CONTENTS

SolveDB$^+$: SQL-Based Prescriptive Analytics

Laurynas Siksnys, Torben Bach Pedersen, Thomas Dyhre Nielsen, Davide Frazzetto
Department of Computer Science, Aalborg University, Denmark
{ siksnys, tbp, tdn }@cs.aau.dk david.frazzetto@gmail.com

ABSTRACT

Today, advanced data analysts make use of both predictive models and optimization problem solving to build data-driven decision making applications, a combination of technologies recently termed *Prescriptive Analytics* (PA). Current PA applications typically have multiple layers of poorly integrated components: a relational DBMS for data storage/management, ML tools for prediction, and specialized software packages for problem modeling and optimization problem solving. This complex stack leads to inefficient, labor-intensive, and error-prone PA workflows, blocking wider adoption of PA. In this paper, we present SolveDB$^+$ – an RDBMS for PA applications which supports all PA steps with *modeling*, *predictive*, and *optimization* functionalities, and integrates these in a common SQL-based framework. Major SolveDB$^+$ novelties are 1) a powerful SQL-based approach for PA problem specification and solving, 2) an extensible in-DBMS infrastructure for prediction and optimization solvers, and 3) in-DBMS modeling and management of PA models. SolveDB$^+$ significantly improves both PA developer productivity and performance.

1 INTRODUCTION

As the next step after Predictive Analytics, *Prescriptive Analytics* (PA) has recently emerged as a new frontier in analytics, combining data management, predictive analytics and ML, and operations research [17]. PA provide a specific course of action for questions such as "How should we maximize our sales in Europe?" PA systems are still in their infancy, typically glued together in an ad-hoc system with separate analytics and optimization tools on top of an RDBMS. There are no integrated PA platforms that combine *data management*, *predictive*, and *optimization* functionalities using a single language, e.g., the frequently used in-DBMS analytics engines only support the first two.

As a running PA example, we consider renewable energy optimization. In a building, PV panels produce intermittent, varying electricity, to run its Heating, Ventilating, and Air Conditioning (HVAC) system. We want to reduce energy costs by using more PV electricity, which requires aligning HVAC operation to PV supply ahead of time, taking forecasted prices and user comfort into account. Table 1 shows a dataset for this case. Input data is a multivariate time series of outdoor (`OutTemp`)/indoor (`inTemp`) temperatures, HVAC consumption (`hLoad`), and PV production (`pvSupply`) per hour. Rows 07:00 - 11:00 are historical data from sensors. Rows 12:00 - 16:00 define future states: `outTemp` contains forecasted outside temperatures; the unknown values of `inTemp`, `hLoad` and `pvSupply` in 12:00 - 16:00 represent decision variables for which PA should compute values by aligning `hLoad` with `pvSupply` at the next 5 hours such that `inTemp` remains within the 20–24°C comfort range and HVAC power limits (0–17kW) are respected. The workflow below exemplifies the 5 overall phases of PA seen in Figure 1.

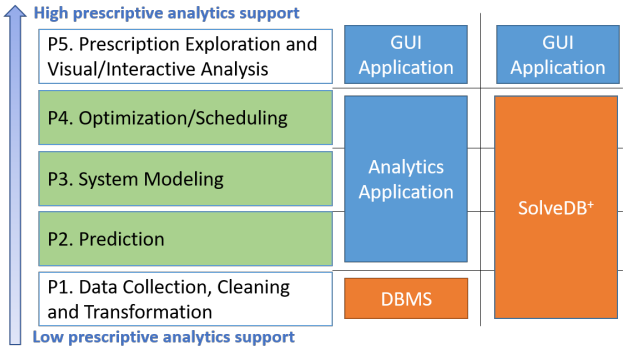

Figure 1: The 5 PA phases and the used software stacks

P1: Collect, clean, validate, and transform the input data.
P2: Predict PV supply given `pvSupply` and `outTemp`.
P3: Model `inTemp` dynamics in relation to `inTemp` and `hLoad`, which requires tuning parameter values specific to this building.
P4: Find optimal `hLoad` values by minimizing electricity cost subject to initial conditions, `pvSupply`, `hLoad`, and comfort constraints, applied over the calibrated model (P3).
P5: Analyze, visualize, and validate the results.

Traditionally, this PA workflow requires a complex software stack with different tools for data management, forecasting, system modeling, and optimization, leading to several *problems*: **Steep Learning Curve:** Different tools have different usage and modeling methodologies, making the learning curve for building PA applications much steeper, which, in turn, leads to more *errors* and *misuse*. **Poor Developer Productivity:** The tools are based on different programming/query languages and have to be glued together in ad-hoc ways to realize PA workflows, leading to *poor developer productivity*, *tool incompatibilities*, and even more *errors* [2]. **Bad performance:** Large amounts of data have to be shipped back and forth between the many tools, leading to high *I/O and memory costs* and *long runtimes* (see Sec. 5). To remedy these problems,these *research challenges* (RCs) must be met:
RC1: Provide a concise yet powerful SQL-based syntax for PA decision problems, supporting efficient query processing.

