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 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.

gpu_pools

pool_id	capacity	mem_gb
H100-E	130	80
A100-W	85	40
T4-S	70	24

… 1,000 rows

workloads

workload_id	demand	min_mem
llm-serve	8	40
embed	3	24
fine-tune	12	80

… 4,000 rows

assignment_costs

pool_id	workload_id	cost
H100-E	llm-serve	42
H100-E	fine-tune	63
A100-W	embed	17

… 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).

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

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828079

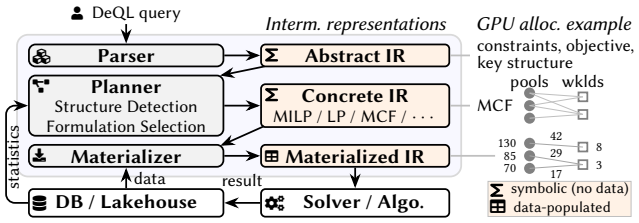

Figure 2: Three-IR pipeline of the DeQL Engine, annotated with the GPU allocation example (right).

a general-purpose solver does not recover the bipartite pattern from it. Even after *presolve* (the solver's automatic preprocessing) eliminates the MILP's binary activation variables, the solver does not switch to a specialized network-flow algorithm, so solving the conventional MILP takes 30× longer than MCF (Section 3).

DeQL [3, 4] (Decision Query Language, pronounced "dee-kwl") is a declarative SQL extension in which users express the optimization problem directly over relational data, and the DeQL Planner determines the formulation. Figure 1 shows the GPU allocation as a DeQL query: `CREATE CANDIDATES` joins the input tables into a candidate space, and `DECIDE` declares the result table, with decision columns whose values the engine determines to satisfy the listed constraints and optimize the stated objective. Because the query stays anchored to the relational schema, its transportation structure is visible to the engine, which reads it directly and solves the problem as MCF instead of MILP, in polynomial time. We demonstrate DeQL Studio, an interactive interface where attendees write DeQL queries, edit data, run conventional scripts for comparison, and inspect the Planner's Structure Detection and Formulation Selection in real time. Editing a query triggers re-detection; editing data can skip it and warm-start the solver, enabling what-if exploration.

2 SYSTEM OVERVIEW

2.1 The DeQL Language

DeQL extends SQL with two constructs designed to express optimization problems over relational data while preserving structure for automatic Formulation Selection (Figure 1). A *candidate* is one possible decision—here, each pool-workload pair. `CREATE CANDIDATES` defines this space via a SQL query annotating the composite key `(pool_id, workload_id)`. At candidate creation time, the engine executes this query against the database, holds the result in memory, and collects *candidate-level statistics* (per-column min, max, count distinct) for Formulation Selection (Section 2.2).

`DECIDE` optimizes over the candidate space and produces a result table. `DECISION COLUMNS` declares new columns in the result table whose values are determined by the engine, analogous to decision variables in mathematical programming: `active` (binary: whether the assignment is used) and `quantity` (continuous: GPU-hours to allocate). The `SELECTION` modifier declares `active` as a binary activation variable: each candidate is either used or not. When `active` = 0, `quantity` is automatically zero; the result table includes only used candidates. Because `SELECTION` handles the *linking* between `active` and `quantity` automatically, `active` does not need to appear in any constraint or in the objective; the user writes constraints only on `quantity`. The `WHERE` clause filters candidates by hardware compatibility. Named constraints enforce resource balance, with `BY` generating one constraint per group: `pool_cap` caps total allocation per pool; `meet_demand` requires each workload's demand to be met. `MINIMIZE` directs the engine to find the lowest-cost feasible allocation; the result is a standard relation, composable with SQL. A formal specification of DeQL, including grammar and type system, is available as a technical report [4].

2.2 DeQL Query Processing

The DeQL engine is a layer on top of existing databases (or lakehouses) and existing solvers (Figure 2). At decide time, it processes each query through three components that produce three intermediate representations (IRs). The *Parser* produces the *Abstract IR*—a symbolic, formulation-agnostic representation that records column references, constraint names with their `SUM...BY` grouping patterns, decision variable types and bounds, and objective structure, but commits to no formulation. The *DeQL Planner* performs *Structure Detection* and *Formulation Selection* [3], producing the *Concrete IR*, bound to a specific formulation such as MCF, MILP, or the linear-programming (LP) relaxation, which drops MILP's integrality requirements. Unlike SQL planning, which selects among strategies within the same complexity class, this choice can change the effective complexity class. The Planner operates on two inputs: the Abstract IR, where `DECISION KEY` components and `SUM...BY` grouping patterns are first-class syntactic elements, and the candidate-level statistics collected at candidate creation (analogous to SQL catalog statistics computed at table creation). The *Materializer* evaluates the Concrete IR's symbolic expressions against candidate data values, producing the *Materialized IR*: the solver-ready numeric structures (constraint matrices, arc lists, bound vectors) that the chosen formulation requires.

