SolveDB: Integrating Optimization Problem Solvers Into SQL Databases

Laurynas Šikšnys, Torben Bach Pedersen
Department of Computer Science, Aalborg University, Denmark
{siksnys, tbp}@cs.aau.dk

ABSTRACT

Many real-world decision problems involve solving optimization problems based on data in an SQL database. Traditionally, solving such problems requires combining a DBMS with optimization software packages for each required class of problems (e.g. linear and constraint programming) – leading to workflows that are cumbersome, complex, inefficient, and error-prone. In this paper, we present *SolveDB* - a DBMS for optimization applications. SolveDB supports solvers for different problem classes and offers seamless data management and optimization problem solving in a pure SQL-based setting. This allows for much simpler and more effective solutions of database-based optimization problems. SolveDB is based on the 3-level ANSI/SPARC architecture and allows formulating, solving, and analysing solutions of optimization problems using a single so-called *solve query*. SolveDB provides (1) an SQL-based syntax for optimization problems, (2) an extensible infrastructure for integrating different solvers, and (3) query optimization techniques to achieve the best execution performance and/or result quality. Extensive experiments with the PostgreSQL-based implementation show that SolveDB is a versatile tool offering much higher developer productivity and order of magnitude better performance for specification-complex and data-intensive problems.

1. INTRODUCTION

In today's world of data, enterprises need simple yet cost-effective ways to get fast and accurate answers to more and more complex questions. Since the late 1990s, this need drives the efforts of bringing computations "close-to-the-data", blending data management and analytical functionalities in a common system with enhanced overall capabilities and performance. As a result, most commercial or open-source SQL-based database management systems (DBMSs) – the *de-facto* choice of most enterprises today – feature support for in-DBMS analytics ranging from simple data mining tools to sophisticated statistical and machine-learning techniques, including techniques for forecasting [8], automated recommendation [16], and decision-support [13]. However, current DBMSs do not yet provide effective functionality to solve *optimization problems* based on data in SQL databases, which is required for many modern *prescriptive analytics* applications [12], e.g., *production planning*, *logistics/resource management*, *what-if* analysis, and *energy planning* and *how-to analysis* covered in this paper. As shown below, users and enterprises can substantially benefit from an easy-to-use SQL-based data management and problem solving system that can integrate a variety of solvers while offering rapid database-based problem specification and solving to (nearly) any application system based on an SQL back-end.

SSDBM '16, July 18-20, 2016, Budapest, Hungary
© 2016 ACM. ISBN 978-1-4503-4215-5/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2949689.2949693

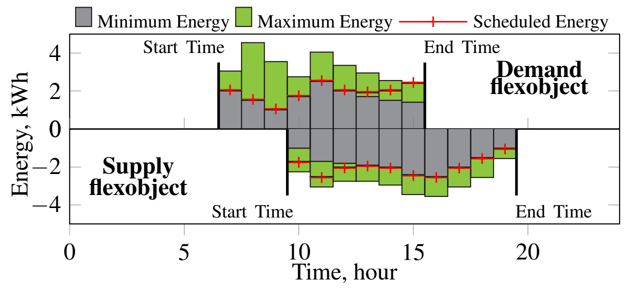
As the running example, consider an energy balancing problem, where the objective is to balance electricity demand and supply based on so-called *flexibility objects* [24] stored in the database. A flexibility object (flexobject), in its basic form, specifies expected demand and/or supply hourly loads (e.g., of a heat pump or battery) with associated *start/end times* as a sequence of minimum/-maximum energy *amount bounds*. Figure 1(a) shows two example flexobjects for demand and supply. A number of flexobjects are stored in the database relation *f_in* shown in Figure 1(b), where fid and tid are flexobject/time interval identifiers (hours) and *e_l* and *e_h* are minimum/maximum energy amounts, respectively. In order to balance the supply and demand (e.g., to avoid imbalance fees for energy suppliers), so-called *schedules*, shown in Figure 1(a), need to be found for every flexobject, e.g., by simply minimizing the sum of absolute total hourly amounts. This requires solving the following *linear programming* (LP) problem, where F is the total number of flexobjects and T is the total number of time intervals:

$$\begin{array}{ll} \underset{e}{\text{Minimize}} & \sum_{t=1:T} \left| \sum_{f=1:F} e_{f,t} \right| \\ \text{Subject To} & e\_l_{f,t} \leq e_{f,t} \leq e\_h_{f,t},\ f=1,\ldots,F,\ t=1,\ldots,T. \end{array}$$

This problem can be viewed as replacing the `NULLs` in *f_in* with energy amount values so that all these constraints are respected and the objective function is minimized, generating the relation *f_out* as shown in Figure 1(c). We call *f_in* (Figure 1(b)) and *f_out* (Figure 1(c)) the *input relation* and *output relation*, respectively.

Traditionally, generating such an output relation from an input relation, i.e., solving the problem, requires using a DBMS and an optimization software package (e.g., Matlab, AMPL, or IBM CPLEX), and as well as developing and executing the model of this optimization problem based on database queries for reading and writing data to/from the database. This process exhibits a number of *complexities* and *inefficiencies*. First, users must have significant *expertise* in a database query language (e.g., SQL) and a modelling languages (e.g., AMPL), dealing with different types of fundamental structures: relations, tuples, and attribute values in databases versus parameters, variables, constraints, and objectives in optimization models. Second, as data is shipped back and forth between two different systems, a significant *performance overhead* is inevitable. Third, users must *manually perform* multiple error-

Figure 1: Examples of demand and supply flexobjects (a) and flexobject data input and output relations (b-c)



(a) Examples of demand and supply flexobjects

fid	tid	e_l	e_h	e?
1	7	2	3	NULL
1	8	1.5	4.5	NULL
...	...	...	...	...
1	15	1.4	2.4	NULL
2	10	-2.2	-1	NULL
...	...	...	...	...
2	15	-3.4	-2.4	NULL
...	...	...	...	...

(b) Flexobject input relation *f_in*

fid	tid	e_l	e_h	e?
1	7	2	3	2
1	8	1.5	4.5	1.5
...	...	...	...	...
1	15	1.4	2.4	2.4
2	10	-2.2	-1	-1.7
...	...	...	...	...
2	15	-3.4	-2.4	-2.4
...	...	...	...	...

(c) Flexobject output relation *f_out*

prone actions (define queries, edit data, create and execute the model file) with different types of software. This is cumbersome and ineffective, particularly when used as part of a larger decision-making workflow, resulting in frequent changes of the data and model. Third, as the model, data, and solution are stored in separate and hard-coded locations, it becomes difficult to *version* models and data so that a particular model version can be used with a particular version of the data. Finally, if additional flexobject constraints are introduced, as we do later, they might be unsupported by the optimization software, leading to much increased user efforts in learning and using a new optimization software package.

To address all these issues, we present *SolveDB* - an extensible purely SQL-based DBMS for rapid database-based optimization problem specification and solving, and simplified solver integration and development (in SQL or DBMS procedural languages). SolveDB is provided with a number of optimized database-based SolveDB-compatible solvers for *linear/mixed integer programming* and *global optimization*, each with a number of modelling tricks that allow the user to concentrate on the high-level models, while SolveDB takes care of the low-level modelling. We now show how both expert and less advanced users can use SolveDB to solve the energy balancing problem above, as well as more advanced ones.

For advanced users, SolveDB offers a set of solvers and their supported modelling constructs (variable types, functions, and operators). Following the mathematical model, the energy balancing problem is solved using the following so-called *solve query* Q_1:

```
1 SOLVESELECT e IN
2            (SELECT fid, tid, e_l, e_h, e FROM f_in) AS r_in
3 MINIMIZE  (SELECT sum(abs(t))
4             FROM (SELECT sum(e) AS t
5                     FROM r_in GROUP BY tid) AS s)
6 SUBJECTTO  (SELECT e_l <= e <= e_h FROM r_in)
7 WITH solverlp();
```

The SELECT statement in Line 2 generates the input relation *f_in* (Figure 1(b)), having an alias *r_in* assigned. The name of the input relation attribute *e* with unknown (decision) variables is set in Line 1. The values of *e* are then treated as variables in all the subsequent SELECT statements in Lines 3-6. The SELECT statement in the MINIMIZE clause (Lines 3-5) specifies the objective function to be minimized, generating a data-specific algebraic combination of values and variables. The SELECT statement in the SUBJECTTO clause (Line 6) specifies flexobject constraints, which are applicable to variables from individual rows of the input relation (*r_in*). For solving this problem, the query specifies the use of *solverlp* – a SolveDB-compliant solver for general linear/mixed integer programming (LP/MIP) problems. Provided this SOLVESELECT formulation, SolveDB invokes a query (sub-)plan generated by *solverlp* that partitions the problem into independent subproblems (for each distinct *tid* value), creates and solves a number of low-level problems (with auxiliary variables/constraints for *abs*), and finally produces the output relation *f_out* (Figure 1(c)).

For less advanced users, SolveDB offers so-called *composite solvers* that hide all problem formulation details and/or offer higher-level constructs to formulate constraints and objective functions, as exemplified using the following solve query Q_2:

```
1 SOLVESELECT e IN (SELECT * FROM f_in) AS r_in
2 SUBJECTTO (SELECT sum(e) <= 100 FROM r_in GROUP BY fid)
3 WITH balancing_solver();
```

This solve query relies on *balancing_solver* – a composite solver that hides all formulation details of the original energy balancing problem. Its workflow is specified declaratively based on the solve query Q_1, reusing *solverlp* for solving. Additionally, it allows specifying (external) user constraints, which, in this case, limit total energy allocations of every flexobject to 100 kWh (Line 2). Thus, the user can focus on data, solutions, and problem customizations instead of detailed problem modelling, allowing SolveDB to reuse and optimize solvers. Later in this paper (Section 6), we use SolveDB to solve this energy balancing problem as a part of a (much more complex) real-world energy planning problem, requiring demand forecasting (based on power and temperature measurements from the database) and special flexobjects with additional constraints on *start time* (see Figure 1(a)) – which allow demand/supply loads to be flexible in time. Our experiments show that, for this complex planning problem, SolveDB allows obtaining comparable forecasts and better schedules in the limited solving time while requiring approximately 15 times less total solver and user problem specification code, compared to traditional approaches.