Table 1: Input dataset for campus energy management.

time	outTemp	inTemp	hLoad	pvSupply
2017/07/02 07:00	05	21	100	0
2017/07/02 08:00	06	20.5	250	0
2017/07/02 09:00	06	21	150	200
2017/07/02 10:00	07	23	120	254
2017/07/02 11:00	08	23	80	320
2017/07/02 12:00	09	?	?	?
2017/07/02 13:00	11	?	?	?
2017/07/02 14:00	12	?	?	?
2017/07/02 15:00	11	?	?	?
2017/07/02 16:00	11	?	?	?

RC2: Provide a concise yet powerful way to share optimization models across sub-problems of the overall PA problem.
RC3: Provide a powerful, easy-to-use, and extensible way of transparently integrating external prediction functionality into PA workflows.
RC4: Seamlessly integrate RC1–RC3 in a SQL-based system.

To meet these challenges, we present SolveDB$^+$. The fact that most PA systems use an RDBMS for data storage [16, 17] combined with the huge popularity of in-DBMS analytics (see Sec. 2), motivates us to propose the first SQL-based in-DBMS platform for PA applications, with these features (www.daisy.aau.dk/solvedb):

Supporting all PA phases: SolveDB$^+$ integrates data management, prediction, system modeling, and optimization in a single tool, yielding better PA productivity. **Extensibility:** SolveDB$^+$ allows developers to add new functionalities for custom PA applications. **Unified SQL-Based PA language:** SolveDB$^+$ extends SQL with new declarative constructs for unified PA problem modeling and analytical functionalities. An entire PA workflow, including forecasting, simulation, and optimization models, can be expressed in a single extended SQL query. **High performance:** The built-in PA algorithms (and user extensions) run in-DBMS, yielding more efficient execution and data exchange. Our experiments show that SolveDB$^+$ yields up to three orders of magnitude better performance for individual PA steps, and up to 3.5 times faster execution and 3 times smaller implementations for complete PA workflows, compared to state-of-the-art baselines, thus combining performance with usability/productivity.

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 describes SolveDB$^+$'s prediction framework. Section 4 presents its new PA problem modeling features. Section 5 provides the experimental evaluation. Finally, Section 6 concludes and points out future work.

2 RELATED WORK

A recent extensive survey [16] identifies major emerging trends, remaining challenges, and available technology in the field of PA. In the classification used in this survey, SolveDB$^+$ falls in the category of analytical DBMSes, where analytical functionality is integrated directly within the DBMS back-end. Efforts within this category can be classified into *prediction* DBMSes, for forecasting and probabilistic analysis, and *optimization* DBMSes, for optimization problem solving. Table 2 summarizes and compares essential relevant systems in these sub-categories. The systems are compared in terms of: 1. *What primary language is used for data management* (Data QL); 2. *What primary language is used to specify analytics (incl., prediction and optimization) tasks* (Anl. QL); 3. *Does the system offer native support for predictions?* (Pred); 4. *Does the system offer native support for physical system models and estimating their parameters?* (Est), 5. *Does the system support optimization problem solving?* (Opt); 6. *Does the system support optimization (sub-)models that can be stored natively and manipulated as first-class citizens in the database, and re-used to forms more complex models?* (Mod). We now review these systems.

In-DBMS analytics is a major trend. Among prediction DBMSes, forecasting and in-database ML is supported by the major commercial DBMSes, Oracle [27], SQL Server [32], and TeraData [13]. Recently, HyPEr [12] and DB4ML [10] provide in-DBMS ML for main memory DBMSes. These systems provide efficient in-DBMS forecasting/ML functions, but lack automatic forecasting model selection, parameter estimation, optimization problem solving, and model management, unlike SolveDB$^+$. The

