Object-Centric Process Querying in SQL: Translation Rules and In-Database Execution

Aaron Küsters, Justin Graß, and Wil M.P. van der Aalst

Chair of Process and Data Science (PADS), RWTH Aachen University
{kuesters,wvdaalst}@pads.rwth-aachen.de
justin.grass@rwth-aachen.de

Abstract. Object-Centric Process Querying (OCPQ) is a visual query language and accompanying tool for expressing queries and constraints over Object-Centric Event Data (OCED). Such queries span multiple object and event types that case-centric query languages cannot natively address. Its existing execution engine processes data in memory: Dataset size is capped by host memory, and the relational database systems that organizations already operate cannot serve as backends, forcing data export or migration. In this paper, we present a translation from OCPQ to SQL, defined as a recursive mapping over the OCPQ query tree with a rule per construct, emitting SQL for SQLite, DuckDB, and PostgreSQL. An *OCEL 2.0 Virtualization Layer* accommodates source schemas that diverge from the OCEL 2.0 layout via direct rewrite or `VIEW` generation, so the translation can target them without data migration. We evaluate using three types of workloads: Seven queries from prior work, an automatically generated corpus of 6 919 OCPQ trees, and a non-OCEL Enterprise Resource Planning (ERP) database schema queried through the virtualization layer. The resulting query outputs match the in-memory engine across all queries, demonstrating that the translation preserves its semantics. Our evaluation also shows the memory decoupling enabled by the translation: SQL backends scale to support large datasets on which the native OCPQ engine runs out of memory. When data fits in memory, the in-memory OCPQ engine remains fastest, with DuckDB roughly 3 – 12× slower and PostgreSQL 14 – 71× slower as drop-in replacements.

Keywords: Object-Centric Process Mining · Process Querying · SQL

1 Introduction

In organizations, process execution data are increasingly stored and queried directly from relational database systems [5,12]. At the same time, the domain of process querying is shifting from case-centric to *Object-Centric Event Data* (OCED), which preserves the multi-object structure of real-life processes [1]. OCED drops the single-case assumption that each event belongs to exactly one defined case. Instead, OCED contains a set of objects and a set of events of specified types, together with relationships between them. This lets OCED represent many real-life processes much more accurately than traditional, flat event

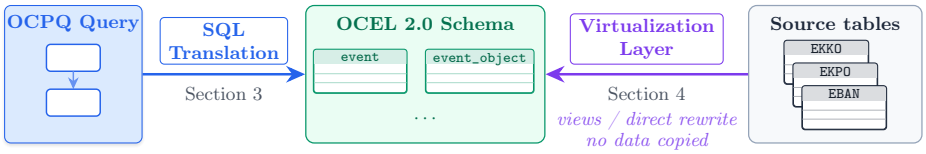

Fig. 1. An OCPQ query is translated to a SQL query that targets the fixed OCEL 2.0 schema. When the data resides in a different operational schema, e.g., the purchasing tables of an SAP system, the virtualization layer presents the OCEL 2.0 schema over the source tables through views or direct rewrite, without copying data.

logs. For example, in an order management process, the different objects interacting in the process may include `customers`, `orders`, `items`, and `packages`. OCEL 2.0 [4] specifies a standardized relational representation of OCED: One table per event and object type, with junction tables for the relationships between them, as well as general event and object tables. Organizational process data, however, rarely follows this layout. It typically resides in the source tables of operational systems, such as Enterprise Resource Planning (ERP) software, in schemas that OCEL-based tooling cannot use directly. Figure 1 previews the setting addressed in this paper: Queries are translated to SQL against the standardized OCEL 2.0 tables, and a thin virtualization layer maps these tables onto whatever schema the database actually holds, so no data has to be migrated or copied.

To query OCED, recent work has proposed Object-Centric Process Querying (*OCPQ*) [9], a graphical nested querying approach that *natively supports multiple object and event types*. OCPQ is attractive for process analysis, as OCED concepts, such as objects and events, are first-class citizens of the language. An analyst expresses a question spanning several objects and event types directly, rather than through the manual joins and subqueries an equivalent SQL formulation requires (cf. the verbose SQL query in Figure 2). OCPQ supports a range of tasks, from querying process data and checking constraints to creating feature tables for machine learning applications.

An OCPQ query is structured as a tree of *binding boxes*. Each binding box introduces typed event and object variables and filters them through predicates, most importantly the three basic predicates event-to-object (E2O), object-to-object (O2O), and time-between-events (TBE), which relate variables through OCED relationships and event timestamps. Children of a binding box extend the bindings of their parent and can be combined with size filters (e.g., "orders with at least three items"). In addition to filter predicates, constraints can be used to specify rules that should hold, marking violations in the output. Figure 2 shows an example OCPQ constraint from [9]: *After an Application was accepted, there should be at least one associated Offer accepted afterwards.* The root node queries combinations of application objects and E2O-related accepted events. The child node queries offer objects that are related to the parent application (O2O) with their offer-accepted events occurring afterwards (TBE). Each parent binding (application and application-accepted event) is marked satisfied if and

