DeciDB: In-Database Processing of Decision Queries

Anh L.Mai[*] Filip Milisov[*] Hatim Rehmanjee[*] Azza Abouzied[*]
Peter J. Haas[†] Alexandra Meliou[†]
[*]New York University Abu Dhabi
{anh.mai,milisov,hatim.rehmanjee,azza}@nyu.edu
[†]University of Massachusetts Amherst
{phaas,ameli}@cs.umass.edu

ABSTRACT

We present DeciDB, a database system with built-in constrained optimization capability, and its SQL extension, DeciQL. The DeciQL language allows specification of a wide variety of optimization problems arising both in decision support settings, such as emergency response, and in common data management tasks such as data cleaning. DeciQL's novel language design generalizes our prior work on package queries and yields optimization queries that are significantly more concise than other language attempts. In DeciDB, decision variables are treated as first class relational values, which lets queries express multiple independently-typed decisions, compose constrained optimization with relational operators, and feed solved results into subsequent SQL processing. DeciDB's query engine, implemented over DuckDB, automatically parses a DeciQL query to generate the underlying mathematical program (MP), send it to a backend solver, and map the solution back to an output relation; importantly, the relational computation and the MP are optimized together. An adapter-based architecture allows the optimizer to choose between multiple backend solvers and ensure that the generated MP leverages the chosen solver's capabilities. Finally, DeciDB provides novel support for explaining optimization-query outcomes and suggesting repairs if needed. Thus, our initial DeciDB prototype is an important step towards democratizing prescriptive analytics and providing a uniform framework for a wide range of data management tasks.

1 INTRODUCTION

Many data-intensive applications ask not only *what the data contains* but also *which actions should be taken*. Solving such problems requires decision variables that have unknown values a priori, constraints that valid value assignments to decision variables must satisfy, and an objective that ranks valid assignments. Together, these requirements form a mathematical program (MP). One class of such problems comprises domain-specific *decision-support* problems (aka "prescriptive analytics"). These include constrained assignment, subset selection, or resource allocation problems such as the following:

Example 1 (Disaster-relief allocation). *Consider a relief agency deciding which supply depots to open and how many units to ship from each depot to each affected region (Figure 1). Opening a depot makes its supply available but incurs a fixed cost; shipping along a route incurs a transportation cost and is limited by both route capacity (line 7) and the stock at its depot (line 8). Critical regions must receive enough supplies to meet their demand (line 9). The choices are interdependent: opening a depot enables its outgoing shipments, while routes from the same depot compete for limited stock. The agency therefore seeks the feasible allocation with the lowest combined depot-opening and transportation cost (line 10). A standard SQL query can evaluate a given allocation, but it cannot choose the allocation itself.*

Depots

depotID	stock	opening_cost
D1	800	12,000
D2	500	8,000

Regions

regionID	demand	priority
R1	450	critical
R2	600	standard

Routes

routeID	depotID	regionID	capacity	unit_cost
T1	D1	R1	500	4
T2	D1	R2	350	6
T3	D2	R2	300	3

```
1  select routeID, depotID, regionID, open, ship
2  decide D.open(BOOL), T.ship(INT)
3  from Depots D
4  join Routes T using depotID
5  join Regions R using regionID
6  such that
7      ship is between 0 and capacity * open and
8      sum(ship) <= stock per depotID and
9      sum(ship) >= demand when priority = 'critical' per regionID
10 minimize sum(unit_cost * ship) + sum(D: opening_cost * open);
```

(query output)

routeID	depotID	regionID	open	ship
T1	D1	R1	true	450
T2	D1	R2	true	0
T3	D2	R2	false	0

Figure 1: Allocating disaster-relief supplies with decisions scoped to depots and routes. The output relation includes the solved decision columns in red.