Table 2: Comparison between relevant tools and SolveDB$^+$

System	Data QL	Anl. QL	Pred	Est	Opt	Mod
Oracle	SQL	SQL	✓	✗	✗	✗
SQL Server	SQL	SQL	✓	✗	✗	✗
TeraData[13]	SQL	SQL	✓	✗	✗	✗
DB4ML[10]	SQL	SQL	✓	✗	✗	✗
HyPer[12]	SQL	SQL	✓	✗	✗	✗
MADlib[5]	SQL	SQL+UDF	✓	✗	✗	✗
F^{2}DB[15]	SQL	ext.SQL	✓	✗	✗	✗
SystemML[1]	R-like	R-like	✓	✗	✗	✗
MLbase[14]	R-like	R-like	✓	✗	✗	✗
SciDB [11]	SQL-like	SQL-like	✓	✗	✗	✗
pgFMU[28]	SQL	SQL+UDF	✓	✓	✗	✗
Searchlight[19]	SQL-like	SQL-like	✗	✗	✗	✗
PaQL[8]	ext.SQL	ext.SQL	✗	✗	✗	✗
InezDB[21]	ext.OCaml	ext.OCaml	✗	✗	✓	✗
Tiresias[23]	SQL	ext.Datalog	✗	✗	✓	✗
LogicBlox[6]	LogiQL	LogiQL	✓	✗	✓	✗
SolveDB[31]	SQL	ext.SQL	✗	✗	✓	✗
SolveDB$^+$	SQL	ext.SQL	✓	✓	✓	✓

open source alternative MADlib [5] extends PostgreSQL with UDFs specialized for ML tasks like clustering, classification, and forecasting. Similar to MADlib, pgFMU [28] offers PostgreSQL UDFs for in-DBMS simulation and parameter estimation of *Functional Mock-up Units* (FMUs). These are interoperable simulation models that can define dynamic behaviour of complex physical systems. While FMUs are often used for predictions (P2, see Figure 1), pgFMU does not support including FMUs into user-defined optimization problems (P4). In comparison, SolveDB$^+$ supports (less detailed) so-called *grey-box* models that can be both simulated and optimized in the same environment. Among stand-alone DBMSes, F^{2}DB [15] focuses on time series forecasting in an SQL-based environment. While F^{2}DB specializes in, and is highly optimized for, time series forecasting tasks and employing specific model reuse and maintenance techniques, it does not support the development and integration of user-defined "do-it-yourself" models and generic library models, unlike SolveDB$^+$. In the Big Data context, systems such as SystemML [1], MLBase [14], and SciDB [11] integrate general-purpose declarative machine learning tools that offer scalable distributed computations. In the context of PA, *all* systems (except pgFMU) in this category *only offer support for the predictive analytics phase (P2).*

The optimization DBMSes have focused on advanced what-if scenarios, in-DBMS optimization problem solving, and search under advanced forms of constraints. Systems such as Searchlight [19] and PaQL [8] exploit powerful constraint solvers when processing advanced data search queries. InezDB [21] proposes a formal logic for the symbolic manipulation of optimization models inside a DBMS. Tiresias [23] and LogicBlox [6] provide users a Datalog-based language for what-if scenario analysis. Being the predecessor of SolveDB$^+$, SolveDB [31] is an extension of PostgreSQL for in-DBMS optimization problem solving and solver integration. SolveDB$^+$ extends SolveDB in the directions covering the highlighted PA phases in Figure 1. These new features in SolveDB$^+$, together with their impact (to be observed in Section 5), are highlighted in Table 3. These correspond 1-1 to the research challenges RC1–RC3 mentioned in Section 1, while the integrated SolveDB$^+$ system corresponds to RC4.

Table 3: New features of SolveDB$^+$ compared to SolveDB.

Feature	Description	Impact
In-DBMS Predictive Framework	Specialized forecasting models that are easy to install, (auto)select, and use.	Forecasting easier to use and up to 6 times faster.
Shared Optimization Models	Allow defining reusable optimization (sub-)models stored in-database with their objective functions, constraints, and data specs.	Up to: 2X less code for P3-P4, 16% less code for P1-P4, similar performance.
New Language Features	Asterisk notation, common decision table expressions, model inlining allow specifying PA problems more concisely/efficiently.	Up to 5X less code for P2-P4, similar performance.

In summary, Table 2 shows that while predictive and optimization DBMSes offer some level of in-DBMS analytics support, they do it only for *some* PA phases and do not offer "SQL for all PA phases" like SolveDB$^+$. In comparison, SolveDB$^+$ is the only system to combine and unify predictions and optimization problem solving within a single SQL-based system.

Explainability, also called interpretability, of ML pipelines has received much attention in recent years. It has been considered both for specific categories of ML pipelines, e.g., user group analytics [25] or data exploration [18], and more generally in a survey of AutoML pipelines [33]. In comparison, SolveDB$^+$ focuses on another category, PA pipelines, and supports explainability in PA phases P1-P4. For P1, we do not claim any new contributions, but simply offer the time-honored explainability of SQL. For (external) Prediction methods (P2), we inherit their existing explainability and add to it by declaratively specifying input and output in the solver specs. For System Modeling (P3) and for overall integration of the phases, our high-level declarative SQL-based syntax and shared models allow a higher level of abstraction which is more compact and explainable than a traditional imperative-style ML pipeline. For Optimization (P4), the declarative specifications of objective functions are immediately explainable. Section 5 provides more details.