In general, the novel SOLVESELECT clause produces an output relation from an input relation according to the problem formulations in the inner SUBJECTTO block and an additional inner MAXIMIZE/MINIMIZE clause defining objective functions. The SOLVESELECT clause supports multiple attributes containing unknown variables of various types, e.g., integers, doubles, or user-defined data types (UDTs). This allows representing any imaginable solution as an output relation. To process problem formulations, SolveDB relies on an extensible infrastructure of SolveDB-compatible so-called *view solvers* (e.g., *solverlp*) dealing with optimization problems formulated as *views* over an input and other relations in the database. Each new view solver registered in SolveDB is either defined in a declarative manner using solve queries and other existing view solvers, or developed as a (native) function realising a specific solving technique or invoking an external solver.

This paper provides a number of contributions. First, we propose the concept of a view solver that allows integrating various existing solvers while offering DBMS query optimizations. Second, we define the complete syntax of the SOLVESELECT clause and propose a common SQL-based problem notation that is intuitive, extensible, and similar to mathematical notation. Third, we present solve query processing and detailed view solver workflows for processing problem descriptors and solutions. Fourth, as these workflows are often expensive to process, we propose a number of workflow tuning (optimization) techniques for both one-time and repeated queries, covering all levels of solve query processing. Finally, we present the ANSI/SPARC-based SolveDB architecture and our PostgreSQL-based implementation. By using this implementation and six view solvers for LP/MIP, black-box, and domain-specific problems, we experimentally compare SolveDB against traditional tools and related systems by solving 11 database-based optimization problems of various sizes. Results demonstrate (1) significantly better tool usability/developer productivity for problems that are complex to specify and (2) order of magnitude better solving performance for problems that are intensive in I/O or can be partitioned into subproblems. SolveDB is thus shown to be a versatile solver integration and efficient easy-to-use SQL-based problem solving system.

The remainder of the paper is structured as follows. Section 2 defines a *solve query* and the SOLVESELECT syntax. Sections 3–4 describe how solve queries are processed and optimized. Section 5 presents the SolveDB architecture and implementation. Section 6 describes the experimental evaluation. Section 7 discusses related work. Section 8 concludes and points to future work.

2. SOLVE QUERIES

We now formulate the concept of a *solve query*, show how SQL is extended to support solve queries, and propose an intuitive common SQL-based notation for optimization problems in solve queries.

2.1 The definition of a solve query

We assume that a (generic) *solver S* produces a solution *s* based on an input (d, p), where *d* is a *model instance descriptor* (*descriptor* for short) and *p* is a set of configuration parameter-value pairs. Based on *d* (e.g., containing data), *S* first produces a *model instance* from a generic *model* [19], and then solves the model instance using some solving technique, finally producing *s*. The solver uses *p* to configure model instance building and solving process.

SolveDB uses solvers of different types. These include existing traditional solvers (e.g., CPLEX), which we refer to as *physical solvers*. As shown later, SolveDB can integrate physical solvers that use different descriptor, solution, and configuration formats.

SolveDB also uses so-called *view solvers* that offer higher-level SQL-based constructs to define user problems. As opposed to physical solvers, view solvers are "white-boxes", meaning that a DBMS can "see inside" and optimize their workflows for the best performance or result quality.A view solver is defined as follows.

Definition 1: *A (generic)* view solver *is a solver S_v that uses a 3-tuple (R_{in}, U, V) as the descriptor d_v and R_{out} as the solution s_v. Here, R_{out} is an output relation that is a special view over an input relation R_{in} (as exemplified in Section 1). U is the set of attribute names representing unknown variables (e.g., {e} in Figure 1(b)), potentially, with initial (non-NULL) values given in R_{in}. $V = \langle V_1, V_2, ..., V_L\rangle$ is the sequence of view definitions (i.e., SELECT queries) over R_{in} and, potentially, other relations in the database.*

A (concrete) view solver in SolveDB rewrites and combines V_1, V_2, ..., V_L, generating a query (sub-)plan to (efficiently) compute R_{out} from R_{in}. The plan, later optimized and executed by SolveDB, may include plans of other existing view solvers and/or calls of physical solver functions (UDFs) for the actual computations. Consequently, physical solvers are hidden from the users and may only be accessed using a *solve query*: a database query that, as a part of its evaluation, invokes the workflow (query plan) of a view solver.

SolveDB invokes the (workflow of the) view solver *solverlp* for LP/MIP problems while processing the solve query Q_1 from Section 1. Initially, a descriptor d_v = (R_{in}, {e}, $\langle V_1,V_3\rangle$) is generated; R_{in} is the input relation *f_in* from Figure 1(b), V_1 is the SELECT statement from the MINIMIZE clause (Lines 3-5), and V_3 is the SELECT statement from the SUBJECTTO clause (Line 6). *solverlp* rewrites and combines V_1 and V_3, generating a query plan that (1) produces linear expressions as instances of a user-defined data type (UDT), (2) partitions the problem based on these expressions, (3) transforms expressions of each problem partition into the matrix-based format adding auxiliary variables and constraints (for the *abs* function), (4) calls the most suitable physical solver (e.g., GLPK) for each partition, (5) combines solutions (if feasible), and (6) generates the output relation *f_out* from Figure 1(c). While SolveDB can integrate different view solvers, the remainder of this paper describes solvers that encapsulate steps 2 to 5 in a single UDF – steps 1 and 6 are still "seen" and optimized by the DBMS query engine.

Next, we show how solve queries are formulated in SQL.

2.2 Extending SQL for solve queries

As exemplified earlier, SolveDB extends SQL with a new SOLVESELECT clause with the following syntax:

```
SOLVESELECT col_name [, ...] IN ( select_stmt ) [AS alias]
[MINIMIZE ( select_stmt ) [MAXIMIZE ( select_stmt )] |
 MAXIMIZE ( select_stmt ) [MINIMIZE ( select_stmt )] ]
[SUBJECTTO ( select_stmt ) [, ...]]
[WITH solver_name [. ...] [( param[:= expr] [, ...] )]]
```

The clause defines a view descriptor d_v=(R_{in}, U, V), solvers, and their configuration parameters. Specifically, the keyword SOLVESELECT is succeeded by a list of attribute names representing unknown variables (U). The first SELECT statement (*select_stmt*) defines the input relation (R_{in}) with an optional alias (*alias*) assigned. This alias is used to reference the input relation in all subsequent SELECT statements in the MINIMIZE/MAXIMIZE clause and the SUBJECTTO block. This clause and the block are optional and they serve as "syntactic sugar" to define V_1, V_2, ..., V_L in V. Irrespective of the order they are applied, MINIMIZE and MAXIMIZE define V_1 or V_2 that specify one or more objective functions (specified as separate SELECT columns) to be minimized and maximized, respectively. The SUBJECTTO block defines V_3, V_4, ..., V_L and represents all remaining model elements such as problem data or constraints. Finally, the WITH clause specifies the name(-s) of the preferred (view/physical) solver(-s) to be involved when processing d_v. Solver names are given such that names of *view solvers* are followed by names of *physical solvers*. The specified solvers can optionally be configured using a list of parameter-value pairs (*param := expr*, see *p* in Section 2.1), where a value is either undefined or computed according to a user-specified SQL expression.

The SOLVESELECT clause results in an output relation R_{out}. Consequently, it can be used as a sub-query in larger and more complex SQL queries or INSERT/UPDATE queries for materializing a solution in the database.

2.3 SQL-based problem descriptors

A view solver S_v rewrites view definitions V_1, ..., V_L and thus poses requirements on how these should be specified in SOLVESELECT (e.g., limits permitted operators and data types of U attributes). It is up to the solver developer to decide what V_1, ..., V_L actually represent and what problem/problem class S_v aims at. At one extreme, S_v can be specialized for a very narrow class of problems by consuming a simple descriptor (R_{in}, U, $\emptyset$), as demonstrated using Q_2 and *balancing_solver* in Section 1. At the other extreme, S_v can utilize the full solving capability of a physical solver by specifying the complete model and data in V_1, ..., V_L using the solver's native modelling language. In the following example, we demonstrate such an S_v using the view solver *wrapper_solver* that just binds data (Line 3) to the OPL model (Line 2) and then passes the model instance to the physical solver *cplex* (Line 4):

```
1 SOLVESELECT x IN (SELECT NULL::real AS x)
2 SUBJECTTO (SELECT 'int␣n=$n;dvar␣float␣x;minimize␣x...'),
3           (SELECT 'n', 123)
4 WITH wrapper_solver.cplex();
```

SolveDB supports view solvers for both extremes. SolveDB also supports *Alias-Based* (*AB*) view solvers, which exploit the input relation alias (*alias*) to offer to the user the formulation of objectives and constraints in V_1, ..., V_L in a syntax similar to mathematical notation. In AB solvers, objective functions and constraints are *first-class citizens* and, for consumption by a solver (rather than a user), are generated using traditional SELECT statements in V_1, ..., V_L (e.g., `SELECT e_l <= e <= e_h FROM r_in`). These SELECTs are independent of each other. To reference common unknown variables from the same input relation, the alias (`r_in`) is used. Note that this notation of constraints is different from traditional WHERE/HAVING clauses (e.g., `WHERE e<=10`) which *filter* rather than *generate* tuples. Using AB SELECTs gives a lot of flexibility in how constraints can be implemented and how a user can select them, while keeping the syntax of the SOLVESELECT clause intact for different AB solvers and optimization problem classes.

AB view solvers rewrite V_1, ..., V_L by substituting the alias of R_{in} with a special view over R_{in} – producing R_{in} with solver-specific UDT values in unknown variable positions (details in Section 3.2). This UDT captures the indices of the unknown variables involved in an optimization model element (e.g., a constraint) and all operations (e.g., assignment or bounding) applied to these variables. The actual set of supported operators depends on the capabilities of the solver, and, among others, may include both common comparison ($<$, $<=$, $=$, $>=$, $>$), equality ($=$), negation ($!=$), and similarity ($\sim$), and specialized (e.g., *is_instance*, *all_different*) operators.