Decision problems are not confined to niche, highly specialized applications, but arise in many common settings. Indeed, many data management problems are fundamentally decision problems. Classic data-manipulation tasks such as data cleaning, entity matching, and data acquisition all require choosing among candidate actions subject to consistency, integrity, or budget constraints. Although these tasks are currently handled by disconnected specialized tools, they can be viewed uniformly as constrained optimization problems such as the following:

Example 2 (Data cleaning). *Consider a customer table in which repeated imports have produced several tuples with the same customer ID. Each tuple has a confidence score indicating the reliability of its source. To restore the key constraint, the analyst must choose exactly one tuple to retain for each customer ID. Among all valid repairs, the analyst chooses the one that maximizes the total confidence while also satisfying specified integrity constraints.*

The need for general in-database optimization. A standard RDBMS typically does not support specification or processing of optimization queries such as the ones in these examples, so analysts must extract relational data, enumerate the variables, constraints,

and objective as an MP in procedural modeling code, invoke an external solver, and translate its solution back into relations. This fragmentation duplicates relational logic and complicates data access control, movement, lineage, optimization, and result interpretation.

The common core across these applications is constrained optimization: relational data determines the available decisions, constraints define valid choices, and an objective ranks them. Many prior in-database approaches center on prescriptive-analytics workloads such as package or subset selection, whose specialized abstractions do not capture the independently-typed, relationally-scoped choices that arise across data-manipulation tasks and complex decision settings. More general language proposals, meanwhile, leave open end-to-end query processing and optimization-specific diagnosis. What is needed is not separate database support for multiple application categories, but a database in which constrained optimization is a general, composable data-processing abstraction.

DeciDB. To address this gap, we present DeciDB, a database system with built-in constrained optimization, and its SQL extension, DeciQL. Its central abstraction is a typed decision column generated over a relational scope. Our first design goal is to make decisions first-class relational values rather than a special-purpose package abstraction or solver-side objects. A decision column is bound to a base relation or derived relation; joins preserve that association, and the solved values return as ordinary columns of a relation. This relation-in/relation-out design lets queries express several independently-typed decisions, compose constrained optimization with relational operators, and feed solved results into subsequent SQL processing. Besides decision columns, "scalar" decisions cover choices that are shared across the entire query. Section 3 develops the language syntax that realizes this abstraction.[1]

Our second design goal is to optimize the relational computation and the mathematical program (MP) together. The relational plan determines which tuples contribute variables and constraint terms, so conventional choices such as filtering, join ordering, and partitioning directly affect the size and structure of the MP presented to a solver. Conversely, solver capabilities determine whether it can handle expressions such as `abs`, `min`, `max` directly or whether they need to be rewritten, often as linear inequalities. DeciDB therefore represents optimization as a query-plan operator, allowing one optimizer to reason about relational cost, MP-solving cost, solution strategy, and solver-specific representation rather than treating the solver as an opaque call after SQL execution. Section 4 describes this architecture.

Our third design goal follows from a new database responsibility: explaining optimization outcomes. A syntactically and semantically valid decision query may still be infeasible, unbounded, or too difficult to solve within a time limit. Solver output, however, includes matrix rows, variable indices, and internal objects rather than the query clauses and input tuples that a user recognizes. As DeciDB constructs the MP, it records the clause and input tuples from which each solver-level row and variable was generated. It can therefore translate solver internals back into the query, identifying the relevant clauses and tuples and suggesting possible repairs needed to resolve unsucessful solves. Section 5 develops this query-level diagnostic abstraction.

[1]An early prototype of DeciDB is available at https://decidb.com.

Contributions. Our paper makes the following contributions:

- We introduce DeciQL, an SQL extension with typed decision columns bound to relational scopes, query-wide scalar decisions, data-generated constraint families, reducers, and optimization objectives.
- We describe the implementation of DeciDB over DuckDB, including its parser, binder, logical and physical `decide` operators, algebraic rewrites, data-dependent MP construction, and a common interface to multiple solver adapters.
- We present query-level handling for infeasible and unbounded decisions, preserving the connection between solver diagnostics and the clauses, variables, and tuples in the user's query.

2 RELATED WORK

Our earlier PaQL and sPaQL languages support package queries by assigning to each input tuple one implicit non-negative integer multiplicity [2, 4, 6]. This syntax is effective for subset and package selection, but cannot directly represent several independently-typed (not necessarily integer-valued) decisions bound to different relations or a query-wide scalar. Moreover, constraints are manually specified as a fixed set rather than as families indexed by the values present in the data.

