

DeQL Studio: Declarative Decision-Making over Relational Data

Matteo Brucato Fjodor Kholodkov Soren Little Jakob Mayer Duc Nguyen

OSM Data

{matteo, fjodor, soren, jakob, duc}@osm-data.com

ABSTRACT

Databases separate what from how: users declare what data they want, and a query optimizer decides how to compute it efficiently. Decision-making has no such separation. Computing optimal actions means solving a mathematical optimization problem, and the user must choose how. That choice is the *formulation*, and alternative formulations reach the same optimum but can differ in solving time by orders of magnitude. A transportation problem, for instance, can be cast as a general mixed-integer linear program (MILP) or, far faster, as min-cost network flow (MCF). We demonstrate DeQL (Decision Query Language), a declarative SQL extension where users state, over relational data, what to decide, what constraints to respect, and what to optimize; the DeQL Planner chooses the formulation. In DeQL Studio, our interactive interface, attendees write DeQL queries, inspect the formulation the Planner selects, and run conventional scripts for comparison: on a GPU allocation problem with 2.7 million possible assignments, the conventional MILP takes Gurobi 26 s to solve to optimality, while the Planner detects the bipartite supply-and-demand structure directly in the query and selects MCF, and the engine reaches the same optimum in 0.88 s, 30× faster. Attendees edit queries and data, watching the engine adapt: removing one constraint changes the detected structure and the Planner switches formulation, while editing data warm-starts the solver from the previous solution.

PVLDB Reference Format:

Matteo Brucato, Fjodor Kholodkov, Soren Little, Jakob Mayer, and Duc Nguyen. DeQL Studio: Declarative Decision-Making over Relational Data. PVLDB, 19(12): 4614 - 4617, 2026. doi:10.14778/3827998.3828079

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/osm-data/deql-studio-analysis>.

1 INTRODUCTION

Prescriptive analytics uses data to compute optimal actions subject to constraints. The inputs to problems such as resource allocation, shift scheduling, and portfolio selection are typically relational: sets of entities and their attributes. Consider allocating 4,000 inference workloads across 1,000 GPU pools at minimum cost, subject to pool capacity limits and workload demand requirements. A database can join the underlying tables (Figure 1, top), filter assignments by hardware compatibility, and compute per-assignment costs; these

gpu_pools			workloads			assignment_costs		
pool_id	capacity	mem_gb	workload_id	demand	min_mem	pool_id	workload_id	cost
H100-E	130	80	llm-serve	8	40	H100-E	llm-serve	42
A100-W	85	40	embed	3	24	H100-E	fine-tune	63
T4-S	70	24	fine-tune	12	80	A100-W	embed	17
... 1,000 rows			... 4,000 rows			... 2.8M rows		

```
CREATE CANDIDATES gpu_assignments
DECISION KEY (pool_id, workload_id) AS
  SELECT pool_id, workload_id, capacity,
         demand, cost, mem_gb, min_mem
  FROM gpu_pools CROSS JOIN workloads
  JOIN assignment_costs
  USING (pool_id, workload_id);
```

```
DECIDE allocation_plan
FROM gpu_assignments
DECISION COLUMNS (
  active SELECTION BINARY,
  quantity CONTINUOUS)
WHERE mem_gb >= min_mem
SUBJECT TO
  CONSTRAINT pool_cap: SUM(quantity) <= capacity BY pool_id,
  CONSTRAINT meet_demand: SUM(quantity) = demand BY workload_id
MINIMIZE SUM(cost * quantity);
```

Figure 1: GPU allocation in our Decision Query Language (DeQL). **CREATE CANDIDATES** defines the candidate space by joining the input tables (top); **DECIDE** expresses the optimization (bottom).

standard relational operations produce millions of possible assignments. But the database cannot determine *which* assignments to activate: that is a constrained optimization problem. For a SQL query, the database handles the full pipeline; for decision problems, the user must choose a mathematical *formulation* (e.g., integer program, linear program, or network flow), encode it, and run a solver.

The choice of formulation determines which algorithms can solve the problem, and hence the running time. At the extreme, the solver faces either a polynomial or an NP-hard problem. Mixed-integer linear programming (MILP) can encode any problem with linear constraints over continuous and integer variables, so many practitioners default to it, but solving a MILP is NP-hard in the worst case, even for problems that admit a polynomial-time algorithm under another formulation. Many real problems do: the GPU allocation above can be solved as min-cost network flow (MCF) in polynomial time [1] or, under tighter conditions, as an assignment problem via the Hungarian algorithm [11]. Both run orders of magnitude faster than MILP, with the same optimal cost. Identifying faster formulations takes expertise, and the analysis must be redone whenever the problem or its data changes.