Another key aspect of ML pipelines is their connectivity to other components/frameworks [33]. As for the "inbound" connectivity, external components are integrated for use in SolveDB$^+$ in two ways. Like other in-DBMS analytics tools (see Tab. 2), SolveDB$^+$ uses UDFs to wrap external functions for direct use in SQL queries, Specifically to SolveDB$^+$, the solver concept is used to integrate external prediction components in a seamless way (see Sec. 3). As for the "outbound" connectivity, SolveDB$^+$ can be integrated in larger pipelines just like other SQL-based in-DBMS analytics tools.

3 PREDICTION

The first phase in Figure 1 *P1: Data Collection, Cleaning, and Transformation* is well supported by the SQL queries, built-in functions, and UDFs of traditional RDBMSes [16], including SolveDB$^+$. Since PA applications need to look ahead in time, effectively supporting the next phase *P2: Prediction* is a key research challenge (RC3). This section describes how we meet RC3. While SolveDB$^+$ can accommodate different models and algorithms for prediction (using both built-in and external tools), it offers dedicated support

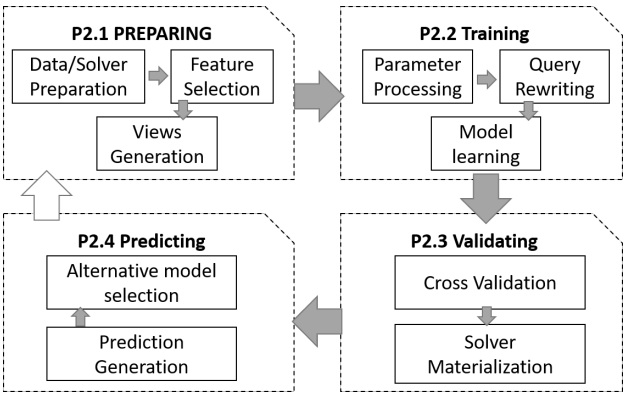

Figure 2: Prediction process + SolveDB$^+$ implementation.

for *time series forecasting* methods. These are widely used for data-driven prediction based on current and historical data.

3.1 Time series forecasting in SolveDB$^+$

Following the energy planning example, the input to the prediction phase is the time series shown in Table 1. The objective is to predict the PV supply for the next 5 hours, by filling in the missing `pvSupply` values in Table 1. This is accomplished by a specific time series forecasting method (e.g., regression) involving a number of steps, as shown in Figure 2: *preparing* – extracting and formatting the data to fit forecasting models, *training* – fitting the forecasting models on the dataset, *validating* – validating the models using cross validation or other evaluation procedures, and *predicting* – forecasting new values.

To support the user in using these methods, SolveDB$^+$ provides its in-DBMS *Predictive Framework*, which (1) exposes various time-series forecasting methods through SQL ("transparently integrating" in RC3), (2) hides the complexity ("easy-to-use" in RC3) of choosing and using these methods (the *preparation*, *training*, *validation*, and *prediction* steps), and (3) offers different extensibility options when a new forecasting method needs to be integrated ("extensible" in RC3). For example, the prediction problem above can be solved in two different ways, using:

Specific forecasting method The following example query invokes the specific forecasting method ARIMA:

```
1 SOLVESELECT t(pvSupply) AS (SELECT * FROM input)
2 USING arima_solver(predictions := 5, time_window := 5,
3                    features := outTemp)
```

To expose the method, SolveDB$^+$ uses the specialized **`SOLVESELECT`** statement (extending the one from SolveDB [31]), to be described in detail in Section 4. It invokes a SolveDB$^+$-native *solver* (*arima_solver*) to derive a so-called *output relation* (a database table) from a so-called *input relation* (**`SELECT * FROM`** `input`) by adding/deleting rows or filling in values in the specified *decision columns*. In this example, the decision column is `pvSupply`, the values of which are requested to be populated by *arima_solver*. The output relation has the same schema as the input relation, but with the `pvSupply` column filled as shown in Table 4. To derive the output relation from the input relation, `arima_solver` additionally takes solver parameters: the number of predictions (*predictions := 5*), the number of time steps to use for training (*time_window:=5*), and the column (*features:=outTemp*) to use as a feature attribute. The solver then performs the steps of *preparation*, *training*, *validation*, and *prediction* (see Figure 2) using the ARIMA model

Table 4: Output of the *Prediction* phase for the example.

time	outTemp	inTemp	hLoad	pvSupply
2017/07/02 07:00	05	21	100	0
2017/07/02 08:00	06	20.5	250	0
2017/07/02 09:00	06	21	150	200
2017/07/02 10:00	07	23	120	254
2017/07/02 11:00	08	23	80	320
2017/07/02 12:00	09	?	?	**200**
2017/07/02 13:00	11	?	?	**220**
2017/07/02 14:00	12	?	?	**260**
2017/07/02 15:00	11	?	?	**140**
2017/07/02 16:00	11	?	?	**0**

trained on data from the input relation with the given parameters. Thus, **SOLVESELECT** allows the user to invoke any specific predictive solver installed in SolveDB$^+$, including solvers for Linear Regression, Logistic Regression, ARIMA, or the powerful *Predictive Advisor* described next. The carefully designed use of the solver ensures the transparency mentioned in RC3.