For example, the view solver *solverlp* from Section 1 is of type AB. It uses the solver-specific data type *LpExp*, which represents an expression of unknown variables in a linear combination $a_1v_1 + a_2v_2 + ... + a_tv_t$, where a_1, a_2, ..., a_t are numerical constants and v_1, v_2, ..., v_t are indices (numbers) of variables. The *LpExp* type supports various linear operations such as expression addition, scalar multiplication, negation, etc., all together offering a rich formulation of linear expressions. Further, *LpExp* supports a number of user-defined functions (e.g., *abs*), aggregation operators (e.g., *sum*), as well as comparison operators (e.g., $<=$: $LpExp \times \mathbb{R} \rightarrow LpCtr$) for building complex linear constraint instances (*LpCtr*) from *LpExp* expressions. Thus, the formulation "`SELECT sum(e) <=100 FROM r_in GROUP BY fid`" from Q_2 (Section 1) produces a relation with *LpCtr* entities, e.g., (1·[1]+1·[2]+...+1·[9], '<=', 100) meaning that the equality $1 \cdot v_1 + 1 \cdot v_2 + ... + 1 \cdot v_9 \leq 100$ holds.

This *Alias-Based* (AB) notation is extensible and user-friendly, yet powerful, as it allows mixing unknown variables with known variables (from other attributes) when binding data to or constraining unknown variables. Depending on the needs, all or some of the operators supported by underlying physical solvers can be exposed to users through view solvers using the AB notation.

3. SOLVE QUERY PROCESSING

We now present SolveDB's query processing and discuss how AB view solver workflows are formed, processed, and integrated.

3.1 Overall solve query processing

Like a standard DBMS, SolveDB prepares, optimizes, and executes a solve query as a single relational workflow. To prepare (establish) a solve query, SolveDB performs the following actions for each SOLVESELECT clause:

1. Forms a problem view descriptor d_v and obtains preferred solver names and their configuration parameters.
2. Finds the most suitable view solver (if there is only one).
3. Initializes and inlines the solver relational workflow.

Step 1 is straightforward (see Section 2.2). Given user-specified preferred solver names, SolveDB in Step 2 searches a *solver catalogue* containing information about all registered solvers and then chooses (if possible) a single view solver to be used when processing the descriptor. For this, SolveDB performs a simple matching of the names and data types of the unknown variable attributes and the operators used in d_v against those supported by the solvers. SolveDB may also consult a *solver advisor* for choosing relevant (view/physical) solvers (not specified by the user) for the best performance or solution quality (see Section 4). When no or multiple solver choices are possible, SolveDB reports an error and requires the user to explicitly specify a valid solver combination in the WITH clause. In Step 3, SolveDB initializes the selected view solver using the configuration parameters (e.g., by invoking its "init" function). This results in a solver-specific relational workflow initialized for a specific solver and configuration parameter setting. The workflow is then inlined (embedded) into the overall solve query workflow – similar to how an SQL function is *inlined* to reduce function call overhead and to allow the query optimizer to "see inside" the function. If the selected view solver utilizes other existing view solvers based on SOLVESELECT, this process is further repeated, thus recursively unfolding all view solver workflows. When the overall workflow is established, it is then optimized (see Section 4) and executed.

As demonstrated earlier, SolveDB uses two kinds of view solvers: *composite* and *atomic*. A composite view solver (e.g., *balancing_solver* from Section 1) takes advantage of other existing view solvers based on SOLVESELECT in their specification. In contrast, so-called *atomic view solvers* (e.g., *solverlp*) use physical solvers directly. We now discuss how the relational workflows of typical atomic and composite AB solvers are formed and processed.

3.2 Atomic AB view solver workflow

An atomic AB view solver is typically used to expose the capabilities of a physical solver, offering higher-level constructs for problem formulation. For this, and Q_1 from Section 1, it generates the *adaptation workflow*, shown in Figure 2 in its simplest form.

The adaptation workflow includes the translation steps of a view solver, a physical solver, and a so-called *relational solver*. Here, the physical solver S_p (e.g., GLPK) consumes a (physical) descriptor d_p and produces a (physical) solution s_p; d_p and s_p are represented in internal/native formats, e.g., MPS/SOL. The *relational solver* S_r translates a descriptor d_r (relational representation) to d_p (physical format), and s_p (physical format) to the solution s_r (relational representation); d_r is a sequence of relations, R_1, ..., R_N, where each relation represents model elements (e.g., constraints) as solver-specific UDT entities (e.g., *LpExp*). s_r is a binary relation $R_s(var\_id, value)$ representing unknown variable indices (*var_id*) and found values (*value*). The view solver S_v^a translates the view descriptor d_v into d_r and s_r into the view solution s_v.

The detailed translation workflow of S_v^a alone is shown in Figure 3. To build d_r, S_v^a rewrites each view definition $V_i \in V$ by creating the three-level nested views R_{in}^o, R_{in}^s, and R_{in}^m over R_{in}.

The *order-level* view R_{in}^o establishes the logical order of tuples in R_{in} so that unknown variables are referenced consistently (with equal offsets) when building the problem instance from R_{in} as well as when embedding the solution back into R_{in}. In the general case,

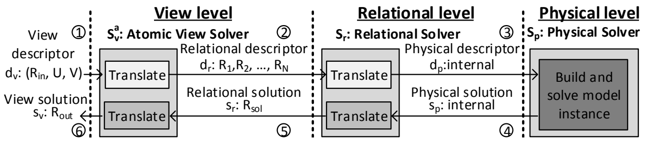


Figure 2: The three-level adaptation workflow

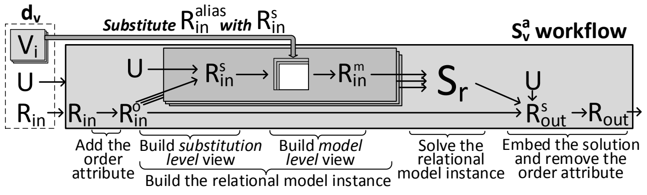

d_v
V_i
U
R_{in}
Substitute R_{in}^{alias} *with* R_{in}^s
S_v^a workflow
U
R_{in}^s
R_{in}^m
S_r
U
R_{in}
R_{in}^o
R_{out}^s
R_{out}
Add the order attribute
Build *substitution level* view
Build *model level* view
Build the relational model instance
Solve the relational model instance
Embed the solution and remove the order attribute

Figure 3: Relational workflow of an atomic AB view solver

S_v^a uses an explicit *order* attribute with unique integer values (see `order` in Table 1 for Q_1), numbering the variables sequentially after the values of the order attribute for every $u \in U$. If single or multi-attribute keys are available in R_{in}, S_v^a uses these instead.

The *substitution-level* view R_{in}^s represents R_{in}^o, but with solver-specific UDT entities embedded at the unknown variable positions (at every $u \in U$). In the case of Q_1, *solverlp* builds R_{in}^s, shown in Table 1, by applying a function (UDF) $f_{LP} : \mathbb{Z} \to LpExp$ that maps variable indices to instances of *LpExp* (see Section 2.3).

The *model-level* view R_{in}^m represents one of d_r relations. The view R_{in}^m is established from V_i by substituting the alias for R_{in} (R_{in}^{alias}) in V_i with the substitution-level view R_{in}^s. As a result, all operations with unknown-variables in V_i are transparently applied on instances of the solver-specific UDT, e.g., *LpExp*. For example, V_3 in Q_1, formulated as "`SELECT e_l<=e<=e_h FROM r_in`", leads to the R_{in}^m shown in Table 2(a). This relation represents all inherent flexobject constraints (see Section 1).

All model-level views, produced from $V_i \in V$, $i = 1..|V|$, represent d_r to be used by S_r. The solution s_r from S_r is embedded back into R_{in}^o by building another substitution-level view R_{out}^s that joins two relations on variable indices. Finally, the projection is applied to remove the order attribute and produce the output relation R_{out}.

Order, *substitution*, and *model* level views are standardized in SolveDB and their generation is supported by an API, available to solver developers. Such views are also used by *solverlp*, which produces relational descriptors as linear expression entities (*LpExp* and *LpCtr*) according to Alias-Based (AB) user-defined views.

3.3 Composite AB view solver workflow

A composite AB view solver relies on another (atomic or composite) view solver, forming its view (rather than a relational) descriptor in a declarative manner using SOLVESELECT.

Consider the solve query Q_2 from Section 1 invoking the composite view solver *balancing_solver*. Suppose that the solver uses the view descriptor (R_{in}^c, U^c, V^c) and consumes the Q_2 constraint "`SELECT sum(e)<=100 FROM r_in GROUP BY fid`" given as $V_3^c \in V^c$. Then, *balancing_solver* simply rewrites V^c based on R_{in}^c as follows:

```
1 SOLVESELECT e IN (SELECT * FROM f_in) AS r_a
2 MINIMIZE (SELECT sum(abs(t))
3           FROM (SELECT sum(e) AS t
4                 FROM r_a
5                 GROUP BY tid) AS s)
6 SUBJECTTO
7   -- All inherent (internal) flexobject constraints
8   (SELECT e_l <= e <= e_h FROM r_a),
9   -- User-defined (external) constraints from V_c
10  (SELECT sum(e)<=100 FROM r_a GROUP BY fid)
11 WITH solverlp();
```

Like Q_1 from Section 1, this solve query specifies the problem with the objective function (Lines 2-5) and inherent flexobject constraints (Line 8) of the original energy balancing problem, and additional user-specified external constraints (Line 10). For the external constraints, *solverlp* generates the model view, shown in Table 2(b).

In the general case, to include external constraints into an internal problem and to produce consistent solutions to user's external problems, *order*, *substitution*, and *model* level views may be built. As for atomic view solvers, SolveDB offers an API for this.

3.4 Integrating a New Solver

For user problems, SolveDB encourages the use of composite AB view solvers, as their workloads can be declaratively specified and optimized (see Section 4). Integrating a new composite solver requires developing and registering a new function that generates and returns the workflow (e.g., as SQL) of a specific solving technique, as part of which an existing view solver is invoked using SOLVESELECT. This might require extending the existing ecosystem of UDTs (e.g., *LpExp*, *LpCtr*) and UDFs (e.g., *sum*), used for encoding and manipulating relational descriptors, to support new problem-specific data types and operators (e.g., *is_instanceof*).