SolveDB [10] moves beyond package-selection queries by allowing relation attributes to hold unknown values and by supporting more general objectives and constraints in SQL. It specifies each constraint with a nested `select–where–group by` subquery inside the constraint block. To express the constraint on line 9, for example, one would write a subquery with a `where` clause filtering the tuples with `priority = 'critical'` and a `group by` clause introducing one constraint per region. This approach is expressive but verbose, as every constraint needs its own subquery. In contrast, DeciQL provides the more concise `when` and `per` constructs, which express filtering and constraint generation directly with no separate subquery, making for better readability.

Finally, DeQL [3] is the closest language comparison to DeciQL[2] because both support typed decisions and data-generated groups of constraints without subqueries. DeciQL improves on DeQL's specification in two main ways. First, DeciQL binds every decision column directly to a base relation or CTE and preserves that association through joins. A DeQL query instead declares candidate rows through a `DECISION KEY`, specifies variable sharing separately through `BY`, and uses an additional `ON` clause to associate a decision column with a joined relation. DeciQL's direct binding therefore avoids coordinating several declarations to establish multiple decision's scopes. These choices make the examples compared in Section 3.3 more concise in DeciQL. DeQL does support window and scheduling constraints that DeciQL cannot yet express; extending DeciQL toward these workloads remains future work.

At the system level, DeQL is a language specification without a concrete query-processing system. SolveDB provides a concrete system, but it does not represent optimization as a native logical and physical query operator. Moreover, in SolveDB and SolveDB+, solver selection is part of the query: the user names the solver that evaluates the model and may additionally specify a solving

[2]DeciDB was public by Feb. 3, 2026 and DeciDB v0.1.0 was released on June 3, 2026; DeQL first appeared on arXiv June 18, 2026.

method [9, 10]. In contrast, DeciDB represents optimization as a native query operator, allowing the database optimizer to jointly plan relational execution, MP construction, and solver selection while also supporting query-level diagnosis of unsuccessful solves.

3 THE DECIQL LANGUAGE

DeciQL extends SQL with a small set of constructs for expressing constrained-optimization problems over relational data. The `decide` clause declares typed unknowns, `such that` specifies the constraints that the assignments must satisfy, and `minimize` or `maximize` defines the objective. We use the disaster-relief query in Example 1 to explain three features: decision columns bound to relational scopes, data-generated constraint families, and reducers that control which tuples, after joins, contribute terms to a given aggregate that appears in an objective or constraint.[3]

3.1 Typed and Scoped Decision Variables

In a programmatic solver interface, the application must enumerate, index, and later map variables back to the input data. A DeciQL query instead declares decision columns directly over relational scopes, avoiding application-side variable indexing and result remapping.

The `decide` clause declares values that are unknown when the query begins and that are assigned by a solver such as Gurobi [5] or Highs [7]. Each declaration specifies a type and a relational scope. In Example 1, `D.open(BOOL)` creates one Boolean decision for each depot tuple, whereas `T.ship(INT)` creates one integer decision for each route tuple. After `Depots` and `Routes` are joined, every route can reference the `open` decision of its depot. A depot that appears on several joined rows still has one `open` variable: the join propagates its reference rather than creating a variable per joined row.

Some decisions are not attached to a tuple. The `scalar` keyword declares one query-wide decision shared by all rows. For example, the disaster-relief query can be extended with `decide scalar max_shortfall(INT)`, an integer upper bound on every region's unmet demand, and the additional constraint

```
demand - sum(ship) <= max_shortfall per regionID.
```

The objective can then penalize large values of this global decision. Per-tuple decision values vary with their depot or route, whereas the query-wide `max_shortfall` value is shared by all result rows. After solving, DeciDB materializes the full query output as an ordinary relation, with the decision assignments added as ordinary columns. A per-tuple assignment is repeated on every output row carrying that tuple's identity (e.g., via the `depotID` attribute of the joined row), while a scalar assignment is repeated on every output row. The resulting relation can be consumed by an enclosing `SELECT`, stored as a view or CTE, or used as the input to another decision query. Thus, DeciQL preserves SQL's relation-in/relation-out model and composes with existing SQL workflows.