Predictive Advisor Users can get automated model selection and configuration by using the *Predictive Advisor*, exposed as `predictive_solver`. This solver hides *model selection*, *feature selection*, and *parameter fitting* from the user, and transparently performs *preparation*, *training*, *validation*, and *prediction* and fills in the missing values in the input relation, thus ensuring "easy-of-use" in RC3. Now, the prediction query above can be rewritten as the following simpler query:

```
1 SOLVESELECT t(pvSupply) AS (SELECT * FROM input)
2 USING predictive_solver()
```

The extensibility offered by SolveDB$^+$ also allows for alternative automated predictive frameworks to be integrated as part of the SolveDB$^+$ predictive advisor ("extensible" in RC3).

3.2 Steps of the Predictive Framework

In SolveDB$^+$, the underlying steps of *preparation*, *training*, *validation*, and *prediction* are standardized and their common routines are shared among different forecasting methods, (ensuring "easy-of-use" in RC3.

P2.1 Preparing When the predictive solver (e.g., `arima_solver`) is invoked, the input relation is first analyzed. The framework extracts *decision* (i.e., to be populated with values) and *feature* (to be used as features) columns specified by the user. After recognizing the types of the input columns, it selects candidate solvers from the pool of predictive solvers by comparing the set of decision and features columns to those supported by the solvers. The framework logically partitions the input relation into the training, test, and validation segments by matching the schema for each candidate solver. The selected solver(s) are then used for the training step.

P2.2 Training Next, the model-specific parameters of the candidate solvers are tuned on the training segment of the input relation. The predictive framework automatically generates a **SOLVESELECT** query that specifies an optimization problem with model parameters as decision variables to optimize. This optimization problem is solved by utilizing the solving capabilities of SolveDB$^+$ (Section 4). For example, the ARIMA solver is installed with the standard ARIMA parameters `ar`, `i`, and `ma`, each associated to the domain $[0, 5]$. Therefore, `predictive_solver` described earlier automatically and transparently invokes the following parameter estimation query:

```
1 SOLVESELECT p(ar, i, ma) AS
2   (SELECT NULL::int AS ar,NULL::int AS i,NULL::int AS ma)
3 MINIMIZE(SELECT arima_rmse(
4          ar:=SELECT ar FROM p,
5          i := SELECT i FROM p,
6          ma := SELECT ma FROM p))
7 SUBJECTTO (
8          SELECT 0 <= ar <= 5, 0 <= i <= 5, 0 <= ma <= 5
9          FROM p)
10 USING swarmops.pso()
```

The above **SOLVESELECT** query specifies a global black-box optimization problem, where the values of the parameters `ar`, `i`, and `ma` are found by minimizing the RMSE between the training set and the ARIMA predictions, computed by the function `arima_rmse` in the **MINIMIZE** clause (line 3). The **SUBJECTTO** clause specifies the range in which the parameters can vary. The optimization solver `swarmops` uses a built-in particle swarm optimization method [20] to iteratively attempt to improve a candidate solution with regards to RMSE.

P2.3 Validating Next, the candidate predictive solvers are compared using cross validation. The solver/model leading to the lowest error is selected. As a side effect, the calibrated model instances are stored in a database as user-defined type (UDT) entities for fast reuse of the solver result later.

P2.4 Predicting Finally, predictions are generated by the selected best candidate solver and returned to the user in the form of an output relation of **SOLVESELECT** (Table 4). As **SOLVESELECT** expresses a view over the input relation (Table 1), no user tables are modified in the database.

3.3 Developer Interface

SolveDB$^+$ addresses the "extensible" in RC3 by providing the user with a developer interface to install new in-DBMS predictive solvers. There exists two categories of solvers: *black box* and *white box*. *Black box* solvers are expected to manually handle the steps of data preparation, feature selection, cross-validation, etc., thus *overriding* the predictive framework functionalities. In contrast, *white box* solvers expose the model specifics (e.g., model parameters, their types, etc.) as well as model training and prediction logic to the predictive framework. This way, the solvers may use the functionalities (e.g., **SOLVESELECT**) provided by SolveDB$^+$ for preparing, training, and validating. Such solvers use the solver extensibility capabilities already present in SolveDB [31]. This allows the developers to easily expand the system by taking advantage of existing SolveDB$^+$ solvers/functionality and integrating new prediction models from existing frameworks, e.g., Scikit-Learn [3], Weka [9], MATLAB [22], Statsmodels [29], and TensorFlow[7].

