG` and `Variants` describe the per-object-type control flow that flattening-and-recombination-based OCPM discovery techniques consume [5]. On P1, the custom `Rust4PM` engine answers in the sub-100 ms band (3.4 ms on `DFG`, 53.0 ms on `Variants`), partly reflecting the native-code vs. query-language gap discussed in Section 5.2. The generic columnar engines `Polars` and `DuckDB` sit up to one band behind at 59.5–114.5 ms. `Pandas`, the graph engines, and `SQLite` are noticeably slower, several crossing into the ≥1 s band, with `SQLite` the extreme at around 24 s on `DFG`. On `Variants`, `Polars` (74.1 ms) edges past `DuckDB` (114.5 ms), likely because the columnar dataframe vectorizes string concatenation very efficiently.

Notably, the durations differ substantially between transactional object types (`Offer` and `Application`) and resource-like object types (`Workflow` and `Case_R`), likely caused by longer trace length for the resource-like types (more events to sort and higher overhead to construct traces). For every engine and both patterns, the resource-like types take longer, ranging from about 15 % on `Kuzu` up to two orders of magnitude on `Rust4PM`, where `Offer` variants take about 1 ms and `Case_R` variants take 134 ms.

4.2 Object-Centric Queries (P2)

OCPQ and EKG-style queries let an OCPM analyst express constraints across multiple object types in one shot, replacing chained filters. On the OCPQ corpus (Q1 – Q7), the custom-native `Rust4PM` (26.7 ms) and three generic engines, `Polars` (22.0 ms), `DuckDB` (47.3 ms), and `Kuzu` (62.2 ms) all respond in the sub-100 ms category. Notably, `Polars` even beats the native `Rust4PM` OCPQ engine. For `EKG`, the ranking is similar with smaller differences between these four engines, and with `Kuzu` and `DuckDB` at a tie for third place. Notably, this workload class uses the dynamic OCPQ engine based on `Rust4PM` instead of a dedicated native implementation for just the specific queries that are used here.

The engine family alone does not predict cost: On OCPQ and EKG, within the graph family, `Neo4j` is about 8.8–11.6× slower than `Kuzu`, and within the dataframe family, `Pandas` (148.3 ms) is about 5.1–6.7× slower than `Polars`, both a band behind on each workload. This pattern also generalizes across all workloads: `Kuzu` is always faster than `Neo4j` and `Polars` is always faster than `Pandas`.

4.3 Object-Centric Performance Analysis (P3)

OC-Perf measures the temporal coupling between events through shared objects and is the OCPM extension of classical performance analysis. The OC-Perf corpus has two scenarios (`Sync` and `Sojourn`) that share one access pattern: for each event, find the events that directly precede it through a shared object, then compute a temporal measure over them. For this kind of workload, the custom engine's advantage is most pronounced: Only `Rust4PM` (49.4–65.7 ms) stays within the <100 ms band comfortably, with `DuckDB` averaging at 99.0 ms for `Sojourn` and 251.5 ms for `Sync`, while `Polars` needs >370 ms for both. `Kuzu` is situated close to `Polars`, and all other engines (`Pandas`, `Neo4j`, `SQLite`) cross into the multi-second band.

4.4 BI/KPI (P4)

Dashboards and KPI reports reduce event- or object-level data to global or type-level summaries. `Types` and `Conv` instantiate the two dominant shapes. On `Types` (the activity-by-object-type distribution over E2O), `Rust4PM` (7.5 ms), `Polars` (13.4 ms), and `DuckDB` (47.0 ms) answer in well under 100 ms, as well as `Pandas` (99.9 ms), if only barely. `Kuzu` (155.6 ms) is a band behind, and `Neo4j` (2.0 s) and `SQLite` (2.5 s) are in the multi-second band. `Conv` (the `Application`-to-`Offer` conversion rate) returns a single number, and every engine answers within a second. On a workload this small, per-query overheads make up a visible share of the time. A notable outlier for `Conv` is `SQLite` with 317.0 ms, the only engine not reaching sub-100 ms for this workload.