For problems that cannot be efficiently handled by, or mapped to the problems of, the existing atomic view solvers, a new physical solver needs to be integrated, involving a number of steps. First, if needed, the ecosystem of UDTs and UDFs needs to be extended to cover all new elements of the native solver descriptors, e.g., new operators for objectives, constraints or data binding. Second, a new *relational solver* needs to be developed for mapping (extended) *relational* descriptors/solutions to/from native *physical* descriptors/-solutions. For this, different objective/constraint low-level modelling tricks and rewriting algorithms might be integrated. Finally, this new relational solver needs to be tied to an existing atomic AB view solver handling this particular class of relational descriptors (UDTs). If no such AB view solver exists (e.g., the new solver uses vectors as decision variables as opposed to the supported real numbers), a new AB view solver needs to be developed using an API and the discussed *order*, *substitution*, *model* level views. These steps are easy to perform in SolveDB due to the provided API and easy-to-use extensibility features of the underlying modern DBMS,

Table 1: Substitution-level view (R_{in}^s) over the flexobject input relation, shown in Figure 1(b)

order :: integer	**fid**	**tid**	**e_l**	**e_h**	**e?** :: LpExp
1	1	7	2	3	(1·[1])
2	1	8	1.5	4.5	(1·[2])
3	1	9	1	3.5	(1·[3])
4	1	10	1.7	2.7	(1·[4])
...	...	...	...	...	...

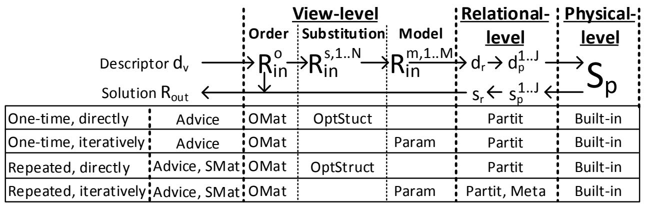
Table 2: Model-level views (R_{in}^m over R_{in}^s) representing all inherent (a) and external (b) flexobject constraints

column1 :: LpCtr
(-1·[1] + 0, '<=', -2)
(1·[1] + 0, '<=', 3)
(-1·[2] + 0, '<=', -1.5)
(1·[2] + 0, '<=', 4.5)
...

(a) Flexobject constraints

column1 :: LpCtr
(1·[1] + ··· + 1·[9], '<=', 100)
(1·[10] + ··· + 1·[20], '<=', 100)
(1·[21] + ··· + 1·[25], '<=', 100)
(1·[26] + ··· + 1·[32], '<=', 100)
...

(b) External user-defined constraints



		Order	Substitution	Model	Relational-level	Physical-level
One-time, directly	Advice	OMat	OptStuct		Partit	Built-in
One-time, iteratively	Advice	OMat		Param	Partit	Built-in
Repeated, directly	Advice, SMat	OMat	OptStruct		Partit	Built-in
Repeated, iteratively	Advice, SMat	OMat		Param	Partit, Meta	Built-in

Figure 4: Types of optimizations applied in SolveDB

leading to significantly reduced efforts in specifying and solving database-based optimization problems, as seen in Section 6.

4. SOLVE QUERY OPTIMIZATION

Solve queries can often be very expensive to process as they rely on (physical) solvers tackling (often) NP-complete problems with no efficient algorithms to find solutions. SolveDB with its view/relational solvers is not optimizing these algorithms, but instead partitions problems, reuses solutions when possible, tunes (optimizes) view solver translation workflows by applying various optimizations at the *view*, *relational*, and *physical* levels, and chooses solvers and solver parameters leading to the best performance or result quality. We now present optimizations specific to one-time and repeated solve queries, processed either iteratively or in a single run (directly). All these optimizations are summarized in Figure 4.

4.1 Solver design optimizations

To increase performance, SolveDB offers the following view and relational solver workflow (design) optimizations.

View solvers At the order level, SolveDB offers a built-in order-level view that uses *materialization* [9] (*"OMat"*) to ensure consistent variable referencing and increase performance of repeated order-level view scans, e.g., when the user provides an expensive input relation query in SOLVESELECT. At the substitution level, for solving problems using an exact numerical technique, SolveDB offers a number of common *data types* (e.g., *LpExp*) provided with a number of standard *operators* (e.g., $+$, $>$) and *aggregation functions* (e.g., *sum*). These are implemented in a low-level language (e.g., $C/C++$), are carefully tuned for performance, and can be easily extended for a specific solver case (*"OptStruct"*). At the model level, SolveDB offers the model-level view *parametrization* (preparation) for solving problems iteratively so that, for instance, the objective function view can be efficiently re-evaluated to produce a new fitness value based on a candidate solution (*"Param"*).

Relational solvers To process problems as independent elements in the DBMS pipeline and because most physical solvers do not have problem partitioning, at the relational level, SolveDB offers built-in *problem partitioning* (*"Partit"*), which splits a problem into a number of sub-problems solved independently and more efficiently. The partitioning is fully transparent to the user and performed in several steps, elaborated in Algorithm 1. First, for a given problem *P*, the sets of dependent variable identifiers are generated by unfolding all constraints and the (independent) terms of the objective function(-s) of *P* (Lines 1–5). Here, the functions *getConstr* and *getObjTerms* return constraint and objective function terms, and *getVars* and *getId* extract variables and their identifiers, respectively. Based on these sets, disjoint partitions of variables are detected (Line 6). If only 1 partition is found, *P* is returned (Line 8). Otherwise, a new sub-problem is initialized for each partition using the function *subproblemInit* (Lines 10–11), which allows reusing the main problem's details (including problem type, variable types, and objective function direction(-s)). Then, the constraints and the objective function terms of *P* are revisited, assigning each to a specific sub-problem using the functions *addConstr* and *addObjTerm* (Lines 12–17). Finally, the generated sub-problems are returned (Line 18) and solved independently.

```
    Input: $P$ - The main problem;
    Output: $S_1,...,S_N$ - Set of sub-problems;
1   $VS \leftarrow \emptyset$;    /* Build sets of variable IDs */
2   foreach $C \in getConstrs(P)$ do
3       $VS \leftarrow VS \cup \{\{getId(v)|v \in getVars(C)\}\}$;
4   foreach $T \in getObjTerms(P)$ do
5       $VS \leftarrow VS \cup \{\{getId(v)|v \in getVars(T)\}\}$;
    /* Detect partitions based on VS */
6   Find $P_1,P_2,...,P_N$ s.t. $\forall V_j \in VS : \exists i \in [1,N]\ s.t.\ V_j \subseteq P_i$ and
    $\forall k,l \in [1,N], k \neq l : P_k \cap P_l = \emptyset$ and $\bigcup_{m\in[1,N]} P_m = \bigcup_{V_n \in VS} V_n$;
7   if $N = 1$ then
8       return P;    /* Only 1 partition found. */
9   else /* Initialize sub-problems */
10      for $i \in [1,N]$ do
11          $S_i \leftarrow subproblemInit(P)$;
        /* Assign model parts to sub-prob. */
12      foreach $C \in getConstrs(P)$ do
13          Find $k \in [1,N]$ s.t. $P_k \supseteq \{getId(v)|v \in getVars(C)\}$;
14          $addConstr(S_k,C)$;
15      foreach $T \in getObjTerms(P)$ do
16          Find $k \in [1,N]$ s.t. $P_k \supseteq \{getId(v)|v \in getVars(T)\}$;
17          $addObjTerm(S_k,T)$;
18      return $S_1,S_2,...,S_N$;
```
Algorithm 1: Partition detection and sub-problem generation

4.2 One-time query optimizations

Given a one-time solve query, SolveDB first uses the *solver advisor* (*"Advice'*) that chooses (if possible) a solver (and its parameters) from a number of alternatives while respecting the user preferences specified in the WITH clause. By using solver priority lists stored in the solver catalogue, the advisor ranks the available solvers based on their performance and solution quality for different groups of problem properties (e.g., problem size, integer/boolean variable count, and objective type). Depending on whether the user requests performance or solution quality (using a configuration parameter), the advisor chooses the best qualifying solver and sets its parameters accordingly. For choosing parameters, the advisor uses either default values or values obtained by *meta-optimization* [23].

When the view solvers are initialized and their workflows are inlined, the overall solve query relational workflow is optimized using the standard cost-based DBMS optimizer, and then executed. Note that SolveDB also benefits from built-in physical solver optimizations (*"Built-in"*) such as *presolving*, *scaling*, *distributed processing*, and *branching heuristics*.

4.3 Repeated query optimizations

Repeated solve queries are those, for example, given in "**CREATE** (**MATERIALIZED**) **VIEW** `vsq` **AS SOLVESELECT** ..." or differing only marginally (e.g., in the solution processing part) from a query executed previously – when both queries lead to identical relational (and physical) descriptors. In such cases, SolveDB is able to reuse (view/relational) *materialized solutions* [9] (*"SMat"*) stored either in a cache or the database, instead of invoking a solver. When problems are solved iteratively, SolveDB additionally uses automatic *meta-optimization* (*"Meta"*) for tuning solver parameters in the background when the SolveDB system idles.

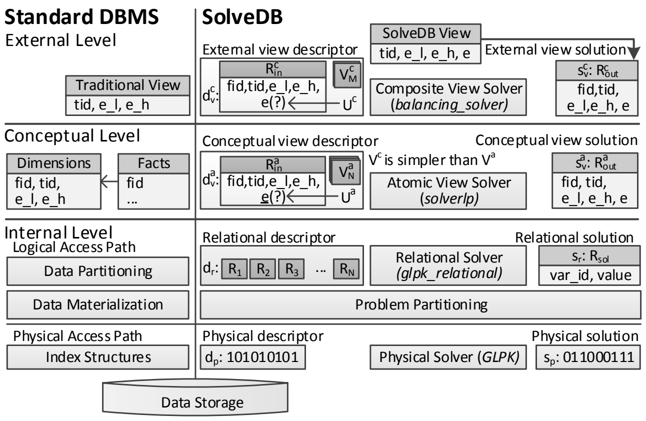


Figure 5: The architecture of SolveDB

5. THE SOLVEDB SYSTEM

SolveDB is based on the standard three-level ANSI/SPARC architecture (the left of Figure 5) with a number of additions, discussed in Sections 2–4, on all three levels (the right of Figure 5). With all these additions, SolveDB allows separating the user's view of an optimization problem from the way the problem is physically formulated and solved, like the user's view of a database is separated from the way the database is physically represented. Analogue to standard DBMS *data independence*, SolveDB offers *solver independence* so user view descriptors can be processed by different combinations of *view*, *relational*, and *physical* solvers. Users do not necessarily need to know the actual combination used and how problems were actually solved. Database administrators may change solvers to increase performance or solution quality, but without changing the semantics of user solve queries.