As we will show in Section 5, SolveDB$^+$ is able to offer reduced PA application development efforts and improved overall performance after the integration of desired solvers, yielding up to 5 times more compact problem specifications and up to 6 times reduced forecasting time, compared to SolveDB and commonly used predictive frameworks.

4 OPTIMIZATIONS AND SYSTEM MODELING

Optimization problem solving is essential in 3 of the 5 PA phases (P2, P3, P4), and it therefore plays an essential role in SolveDB$^+$. To deal with optimization problems, SolveDB$^+$ borrows a number of solvers from SolveDB for the different classes of optimization problems, including *linear programming* (LP), *mixed-integer programming* (MIP), and blackbox global optimization (GO), some of

Table 5: LR problem variable layout and a new c_mask column introduced during the CDTE rewrite

id	*pOTemp*	*pMonth*	*pEps*	*error*	**c_mask**
1	*pOTemp*	*pMonth*	*pEps*	e_1	B'11'
2				e_2	B'01'
...				...	B'01'
M				e_M	B'01'

which were already demonstrated in Section 3. To address RC1, SolveDB$^+$ further extends the query syntax used for accessing these solvers. We now elaborate on these new language features.

4.1 Model Specification Syntax

SolveDB$^+$ uses the following syntax to interact with various (e.g., LP/MIP) solvers registered in the active database:

```
1 {SOLVESELECT | SOLVEMODEL}
2         [alias[(col_name[,...])] AS](select_stmt)
3 [INLINE [alias                   AS](select_stmt) [,...]]
4 [WITH   [alias[(col_name[,...])] AS](select_stmt) [,...]]
5 [MINIMIZE (select_stmt) [MAXIMIZE (select_stmt)] |
6  MAXIMIZE (select_stmt) [MINIMIZE (select_stmt)]]
7 [SUBJECTTO [alias AS] (select_stmt) [,...]]
8 [USING solver_name[.method_name][(param[:=expr][,...])]]
```

As shown earlier, the user can use **SOLVESELECT** to define a *model* and pass it to SolveDB$^+$-compliant solver `solver_name` for evaluation using an optionally specified solving method, `method_name`, all defined as follows.

A *problem model* m is defined as a 4-tuple (D, R, s, m). D is the specification of *data and decision variable columns* (lines 2,4). R is the specification of rules that define how the values of the decision variable columns should be instantiated (lines 5-7). s is the name of the solver (`solver_name`) that should evaluate the rules R on the given D using some method m (`method_name`, line 8). Both D and R define two separate sets of specially annotated database relations. Specifically, $D = (D_1^{a_1}, D^{a_2}, \ldots, D_N^{a_N})$ where, $\forall i \in 1 : N, D_i^{a_i} = (c_1, \ldots, c_k, \overline{c_1}, \ldots, \overline{c_l})$ is a SELECT statement (`select_stmt`) defining a database relation with the alias a_i (`alias`) assigned and defined by k *data columns* $c_1, \ldots, c_k$ and l so-called *decision columns* $\overline{c_1}, \ldots, \overline{c_l}$ (`col_name`). Decision columns denote that their rows are decision variables, the values of which should be computed by s. Here, $D_1^{a_1}$ (line 2) is denoted as *input relation*. In a similar way, $R = (R_1^{min}, R_2^{max}, R_3^{a_3}, \ldots, R_M^{a_M})$ is a set of relations that contain s-specific representations of rules defining how decision column values in D should be computed. For convenience, the aliases of R_1^{min} and R_2^{max} are fixed and they are specified in the **MINIMIZE** and **MAXIMIZE** clauses, respectively (line 5-6). The remaining $R_3^{a_3}$ to $R_M^{a_M}$ are specified in the **SUBJECTTO** block along with their respective aliases (line 7). This provides powerful yet concise model specs for RC1.

A *solver* in SolveDB is a user-defined function (UDF) capable of producing (a query for) a so-called *output relation* O in the schema of the *input relation* $D_1^{a_1}$ from a given problem model instance (D, R, s, m) and additionally supplied solver parameters `param` (line 8). SolveDB$^+$ assumes the following standard scoping rules within SOLVESELECT. Each $d_i^{a_i} \in D$ may access a relation $d_j^{a_j} \in D$ using the alias a_j if $j < i$, i.e., $\forall d_i^{a_i} \in D : scope(d_i^{a_i}) = \{(a_j \mapsto d_j^{a_j} | d_j^{a_j} \in D, j < i\}$. Each $r_i^{a_i} \in R$ may access all data and decision variable tables, i.e., $\forall r_i^{a_i} \in R : scope(r_i^{a_i}) = \{(a \mapsto d^a | d^a \in D\}$.