4.5 Typing Sub-Study

To quantify the effect of weakly typed schemas, we calculate a speedup factor as the average runtime of the weak schema divided by the baseline results from Table 1, per workload. A factor >1 indicates that the strongly-typed (baseline)

schema is faster, and <1 that the weak schema is faster. Typing helps the most on workloads where typed tables can directly be targeted in queries, speeding up `OCPQ` and `EKG` across all tested engines. For example, OCPQ trees filter by object or event type at every node, so giving each type its own table lets the engine jump straight to the matching rows instead of scanning a generic store and filtering by type. The highest improvement factors are reached by `Kuzu` with 1.9× – 3.4× on the P2 workloads and `Conv` of P4. On others (`DFG` of P1, all of P3, and `Types` of P4), the performance generally degrades with typing (0.37× – 0.57×), while `Variants` is unaffected (1.1×). `DuckDB` significantly benefits from typing on the flattened control-flow workloads (P1), with improvements of about 1.4× on P1 and 1.2× on P2, while the other workloads are largely unaffected (0.94× – 1.06×). For `SQLite`, the weakly typed queries for P2 ran into a timeout, while the default-schema ones did not, indicating significant but not quantifiable improvements for P2. For P4, `SQLite` is largely unaffected, and for P1 and P3, weak typing is significantly faster (0.14× – 0.78×), indicating that it struggles with the unions across typed tables required for these workloads.

5 Discussion

We interpret results in Section 5.1 and discuss threats to validity in Section 5.2.

5.1 Interpretation

Across the four workload classes, the custom-native `Rust4PM` is the only one to reach the fastest band on every workload. Of the general-purpose engines, `Polars` and `DuckDB` are clearly the fastest and reach the top-band in 4 (`Polars`) and 3 (`DuckDB`) of the eight specific workloads, while `Kuzu` still reaches the top-band in two workloads, and all these three engines reach sub-second performance in the remaining workloads. Thus, fundamentally, each general-purpose engine family has one member capable of reaching interactive response times in all workloads, though only the custom `Rust4PM` stays in the fastest band, likely with reserves.

The gap between native and general-purpose engines depends on the workload: It is widest for `DFG` and `Conv`, and narrowest on the query workloads, where `Polars` even beats the `Rust4PM`-based OCPQ engine, and `DuckDB` and `Kuzu` are both also in the fastest band.

5.2 Threats to Validity

Scope All numbers come from BPIC2017 and the four workload classes that admit comparable implementations across every engine family. Larger object-centric datasets exist (e.g., BPIC2016 [12] as OCEL 1.0), but we focus on BPIC2017 for the existing query workloads we reuse [9,15]. Our results thus characterize one dataset, not scaling behavior, and datasets of a different shape or engine-specific workloads could shift the picture (Section 6). We also focus on engines used by publicly available OCPM tools on a single machine, leaving larger commercial or distributed systems out of scope.

Fairness Beyond enough memory to hold the dataset natively and the structural indexes each schema provides (Section 3), engines run without per-query tuning. We add no workload-specific indexes, so the results characterize engines under one uniform configuration rather than a per-query tuned ceiling. The most impactful such index, on the directly-follows edge, is quantified in Section 3. In contrast to the other engines, `Rust4PM` does not interface through a query language (SQL, Cypher, dataframe operations), so part of its lead is a native-code gap rather than a pure data-model effect. The P2 workloads are an exception to this, as `Rust4PM` uses the dynamic OCPQ query engine for them.

6 Conclusion

We benchmarked seven engines across four OCPM workload classes (Section 2.3) on the BPI Challenge 2017 log, as a publicly available, real-life object-centric dataset. Engine choice alone can change response times for the same workload by up to almost four orders of magnitude, from a few milliseconds to tens of seconds. The native `Rust4PM` engine implementation reaches the fastest band on every class, a lower bound on what an OCED-native implementation can achieve. The gap to this lower bound differs across workloads. Each engine family has a general-purpose engine that achieves sub-second response times on all workloads. Overall, the columnar engines `Polars`, `DuckDB`, and `Kuzu` come closest to the lower bound set by `Rust4PM` on 4/3/2 of the eight workloads. The row-oriented `SQLite` is the slowest, followed by `Neo4j`. Strong typing mainly helps type-filtering workloads, speeding up `Kuzu` by up to 3.4× and `DuckDB` by 1.4×, and taking `SQLite` from a timeout to computing the results within the duration limit. However, for both `Kuzu` and `SQLite`, typing is also slower elsewhere, for example, where splitting events and objects into per-type tables forces a union on every traversal. All numbers come from untuned engines on a single log and a single container. They do not cover per-query tuning, concurrent queries, or the zero-copy cloud architectures that high-end process mining increasingly uses.