[3] DeciQL can also express stochastic optimization problems where the data is uncertain, optimizing expected objective values subject to mean, value-at-risk, or conditional value-at-risk constraints [6]. We omit a detailed description due to limited space.

3.2 Constraint & Objective Generation

Solver APIs require a mathematical program (MP) expressed as a scalar objective (i.e. an expression that evaluates to a single value), decision variables with specified domains, and a conjunctive set of scalar constraints. In DeciQL, users specify these components declaratively over relational data, and DeciDB generates the corresponding terms without requiring handwritten procedural loops to enumerate coefficients or construct constraints individually.

The `such that` clause specifies one or more constraints separated by `and`, reflecting their conjunctive interpretation. Each constraint compares two scalar expressions using one of $\{=, \leq, \geq\}$, and may be followed by optional `when` and `per` modifiers (see §3.2.1). Either side of a constraint, as well as the objective, may reference constants, stored attributes, decision columns, or *reducers*. A reducer is an aggregate-style expression, such as `sum`, that combines contributions from multiple rows into a single scalar expression in the MP (see §3.2.2). When applied only to stored attributes, as in `sum(stock)`, it evaluates to a scalar constant derived from the data. An attribute or decision column referenced within a reducer is *reduced*; one referenced outside a reducer is *non-reduced*. Thus, `sum(ship) <= stock` compares the reduced expression `sum(ship)` with the non-reduced expression `stock`.

3.2.1 Constraint Modifiers. The `when` modifier restricts constraint-generation to only tuples satisfying a condition. For example,

```
sum(ship) >= demand  when priority = 'critical'
                     per regionID
```

generates the constraint `sum(ship)>=demand` only from tuples whose priority is `critical`.[4] No corresponding constraint is generated from non-critical tuples.

After `when` selects the tuples that contribute to a constraint, the optional `per` modifier determines whether constraints are generated per tuple or per group.

Without `per`: If `per` is omitted, DeciQL generates one constraint expression for each tuple in the `when` selection (or entire join table if no `when` modifier exists). Each reducer is evaluated once over the full selection and reused in every generated per-tuple constraint, with non-reduced values taken from the corresponding tuple. An expression such as `sum(ship)<=1000` will 'conceptually' result in as many identical constraints as tuples in the selection even though every instance will express the same global shipment limit. The DeciDB query processing pipeline (§4.1.4), however, eliminates such duplicate generation.

With `per`: DeciQL partitions the `when` selection (or entire join if no `when` exists) by the `per` key and generates one constraint for each partition. Every non-reduced attribute referenced by the constraint must have a single value within a partition. In the example above, `when` retains only routes associated with critical regions, and `per regionID` groups those routes by region. This produces one constraint per critical region: `sum(ship)` aggregates shipments across the region's routes, while `demand` supplies that region's single demand value. Using `sum(demand)` instead would add the same regional demand once for every joined route, thereby overstating

[4] `when` is similar to SQL's `WHERE` in that it retains only tuples satisfying a condition. However, `WHERE` changes the relational input and therefore affects the entire query, whereas `when` applies only to the constraint or objective expression it modifies.

the demand. If the `per` key does not determine a unique value for a non-reduced attribute, DeciDB rejects the query as the constraint is ambiguous, analogous to rejecting an ungrouped column in a SQL `GROUP BY` query.

3.2.2 Qualified Reducers. A reducer combines multiple terms into a single scalar expression. By default, it reduces the terms contributed by rows of the join result, after any `when` filtering and `per` partitioning. For example, the `minimize` reducer expression `sum(unit_cost * ship)` in Example 1 produces

$$4 * \texttt{ship}_{T_1} + 6 * \texttt{ship}_{T_2} + 3 * \texttt{ship}_{T_3}$$

with `unit_cost` coefficients and `ship` decision variables bound to the Routes T relation. DeciQL allows reductions to operate on relations other than the full join relations that appear in the query. For example, the reducer `sum(D: opening_cost * open)` combines terms from only the tuples of the `Depots` table, referenced here by its alias `D`; this is done by grouping tuples in the join by `depotID` and then de-duplicating by sending the common value of `opening_cost * open` in each group to the aggregate. Qualifiers can refer to base relations within the query (e.g. '`D:`' or '`Depots:`') or intermediate joins (e.g. '`D,T:`', which maps to `Depots join Routes`).