We implemented SolveDB (www.daisy.aau.dk/solvedb) based on PostgreSQL 9.3.1 while using a simplified solve query processing where each SOLVESELECT is handled by a special PL/pgSQL function, selecting and initializing solvers in a single (execution) step, rather than in three preparation/optimization/execution steps. Thus, the present implementation supports only manual (not yet automatic) *meta-optimization* and *solution materialization*.

We extended the parser of PostgreSQL to support the SOLVESELECT syntax and used PostgreSQL ≥9.1 extensions to encapsulate the remaining SolveDB functionality. The function handling each SOLVESELECT was included as part of a PostgreSQL extension for *SolveAPI* (Sections 3.2–3.3), which additionally defines the *solver catalogue* and a number of generic data types and functions that allow conveniently specifying workflows of atomic and composite view solvers in UDFs (in C/C++ and PL/pgSQL). By utilizing *SolveAPI* and following the steps from Section 3.4, we implemented two proof-of-concept atomic AB view solvers, abbr. as *LP* and *BB*, and packaged them as UDFs in C in two independent PostgreSQL extensions together with the underlying relational (logically integrated) and physical solvers. The LP solver, the implementation of *solverlp* from Section 1, generates instances of the workflow from Section 3.2 for delivering an exact solution after a finite sequence of operations for either LP or MIP problems. It uses the *LpExp* and *LpCtr* constructs for representing objective functions and constraints of these LP/MIP problems (Section 3.2). As the default option, LP invokes a built-in physical solver from the GNU Linear Programming Kit (GLPK) v4.47. For more challenging MIP problems, the user can choose COIN CBC v2.8.12, which we also integrated into the LP solver package. In contrast, the BB solver generates instances of the workflow from Section 3.2 for delivering an approximate solution after a number of iterations for the class of black-box global optimization problems. To produce a solution, it uses 1 of 15 applicable physical solvers from the integrated SwarmOps optimization library[21], e.g., *Simulated Annealing* (SA) or *Particle Swarm Optimization* (PSO). The physical solvers from GLPK and SwarmOPS were patched to use PostgreSQL's I/O and hierarchical memory management routines, and all the view and physical solvers (including CBC) run completely in the PostgreSQL backend's process (address) space.

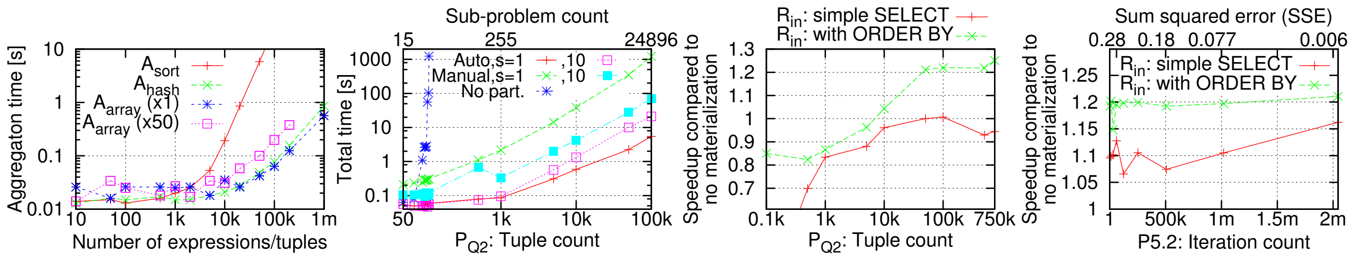
The view solvers LP and BB integrate a number of solver workflow (design) optimizations from Section 4. If not overridden by the user, LP automatically switches from the GLPK LP to GLPK MIP physical solver when a given solve query uses integers or booleans as decision variables (*"Advice"*). At the *substitution level*, LP uses an efficient hash-based *aggregation function* implemented in C for adding linear expressions represented by *LpExp* (*"OptStruct"*). This aggregation implementation is generally faster than the alternatives, studied in Section 6. At the *relational level*, LP employs *problem partitioning* (*"Partit"*) using Algorithm 1 and the *rank-based heuristics* [5] of *disjoint sets* to partition problems based on the *LpExp* and *LpCtr* constructs provided as input. The present LP implementation solves these partitions on a single node in serial. At the *physical level*, LP takes advantage of the default built-in optimizations of GLPK and CBC solvers (*"Built-in"*). At the *model level*, BB uses *parametrization* (*"Param"*) and thus it prepares (the model-level view of) an objective function defined in the MAXIMIZE/MINIMZE clause, and then a SwarmOps physical solver repeatedly re-evaluates the query to produce a new fitness value based on a candidate solution. Experiments show that the parametrization reduces total solving time by approx. 1.7 times on average.

6. EXPERIMENTAL EVALUATION

We now evaluate the performance of the built-in SolveDB optimizations and demonstrate the versatility, usability/developer productivity, and performance of SolveDB. For *versatility*, we developed four additional composite view solvers and used them, together with LP and BB, to solve 7 LP/MIP problems, 3 global optimization (GO) problems, and 1 energy planning problem (based on Q_1 from Section 1) – covering all combinations of specification complexity, I/O-, and CPU-intensity, as shown in Table 3. For *performance*, we compared SolveDB against existing tools and related systems by measuring (1) *solution quality* (in terms of error or imbalance), (2) *solving time*, which is spent by a physical solver to solve a problem based on a descriptor stored in a file or main memory, (3) *I/O time*, which is needed to form the descriptor from data in the database and to process and write the solution back to the database, excluding time spent solving, and (4) *total time* (*I/O time* + *solving time*), which is needed to build the descriptor, solve the problem, and write the solution. For *productivity/usability*, we compared the number of effective source lines of code (eLOC) of problem descriptors (including data bindings) and/or solver implementations. We used default PostgreSQL 9.3.1 configuration and a laptop with Ubuntu 12.04 64bit OS, Intel Core i7 CPU, 500GB HDD, and 4GB of RAM for SolveDB and all competitor systems.

6.1 Evaluation of SolveDB optimizations

Linear expression aggregation First, we run an experiment to determine the most suitable *LpExp* aggregation function (*"OptStruct"*) for the use by LP (see Section 4.1). We considered three function alternatives implemented in C. A_{sort} combines linear expressions into one taking advantage of and maintaining sorted variable indices. A_{hash} combines expressions by building a hash table



(a) *LpExp* aggr. functions (b) Partitioning techniques (c) Materialization effect on LP (d) Materialization effect on BB

Figure 6: Performance of SolveDB built-in optimizations

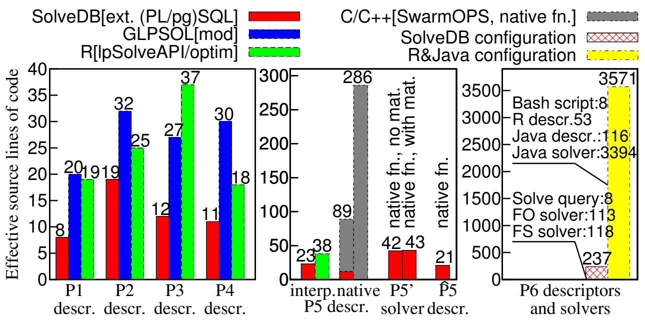
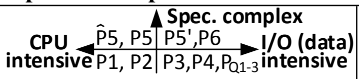


Figure 7: Sizes of P1–P6 implementations in eLOC

Table 3: Optimization problems solved

Nr.	Problem  Spec. complex CPU intensive I/O (data) intensive P̂5, P5 P5',P6 P1, P2 P3,P4,P$_{Q1-3}$	# of Vars
P1.1	Maximum independent set (619 edges)	50
P1.2	Maximum independent set (928 edges)	50
P1.3	Maximum independent set (1140 edges)	50
P2.1	Linear model fitting (10 regr. coef., 100 pnt.)	111
P2.2	Linear model fitting (10 regr. coef., 1000 pnt.)	1011
P2.3	Linear model fitting (10 regr. coef., 2000 pnt.)	2011
P3.1	Sudoku 9x9 (23 initial values)	729
P3.2	Sudoku 16x16 (97 initial values)	4096
P3.3	Sudoku 36x36 (672 initial values)	46656
P4.1	Stigler diet (923 foods)	923
P4.2	Stigler diet (9923 foods)	9923
P4.3	Stigler diet (49928 foods)	49928
P5.1	Neural network black-box training (1x4x3x1 neurons, 10 points)	27
P5'.1	Neural network training and output generation	27
P̂5.1	Meta-optimization of PSO solver parameters	31
P5.2	Neural network black-box training (1x4x3x1 neurons, 100 points)	27
P6	Energy planning – forecasting (3000 measurements) and scheduling (500 flexobjects)	7983
P_{Q1}	"How to": decreasing line item quantities [18]	≤50k
P_{Q2}	"How to": deleting line items (tuples) [18]	≤50k
P_{Q3}	"How to": choosing new suppliers [18]	≤200k

of factors with variable indices used as keys, storing the table in the intermediate aggregation state. A_{array} is similar to A_{hash}, but instead of the hash table it builds a dynamic array, the size of which depends on the difference between the highest and the lowest variable indices in the expression. Figure 6(a) shows aggregation time when varying the number of single-variable expressions with the distinct variable indices. As seen in the figure, A_{sort} is slow when more than 10k expressions/variables are used. A_{array} is the fastest (up to 1.5x compared to A_{hash}) for a narrow range of variable indices (x1). When we scale the range of indices by 50 times (x50), A_{array} performs approx. 3 times slower as it is sensitive to the variable indices influencing the size of the internal array. Finally, A_{hash} is not dependent on the indices of variables and performed up to 3 times faster than A_{array} (x50). Thus, since A_{hash} is robust over a wide variety of scenarios, it is used as default by LP.

