

Object-Centric Process Querying in SQL

Translation Rules and In-Database Execution

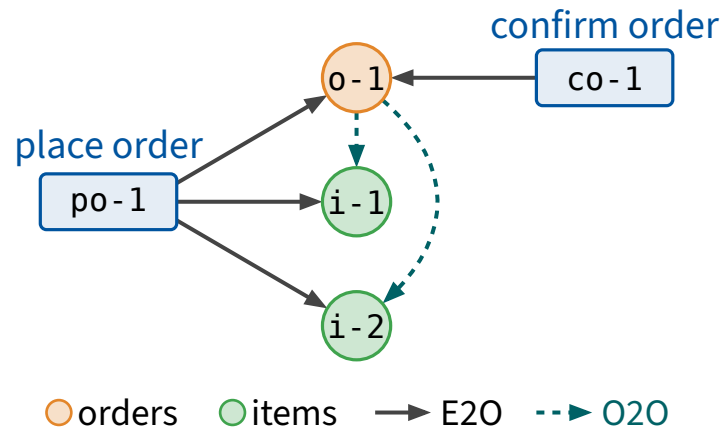
Aaron Küsters, Justin Graß, Wil M.P. van der Aalst

2026-09-29

Object-centric event data

An event can involve several objects of different types (event-to-object, E2O), and objects relate to each other (object-to-object, O2O).

The OCEL 2.0 standard specifies SQL tables for this data:



event_place_order

ocel_id	ocel_time
po-1	09:12

event_object

ocel_event_id	ocel_object_id
po-1	o-1
po-1	i-1
po-1	i-2
co-1	o-1

object_orders

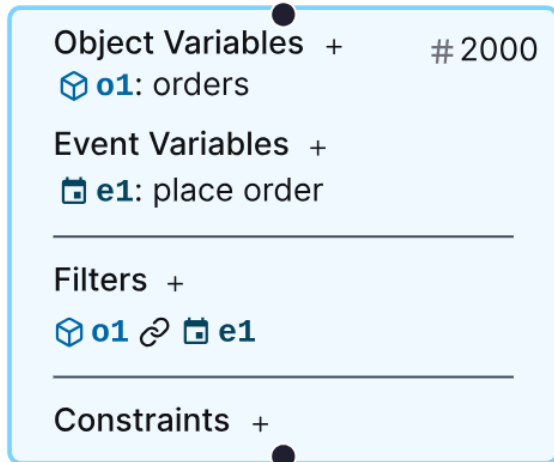
ocel_id
o-1

object_object

ocel_source_id	ocel_target_id
o-1	i-1
o-1	i-2

Binding boxes

OCPQ (Object-Centric Process Querying) is a visual query language. A query is a tree of binding boxes. Each box declares typed object and event variables and filters them, e.g., an E2O filter requires that e1 involves o1.