Inside a relation-qualified reducer, any attribute or decision column used must come from the qualified relation. Constants and query-wide decisions are also allowed because they do not vary across tuples. Thus, `sum(D: opening_cost * open)` is valid because both `opening_cost` and `open` come from `D`. In contrast, `sum(D: opening_cost + unit_cost * ship)` is invalid because `unit_cost` and `ship` come from `T`, not `D`. DeciDB rejects this expression rather than choosing an arbitrary route for each depot.

Constraint-expression construction follows a fixed order:

`when` selection ⟶ `per` partitioning ⟶ qualifier-key grouping and de-duplication ⟶ aggregation.

First, `when` selects the relevant rows. Next, `per` divides them into the groups for which separate expressions are generated. A qualifier such as `D:` then specifies the tuple-identity key by which the reducer's input is grouped and de-duplicated. Finally, the reducer aggregates one contribution from each tuple identity.

3.3 Language Conciseness

To assess syntactic conciseness, we rewrote in DeciQL every in-scope example query from the DeQL [3] and SolveDB [10] papers and repositories, preserving the original identifiers and labelling in each translation, and computed per-example DeciQL-to-baseline size ratios in words. The mean ratio was 0.71 (95% CI [0.62, 0.80]) against DeQL (12 examples) and 0.60 ([0.49, 0.72]) against SolveDB (9 examples); DeciQL was shorter in all 21 cases.[5]

4 THE DECIDB ARCHITECTURE

This section describes the principal design choices and components of DeciDB. Processing a decision query comprises of (1) assembling the mathematical program (MP) within the DeciDB system, and (2) solving the MP within a solver. DeciDB generates a query plan that represents the MP symbolically and fills it in from the data during execution. This allows joint optimization of the relational computations and the constructed MP.

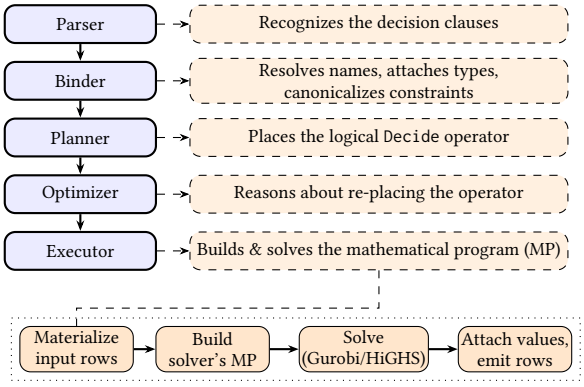

Figure 2: Decision-query processing through the standard query pipeline and the DeciDB task added at each stage.

Before describing how DeciDB processes a query, we explain why we implement it on DuckDB rather than on a server-oriented row store such as PostgreSQL. There are three reasons:

① *Efficiency.* Constructing an MP is itself an analytical workload: it scans, joins, and aggregates many tuples to produce arrays of variable bounds, constraint coefficients, and objective coefficients. DuckDB's columnar, vectorized executor passes these values through the plan in batches, allowing the physical `decide` operator to construct many MP entries at once versus processing one tuple at a time. Its logical and physical operator framework also makes `decide` part of the query plan, so relational optimization can reduce and reshape the data before MP construction.

② *Ease of Deployment.* DeciDB changes the parser, binder, optimizer, and executor rather than adding only a client-side modeling layer. A customized DuckDB compiles into an in-process library over a single-file database, so users run DeciDB without installing or administering a modified server.

③ *Data Interoperability.* DuckDB can query data where analytical applications already keep it, including columnar files, Arrow objects, and in-memory data frames. A decision query can operate directly on these representations without first loading them into a separate server or exporting them to a modeling application.

4.1 Assembling the Mathematical Program

DeciDB represents an MP at two levels. Before optimization, the query plan stores a *logical* MP with bound expressions for the variables, constraints, and objective. During optimization and execution, the data instantiates the *physical* MP that is passed to the solver with coefficients and constraint bounds.