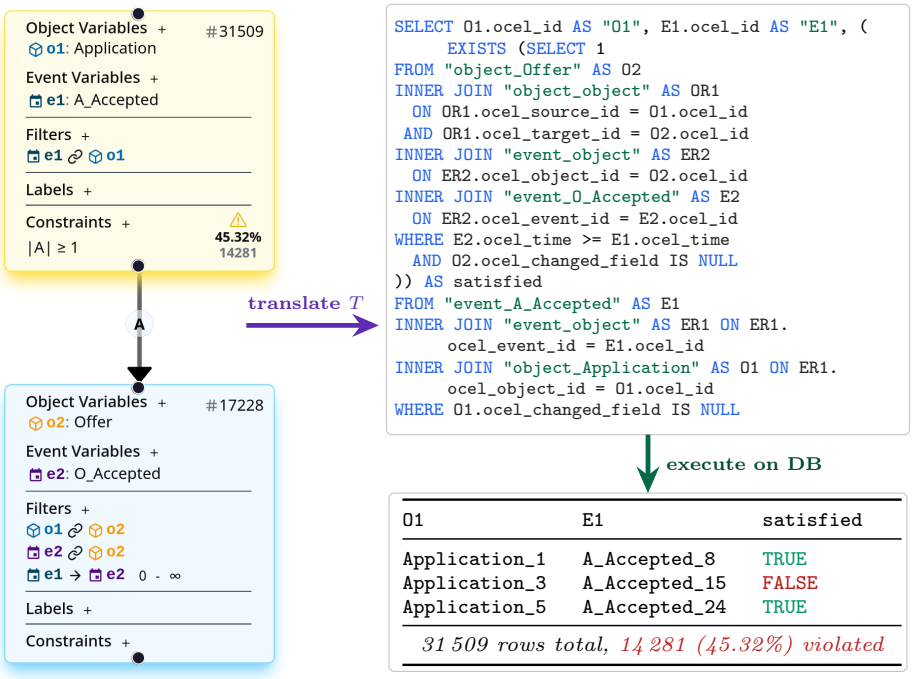

Fig. 2. *Left:* an OCPQ query tree modeling "*After an Application was accepted, there should be at least one associated Offer accepted afterwards*". The root node binds `o1: Application` objects, and `e1: A_Accepted` events linked by `E2O`. The child node binds `o2: Offer`, `e2: O_Accepted` linked by `O2O` and `E2O`, and specifies a time filter (`e2` after `e1`). The parent's size constraint $|A| \geq 1$ marks each output binding as satisfied if and only if the child set is non-empty. *Right:* the SQL query Q produced by T (Section 3) based on the OCPQ query tree. *Bottom:* schematic of the result rows on BPIC 2017.

only if the child binding set (offer and offer-accepted event combinations) is non-empty, as per the $|A| \geq 1$ Child-Binding-Set (CBS) size constraint.

The OCPQ tool has a high-performance execution engine that achieves fast runtime, even on datasets with millions of events [9]. However, the existing engine relies on in-memory processing of OCEL files using custom data structures. For organizations that store their process data in a database, this design is restrictive in several ways: The data must first be exported to an OCEL 2.0 [4] file before queries can be evaluated, query results then go stale as soon as the underlying data in the database changes, and the mature infrastructure that database systems already offer (indexing, distributed processing, etc.) cannot be leveraged. Moreover, in-memory processing has natural scalability limits: When the data exceeds host memory, the in-memory engine cannot produce results.

This paper investigates the following question: *How can OCPQ queries be translated to SQL so they execute directly on the relational databases that hold process data?* We answer it with a translation from OCPQ to SQL, defined as a recursive mapping over the binding-box tree with one rule per construct.

The structural predicates (the basic relations E2O, O2O, TBE, together with attribute filters, child-binding-set (CBS) predicates, and child-composition constraints) become SQL joins, `WHERE` clauses, correlated count or `EXISTS` subqueries, and a `satisfied` column. The same rules apply to SQLite, DuckDB, and PostgreSQL with small variations (e.g., for timestamp arithmetic). The right side of Figure 2 shows the SQL query produced for this OCPQ tree, and a schematic of the resulting rows after execution on a database.

OCPQ returns a set of output bindings for every tree node, as a table with one column per variable. SQL, in contrast, returns a single table per query. Our translation therefore targets the output bindings of the root node. Child results can be used in the parent output, for example through CBS predicates, but are not returned themselves. Retrieving child outputs on demand follows the same scheme and is left for future work.

Most OCPQ queries translate directly to SQL that can run on the database. However, OCPQ also allows for scripting using the Common Expression Language (CEL) (`https://cel.dev`), which has no direct SQL equivalent. We support these as an extension (cf. Section 3.2). When used, as much work as possible is pushed to the database, and only CEL predicates are post-processed host-side as residuals.

The translation targets the relational schema defined by OCEL 2.0. On such a schema, the emitted SQL runs directly without exporting or migrating data. To support scenarios where process data resides in a different schema, we add an optional *OCEL 2.0 Virtualization Layer* (Section 4) that mediates between this OCEL 2.0 surface and the actual source schema. It offers two paths: a *direct rewrite* that maps OCEL 2.0 table and column names to source names, so the translation targets the source schema with no view indirection; or, when the schema diverges further, generated `VIEW`s over the source tables that reshape the data into an OCEL 2.0-like layout.

We integrate the translation into the open-source OCPQ tool [9] and evaluate it on three relational backends: SQLite, DuckDB, and PostgreSQL. We evaluate and verify the translation on three workloads: Seven BPIC 2017 queries reused from prior work, an automatically generated corpus of 6 919 OCPQ trees, and an ERP-like database schema that differs from the OCEL 2.0 layout and is queried through the virtualization layer.

Contributions This paper contributes a SQL backend for OCPQ that enables queries to execute directly on the relational systems already deployed in organizations. It produces the same root output bindings as the in-memory engine, so downstream uses that consume root bindings are unaffected. Concretely, we make three contributions. First, we define a recursive translation T from OCPQ to SQL whose per-construct rules cover all binding-box predicates, with dialect-aware emission for SQLite, DuckDB, and PostgreSQL. As an extension, CEL filters, label functions, and AdvancedCEL quantifiers (CEL expressions that access the child binding set, e.g., for aggregation) are evaluated host-side. Second, we add support for diverging database schemas through a declarative OCEL 2.0 Virtualization Layer with direct-rewrite and generated-view options. Third, we

integrate this into the OCPQ backend as an open-source extension and evaluate it for correctness, runtime, and memory across different workloads.

2 Background

Object-Centric Event Data (OCED) models a process as events and objects of different types, related through qualifiers [1]. Objects may relate to other objects, and object attributes are time-varying. The OCEL 2.0 standard [4] specifies a SQL schema for OCED. The junction tables `event_object` and `object_object` carry the E2O and O2O relationships with an `ocel_qualifier` column. Per-type tables `event_<T>` and `object_<T>` hold attributes, with an `ocel_time` timestamp for events as well as object attribute history. The translator targets this schema as its logical surface. When the source schema differs, the OCEL 2.0 Virtualization Layer (Section 4) mediates the difference.