Problem partitioning By using SolveDB with LP (GLPK), we evaluated the built-in problem partitioning (*"Partit"*) by considering P_{Q2} – one of our solved MIP problems that can be partitioned into a number of independent sub-problems (details below). By varying the number of tuples in the input relation (problem size) and grouping sub-problems into partitions of size 1 and 10, we solved P_{Q2} instances by using (1) *no partitioning*, (2) the built-in *automatic partitioning*, and (3) *manual partitioning* where solve queries targeting each individual partition are dynamically generated at run-time. As seen in Figure 6(b) (note the log scale), the P_{Q2} instances with more than 100 tuples can hardly be solved with GLPK without partitioning. By requiring extra work from the user, the manual partitioning allowed to cope with P_{Q2} instances approx. 1000 times larger than in the no partitioning case. Provided the same descriptor as in the no partitioning case (no extra work for the user), automatic partitioning not only discovered partitions transparently to the user, but also allowed to solve P_{Q2} instances two orders of magnitude faster than with manual partitioning, which required query re-parsing and re-optimization for every partition. The overhead of the automatic partitioning was only 0.06 sec (measured) in the 100k case. For a partition size of 10, the relative speedup of automatic partitioning was lower (approx. 3 times faster compared to manual partitioning). In this case, manual partitioning was approx. 10 times faster, and automatic partitioning was approx. 5 times slower. In both cases, manual and automatic partitioning were I/O- and CPU- bound, respectively. As automatic partitioning is a very effective low-cost optimization, it is enabled by default for the LP view solver and used with a partition size of 1.

Order-level materialization We studied the impact of *order-level* view materialization (*"OMat"*) for MIP and global optimization problems. We used P_{Q2} solved with LP in various sizes and P5.2 solved with BB for various numbers of iterations (details below). We formulated the input relations of P_{Q2} and P5.2 as simple SELECTs and, for making order-level view evaluation more challenging, as SELECTs with an additional ORDER BY influencing the ordering of tuples in R_{in} and R_{out} but not the actual encoded solutions. For the materialization, we built temporary tables over the order-level view. As seen in Figure 6(c-d), materialization is effective for larger P$_{Q2}$ instances ($>$10k tuples) based on ORDER BY, as well as the ALL cases of P5.2 requiring repeated re-evaluation of the order-level view. The overhead of the materialization is insignificant for smaller P$_{Q2}$ instances, solved with partitioning in less than a second (see Figure 6(b)). Therefore, order-level view materialization is enabled by default for LP and BB in SolveDB.

6.2 Comparison against traditional tools

Classical LP/MIP We used SolveDB with LP (GLPK), GLPSOL (GLPK), and *GNU R* to solve two LP problems, *Linear Model Fitting* (by least absolute deviations) and *Stigler Diet*, and two MIP problems, *Maximum Independent Set* (MIS) and *Sudoku*, in three different sizes leading to the 12 problem instances P1.1, ..., P4.3, shown in Table 3. For each instance, we prepared a database with initial data and an empty solution table, and described each problem as (1) a solve query for SolveDB, (2) a MathProg (mod) program for GLPSOL, and (3) an R program using the *lpSolveAPI* library. For example, P1 uses tables *vertex(vid, m)* and *edge(vid1, vid2)*. Given all edges in *edge*, the boolean values of *m* (treated as 0/1 integers) indicate vertices (with IDs in *vid*) in *vertex* belonging to the MIS. The overall query of $P1$ includes the following solve query:

```
SOLVESELECT m IN (SELECT vid, m FROM vertex) AS t
MAXIMIZE         (SELECT sum(m) FROM t)
SUBJECTTO (
 SELECT t1.m + t2.m <= 1 FROM t as t1, t as t2
 WHERE (t1.vid, t2.vid) IN (SELECT vid1, vid2 FROM edge))
WITH solverlp()
```

We solved all P1-4 instances, measured *total time* and *solving time*, and calculated *I/O time* (*total time - solving time*). The results are shown in Figure 8(a-d) which, together with Figure 7, shows that SolveDB allows defining the problems much more compactly (approx. 1.5-3 times less code) and processing them much more efficiently (I/O time is 2-28 times smaller). As expected, SolveDB and GLPSOL solving times (and total times for solving-intensive cases) are similar as they share the same physical GLPK solvers. The *lpSolveAPI* of R could not find solutions for P3.1–P3.3 within 1 hour; increasing the number of given Sudoku digits from 23 to 62 in P3.1 yielded a solution after 11.6 sec (denoted as R*).

Black-box optimization We used SolveDB with BB (PSO), R, and SwarmOPS (PSO) to train a neural network with a sigmoidal activation and 4 and 3 nodes in 2 hidden layers, respectively, by formulating a (derivative-free) optimization problem P5, minimizing the SSE. We used 2 datasets with 10 and 100 data points (P5.1 and P5.2 respectively). We used 2 solving configurations: an *interpreted* variant where we defined the problem as (1) a solve query using a recursive SELECT in the MINIMIZE clause and (2) an R program using the *optim* function, and a *native machine code* variant where we defined the problem as (1) a solve query specifying (in MINIMIZE) the use of an external C library function for computing output based on input and neuron weights, and (2) a C++ program relying on SwarmOPS and ODBC. Using these two configurations and default solver parameters, we solved P5.1-2 6 times by varying the total number of solving iterations and measured the average resulting error (SSE) and total time. The results, seen in Figure 9(a), show that the C++ implementation has a lower error than SolveDB using the native function, but at a very substantial expense of usability/productivity (286 vs. 89 lines). Compared to R (*optim*), SolveDB (BB) performs approx. 30–100 times better. In both configurations, SolveDB problem formulations have substantially less code (23 vs. 38 and 89 vs. 286 lines), see Figure 7.

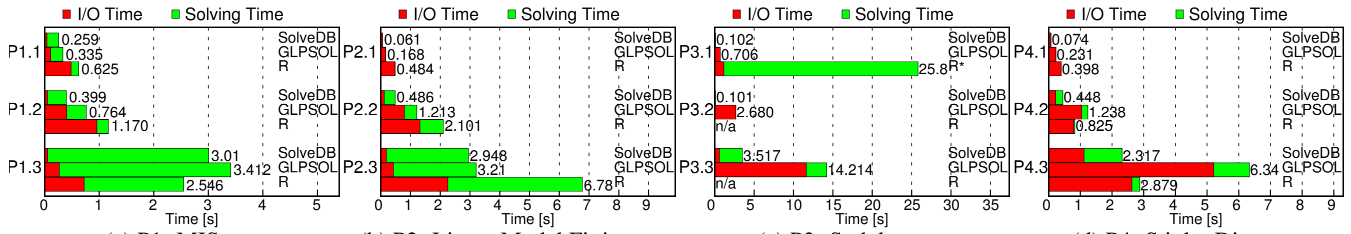
Solution materialization/meta-optimization We used P5.1 as a basis for evaluating *solution materialization* (*"SMat"*) and *meta-optimization* (*"Meta"*, see Section 4.3). For this, we considered P5'.1 - the joint problem of both (1) training a neural network (P5.1) and (2) computing neural network output values based on test inputs and optimized weights. We built two composite view solvers taking test inputs and producing network outputs. The first solves P5.1 each time it is invoked, while the second uses a *materialized view* over the corresponding SOLVESELECT for P5.1. As seen in Figure 9(b), materialization reduces the total P5'.1 solving time by 2–3 orders of magnitude for $SSE = 0.001$ and 1000 input points, both for *interpreted* and *native* configurations. Compared with existing tools (R and C++ implementations), computing neural network outputs using SolveDB takes 11–62 times less time compared to reading materialized weights and network inputs into the external tools and writing outputs back to the database.

As a one-time investment paying off after several materialized view refreshes, we considered a *meta-optimization* problem $\hat{\text{P}}$5.1 for tuning P5.1 solver parameters for smaller error (and time for a desired error). We formulated $\hat{\text{P}}$5.1 as a nested solve query with the inner SOLVESELECT for P5.1 and the outer SOLVESELECT for PSO solver parameter (S,ω,ϕ_p,ϕ_g) tuning. We used BB with *Local Unimodal Sampling* [21] and standard parameters as an overlying meta-solver, which was run 5 times for 100 iterations, solving P5.1 6 times in each iteration (3000 times in total). The meta-optimization results in Figure 9(a) show that meta-optimization is very effective (approx. 10-100 times smaller error) for time-consuming ($>$10 secs) cases with larger numbers of P5.1 solving iterations, while being very easy to specify in SolveDB.