For the GPU allocation, the *structure* that enables MCF is a bipartite transportation graph: two entity types (pools and workloads) with independent capacity and demand constraints. A MILP formulation encodes the problem as a flat coefficient matrix in which the pool and workload constraints are indistinguishable rows, and

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097. doi:10.14778/3827998.3828079

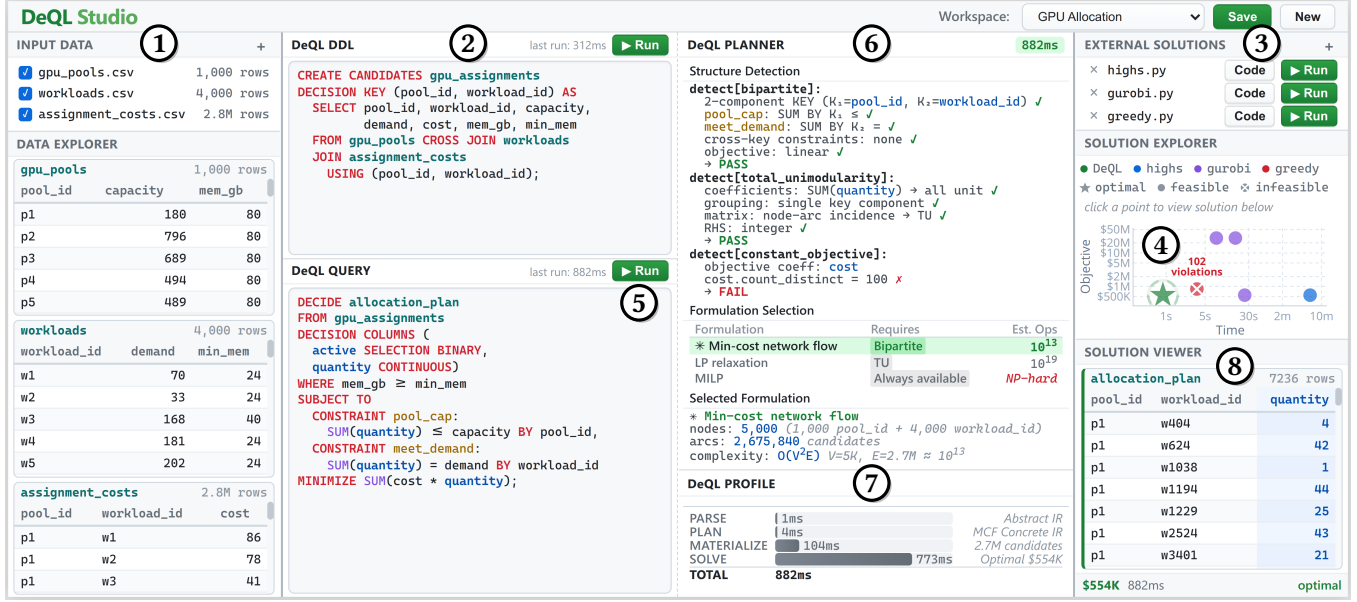


Figure 3: DeQL Studio during the GPU allocation demonstration (1,000 pools, 4,000 workloads, 2.8M assignment costs). ① Data Explorer with editable input tables. ② DeQL query editor. ③ Conventional solver scripts. ④ Solution Explorer chart. ⑤ Run DeQL query. ⑥ Planner: Structure Detection and Formulation Selection. ⑦ Profile panel. ⑧ Solution Viewer (result table).

(demand); and (5) the objective is linear. When all five hold, the component with the $\leq$ constraint provides the supply nodes and the $=/ \geq$ component the demand nodes; candidates become arcs (directed edges carrying flow), and the engine solves an MCF problem in polynomial time [1]. For example, a global constraint (no BY) violates (2).

Constant-objective (statistics-based) When the bipartite detector passes, this detector additionally consults candidate-level statistics. If `COUNT DISTINCT` on the objective’s coefficient column equals 1 and the demand constraint is an equality (so total flow is fixed), the objective is constant over all feasible solutions and the problem reduces from MCF to max-flow, which is $4\times$ faster.

Unit-demand/supply (statistics-based) When the bipartite detector passes, this detector checks `MIN` and `MAX` on the constraint right-hand sides; if both equal 1, all supplies and demands are unit and the transportation problem reduces to assignment, solvable by the Hungarian algorithm [11].

Total unimodularity (static) This detector relaxes the bipartite conditions, so it recognizes strictly more queries: every constraint need only be a bare `SUM` grouped by *at most* one component of a two-component `DECISION KEY` (so global constraints are allowed). The constraint matrix is then a bipartite incidence matrix, hence totally unimodular (TU) [13]. For an integer decision column, the detector also checks that the right-hand sides are integral; the Hoffman–Kruskal theorem [9] then guarantees an integral LP optimum, so the engine solves the LP, not the MILP.