Object-Centric Process Querying (OCPQ) structures a query as a tree of *binding boxes* [9]. Each box declares typed event and object variables (V_E, V_O) and constrains them with predicates. The three *basic relational predicates* reference OCED structure directly: E2O(e, o, q) and O2O(o_1, o_2, q) require an event-to-object or object-to-object relationship between their arguments (with optional qualifier q), and TBE$(e_1, e_2, t_{\min}, t_{\max})$ bounds elapsed time between events. *Attribute filters* test an event or object attribute against a constant or range, where object attribute filters carry a time qualifier: `Always` and `Sometime` quantify over the object's attribute history, while `AtEvent` evaluates at a specified event's timestamp. *CEL filters* carry an expression in the Common Expression Language (CEL) evaluated per candidate binding, e.g., `e1.attr("amount") > 1000`. *Label functions* attach computed CEL values to surviving bindings under a named tag. Child edges are named (`A`, `B`, ...) so a parent can reference its children by edge label. A *binding* of box B is a mapping that assigns concrete events and objects to all variables in $V_E \cup V_O$ such that every predicate of B holds. Binding multiple events and objects of arbitrary types at once lets OCPQ express joint conditions across the process graph that case-centric query languages cannot. Evaluation is bottom-up: A binding c *extends* a parent binding b, written $b \sqsubseteq c$, if the two agree on every variable that b assigns. For a node N and an ancestor binding b, we write $out_b(N)$ for the set of all bindings that satisfy N's box and extend b; this is the *child binding set* produced for b on N's edge, and at the root, where b is empty, $out(N)$ is the full result of N. We follow the binding formalization of [9]. The *child-binding-set predicate* CBS$(A, n_{\min}, n_{\max})$ requires the child binding set $out_b(M)$ on edge A (to child M) to have cardinality in $[n_{\min}, n_{\max}]$. The shorthand $|A| \geq 1$ used in Figure 2 stands for CBS$(A, 1, \infty)$. Each predicate above can serve two roles: As a *filter*, which drops bindings that violate it, or as a *constraint*, which keeps the binding but labels it as satisfied or violated. Constraint results of child nodes can be used with logical compositions (e.g., OR). For more details, we refer to [9] or `https://ocpq.aarkue.eu`.

Figure 2 demonstrates these definitions. The root box binds an `Application` object and its `A_Accepted` event through E2O, the child binds an `Offer` and

its `O_Accepted` event (linked by O2O, E2O, and a TBE ordering), and the root constraint CBS($\texttt{A}, 1, \infty$) marks each parent satisfied when its child set is non-empty. Next, in Section 3 we sketch how such a query is translated to SQL.

3 Translating OCPQ to SQL

We define translation as a recursive mapping $T(N) = (Q_N, R_N)$ over each binding-box node N of the OCPQ tree. Q_N is the SQL fragment for N's subtree and R_N is a list of *residuals*: Host-side predicates that N (or one of its descendants) contributes but that have no SQL counterpart. The root node's Q is the query sent to the database. When a query carries no CEL ($R = \emptyset$), the returned rows are the final binding set, so a single SQL query fully captures the OCPQ semantics. The mapping recurses into children, and per-construct rules (Section 3.1) define how each predicate contributes to Q. Figure 3 shows the architecture. In the common case where no CEL predicates are involved, the translation maps the OCPQ tree to a single SQL query Q, which is then adapted by the virtualization layer to the source schema and executed on the database, with the returned rows as the final result.

When CEL predicates are used, the SQL query is relaxed to over-approximate OCPQ semantics. Thus, as much of the filtering and work as possible is pushed to SQL and the database, leaving only the CEL predicates as residuals to post-process on top of that. These residuals are then evaluated host-side over the returned rows, dropping or labeling each binding accordingly.

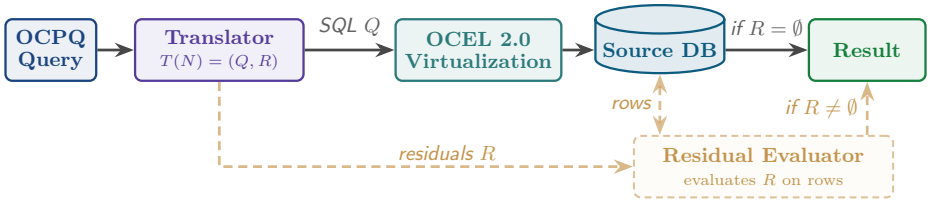

Fig. 3. Translation architecture. The translator T emits a SQL query Q and a residual list R. The query Q is executed against the source database via the Virtualization Layer. When $R = \emptyset$, the database rows are the final result. Otherwise, the CEL extension (Section 3.2) invokes an optional residual evaluator host-side.

3.1 Per-Construct Rules

Each variable in N gets a stable alias in the SQL query: `E`m for the m-th event variable and `O`k for the k-th object. Junction joins use `ER`n on `event_object` and `OR`n on `object_object`. Children inherit their parent's alias set so cross-node variable references resolve outward. Table 1 states each rule by the SQL it emits. A full per-rule formalization is beyond the scope of this paper; we state the semantics T preserves and its soundness below, and refer to the released implementation (Section 7) for the remaining details. As an example, E2O(e, o, q)

contributes an `INNER JOIN` on `event_object` with a `WHERE` on the qualifier q if specified. An event attribute filter on e contributes a `WHERE` on the per-type attribute column of e. Constraints *label* bindings rather than filtering them out. Every node, root or non-root, exposes a boolean `satisfied` column built from its constraint list via a `CASE` expression. Empty-child-set rules (SAT vacuously true, ANY false) match the in-memory engine.

The recursion composes SQL as follows: For each child edge ℓ of N, the child's query is embedded into Q_N as the body of the correlated subqueries that the CBS and composition rules referencing ℓ require (e.g., inside `EXISTS` or a `COUNT`), with parent variables resolved through the inherited aliases and the child's residuals appended to R_N.