We extend each stage of DuckDB's query-processing pipeline just enough to carry a decision query into a logical `decide` operator (Figure 2), which holds the logical MP.

4.1.1 Parser. The parser recognizes the `decide`, `such that`, `minimize`, and `maximize` clauses and represents `when`, `per`, and reducers as explicit syntax-tree nodes. It checks syntax, clause placement, and the decision-type vocabulary, but does not expand

[5] We choose word measure as it reflects syntax; characters depend on identifier names, while sentence counts treat most SQL queries as one statement. We exclude out-of-scope queries that require: scheduling, learned prediction, external callbacks.

data-dependent constructs. For example, the number of groups created by `per` is unknown until execution.

4.1.2 Binder. The binder resolves stored and decision columns against the relations and aliases in `FROM`. For each decision declaration, it records the column's name, declared type, and relational scope and adds the column to the `decide` operator's output schema without modifying the base relation. It then normalizes each constraint by moving decision-dependent terms to the left and data-only terms to the right. This normalized form exposes the coefficient expressions and bounds, allowing the optimizer to classify the MP—for example, as linear or nonlinear—and choose a suitable evaluation strategy or declare unsupported queries.

4.1.3 Planner. The planner inserts a logical `decide` operator after the scans, joins, and filters that produce the candidate rows, but before operators such as projection and grouping. The operator carries the bound decision variables, constraints, and objective and appends solved decisions as ordinary output columns. Later SQL operators can therefore consume these columns.

4.1.4 Optimizer. Given the logical plan, the optimizer determines where to place the `decide` operator. It may push the operator below a join when the variables, constraints, and objective depend on only one input and the join preserves that input's tuples one-for-one. This allows the MP to be constructed over a smaller base relation, reducing constraint-generation cost.

The optimizer chooses the physical implementation of the `decide` operator. It estimates the size of the MP—the number of its decision variables and constraints—to determine whether to choose a physical `decide` that constructs the full physical MP and passes it directly to the solver, or to instead apply an approximation algorithm such as Progressive Shading [8] or SketchRefine [2]. These algorithms solve several smaller, tractable mathematical subproblems to produce approximate decisions.[6] Because these strategies operate over precomputed data partitions, choosing one of them also inserts the required partitioning into the relational plan if the data is not pre-partitioned.

In some cases, an optimizer could potentially eliminate the solver entirely by replacing `decide` with relational operators. For example, this is possible in fractional knapsack problems, such as:

```
decide quantity(REAL)
such that
    quantity is between 0 and upperBound and
    sum(quantity * weight) <= fixedCapacity
maximize sum(quantity * profit);
```

When profits and weights are positive and quantities are continuous, the greedy choice is optimal: take tuples in descending order of profit per unit of weight, filling each to its upper bound until capacity runs out. The optimizer therefore would replace the solver call with a sort and a cumulative scan; such advanced optimizer functionality is a topic for future work.

4.1.5 Executor. The executor constructs and solves the MP, and then emits an output relation with the decision columns. Given the logical MP representation (symbolic expression trees for constraints and the objective), it encodes the physical MP according to its complexity, and the capabilities and interface of the underlying solver. For example, if the MP is strictly linear, it constructs an in-memory sparse constraint matrix with one column per variable and one row per constraint; each nonzero entry records a variable's coefficient in that constraint. To populate this matrix, it follows the construction order described in Section 3: it reads DuckDB data chunks, filters by the `when` condition, partitions by the `per` key and applies any qualified reductions. It minimizes the number of passes over the data by processing all non-reduced expressions simultaneously, and identically scoped or qualified reductions together. All coefficients are generated as vectorized batches. The operator efficiently passes a pointer to this matrix to the solver. For non-linear MPs, it follows the same process but formulates solver-API calls through solver adapters (§4.2) to add constraints and the objective. If the query has linearizable reducers (e.g. `abs, min, max`), it retains them as is for solvers that can handle them (e.g. Gurobi); otherwise, it linearizes them. Overall, the operator keeps MP constructs supported by the solver intact and rewrites only unsupported ones.

After the solver returns, the operator maps assignments back through the variable references and exposes them as ordinary output columns. It retains the constructed model until it finishes, so a query can be diagnosed (Section 5) without rebuilding it.