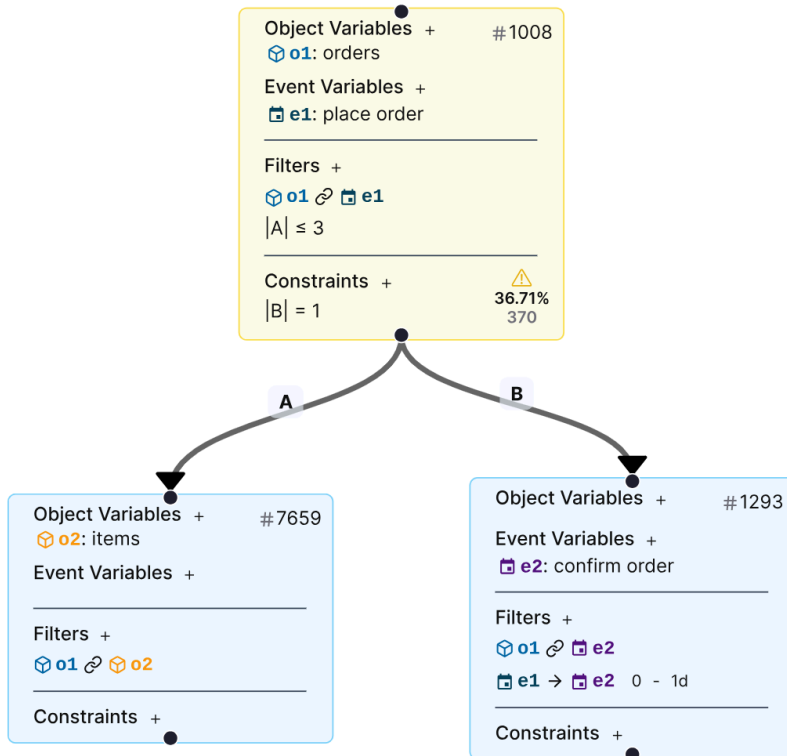


Output: 2,000 bindings

o1	e1
o - 1	po - 1
o - 2	po - 2
o - 3	po - 3
...	...

A **binding** assigns concrete objects and events to the variables. The output contains all bindings that pass every filter, here each order with its place order event.

Nesting and constraints



For each root binding of an order $o1$ and its place_order event $e1$:

- child A binds the items of $o1$
- child B binds confirm_order events of $o1$ within one day after $e1$
- $|A|$ counts the bindings of child A
- **Filter** $|A| \leq 3$: keeps 1,008 bindings
- **Constraint** $|B| = 1$: marks 370 bindings as violated

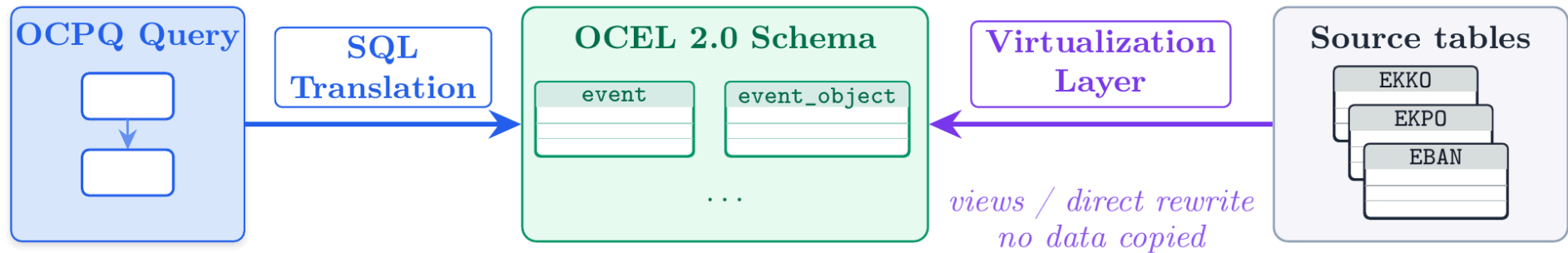
In-memory limits

The OCPQ tool evaluates queries with its own in-memory engine. It is fast, even on logs with millions of events, but it works on an OCEL file loaded into memory:

-  data must be **exported** to an OCEL file first
-  results are **outdated** as soon as the source database changes
-  indexes and other database features stay **unused**
-  the dataset has to fit into **main memory**

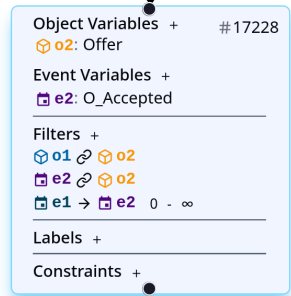
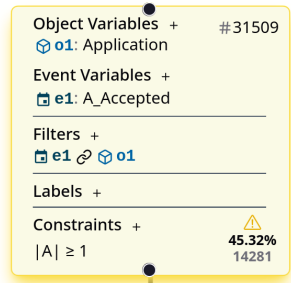
This work translates OCPQ queries to SQL, so they run **directly** on the database that holds the data.