Energy planning We used SolveDB to solve the complex energy planning problem for balancing demand and supply within 24 hours according to hourly *demand forecasts* and *flexobjects* (see Figure 1(a)), extended to include additional *start time* flexibilities given as *earliest/latest start time* bounds (constraints). This inherently requires solving complex interlinked energy *forecasting* (using the EGRV model [22]) and *energy balancing* sub-problems. Following Section 3.4, we implemented two composite AB view solvers based on PL/pgSQL, *SolveAPI*, and SOLVESELECT. The *forecasting solver* (FO) uses the LP solver to minimize least absolute deviations to estimate EGRV model parameters and produce forecasts. The *scheduling solver* (FS) uses a heuristic technique, where BB (with SA) fixes flexobject time flexibilities and LP (with GLPK) uses Q_1 from Section 1 to fix energy amount flexibilities. While using UDTs to encapsulate flexobjects and thier schedules, FO and FB are interlinked using the following query:

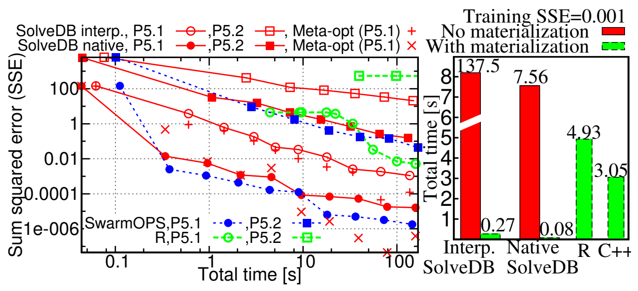
```
SOLVESELECT sch IN (SELECT fo, NULL::schedule AS sch
                    FROM flexobject) AS t
SUBJECTTO (SELECT is_instanceof(sch, fo) FROM t),
          (SOLVESELECT load IN (SELECT time, load
                                FROM hist_load) AS s
           SUBJECTTO (SELECT time, temp FROM temp_data)
           WITH solverFO)
WITH solverFS(rndseed:=12345, sn:=3176)
```

We compared with a traditional configuration (R&Java) using a Bash script, an R program (based on the model fitting function *lm*) for EGRV forecasting, and a third-party Java solver using an evolutionary scheduling technique [24]. The R&Java and SolveDB configurations together with component dependencies (lines), sizes, and invocation order (numbers in circles) are shown in Figure 10, respectively. We used 3000 (load/temperature) measurements and 500 flexobjects representing supply and demand loads with complex flexibility patterns. The results are given in Figure 11. As seen in Figures 10–11, SolveDB required approx. 15 times (237 vs. 3571 lines) less total solver and descriptor code. SolveDB obtained comparable forecasts ($9\% \pm 1$ of Mean Absolute Percentage Error) and a better schedule in less time, due to substantially reduced I/O time (>8 times) and the scheduling technique providing



(a) P1: MIS (b) P2: Linear Model Fitting (c) P3: Sudoku (d) P4: Stigler Diet

Figure 8: Performance of SolveDB (LP/GLPK), GLPSOL (GLPK), and R (lpSolveAPI) when solving the classical LP/MIP problems



(a) P5: Neural network black-box training, 10/100 points (b) P5'.1: Computing 1000 outputs

Figure 9: Performance of SolveDB (BB/PSO), C++ (SwarmOPS/PSO), and R (optim) when solving black-box problems

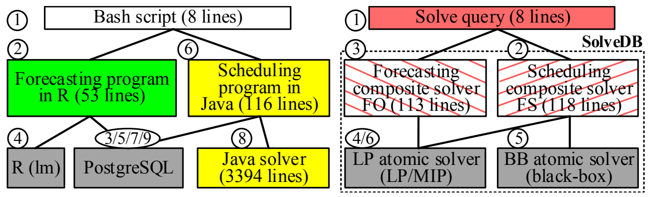

1
Bash script (8 lines)
2
6
Forecasting program in R (53 lines)
Scheduling program in Java (116 lines)
4
3/5/7/9
8
R (lm)
PostgreSQL
Java solver (3394 lines)
1
Solve query (8 lines)
SolveDB
3
2
Forecasting composite solver FO (113 lines)
Scheduling composite solver FS (118 lines)
4/6
5
LP atomic solver (LP/MIP)
BB atomic solver (black-box)

Figure 10: R&Java and SolveDB configurations for the energy planning problem P6

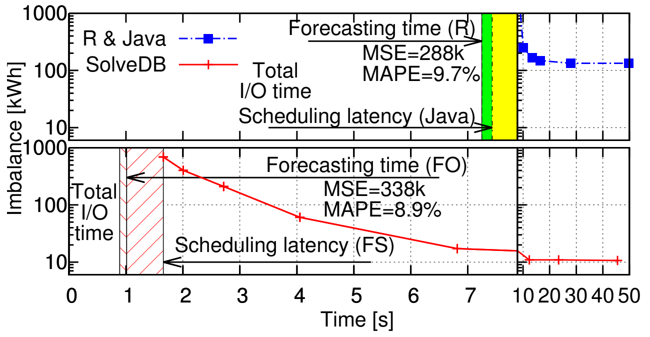


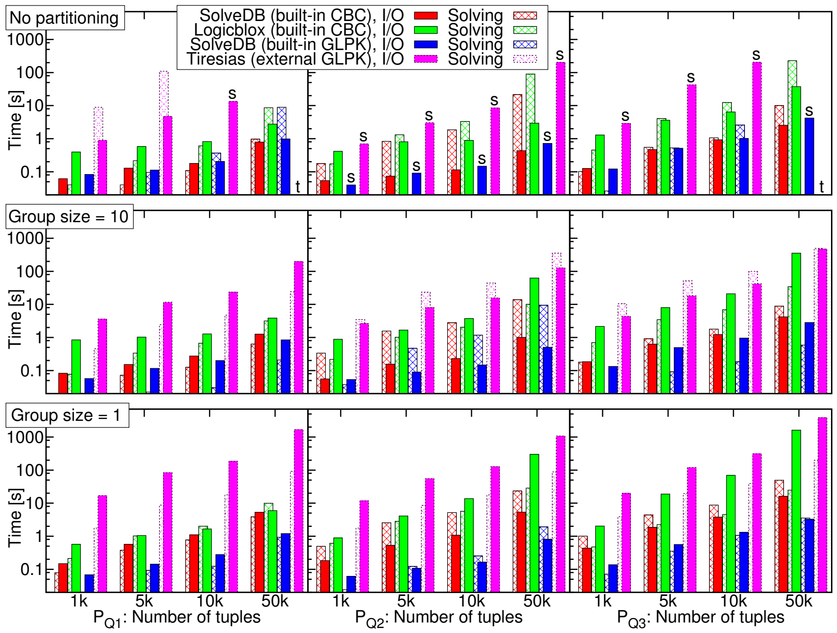
Figure 11: Performance of SolveDB and R&Java when solving the energy planning problem P6

better immediate (first) solution due to exact solving with LP. Furthermore, FO and FS transparently took advantage of the LP and BB solver optimizations. Due to problem partitioning, for example, both the forecasting and scheduling problems are partitioned into sub-problems, resulting in much lower overall complexity. If we disable partitioning, the FO solving time and the FS latency increases approx. 2 times.

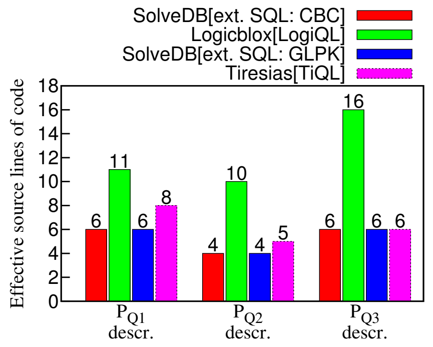
6.3 Comparison against related systems

Lastly, we compared SolveDB with LP (GLPK and CBC) against Logicblox [10] and Tiresias[18] – two related data management and optimization problem solving systems. We used Logicblox 3.10.14 with the built-in Coins CBC solver and Tiresias on PostgreSQL 9.3.1 with the external GLPK v4.47 solver. All three systems support problem partitioning. For partitioning, Tiresias statically analyses TiQL programs, whereas SolveDB uses Algorithm 1 based on relational descriptors for detecting data-dependent partitions that cannot be identified from data-decoupled descriptors such as TiQL programs. We used the three "how-to queries" Q_1, Q_2, and Q_3 and experimental setup as in the Tiresias paper[18]. The queries Q_{1-3} use the TPC-H *lineitem* relation and represent the 3 MIP problems of (1) decreasing line item quantities (P_{Q1}), (2) deleting line item tuples (P_{Q2}), and (3) choosing new suppliers (P_{Q3}) so that total quantities per order (P_{Q1}, P_{Q2}) or order and supplier (P_{Q3}) do not exceed 50 units. We encoded the problems as solve queries (SolveDB) and Datalog-like LogiQL (Logicblox) and TiQL (Tiresias) programs. The descriptor sizes are given in Figure 12(b). For P_{Q1-3} instances with 1k to 50k *lineitem* tuples, we loaded all relevant data into the database and solved the instances using *no partitioning* and *partitioning* into groups of size 1 and 10. Solving and I/O times are shown in Figure 12(a) (note the log scale). If a solver cannot find a solution in 10 minutes, we show an "*s*". If a descriptor size exceeds 10GB, we terminate the solving and show a "*t*". The results have been verified by Logicblox and Tiresias.

By distinctly considering the successful CBC and GLPK solving cases, we summarized and generalized results in Figure 12(c). The results show that SolveDB is significantly faster (up to 3 orders of magnitude) in terms of I/O, solving, and total time (I/O + solving). Compared to Tiresias, the I/O and solving time is up to 3 orders of magnitude smaller, due to (1) the built-in GLPK solver, (2) the in-DBMS approach of processing descriptors and solutions, and (3) more efficient generated descriptors, processed solely in main memory. Compared to Logicblox, the I/O time is more than 7 times smaller on average, mainly because the Logicblox translation procedure is less optimized for partitioned problems than SolveDB's and because Logicblox employs mandatory incremental solution maintenance. The encountered differences in solving time seem to be caused by different CBC versions. Furthermore, SolveDB uses approx. 30% less memory than Logicblox and Tiresias for the successful solving cases. Lastly, Logicblox and Tiresias use somewhat larger descriptors and can handle only LP/MIP problems, while SolveDB uses more compact descriptors and supports a much broader range of problems, see Figure 12(b-c).