4.2 Solver Adapters

Solvers are linked to DeciDB via *adapters*; adding a new solver requires implementing a corresponding adapter and declaring its supported variable domains and constraint types. Currently, DeciDB provides adapters for the open-source Highs solver [7] as well as Gurobi [5]. An adapter has three responsibilities. First, it translates variable domains, terms, coefficients, and retained native constraints into calls to the solver's API to construct MPs. Second, it supplies the static capability table used by the physical decide operator to recognize supported MP constructs. Third, it converts solver-specific output signals into common statuses—optimal, infeasible, unbounded, or stopped at a limit—together with assignments and objective information when available.

5 DIAGNOSTICS

A decision query can execute completely and still return no assignment. This occurs in two cases:[7]

(1) *Infeasibility.* The `such that` constraints cannot all be satisfied, so no valid assignment exists. DeciDB logs and raises an exception: 'Infeasible Query'.
(2) *Unboundedness.* The objective improves without limit while the constraints hold. Again, the solver produces no decision values here. DeciDB logs and raises an exception: 'Unbounded Query'.

To diagnose these two cases, a user simply prefixes their decision query with `DIAGNOSE`. DeciDB then suggests *repairs*: a few edits to the query[8] that may result in a feasible and optimal answer.[9]

[6] This functionality is not yet fully implemented in the current DeciDB prototype, because these algorithms must first be reimplemented within DuckDB.

[7] A timeout is a third, ambiguous outcome: the MP may be slow, infeasible, or unbounded. Diagnosing timeouts is future work.

[8] We are also exploring the potential for data repairs: changing the input rows rather than the query.

[9] Our diagnostic methods prefers fewer edits but neither guarantee minimality nor fixes that will always result in optimal answers. This is a known hard problem [1].

Feasibility and boundedness generally depend on both the query and the data. The query in Ex. 1 is feasible on the data shown, but if R2 becomes a critical region and route T2 capacity drops to 250, the routes into R2 carry 50 units less than R2's demand and the query becomes infeasible as no assignment of values to `ship` would satisfy `sum(ship) >= demand ... per regionID`.

For each case, we briefly review standard solver diagnostics, explain why they do not apply directly in DeciDB, and describe the modifications needed to make them query-aware.

Case 1: Infeasibility. Standard solvers explain infeasibility using *elastic relaxation*: they add non-negative slack variables to constraints and minimize a penalty on those slacks. Nonzero slacks identify which constraints must be relaxed and by how much. Gurobi exposes this through `feasRelax`, but reusing this functionality directly in DeciDB raises the following issues:

(1) A single DeciQL clause may generate many MP constraints. For example, `sum(ship) >= demand ... per regionID` generates one constraint, and hence one slack, for each critical region. These MP row-level slacks are less intuitive than a query-level repair such as `sum(ship) >= demand + s ... per regionID`. However, mapping multiple, potentially different slacks to one value s is not straightforward: taking the maximum may overstate the required change, while aggregating them may have no clear query-level meaning. Moreover, because elastic relaxation minimizes a penalty over all row-level slacks, it may concentrate the relaxations in a few generated constraints or distribute them across many, making the resulting query-level repair difficult to interpret.

(2) The solver does not distinguish between data-derived and user-specified bounds. For example, `sum(ship) >= demand` and `sum(ship) >= 500` are represented similarly because DeciDB evaluates `demand` to a constant before constructing the MP. Yet these bounds may warrant different repairs: in some settings, relaxing an explicit query bound may be more appropriate than modifying one derived from the data.

(3) Finally, diagnostic support varies across solver backends: Gurobi provides `feasRelax`, whereas Highs offers no equivalent.

We adapt standard elastic relaxation in two ways. First, all MP constraints generated from a `such that` clause that contains a `per` specifier share one slack variable, so each relaxation maps directly back to the query. Second, DeciDB prefers relaxing explicit user-provided bounds over bounds derived from data. To enforce this preference, DeciDB solves elastic programs in stages. It first minimizes slack on data-derived bounds while leaving slack on user-provided bounds unpenalized, pushing repairs toward the user-provided bounds. It then fixes the resulting data-derived bounds' slack values (if any) and minimizes the slack on user bounds.