The boundary between Concrete and Materialized IR separates formulation from data. When a user modifies data without changing the query, the engine reuses the cached Abstract and Concrete IRs and re-evaluates the Materialized IR against the new data (the Concrete IR records which table columns map to which coefficients and bounds), unless the edit changes a candidate-level statistic, in which case it re-runs Structure Detection and Formulation Selection. A query edit (e.g., removing a constraint) invalidates the Abstract IR, triggering full re-processing.

Default formulation. DeQL Studio focuses on linear decision problems, for which a MILP is the always-available default.

Structure Detection. Before committing to this default, the Planner runs a set of *detectors*, each checking a structural condition under which a more efficient formulation attains the MILP's optimum. The conditions are sufficient but not necessary, so a query whose structure no detector recognizes is still solved exactly, at the default's running time. The engine implements four detectors in an extensible framework.

Bipartite network *(static)* This detector analyzes the Abstract IR and checks five conditions: (1) the `DECISION KEY` has exactly two components, K_1 and K_2; (2) there are exactly two constraints, one grouped by K_1 and one by K_2; (3) every constraint is a `SUM` of a bare decision column (`SUM(quantity)`, not `SUM(cost*quantity)`); (4) one component's constraint is `<=` (*supply*), the other's `=` or `>=`

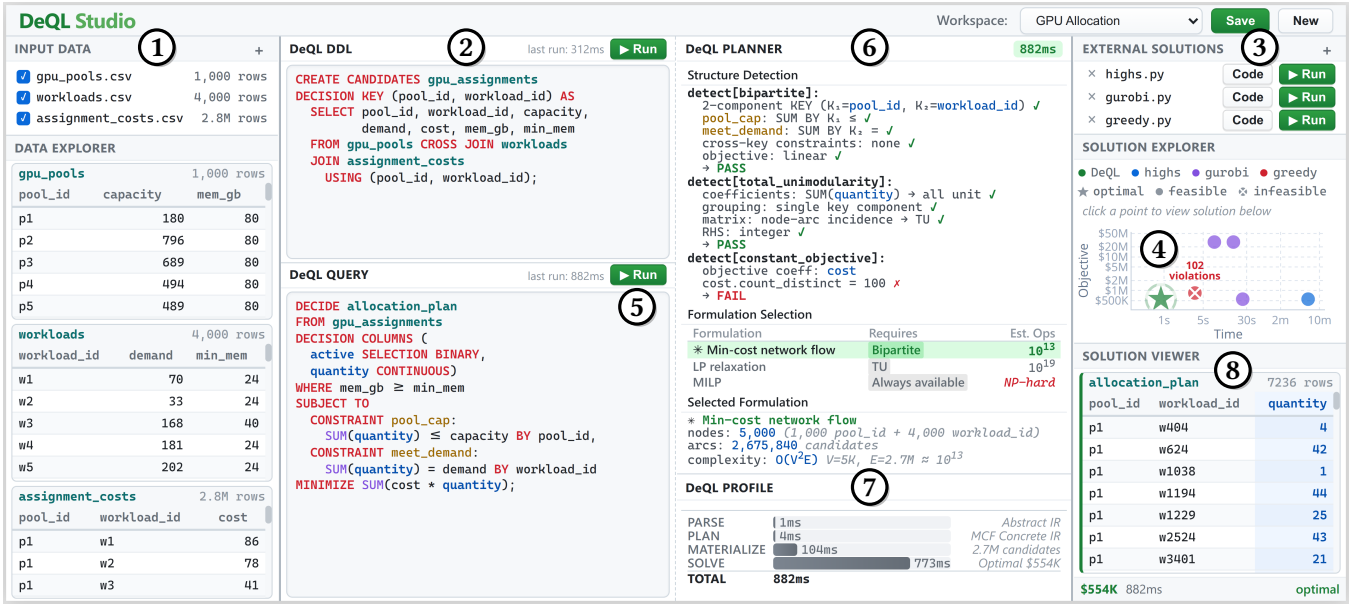

Figure 3: DeQL Studio during the GPU allocation demonstration (1,000 pools, 4,000 workloads, 2.8M assignment costs). ① Data Explorer with editable input tables. ② DeQL query editor. ③ Conventional solver scripts. ④ Solution Explorer chart. ⑤ Run DeQL query. ⑥ Planner: Structure Detection and Formulation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

(*demand*); and (5) the objective is linear. When all five hold, the component with the `<=` constraint provides the supply nodes and the `=/>=` 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× 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.

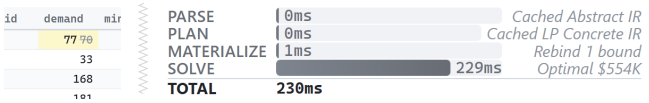

Figure 4: A demand edit (left) and resulting execution profile (right): the solver warm-starts from the previous solution.

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.*