Formulation Selection. Each detector that passes enables one or more formulations. The engine substitutes the candidate count and the entity count (pools and workloads) into each formulation’s complexity bound to estimate a worst-case operation count, then

selects the lowest. For the query in Figure 1, three are available: MCF (requires bipartite structure; optimal, polynomial), LP relaxation (requires TU alone; optimal, polynomial but slower), and MILP (always available; NP-hard in the worst case), so it picks MCF. Alternatively, it selects max-flow if the objective is constant, or the Hungarian algorithm if the problem reduces to assignment.

Implementation and scope. The DeQL engine is implemented in Rust (parser, three-IR pipeline, solver integration): HiGHS [10] and Gurobi [7] for LP and MILP, and OR-Tools’ [12] cost-scaling solver for MCF. Structure Detection is static pattern matching on the Abstract IR and candidate-level statistics, so its outcome is deterministic. DeQL Studio is a browser-based interface that communicates with the engine, executes user-provided solver scripts, and visualizes the pipeline alongside results.

Prior work. PaQL [2] introduced declarative package queries with ILP as the fixed formulation; sPaQLTools [5] demonstrated this interactively. DeQL generalizes PaQL’s translation to typed decision variables and grouped constraints, and its engine adds automatic Formulation Selection. SolveDB [14] embeds solvers in SQL but requires the user to choose the formulation; modeling languages (AMPL [6], Pyomo [8]) express a user-chosen formulation.

3 DEMONSTRATION

Figure 3 shows DeQL Studio during the GPU allocation demonstration: 1,000 GPU pools, 4,000 workloads, 2.7 million candidate assignments after hardware-compatibility filtering. We guide attendees through eight steps, then invite exploration.

Step ①: Inspect the input data. The presenter opens the Data Explorer (①), which lists three inputs: `gpu_pools` (capacity, memory tier), `workloads` (demand, minimum memory), and `assignment_`

costs (one row per pair). The goal is to allocate workloads to pools at minimum cost, respecting pool capacities and workload demands.

Step ②: Read the DeQL query. The presenter examines the DeQL query (②; Figure 1). `CREATE CANDIDATES` joins the three tables into a candidate space, materialized once and shared by the scripts and the engine alike; `DECIDE` declares decision columns, named constraints with `SUM...BY` grouping, and a cost-minimizing objective: the full problem, declaratively, in two short statements.

Step ③: Run conventional scripts. DeQL Studio can execute external solver scripts (③) and display their results alongside the DeQL result on the Solution Explorer chart (④). `greedy.py` sorts workloads by demand (largest first) and assigns each to the cheapest compatible pool with enough remaining capacity, overflowing onto its cheapest pool when none fits: ~4 s, \$844K, but violates 102 of 1,000 capacity constraints. `gurobi.py` and `highs.py` each build the same MILP formulation (2.7M binary and 2.7M continuous variables, plus explicit linking for `SELECTION` variables) and pass it to Gurobi [7] and HiGHS [10], respectively. Both solvers’ presolve eliminates the binary variables through generic reductions; the resulting LP relaxation produces integer solutions automatically due to total unimodularity [9]. Gurobi’s first feasible solution appears at 8 s costing \$27.7M (50× the optimum); the optimum follows at 26 s. HiGHS takes over 6 minutes.

Step ④: Compare results. The Solution Explorer (④) distinguishes optimal, feasible, and infeasible solutions on a cost-vs-time chart.

Step ⑤: Run the DeQL query. While Gurobi runs, the presenter clicks Run on the DeQL query (⑤). The engine returns in **0.88 seconds**: provably optimal at \$554,053, the same cost Gurobi reaches after 26 s and HiGHS after over 6 minutes.

Step ⑥: Inspect the Planner. The Planner panel (⑥) reveals the Planner’s reasoning: the bipartite network detector passes, enabling MCF; the TU detector also passes, making LP relaxation available as a fallback; the constant-objective detector fails (costs vary). The Planner selects MCF over LP relaxation and MILP.

Step ⑦: Review the pipeline. The Profile panel (⑦) shows the `DECIDE` execution breakdown: Parse 1 ms, Plan 4 ms, Materialize 104 ms, Solve 773 ms, totaling 882 ms. Detection and planning add negligible overhead, reading the Abstract IR and pre-collected candidate statistics rather than scanning candidates; the solver accounts for 88% of the total.