Suppose `sum(ship) >= demand when priority = 'critical' per regionID` makes the query in Ex. 1 infeasible. `DIAGNOSE` reports the clause to change, the changed clause, and the objective the repaired query achieves:

```
clause      sum(ship) >= demand when ... per ...
change      sum(ship) >= demand - 50 when ... per ...
objective   24200
```

Since `demand` is a column, the change applies to each entry.

Case 2: Unboundedness. For linear programs of the form "maximize $c^\top x$ s.t. $Ax \leq b$", standard solvers diagnose unboundedness using an *unbounded ray*: a direction d such that $Ad \leq 0$ and $c^\top d > 0$. Moving along d preserves feasibility while improving the objective indefinitely. The nonzero entries of d identify the *escaping* decision variables, suggesting a repair such as adding upper or lower bounds.

Gurobi exposes this information through `UnbdRay`, but some solvers provide no equivalent. DeciDB therefore computes the ray independently. (For MPs with integer decisions, it first relaxes integrality and analyzes the resulting LP.) The ray then identifies which decision variables can escape and may need additional bounds.

Consider Ex. 1 with the following edits: set the objective to *maximize* `sum(ship)` and remove lines 7–8. Now, nothing caps `ship`. `DIAGNOSE` reports the escaping variable, the rows it escapes on, and the repair:

```
variable   ship
rows       all rows
repair     ship <= <cap>
```

DeciDB does not choose the cap since that number is use-case dependent. If only a subset of `ship` escaped, 'all rows' would be replaced by the subset's group.

6 CONCLUSION

We have presented DeciDB, a database system that makes constrained optimization a composable part of relational query processing, and DeciQL, its SQL extension for concisely specifying decisions, constraints, and objectives over data. DeciDB jointly processes relational computation and MP construction, supports multiple solver backends, and diagnoses infeasible and unbounded outcomes by identifying relevant query clauses, decision variables, and candidate repairs. DeciDB is under active development; the extended version of our paper will include more details on the system components and a full performance evaluation against SolveDB.

REFERENCES

[1] Edoardo Amaldi and Viggo Kann. 1995. The complexity and approximability of finding maximum feasible subsystems of linear relations. *Theoretical Computer Science* 147, 1–2 (1995), 181–210. doi:10.1016/0304-3975(94)00254-G

[2] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. 2018. Package queries: efficient and scalable computation of high-order constraints. *VLDB J.* 27, 5 (Oct. 2018), 693–718. doi:10.1007/s00778-017-0483-4

[3] 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 [cs.DB]

[4] Matteo Brucato, Nishant Yadav, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2020. Stochastic Package Queries in Probabilistic Databases. In *Proc. ACM SIGMOD*. 269–283. doi:10.1145/3318464.3389765

[5] Gurobi Optimization, LLC. 2026. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[6] Riddho R. Haque, Anh L. Mai, Matteo Brucato, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2025. Stochastic SketchRefine: Scaling In-Database Decision-Making under Uncertainty to Millions of Tuples. *Proc. VLDB Endow.* 18, 9 (2025), 3106–3118. doi:10.14778/3746405.3746431

[7] Q. Huangfu and J. A. J. Hall. 2015. Parallelizing the dual revised simplex method. arXiv:1503.01889 [math.OC]

[8] Anh L. Mai, Pengyu Wang, Azza Abouzied, Matteo Brucato, Peter J. Haas, and Alexandra Meliou. 2024. Scaling Package Queries to a Billion Tuples via Hierarchical Partitioning and Customized Optimization. *Proc. VLDB Endow.* 17, 5 (2024), 1146–1158. doi:10.14778/3641204.3641222

[9] Laurynas Šikšnys, Torben Bach Pedersen, Thomas Dyhre Nielsen, and Davide Frazzetto. 2021. SolveDB+: SQL-Based Prescriptive Analytics. In *Proc. EDBT*. 133–144. doi:10.5441/002/edbt.2021.13

[10] Laurynas Šikšnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *Proc. SSDBM*. Article 14. doi:10.1145/2949689.2949693