Approach



-  The **translation** turns an OCPQ query into a SQL query over the standard OCEL 2.0 tables.
-  The **virtualization layer** presents other schemas, e.g., SAP purchasing tables, as OCEL 2.0 tables without copying data.

Running example



Q4 of the seven queries from the OCPQ paper, on the BPIC2017 loan application log: *After an Application was accepted, there should be at least one associated Offer accepted afterwards.*

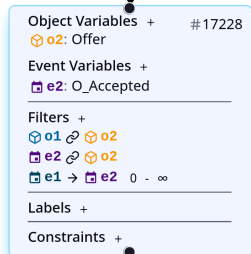
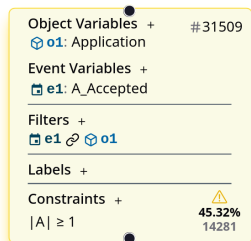
- Root:** each Application with its A_Accepted event
- Child:** each Offer of that Application with an O_Accepted event afterwards
- Constraint** of the root: $|A| \geq 1$

Output on BPIC2017: 31,509 rows, **14,281** (45%) violated.

o1	e1	satisfied
Application_1	A_Accepted_8	TRUE
Application_3	A_Accepted_15	FALSE
Application_5	A_Accepted_24	TRUE

Example translation

The root box becomes the outer query, the child box a correlated subquery inside EXISTS:



```
SELECT 01.ocel_id AS "01", E1.ocel_id AS "E1", (EXISTS (  
  SELECT 1  
  FROM "object_offer" AS 02  
  INNER JOIN "object_object" AS OR1  
    ON OR1.ocel_source_id = 01.ocel_id  
    AND OR1.ocel_target_id = 02.ocel_id  
  INNER JOIN "event_object" AS ER2  
    ON ER2.ocel_object_id = 02.ocel_id  
  INNER JOIN "event_0_Accepted" AS E2  
    ON ER2.ocel_event_id = E2.ocel_id  
  WHERE E2.ocel_time >= E1.ocel_time  
    AND 02.ocel_changed_field IS NULL  
)) AS satisfied  
FROM "event_A_Accepted" AS E1  
INNER JOIN "event_object" AS ER1 ON ER1.ocel_event_id = E1.ocel_id  
INNER JOIN "object_Application" AS 01 ON ER1.ocel_object_id = 01.ocel_id  
WHERE 01.ocel_changed_field IS NULL -- first row of each object
```

Translation rules

Each construct except CEL maps to one SQL pattern. Child boxes become correlated subqueries of their parent.

OCPQ construct	SQL
E2O, O2O: event-to-object, object-to-object	INNER JOIN on event_object / object_object
TBE: time between two events	WHERE on both timestamps
CBS: number of child bindings, e.g., $ A \geq 1$	correlated COUNT, or EXISTS / NOT EXISTS
SAT / ANY: all / some children satisfied	NOT EXISTS / correlated count on satisfied
CEL (Common Expression Language), e.g., regex matches	none, OCPQ evaluates them on the returned rows

The same rules produce SQL for SQLite, DuckDB and PostgreSQL. The main difference is timestamp arithmetic. With CEL, the SQL query returns a superset of the bindings, and OCPQ removes the extra ones afterwards.

Virtualization layer

Organizations rarely store process data in the OCEL 2.0 layout. A declarative description of the source schema maps it to OCEL 2.0 in one of two ways:

Direct rewrite

The source differs only in table and column names. The translator substitutes them, so the query runs on the source tables at no extra cost.

Generated views

The source needs more than renaming, e.g., row splits, type columns, deduplication or composite IDs. CREATE VIEW statements, installed once, present the OCEL 2.0 tables.

Evaluation setup

Seven queries

Q1 to Q7 from the OCPQ paper on BPIC2017: 1.2M events, 106,162 objects

Generated corpus

6,919 query trees that enumerate filters, child counts, CEL options and compositions such as SAT and ANY

ERP-like schema

three queries through generated views, at 10k and 100k rows

 In-memory engine vs. SQLite 3.52, DuckDB 1.4, PostgreSQL 18.3. Intel Core Ultra 7 155U, 32 GiB RAM.