Correctness We argue that T preserves OCPQ semantics, referring to the notation and definitions of Section 2 and [9]. Let N be a node with $T(N) = (Q_N, R_N)$ and b an output binding of N's parent (empty at the root). Assume N's subtree contains no CEL predicate, so $R_N = \emptyset$. Then the rows Q_N returns for b, projected onto the columns of $V_E^N \cup V_O^N$, are exactly $out_b(N)$. Constraints do not filter out rows, but indicate violations, so each returned row additionally carries a `satisfied` value built using a `CASE` expression and slightly modified SQL fragments from Table 1. The invariant follows by induction over the tree:

Leaf Nodes A leaf declares variables and filters only. Its query joins the per-type table of each variable through the `event_object` and `object_object` junctions, so one row is one type-correct assignment to $V_E^N \cup V_O^N$. Every filter is mainly encoded by the clause in the corresponding row of Table 1, and each such clause is simply a relational implementation of the predicate's definition. For example, $\mathrm{E2O}(e, o, q)$ holds iff the qualified `event_object` join has a matching row, and the `Always`/`Sometime`/`AtEvent` scopes select exactly the attribute rows their definition quantifies over. A row survives Q_N iff its assignment satisfies every filter, i.e. iff it lies in $out_b(N)$, where identical assignments arising from several witnessing junction rows are coalesced by the `DISTINCT` the emitter applies.

Inner Nodes Assume the invariant for every child. The query of a child node M is embedded in Q_N as a subquery whose outer reference is a parent binding b (the current row), so by hypothesis it enumerates only child bindings $out_b(M)$ of the parent binding b. The CBS rule reads this subquery as a `COUNT` or `EXISTS`, so the test $n_{\min} \leq |out_c(M)| \leq n_{\max}$ is decided exactly. Other compositions (e.g., SAT) combine the children's `satisfied` values similarly. Basic and attribute predicates of the inner node itself are handled as in the leaf case.

CEL Predicates A subtree with CEL predicates has no exact clause in Table 1. Instead, T widens Q_N to a superset of $out_b(N)$, and the residual evaluator removes the extra output rows host-side (Section 3.2).

Example Figure 2 shows an example mapping T. At the root, the $\mathrm{E2O}(e_1, o_1)$ rule aliases the per-type tables `E1`/`O1` and adds the `event_object` junction `ER1`, and the constraint $\mathrm{CBS}(A, 1, \infty)$ takes the `EXISTS` shortcut into the root's `satisfied` `CASE`, with the child's Q as its body. In the child, O2O and E2O emit the `object_object` (`OR1`) and `event_object` (`ER2`) joins to tables `O2`/`E2`, and the

Table 1. Per-construct translation rules. Each row shows the SQL fragment a construct contributes to Q. Used as a filter, the fragment restricts Q. Used as a constraint, the same fragment forms a branch of the parent's `satisfied CASE`. Predicates without a SQL counterpart (CEL, AdvancedCEL) are covered in Section 3.2.

	Construct	Contribution to Q
Basic relations	E2O(e, o, q), O2O(o_1, o_2, q) (event-/object-to-object)	`INNER JOIN` on `event_object` / `object_object`, and q adds an `ocel_qualifier WHERE`
	TBE$(e_1, e_2, t_{\min}, t_{\max})$ (time-between-events)	`WHERE` on the e_1, e_2 timestamps; $e_2 \geq e_1$ shortcut for $t_{\min} = 0$, otherwise dialect-specific epoch arithmetic (see *Dialect Handling*)
	Variable disequality $x \neq y$	`WHERE` on `ocel_id` inequality
Attributes	Event attribute filter	`WHERE` on the per-type attribute column of e
	Object attribute filter (`Always` / `Sometime` / `AtEvent`)	Scoped `EXISTS` on `object_<T>`; the time qualifier selects the historical rows considered (all, any, or the row valid at the named event, respectively)
Child-binding-set	CBS$(A, n_{\min}, n_{\max})$ (child binding count)	Correlated `COUNT(DISTINCT... )` over edge A (defaulting to 0 via `COALESCE`), checked against $[n_{\min}, n_{\max}]$; `NOT EXISTS` / `EXISTS` shortcuts for $[0, 0]$ / $[1, \infty)$
	CBS over projection variable v	As above, with `COUNT(DISTINCT)` over v instead of the full binding key
	Binding-set equality $A = B$	Two `EXCEPT` subqueries checking $A \setminus B$ and $B \setminus A$; both must return no rows
Composition	SAT ℓ (all children satisfied)	`NOT EXISTS` child row on edge ℓ with `satisfied = FALSE`; vacuously true on the empty set
	ANY ℓ (some child satisfied)	Correlated count of `satisfied = TRUE` rows on edge ℓ must be ≥ 1; false on the empty set
	AND, OR, NOT over $\{\ell_i\}$	Boolean composition of the per-edge SAT forms

zero-bound TBE reduces to `E2.ocel_time >= E1.ocel_time`. With $R = \emptyset$, Q yields 31 509 root bindings when executed on BPIC 2017.

Dialect Handling Generally, the same rules apply across SQLite, DuckDB, and PostgreSQL. The main difference is timestamp arithmetic for general TBE bounds: Writing t for the event timestamp column, SQLite emits `(julianday(t) - 2440587.5) * 86400`, DuckDB `EPOCH(t)`, and PostgreSQL `EXTRACT(EPOCH FROM t)`. When either bound is zero, the translation compares the timestamp columns directly (`E2.ocel_time >= E1.ocel_time`) on all three dialects.

3.2 Extension: Predicates Without a SQL Counterpart

The rules in Section 3.1 cover every OCPQ predicate that maps to a relational operator, so a CEL-free query translates to a single self-contained SQL statement ($R = \emptyset$). CEL filters and label functions are extensions appended to R and evaluated host-side over the rows returned by Q. Simple CEL expressions could

be pushed into a `WHERE` clause, which we leave as future work (Section 7). Other operators cannot be translated to SQL. For example, a regular-expression match in `attr("name").matches("iPhone \d+")` has no portable SQL form (SQLite, for instance, lacks regex support by default), so it is incompatible with the uniform, multi-dialect translation. More fundamentally, CEL is a general expression language that OCPQ extends with custom host-evaluated extensions, allowing for any predicate or functionality implementable as Rust functions.

Residuals are handled in two steps. First, T relaxes the SQL query so that Q over-approximates OCPQ semantics and never wrongly excludes a valid binding. Second, the residual evaluator post-processes the rows Q returns. It evaluates the CEL filters and labels and re-checks the structural clauses that Q could not enforce, such as upper-bounded child-binding-set (CBS) counts, then drops or labels each binding accordingly.