Step ⑧: Examine the result. Clicking a point in the Solution Explorer displays that solution’s allocation plan in the Solution Viewer (⑧): one row per **active** assignment with its **quantity**. All formulations yield the same optimal cost. The 26 s and over-6-minute times come from the conventional MILP a user writes without analyzing the problem’s structure. Both reach the LP relaxation through standard presolve reductions, not by detecting total unimodularity, and neither detects the transportation structure that enables MCF. Gurobi includes a network-simplex algorithm, but requesting it via configuration has no effect here, because the network structure is not apparent in the coefficient matrix; exploiting it requires remodeling as a network. The DeQL engine detects the structure from the query and selects MCF automatically. A larger instance (10.7M candidates) exhausts the MILP solvers’ memory or time; MCF still solves in ~4 s.

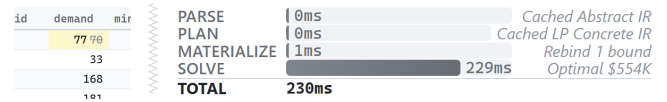


Figure 4: A demand edit (left) and resulting execution profile (right): the solver warm-starts from the previous solution.

What-if analysis. Attendees edit queries (②) and data values (①) directly. First, removing `pool_cap` triggers re-detection: the bipartite detector fails (the pool side is no longer constrained), but TU still holds, so the engine falls back to LP relaxation. Solve time jumps to ~6 s: the feasible set is larger but MCF no longer applies. Next, editing a workload’s demand reuses the LP model and warm-starts the solver (Figure 4). This re-solve takes ~0.2 s, 25× faster than the cold LP solve. Other edits: a global budget (`SUM(cost*quantity) <= budget`) has a non-unit coefficient, forcing MILP; `MAX` instead of `SUM` breaks flow conservation.

Additional scenarios. A *constant-objective* scenario uses the same query with a uniform cost per GPU-hour. The static detectors pass as before, but the constant-objective detector also passes (`COUNT DISTINCT` on `cost` falls to one), enabling max-flow, which the Planner selects (4× faster than MCF). Editing one assignment’s cost breaks uniformity; the Planner reverts to MCF. A *worker-shift scheduling* scenario has a global constraint (a total cap, no `BY`) that breaks the bipartite detector but preserves TU; the Planner selects LP relaxation. A *product-selection* scenario has non-unit coefficients; both detectors fail and the Planner falls back to MILP.

REFERENCES

- [1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.
- [2] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. 2018. Package Queries: Efficient and Scalable Computation of High-Order Constraints. *The VLDB Journal* 27, 5 (2018).
- [3] Matteo Brucato, Fjodor Kholodkov, Soren Little, Jakob Mayer, and Duc Nguyen. 2026. Decisionhouse: Prescriptive Analytics in the Data Stack. In *PVLDB*, Vol. 19. 2755–2762. <https://doi.org/10.14778/3828612.3828629>
- [4] Matteo Brucato, Fjodor Kholodkov, Soren Little, Jakob Mayer, and Duc Nguyen. 2026. DeQL: A Decision Query Language for Prescriptive Analytics over Relational Data. arXiv:2606.19751
- [5] Matteo Brucato, Miro Mannino, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2020. sPaQLTools: A Stochastic Package Query Interface for Scalable Constrained Optimization. In *PVLDB*, Vol. 13.
- [6] Robert Fourer, David M. Gay, and Brian W. Kernighan. 2003. *AMPL: A Modeling Language for Mathematical Programming* (2nd ed.). Brooks/Cole.
- [7] Gurobi Optimization, LLC. 2026. Gurobi Optimizer Reference Manual. <https://www.gurobi.com>
- [8] William E. Hart, Jean-Paul Watson, and David L. Woodruff. 2011. Pyomo: Modeling and Solving Mathematical Programs in Python. *Mathematical Programming Computation* 3, 3 (2011).
- [9] A. J. Hoffman and J. B. Kruskal. 1956. Integral Boundary Points of Convex Polyhedra. In *Linear Inequalities and Related Systems*, H. W. Kuhn and A. W. Tucker (Eds.). Number 38 in Annals of Mathematics Studies. Princeton University Press.
- [10] Qi Huangfu and J. A. J. Hall. 2018. Parallelizing the Dual Revised Simplex Method. *Mathematical Programming Computation* 10, 1 (2018), 119–142.
- [11] H. W. Kuhn. 1955. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly* 2, 1-2 (1955).
- [12] Laurent Perron and Vincent Furnon. 2026. OR-Tools v9.15. Google. <https://developers.google.com/optimization/>
- [13] Alexander Schrijver. 2003. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics, Vol. 24. Springer.
- [14] Laurynas Siksnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *SSDBM*.