 Mean of 10 warm runs, from query to result bindings. Every result matches the in-memory engine.

Seven queries

Mean runtime in ms on BPIC2017:

Query	Constructs	In-memory	DuckDB	PostgreSQL	SQLite
Q1	E2O, CBS	8.7	40.2	615	365
Q2	2×E2O, TBE, CBS	19.4	80.9	713	1,580
Q3	O2O, CBS	6.5	33.0	187	92.0
Q4	2×E2O, O2O, TBE, CBS	18.3	73.5	870	2,529
Q5	4×E2O, O2O, SAT	25.5	160	1,215	2,469
Q6	2×E2O, CEL aggregate	37.0	448	598	> 60 s
Q7	2×E2O, 2×O2O	27.5	226	397	4,433

The in-memory engine is fastest on all seven. **DuckDB** is 4 to 12× slower, **PostgreSQL** 14 to 71×. **SQLite** beats PostgreSQL on Q1 and Q3, but hits the 60 s timeout on Q6.

Generated corpus

6,919 generated query trees, run on the in-memory engine, DuckDB and PostgreSQL (SQLite is too slow at this scale):

✓ **All 6,919** trees return the same bindings on all three engines.

0

SQL errors or timeouts (120 s per tree)

0.11 ms

median translation time per tree

3.3×

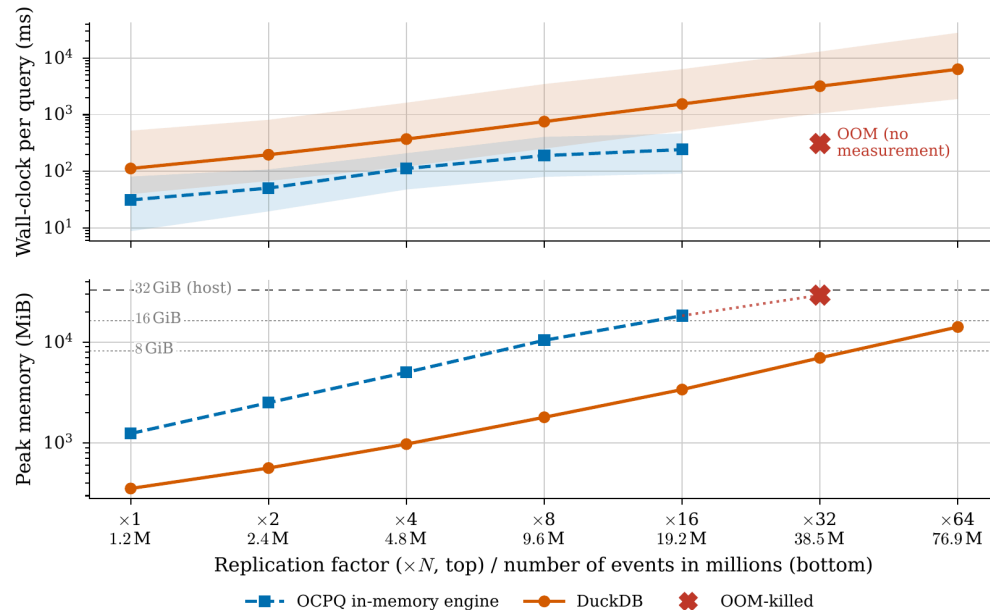
median slowdown of DuckDB vs. in-memory

31×

median slowdown of PostgreSQL vs. in-memory

Memory scaling

BPIC2017 copied up to 64 times (76.9M events), Q1 to Q7, in-memory engine vs. DuckDB:



The **in-memory engine** runs out of memory at **x32**.

DuckDB completes x64 with a 13.8 GiB peak.

Up to x16, the in-memory engine is 3.3 to 6.4× faster than DuckDB.

With a 400 MiB memory limit, all three SQL backends run Q1 to Q7 on BPIC2017. The in-memory engine does not.