Relaxation targets any parent clause that constrains a child whose subtree carries residuals (i.e., CEL predicates). In such cases, the child count seen in SQL is only an upper bound and not yet final, because the evaluator may still drop child rows, so the parent rule cannot commit to it. The SQL form is allowed to over-include but never under-include. Clauses for which over-inclusion is safe stay in Q exactly: ANY, a CBS with no upper bound, and SAT or AND over CEL-free children. The remaining clauses, NOT, OR, an upper-bounded CBS, and binding-set equality, are relaxed from Q and re-checked host-side. For example, $\mathrm{CBS}(A, 0, 2)$ over a CEL-filtered child cannot be enforced in Q, since CEL may later drop child rows and lower the count. Thus, Q keeps every parent binding and the evaluator applies the exact bound afterward.

Post-processing the CEL predicates in R needs referenced attribute values, so T widens the relevant `SELECT` to project them if possible. Attributes unreachable through the parent's projection, e.g., across a CBS predicate, are fetched by a dedicated query against the referenced IDs, so the SQL path never materializes the full linked OCEL graph. CEL based on child bindings additionally needs the parent's bindings in scope for its child query, which the emitter provides with a correlated join where the dialect supports it, and otherwise by substituting the parent bindings into a batched subquery.

4 OCEL 2.0 Virtualization Layer

The translation so far targets the OCEL 2.0 database schema (e.g., assuming per-type event and object tables `event_<T>`, `object_<T>`, and `event_object` junction tables). Organizations rarely store process data this way. Source schemas diverge from this layout to varying degrees. Some differ only in naming, where a table maps to an OCEL type by renaming alone. Others differ more substantially, for example when an object's identifier spans several columns. The OCEL 2.0 Virtualization Layer bridges this gap. Based on a declarative description of the source schema, it presents the standard OCEL 2.0 SQL surface over an arbitrary source layout, so the translation runs against an organization's existing database without first migrating the data. The layer realizes the description in two ways.

Direct rewrite handles sources that differ only in their table and column names. The emitter substitutes the source names at emission time, producing plain SQL over the source tables with no view indirection and no overhead of its own. Defaults reproduce the standard OCEL 2.0 SQL exporter convention, so stock SQLite, DuckDB, and PostgreSQL exports work without configuration.

Generated views handle anything beyond renaming. The description carries `CREATE OR REPLACE VIEW` statements, installed once before queries run, that the queries then read from. The views hold the reshaping (row splits, type discrimination, deduplication, computed identifiers) and expose the OCEL 2.0 surface as named relations that any OCEL 2.0 consumer can query, not only our translator. Because the view definition can contain arbitrary SQL, this path is the more general of the two, able to persist, index, and reuse the reshaped surface in ways inline substitution cannot.

We evaluate the generated-view path as RQ3 in Section 5 on a synthetic ERP-shape source with deliberately stressful reshaping. The OCEL 2.0 Virtualization Layer is a first step towards executing object-centric queries directly on the databases where process data already lives. Federation across sources, automatic schema inference, and query optimization remain future work (Section 7).

5 Evaluation

We organize the evaluation around three research questions:

RQ1. How do the runtimes of the SQL backends compare to the existing in-memory engine? We investigate this on the query set from [9] and across a generated corpus of OCPQ queries, both on BPIC 2017.

RQ2. Do SQL backends decouple memory from data size and scale out to support larger datasets, and at what runtime cost? We evaluate this using memory stress on BPIC 2017 in two modes: Tight memory limits and scale-out through replication.

RQ3. What is the runtime overhead of running OCPQ over a non-OCEL SQL source schema via generated views?

Correctness is a precondition for every result we report. For each query we require the resulting binding set to match that of the in-memory engine. All cells reported below pass the gate, including all 6 919 trees of the generated corpus. These checks reinforce the translation correctness argued in Section 3.1.

Setup and Coverage We use the BPIC 2017 dataset (1 202 267 events, 106 162 objects) because of the pre-existing query corpus [9] and as it is among the largest publicly available real-life object-centric datasets. Each SQL backend opens the DB connection once, and all four engines report mean ± standard deviation over 10 warm runs (1 untimed warmup). We time the whole path from a query to its output bindings in OCPQ, including translation, execution, rebuilding bindings from the result rows, and any residual evaluation on the host. All experiments ran on an Intel Core Ultra 7 155U with 32 GiB RAM and NVMe SSD storage, running PostgreSQL 18.3, SQLite 3.52, and DuckDB 1.4. All three SQL backends

Table 2. Per-query wall-clock on BPIC 2017 (ms, mean with standard deviation subscript over 10 warm runs, darker cells faster). Construct tags follow Section 3.1, and Q4 is the running example of Figure 2.

Query	Features	In-mem	DuckDB	Postgres	SQLite
Q1	E2O, CBS	$8.7_{\pm 0.8}$	$40.2_{\pm 4.3}$	$615_{\pm 22.1}$	$365_{\pm 24.3}$
Q2	2×E2O, TBE, CBS	$19.4_{\pm 1.2}$	$80.9_{\pm 3.4}$	$713_{\pm 57.0}$	$1\,580_{\pm 734}$
Q3	O2O, CBS	$6.5_{\pm 0.7}$	$33.0_{\pm 1.5}$	$187_{\pm 23.1}$	$92.0_{\pm 7.3}$
Q4	2×E2O, O2O, TBE, CBS	$18.3_{\pm 0.9}$	$73.5_{\pm 3.5}$	$870_{\pm 75.8}$	$2\,529_{\pm 1\,321}$
Q5	4×E2O, O2O, SAT	$25.5_{\pm 2.5}$	$160_{\pm 6.7}$	$1\,215_{\pm 45.6}$	$2\,469_{\pm 1\,160}$
Q6	2×E2O, CEL aggregate	$37.0_{\pm 1.2}$	$448_{\pm 31.8}$	$598_{\pm 16.3}$	>60 s
Q7	2×E2O, 2×O2O	$27.5_{\pm 4.2}$	$226_{\pm 10.1}$	$397_{\pm 6.1}$	$4\,433_{\pm 113}$

share a uniform index set (primary keys on the four OCEL tables, indexes on both sides of each junction, and an `ocel_id` index on each per-type table). The query-set runtime and the virtualization study use all four engines. The corpus and the scale-out drop SQLite because it is not competitive at corpus and replication scale, and the virtualization study drops the in-memory engine because the layer targets SQL backends.