Future work We plan to extend the benchmark along all dimensions: First, we aim to add more datasets, in particular configurable synthetic logs relating characteristics (object types, events per object, looping, concurrency) to performance, which also needs workloads that select types structurally rather than by name. Second, we aim to cover more workloads beyond the four shared access patterns, including advanced discovery that no single query expresses. Third, we aim to include more engines, such as server-based databases like PostgreSQL and distributed backends. Beyond per-class runtimes, end-to-end stress tests on larger logs would expose hidden costs such as intermediate-result handoff and setup amortization, alongside adoption factors such as memory usage, implementation effort, and deployment and maintenance cost.

The benchmark harness, per-engine implementations, and raw measurements are publicly available at: `https://github.com/aarkue/oc-performance-code`.

Acknowledgments. The authors gratefully acknowledge the German Federal Ministry of Research, Technology and Space (BMFTR) and the state government of North Rhine-Westphalia for supporting this work as part of the NHR funding.

References

1. van der Aalst, W.M.P.: Object-centric process mining: Dealing with divergence and convergence in event data. In: SEFM. pp. 3–25. LNCS, Springer (2019)
2. Abu Sbeit, A.A., Cavadini, F., Fahland, D.: Enabling object-centric process mining from time-series. In: ICPM 2025 Workshops. LNBIP, vol. 575. Springer (2026)
3. Adams, J.N., Park, G., van der Aalst, W.M.P.: OC-PM: analyzing object-centric event logs and process models. Int. J. Softw. Tools Technol. Transf. **24**(6), 847–861 (2022)
4. Berti, A., et al.: OCEL (object-centric event log) 2.0 specification. CoRR (2024)
5. Berti, A., Montali, M., van der Aalst, W.M.P.: Advancements and challenges in object-centric process mining: A systematic literature review. CoRR (2023)
6. Berti, A., van Zelst, S.J., Schuster, D.: Pm4py: A process mining library for python. Softw. Impacts **17**, 100556 (2023)
7. Bosmans, L., Peeperkorn, J., Smedt, J.D.: Pystack't: Real-life data for object-centric process mining. In: BPM (Demos / Resources Forum). pp. 208–215. CEUR Workshop Proceedings, CEUR-WS.org (2025)
8. van Dongen, B.: BPI challenge 2017 (2017), `https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b`
9. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. J. Data Semant. **10**(1-2), 109–141 (2021)
10. Fahland, D., et al.: Towards a simple and extensible standard for object-centric event data (OCED) - core model, design space, and lessons learned. CoRR (2024)
11. Kampik, T., Lücke, A., Horstmann, J., Wheeler, M., Eickhoff, D.: SIGNAL - the SAP signavio analytics query language. CoRR (2023)
12. Khayatbashi, S., Hartig, O., Jalali, A.: Transforming event knowledge graph to object-centric event logs: A comparative study for multi-dimensional process analysis. In: ER. pp. 220–238. LNCS, Springer (2023)
13. Klijn, E.L., Preuss, D., Imeri, L., Baumann, F., Mannhardt, F., Fahland, D.: Event knowledge graphs for auditing: A case study. In: ICPM 2023 Workshops. pp. 84–97. LNBIP, Springer (2023)
14. Küsters, A., van der Aalst, W.M.P.: Rust4pm: A versatile process mining library for when performance matters. In: BPM (Demos / Resources Forum). pp. 91–95. CEUR Workshop Proceedings, CEUR-WS.org (2024)
15. Küsters, A., van der Aalst, W.M.P.: OCPQ: object-centric process querying and constraints. In: RCIS (1). pp. 383–400. LNBIP, Springer (2025)
16. Miller, R.B.: Response time in man-computer conversational transactions. In: AFIPS Fall Joint Computing Conference (1). vol. 33, pp. 267–277. AFIPS / ACM / Thomson Book Company, Washington D.C. (1968)
17. Park, G., Adams, J.N., van der Aalst, W.M.P.: OPerA: Object-centric performance analysis. In: ER. LNCS, Springer (2022)
18. Seidel, A., et al.: Object-centric process management: A research manifesto. Information Systems **141** (2026)
19. Vogelgesang, T., Ambrosy, J., Becher, D., Seilbeck, R., Geyer-Klingeberg, J., Klenk, M.: Celonis PQL: A query language for process mining. In: Process Querying Methods, pp. 377–408. Springer (2022)