View overhead

ERP-like source with 100k rows, runtime in ms, native OCEL 2.0 tables vs. generated views:

	DuckDB		PostgreSQL		SQLite	
Query	native	views	native	views	native	views
V1: E2O	136	174	225	355	671	970
V2: CEL	248	292	355	470	789	1,095
V3: CBS filter	99.5	97.0	1,098	> 180 s	493	> 180 s

On V1 and V2, the views add a factor of 1.2 to 1.6 (1.2 to 1.7 at 10k rows). On V3, PostgreSQL and SQLite recompute the view's composite IDs for every parent row and time out. DuckDB runs V3 through views as fast as on native tables (97.0 vs. 99.5 ms).

Backend choice

In-memory engine

fastest when the log fits comfortably in RAM

PostgreSQL

integration with the operational database

DuckDB

analytical workloads and data larger than memory

SQLite

small embedded deployments

Takeaways



One set of rules translates OCPQ queries to SQL for SQLite, DuckDB and PostgreSQL.



The SQL backends return the same bindings as the in-memory engine on Q1 to Q7 and on all 6,919 generated trees.



All three SQL backends complete Q1 to Q7 under a 400 MiB cap, and DuckDB at 76.9M events. The in-memory engine runs out of memory in both settings. When the log fits, DuckDB is 4 to 12× slower.



With generated views, queries run on non-OCEL schemas without copying data. V1 and V2 are 1.2 to 1.7× slower, V3 times out on PostgreSQL and SQLite.



Open source, integrated in OCPQ: github.com/aarkue/OCPQ

Thank you

Thanks!

 github.com/aarkue/OCPQ

 github.com/aarkue/ocpq-sql-eval

 kuesters@pads.rwth-aachen.de

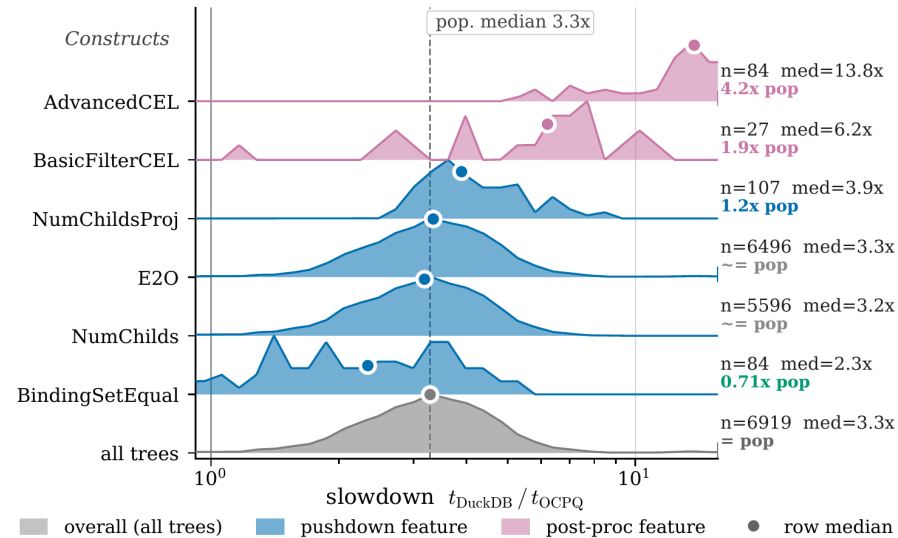


github.com/aarkue/OCPQ



Bonus

Corpus slowdown per construct



DuckDB over in-memory, 6,919 trees. NumChilds = CBS. Most structural constructs sit near the median. CEL-bearing constructs are slower, since OCPQ evaluates their CEL on the returned rows.

Next steps

- 🔹 Push simple CEL expressions, e.g., `attr("X") == const`, into SQL.
- 🏠 Run the generated-view path on a real production schema.
- 📄 More backends, and choosing between EXISTS, COUNT and LEFT JOIN from index statistics.
- 🔗 Target a second query language, e.g., Cypher.