Runtime (RQ1) Table 2 reports per-query mean durations across all four engines on the seven-query BPIC 2017 set (Q1–Q7, reused verbatim from the OCPQ paper [9]). Construct coverage beyond these seven queries is provided by the generated corpus below. DuckDB stays within roughly 4 to 12× of in-memory. The only time it crosses the 10× factor is for Q6, where CEL is evaluated as residual. PostgreSQL trails by roughly 14 to 71×. SQLite is competitive on a few cheaper queries but drops to the bottom of the benchmark on Q2, Q4, Q5, and Q7. The Q6 outlier is SQLite-specific, where the CEL aggregate over child bindings does not complete within the 60 s timeout.
The in-memory engine wins where the OCEL fits in RAM. The SQL backends matter when it does not, avoiding export, staying fresh against database mutation, reusing existing indexes, and (per RQ2) lifting the RAM ceiling.

To test correctness beyond BPIC 2017, we additionally create seven advanced OCPQ queries over three simulated OCEL 2.0 logs (order-management, container-logistics, purchase-to-pay), spanning qualified E2O, more nesting, TBE windows, and advanced CEL labels. All three backends return the same output on all these queries.

Corpus Runtime Distribution (RQ1) Beyond Q1–Q7, we automatically generate a corpus of OCPQ trees on BPIC 2017 over five event types and two object types. Each tree shape satisfies $|V_E^N| + |V_O^N| \leq 3$ per node, depth ≤ 2, and at most two structurally identical sibling children; every variable must participate in at least one E2O or O2O predicate. On top of each connected shape, the generator enumerates every filter, every CBS filter variant, every AdvancedCEL configuration, and every composition operator (SAT, ANY, AND, OR, NOT). Labels are layered at the root only. Four construct families fall outside the corpus

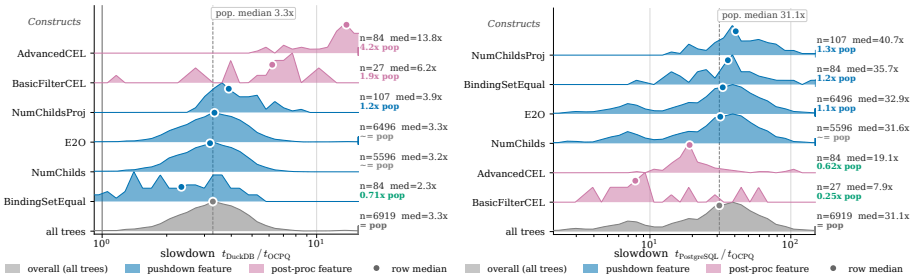

Fig. 4. Slowdown over the in-memory engine by OCPQ construct across the 6 919-tree corpus. Left: DuckDB. Right: PostgreSQL. Each row shows the $\log_{10}$ distribution of slowdowns over trees carrying the construct. The dashed lines indicate each engine's population median (DuckDB 3.28×, PostgreSQL 31.1×).

and are instead covered manually by tests: Constraint-mode CBS, E2O/O2O qualifier variation, non-zero TBE windows, and the `AtEvent`/`Always` object-attribute time qualifiers.

The generator produces **6 919 distinct trees** and the translator accepts every one. We run each on three engines (in-memory, DuckDB, PostgreSQL) under a 120 s per-tree timeout. All 6 919 trees yield exactly the same output binding sets (variable, `ocel_id` tuples, label values, the per-binding satisfied flag), with zero SQL errors and zero timeouts. Across the corpus, 99.78% of per-tree predicates land in SQL (51 130 of 51 241). The remaining 0.22% are residuals. Translation takes a median 0.11 ms per tree, under 1% of any backend's runtime.

In Figure 4, we break down the slowdown of each backend over the in-memory engine by query construct, showing the distribution of slowdowns for trees carrying each construct. On DuckDB (median 3.28×) the structural constructs (like NumChilds and E2O) are close to the median, while CEL-bearing constructs sit above it because their predicate is re-evaluated host-side. PostgreSQL is uniformly slower (median 31.1×). There, CEL-bearing trees sit *below* the median, likely reflecting structural corpus features (e.g., simpler query shape when CEL is generated) rather than a real speedup. Such residual predicates occur in only 2.2% of trees, so excluding them barely shifts either median.

Takeaway. When the data fits in RAM the in-memory engine is fastest, and running the same queries inside the database costs a small constant factor, a corpus median of 3.28× for DuckDB. Across the full 6 919-tree corpus every backend returns the same bindings, so the translation is correct over the construct combinations the corpus enumerates.

Memory Decoupling (RQ2) We run two memory stress tests: A *scale-out* replication ladder and a *tight* host-memory *cap*. *Scale-out.* We replicate BPIC 2017 by ID-suffixing from ×1 up to ×64 (76.9 M events, 153.9 M E2O at ×64), evaluating the Q1–Q7 queries from [9] at each step. Figure 5 reports the per-engine runtime (geometric mean) and peak memory usage across Q1–Q7. The in-memory engine runs out of memory at ×32 (terminated during OCEL load), while DuckDB completes the full ladder (13.8 GiB at ×64), a peak-memory gap of roughly 3.5 to 5.8× across the ladder. Notably, DuckDB also supports data larger than