For example, consider a predictive solver (for P2) based on linear regression (LR). In SolveDB$^+$, LR model parameter estimation is specified using the following **SOLVESELECT**:

```
1 SOLVESELECT p(pOTemp, pMonth, pEps) AS (SELECT * FROM pars)
2 WITH e(error) AS (SELECT *, NULL::float8 AS error
3                   FROM input)
4 MINIMIZE (SELECT sum(error) FROM e)
5 SUBJECTTO(SELECT -1*error <=
6   (pOTemp*outTemp + pMonth*month(time) +
7   pEps - pvSupply) <= error FROM e, p)
8 USING solverlp.cbc()
```

Here, lines 1-3 specify model *data and decision columns*. Lines 4-7 specify *rules* that define an objective function and constraints that involve decision variables from the tables p and e. Finally, line 8 specifies *solverlp* and *cbc* as a SolveDB$^+$-compatible solver and a solving method, respectively.

This general **SOLVESELECT** syntax based on standard SQL SELECTs allows exposing different kinds of models and solvers to user queries in a powerful yet concise way (RC1). Compared to SolveDB, SolveDB$^+$ uses a number of novel modeling features unavailable in SolveDB. These are outlined in the remainder of this section.

4.2 Asterisk notation

To support RC1's need for concise and powerful syntax , SolveDB$^+$ proposes the asterisk (*) notation for decision variable column specification (`col_name`). Like SELECT * in the standard SQL, this allows declaring all table columns as decision variables, thus offering more compact problem specifications. Using asterisks, Line 1 in the above optimization problem can be concisely specified as **SOLVESELECT** `p(*)` **AS** (**SELECT** * **FROM** `pars`).

4.3 Common Decision Table Expressions

In SolveDB, the **WITH** clause within **SOLVESELECT** is not supported. Consequently, decision columns (variables) are only allowed in a single (input) relation $D_1^{a_1}$ (i.e., $N = 1$). Therefore, objective and constraint (SELECT) expressions in the **MINIMIZE**/ **MAXIMIZE** and **SUBJECTTO** blocks may become unnecessarily large and complex. Consider the LR model fitting example. This problem uses 2 collections of decision variables: *pOTemp*, *pMonth*, *pEps* as model parameters and e_1, e_2, ..., e_M ($M >> 3$) as prediction errors. One of the most convenient ways to arrange these variables in a single input relation in SolveDB is depicted in Table 5. Here, *pOTemp*, *pMonth*, *pEps* are contained within a single row and e_1, ..., e_M contained within a single column, with many "empty cells" representing *unbound* decision variables. When not referenced within **MINIMIZE**/**MAXIMIZE** and **SUBJECTTO** expressions, such unbound variables are automatically excluded from computations by SolveDB$^+$. Still, referencing *pOTemp*, *pMonth*, *pEps* in the objective and constraint expressions is quite cumbersome - the user is required to supply the predicate **WHERE** `id=1` in all relevant **MINIMIZE**/ **MAXIMIZE**, and **SUBJECTTO** expressions. This makes problem specifications complex and less readable, especially when more than two variable collections are modeled.

Again meeting RC1's need for concise and powerful syntax, SolveDB$^+$ proposes to extend the SOLVESELECT clause with so-called *Common Decision Table Expressions* (CDTEs). As an extension of Common Table Expressions (CTEs, i.e. WITH queries), these allow specifying additional temporary relations, $D_2^{a_2}$, ..., $D_N^{a_N}$, with or without decision columns, where each relation $D_i^{a_i}$ can be accessed from SELECTs of $D_j^{a_j}$, $j > i$, and in the

MINIMIZE/**MAXIMIZE** and **SUBJECTTO** blocks ($R_1^{min}, \ldots, R_M^{ctr_M}$) using the alias a_i. All decision variables of $D_1^{a_1}, \ldots, D_N^{a_N}$ are solved together in a single optimization problem. Note, when the list of the decision columns is empty ($|\{\overline{c} \in D_i^{a_i}\}| = 0$), the CDTE has the semantics of the standard CTE. As demonstrated earlier, CDTEs in SolveDB$^+$ allow conveniently modeling two or more collections of decision variables, unlike SolveDB.