(a) Performance of SolveDB, Logicblox, and Tiresias when solving P_{Q1}–P_{Q3}



(b) Sizes of P_{Q1}–P_{Q3} implementations in eLOC

Prb. type	**Part.**	**Logicblox**	**Tiresias**
LP/MIP, P_{Q1}-P_{Q3}, excl., P1-4	No	7.6/6.7/7x	82/1437/330x
	10	21/3.1/7x	108/295/112x
	1	19/1.4/5x	623/79/348x
BB opt., P5, P5'		N/A	N/A
Meta opt., $\hat{P}5$		N/A	N/A
Hybrid, P6		N/A	N/A

(c) SolveDB speedups in I/O, solving, and total time

Figure 12: Results of SolveDB, Logicblox, and Tiresias comparison

7. RELATED WORK

Most previous efforts of integrating problem solving into a relational DBMS focus on LP/MIP problems only. The approaches supporting the block-schematic view [11] of an LP/ MIP problem require users to specify either a stored procedure [15] or multiple SQL views [1] to define the "blocks" of a matrix used as input to one particular physical solver, the simplex method [6]. This solver-oriented problem formulation is very inconvenient for the user. In contrast, SolveDB (with LP) uses this representation only at the intermediate translation step when processing algebraic problem descriptors defined as solve queries and does so transparently to the user. SQLMP (SQL for Mathematical Programming) [4] offers an SQL-like syntax to define objectives/constraints and extract data from the database. However, unlike the SELECTSOLVE clause, SQLMP's COMPUTE statement cannot be nested or used in a single query to transform data and solution between user-convenient formats. A similar limitation is encountered in DGQL [2] (Decision Guidance Query Language), where each problem formulation leads into an excessive number of SQL views, each requiring heavy annotation. In comparison, SolveDB problem descriptors are processed using a DBMS query processor and the translation procedure can be extended and customized, depending on the problem type or the user's perspective of it. Most importantly, SQLMP and DGQL support only LP/MIP problems and mainly propose just the query languages, not query processing, optimization, DBMS integration, and experimental evaluation, unlike SolveDB.

Among related systems, Tiresias [18] is built on top of a relational DBMS and uses two languages, SQL for data management and the Datalog-like TiQL for "how-to" queries. Logicblox [10] is a deductive engine using the Datalog-like LogiQL language. In comparison, SolveDB's integrated SQL-based approach is considerably easier to learn and use for most database users/developers. As shown above, SolveDB uses more effective data-dependent partitioning and outperforms Tiresias by more than two orders of magnitude on average, due to SolveDB's efficiently integrated solver workflows. SolveDB outperforms Logicblox for all cases, and by an order of magnitude in terms of I/O and total time for partitionable problems. Similarly, InezDB [17] is a Quantifier-Free Linear Integer Arithmetic (QFLIA) solver, extended to allow referencing DBMS tables via membership constraints that tie model variables to existing tuples. Although InezDB improves solving performance by exploiting the database structure, it uses two different languages, performs no solving during database query evaluation, and offers no DBMS-specific optimizations, unlike SolveDB. Searchlight [14] is an interactive data exploration system for very large data sets, bringing constraint programming (CP) inside an array DBMS. Seachlight focuses on, and is optimized for, a very narrow class of search problems, where solutions are (groups of) database objects satisfying constraints on shape and content. By partitioning the search space of a single problem, Seachlight offers load balancing techniques and efficient heuristics based on synopsis for processing these partitions efficiently. Inspired by Tiresias, PaQL [3] is an SQL-based language for "package queries" that retrieve tuples based on user-specified linear constraints and preferences. Similar to Seachlight, PaQL query approximation techniques allow scaling to large datasets for this very narrow class of search problems. In contrast, SolveDB is a common platform for in-DBMS solver integration, offering solve query optimizations and unified solver interfaces, while supporting much broader classes of problems. Furthermore, Logicblox, Tiresias, PaQL, InezDB, and Seachlight are limited to LP/MIP/CP only, unlike SolveDB.

Existing in-DBMS analytics tools, including *MADlib* [13], commercial products from the big DBMS vendors, and *statistical* [20] and *forecasting* [8] methods, either implement or rely on specific (non-linear) problem solvers such as *simulated annealing* or *conjugate gradient*. Sharing SolveDB's aim of unifying these in-DBMS tools and techniques, Bismarck [7] takes a narrower approach in which data analytics tasks are formulated/solved as convex optimization problems using a hardcoded *incremental gradient descent* (IGD) solver based on user-defined aggregates. In comparison, SolveDB is much more general and versatile, supporting all the

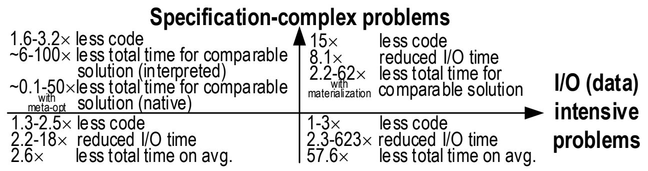

Specification-complex problems
1.6-3.2× less code
~6-100× less total time for comparable solution (interpreted)
~0.1-50× with meta-opt less total time for comparable solution (native)
15× less code
8.1× reduced I/O time
2.2-62× with materialization less total time for comparable solution
I/O (data) intensive problems
1.3-2.5× less code
2.2-18× reduced I/O time
2.6× less total time on avg.
1-3× less code
2.3-623× reduced I/O time
57.6× less total time on avg.

Figure 13: Summary of SolveDB comparison results

mentioned types of physical solvers and exposing their capabilities (but not their complexities) to the query level. It allows database users/developers dealing with almost any kind of database-based optimization problem in a simple, integrated, and SQL-based way.

8. CONCLUSION AND FUTURE WORK

This paper presented SolveDB – the first purely SQL-based DBMS for optimization applications, integrating solvers for different problem classes and offering rapid problem specification and solving to (nearly) any application system using an SQL back-end. SolveDB allows defining, solving, and processing problem solutions using a single *solve query* with an intuitive SQL-based syntax, similar to mathematical notation. To process the solve query, SolveDB uses one (of the many supported) SolveDB-compliant *view solvers* (automatically or manually selected), which bridge the gap between a DBMS and solver applications while offering additional optimizations and modelling tricks, and performing multi-level translations to provide problem formulations and solutions in formats that are convenient for the user rather than for the underlying solving technique. The paper presented solve query processing and elaborated on how view solvers, both a general-purpose (LP/MIP) and a problem-specific (energy balancing), solve problems. Finally, it presented solver optimizations (*problem partitioning*, *parametrization*), solve query optimizations (*solver advising*, *materialization*, and *meta-optimization*), the SolveDB architecture, and the PostgreSQL-based implementation with view solvers for LP/MIP, black-box, and domain-specific problems. The results of the extensive experimental evaluation, summarized in Figure 13, showed that SolveDB is a versatile solver integration, and a competitive problem solving, system offering significantly increased tool usability (order of magnitude less code), while in most cases providing much (up to >2 orders of magnitude) better performance.

Future work will develop cost-based optimization techniques for solver advising, order-level view materialization, model-level view multi-processing, and selection predicate pushdown from the output to the input relation to prune irrelevant problem partitions early. Additionally, further solver and solver workflow optimizations will be explored, including solvers that incrementally update solutions, partition and parallelize search in complex solution spaces, support automatic differentiation, and compile problem formulations into efficient code (e.g., JIT-compiled Java) to improve performance for queries requiring iterative solving. Lastly, more general issues such as language expressiveness and design choices on different DBMS platforms will be investigated.

9. REFERENCES

[1] A. Atamtürk, E. L. Johnson, J. T. Linderoth, and M. W. P. Savelsbergh. A Relational Modeling System for Linear and Integer Programming. *OR*, 48(6):846–857, 2000.

[2] A. Brodsky, N. Egge, and X. Wang. Supporting Agile Organizations with a Decision Guidance Query Language. *J MANAGE INFORM SYST*, 28(4):39–68, 2012.

[3] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou. Scalable package queries in relational database systems. *PVLDB*, 9(7):576–587, 2016.

[4] J. Choobiheh. SQLMP: A Data Sublanguage for Representation and Formulation of Linear Mathematical Models. *ORSA Journal of Computing*, 3(4):358–375, 1991.

[5] S. Conchon and J.-C. Filliâtre. A persistent union-find data structure. In *Proc. of Workshop on ML*, pages 37–46, 2007.

[6] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[7] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. In *Proc. of SIGMOD*, pages 325–336, 2012.

[8] U. Fischer, L. Dannecker, L. Siksnys, F. Rosenthal, M. Böhm, and W. Lehner. Towards Integrated Data Analytics: Time Series Forecasting in DBMS. *Datenbank-Spektrum*, 13(1):45–53, 2013.

[9] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proc. of SIGMOD*, pages 331–342, 2001.

[10] T. J. Green, M. Aref, and G. Karvounarakis. LogicBlox, Platform and Language: A Tutorial. In *Datalog in Academia and Industry*, pages 1–8, 2012.

[11] H. J. Greenberg and F. H. Murphy. Views of mathematical programming models and their instances. *DSS*, 13(1):3–34, 1995.

[12] P. J. Haas, P. P. Maglio, P. G. Selinger, and W. C. Tan. Data is Dead... Without What-If Models. *PVLDB*, 4(12):1486–1489, 2011.

[13] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library:or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[14] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[15] A. Kawaguchi and A. Nagel. *Linear Programming in Database*, chapter 19, pages 339–354. In-Tech Press, 2008.

[16] J. J. Levandoski, M. F. Mokbel, and M. E. Khalefa. CareDB: a context and preference-aware location-based database system. *PVLDB*, 3(2):1529–1532, 2010.

[17] P. Manolios, V. Papavasileiou, and M. Riedewald. Ilp modulo data. In *Proc. of FMCAD*, pages 28:171–28:178, 2014.

[18] A. Meliou and D. Suciu. Tiresias: The Database Oracle for How-to Queries. In *Proc of SIGMOD*, pages 337–348, 2012.

[19] W. A. Muhanna and R. A. Pick. Meta-modeling concepts and tools for model management: a systems approach. *Manage. Sci.*, 40(9):1093–1123, 1994.

[20] C. Ordonez. Statistical Model Computation with UDFs. *TKDE*, 22(12):1752–1765, 2010.

[21] M. E. H. Pedersen and A. J. Chipperfield. Simplifying Particle Swarm Optimization. *APPL SOFT COMPU*, 10(2):618–628, 2010.

[22] R. Ramanathan, R. Engle, C. W. J. Granger, F. Vahid-Araghi, and C. Brace. Short-run Forecasts of Electricity Loads and Peaks. *INT J FORECASTING*, 13(2):161–174, 1997.

[23] S. K. Smit and A. E. Eiben. Comparing Parameter Tuning Methods for Evolutionary Algorithms. In *IEEE CEC*, pages 399–406, 2009.

[24] T. Tusar, E. Dovgan, and B. Filipic. Evolutionary scheduling of flexible offers for balancing electricity supply and demand. In *Proc. of IEEE CEC*, pages 1–8, 2012.