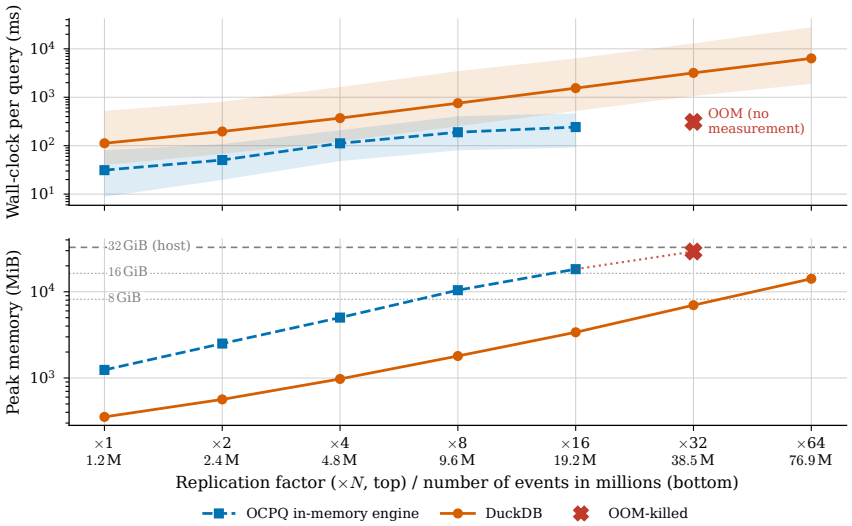

Fig. 5. BPIC 2017 replication ladder, in-memory engine vs. DuckDB. Runtime (top) is the per-engine Q1–Q7 geometric mean with min–max band. Memory (bottom): the heavy dashed line is the 32 GiB host, and the ×32 red "X" is the last observed in-memory value (28.6 GiB) before it ran out of memory. The runtime "X" is a log-log extrapolation of the in-memory trend (no measurement).

memory [11], so the scalability argument is not only a constant-factor memory reduction. On warm RAM-resident data the in-memory engine is 3.6 to 6.4× faster than DuckDB, and the gap does not close as data grows. *Tight cap.* We re-run Q1–Q7 on BPIC 2017 under a 400 MiB cap enforced as a hard per-process memory limit. The in-memory engine runs out of memory before the warmup completes. All three SQL backends complete the query set under the cap. SQLite peaks at 162 MiB and DuckDB at 350 MiB. PostgreSQL saturates the cap with default settings, and its client process holding query results adds 37 to 134 MiB outside the cap. DuckDB's higher footprint reflects its default buffer-pool target (80% of system RAM).

Takeaway. Memory is decoupled from data size. The in-memory engine runs out of memory at ×32 replication and under a 400 MiB cap, while the SQL backends complete every query in both settings.

Virtualization layer (RQ3) We evaluate the generated-view path (as direct rewrite only renames) on a synthetic ERP source with three reshapes from Section 4: A type-discriminated event table (i.e., multiple event types in one table), a deduplicated `Customer` object (i.e., an object type without a dedicated table), and a composite `ocel_id` via concatenation, at 10 000 and 100 000 source rows. Three queries V1–V3 run across SQLite, DuckDB, and PostgreSQL. View indirection on V1 and V2 is a constant overhead across both scales (DuckDB 1.2 to 1.4×, PostgreSQL 1.3 to 1.6×, SQLite 1.4 to 1.7×). V3 exposes a cliff on the row-oriented SQLite and PostgreSQL: the CBS's correlated count re-evaluates the view's per-row composite-identifier concatenation, which no underlying in-

Table 3. Virtualization cost on the synthetic ERP schema (ms, mean ± standard deviation over 10 warm runs). Native is the standard OCEL 2.0 tables, and Virt. is generated VIEWs over a non-OCEL source (darker is faster, and Virt./Native $\geq 10\times$ is shown in bold).

Rows	Query	DuckDB		PostgreSQL		SQLite	
		Native	Virt.	Native	Virt.	Native	Virt.
10 k	V1 (E2O)	$13.2_{\pm 2.6}$	$18.1_{\pm 2.5}$	$20.4_{\pm 2.6}$	$25.5_{\pm 1.2}$	$39.9_{\pm 9.9}$	$67.1_{\pm 3.4}$
10 k	V2 (CEL)	$21.4_{\pm 0.8}$	$25.5_{\pm 1.2}$	$28.8_{\pm 1.0}$	$38.0_{\pm 3.0}$	$53.2_{\pm 10.9}$	$73.8_{\pm 2.2}$
10 k	V3 (size filter)	$14.9_{\pm 2.7}$	$13.9_{\pm 1.7}$	$119_{\pm 11.6}$	**$18\,179_{\pm 1\,140}$**	$39.0_{\pm 4.1}$	**$72\,027_{\pm 3\,461}$**
100 k	V1 (E2O)	$136_{\pm 7.2}$	$174_{\pm 6.6}$	$225_{\pm 12.1}$	$355_{\pm 14.0}$	$671_{\pm 267}$	$970_{\pm 47.5}$
100 k	V2 (CEL)	$248_{\pm 7.7}$	$292_{\pm 24.2}$	$355_{\pm 12.5}$	$470_{\pm 7.4}$	$789_{\pm 272}$	$1\,095_{\pm 37.9}$
100 k	V3 (size filter)	$99.5_{\pm 4.8}$	$97.0_{\pm 4.0}$	$1\,098_{\pm 73.8}$	**>180 s**	$493_{\pm 32.8}$	**>180 s**

dex satisfies, while DuckDB's columnar engine processes it in a single pass. A stored generated column on the composite identifier could close the gap. Overall, querying non-OCEL source schemas through generated views is feasible, with performance depending on the engine and the schema's shape.

Limitations Soundness is argued by construction (Section 3.1) and checked through our evaluation query corpus but not established by a full formal proof. Under the tight cap the PostgreSQL client process is not subject to the cap (only the server), so we report client and server peaks separately. Scale-out scales by ID-suffixed replication and so stresses size, not graph complexity. RQ3 uses one synthetic ERP schema with worst-case reshape, and other production shapes remain untested. Host-side residuals carry a transfer cost when an unselective SQL fragment meets a selective CEL filter. Runtime figures use default engine configurations and one uniform SQL shape, not per-engine tuning, so cross-engine factors likely do not reflect the engines' full potential.

6 Related Work

Process query languages Prior process query languages (BP-QL [3], APQL [8], PIQL [10], Celonis PQL [15]) target case-centric settings and do not natively span multiple object types or multiple instances of the same object type. OCPQ [9] closes that gap at the language level. We contribute an orthogonal execution backend that maps OCPQ to SQL.

Process mining on relational databases Keeping process data inside the database during execution predates this work in the case-centric setting: DB-XES [14] computes intermediate structures via SQL rather than over an in-memory log, and Dijkman et al. [6] realize a discovery operator running entirely inside the database engine. Schönig et al. [13] and Riva et al. [12] translate DECLARE-style templates to SQL for constraint checking in the single-case setting. The object-centric setting adds multi-typed variable bindings joined through junction tables and correlated child-binding-set counts, neither of which arises in single-case DECLARE templates. We additionally handle non-translatable predicates as host residuals, a boundary those works do not encounter.

Object-centric process mining Object-centric behavioral constraint models [2] introduce object-centric predicates at the modeling level by grounding behavior in an ER-style schema of typed entities and relationships, anticipating the relational execution target we adopt for OCPQ. Esser and Fahland [7] query OCED in Neo4j via hand-written Cypher; graph databases fit OCED's structure naturally, but organizations rarely store process data in them, so querying one requires a data export that our approach avoids. Berti et al. [5] and Xiong et al. [16] extract OCEL from ERP and via virtual knowledge graphs. We contribute the executable OCPQ-to-SQL query layer over OCED already in relational storage, and assume a mapping from the raw data to an OCEL 2.0 surface to be given as input [1,4].

7 Conclusion

We presented a translation from OCPQ to SQL: A recursive mapping $T(N) = (Q, R)$ over the binding-box tree, with per-construct rules emitting SQL for SQLite, DuckDB, and PostgreSQL and routing CEL filters, labels, and AdvancedCEL quantifiers to host-side residuals. An OCEL 2.0 Virtualization Layer extends it to source schemas that diverge from the standard layout. Memory decouples from dataset size: The SQL backends complete BPIC 2017 under a tight memory cap and DuckDB scales to 76.9 M events, both of which exhaust the in-memory engine, while binding sets match it across all 6 919 corpus trees.

Practitioner guidance Use the in-memory engine when the log fits comfortably in RAM. Switch to a SQL backend when data approaches or exceeds the memory ceiling, when results must stay fresh against a live database, or when the data already resides in one. Among the SQL backends, DuckDB is the default for analytical and oversized workloads, PostgreSQL is suited to operational-database integration, and SQLite suits small embedded deployments.

Future work Simple CEL shapes such as `attr("X") == const` could be pushed into the SQL query instead of evaluated host-side. The generated-view path could be exercised on a real production schema. More database backends could be supported, and the emitter could pick between `EXISTS`, `COUNT`, and `LEFT JOIN` shapes based on index statistics. Moreover, the translation can be improved by optimizing the simplicity and performance of the emitted SQL. Finally, similar translation rules could target a second query language such as Cypher.

Reproducibility The translation code is available and integrated in OCPQ (see `https://github.com/aarkue/OCPQ/`). The evaluation setup, datasets, and scripts are available at `https://github.com/aarkue/ocpq-sql-eval`.

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: Unraveling the Fabric of Real Processes. Mathematics **11**(12) (2023)
2. van der Aalst, W.M.P., Artale, A., Montali, M., Tritini, S.: Object-centric behavioral constraints: Integrating data and declarative process modelling. In: Description Logics. CEUR Workshop Proceedings, CEUR-WS.org (2017)
3. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes with BP-QL. In: VLDB. pp. 1255–1258. ACM (2005)
4. Berti, A., Koren, I., Adams, J.N., Park, G., Knopp, B., Graves, N., Rafiei, M., Liß, L., genannt Unterberg, L.T., Zhang, Y., Schwanen, C.T., Pegoraro, M., van der Aalst, W.M.P.: OCEL (object-centric event log) 2.0 specification. CoRR **abs/2403.01975** (2024)
5. Berti, A., Park, G., Rafiei, M., van der Aalst, W.M.P.: A generic approach to extract object-centric event data from databases supporting SAP ERP. J. Intell. Inf. Syst. **61**(3), 835–857 (2023)
6. Dijkman, R.M., Gao, J., Syamsiyah, A., van Dongen, B.F., Grefen, P., ter Hofstede, A.H.M.: Enabling efficient process mining on large data sets: realizing an in-database process mining operator. Distributed Parallel Databases **38**(1), 227–253 (2020)
7. Esser, S., Fahland, D.: Multi-Dimensional Event Data in Graph Databases. Journal on Data Semantics **10**(1-2), 109–141 (2021)
8. ter Hofstede, A.H.M., Ouyang, C., Rosa, M.L., Song, L., Wang, J., Polyvyanyy, A.: APQL: A process-model query language. In: AP-BPM. pp. 23–38. LNBIP, Springer (2013)
9. Küsters, A., van der Aalst, W.M.P.: OCPQ: object-centric process querying and constraints. In: RCIS (1). pp. 383–400. LNBIP, Springer (2025)
10. Pérez-Álvarez, J.M., Díaz, A.C., Parody, L., Quintero, A.M.R., Gómez-López, M.T.: Process instance query language and the process querying framework. In: Process Querying Methods, pp. 85–111. Springer (2022)
11. Raasveldt, M., Mühleisen, H.: DuckDB: an embeddable analytical database. In: SIGMOD Conference. pp. 1981–1984. ACM (2019)
12. Riva, F., Benvenuti, D., Maggi, F.M., Marrella, A., Montali, M.: An SQL-Based Declarative Process Mining Framework for Analyzing Process Data Stored in Relational Databases. In: BPM Forum. LNBIP, vol. 490, pp. 214–231. Springer (2023)
13. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and Customisable Declarative Process Mining with SQL. In: CAiSE. LNCS, vol. 9694, pp. 290–305. Springer (2016)
14. Syamsiyah, A., van Dongen, B.F., van der Aalst, W.M.P.: DB-XES: enabling process discovery in the large. In: SIMPDA. pp. 63–77. CEUR Workshop Proceedings, CEUR-WS.org (2016)
15. 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)
16. Xiong, J., Xiao, G., Kalayci, T.E., Montali, M., Gu, Z., Calvanese, D.: A virtual knowledge graph based approach for object-centric event logs extraction. In: ICPM Workshops. pp. 466–478. LNBIP, Springer (2022)