Efficient CDTE query evaluation ("efficient query processing" in RC1): SolveDB$^+$ efficiently evaluates SOLVESELECT queries with CDTEs in two different ways. SolveDB$^+$ either rewrites the CDTEs to a single input relation and standard CTEs, or passes them to a solver for specialized processing. The first approach is preferred, as it is transparent and applicable to all registered SolveDB$^+$ solvers. Here, SolveDB$^+$ first generates a new input relation ($D_1'^{a_1}$) by joining all CDTEs with decision variables and adding a special bit string attribute `c_mask` (see Table 5) to denote CDTEs relevant to specific rows. Then, SolveDB$^+$ generates and processes a new SOLVESELECT *without* decision variables in CDTEs, by using different projections over the new input relation:

```
1 SOLVESELECT l(pOTemp,pMonth,pEps,error) AS
2             (SELECT * FROM input)
3 WITH p AS (SELECT pOTemp, pMonth, pEps FROM l
4            WHERE (c_mask & b'10') <> b'00'),
5      e AS (SELECT error FROM l
6            WHERE (c_mask & b'01') <> b'00')
7 MINIMIZE(SELECT sum(error) FROM e) ...
```

This syntactical extension does not increase the expressive power of SOLVESELECT as the WITH sub-expressions can always be combined into a joint input relation. Instead, CDTEs allow a more intuitive and comcise organization of decision variables in a SOLVESELECT query ("powerful yet concise" in RC1), which is particularly useful when dealing with many auxiliary variables in complex PA cases.

4.4 Shared Models and Model Management

PA applications often build (optimization) models by combining several existing models, e.g. for P3 in our use-case we want to use a generic linear time-invariant *state-space model* (LTI) for capturing temperature dynamics of the HVAC-equipped campus building, and then apply this model in two optimization problems – *LTI model parameter estimation* and *electricity cost optimization* – P3 and P4 in Figure 1. For the first problem, we want to use our input data to estimate the parameters a_1, b_1, and b_2 of the following discrete LTI model for this specific building:

$$x[n+1] = \begin{bmatrix} a_1 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} b_1, b_2 \end{bmatrix} \mathbf{u}[n]$$
$$y[n] = \begin{bmatrix} 1 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 0, 0 \end{bmatrix} \mathbf{u}[n]$$

Here, $\mathbf{x}$ is the system 1×1 *state vector* denoting the *inside temperature* of the building; $\mathbf{u}$ is the system 2×1 *input vector* denoting *outside temperature* and applied *HVAC load*, and $\mathbf{y}$ is the 1×1 *output vector* which just "feeds forward" the inside temperature.

In the second problem, we want to use this LTI model with instantiated parameters a_1, b_1, and b_2 inside the cost optimization problem with additionally specified constraints on state variables (inside temperature bounds) and input variables (HVAC power bounds). Obviously, these two problems share the common specification of the generic LTI model (i.e., equations above). However, the LTI model constraints have to be redefined in each of the problems when using SolveDB, as there is no way to reuse them.

Algorithm 1: Problem model instantiation

Input: m - a generic model; Δm - instantiation model
Output: m' - an instantiated model

1 $D \leftarrow \{d^{alias} \in m.D | alias \notin \{alias | d^{alias} \in \Delta m.D\}\} \cup \Delta m.D$
2 $R \leftarrow \{r^{alias} \in m.R | alias \notin \{alias | r^{alias} \in \Delta m.R\}\} \cup \Delta m.R$
3 **return** *(D, R, m.s, m.m)*

To address RC2, SolveDB$^+$ proposes the concept of a *shared problem model*. The shared problem model is a special user-defined data type (UDT), which can be created via the **SOLVEMODEL** clause sharing the same syntax as **SOLVESELECT** (see above). Instead of returning an output relation, this new clause returns the UDT with the complete problem specification inside, i.e., (D, R, s, m). In SolveDB$^+$, such UDTs can be *transformed*, *used in computations*, or *stored* in a database using SolveDB$^+$ queries. The shared LTI model of the building inside temperature can be specified, for example, as:

```
1 SELECT (SOLVEMODEL
2   pars  AS (SELECT 0.0 AS a1,0.0 AS b1,0.0 AS b2)
3 WITH
4   data0 AS (SELECT 21.0 AS inTemp),
5   data  AS (SELECT time,outTemp,inTemp,hLoad FROM input),
6   simul AS (
7     WITH RECURSIVE t(time, x, inTemp) AS (
8       -- Initial data, for step 0
9       SELECT (SELECT min(ts) FROM data) AS time,
10             (SELECT x0 FROM data0) AS x,
11             (SELECT intemp0 FROM data0) AS inTemp
12    UNION ALL
13      -- Computed data, for steps > 0
14      SELECT (SELECT time+interval '1 hour'),
15             (SELECT a1*x+b1*outTemp+b2*hLoad FROM pars),
16              n.inTemp
17      FROM t LEFT JOIN LATERAL
18               (SELECT time, inTemp, outTemp, hLoad
19                FROM data) AS n
20             ON t.time = n.time - interval '1 hour'
21      WHERE (time < (SELECT max(time) FROM data))
22    SELECT time, x, intemp FROM t)))
```

As seen in the example, this model is, essentially, a placeholder with (dummy) relations for LTI model parameters (*pars*), initial values of the state variables (*data*0), and system inputs to be used for model training or predictions (*data*); and relations that represent simulated system states and outputs (*simul*). This model is fairly useless without actual model parameters and data being specified. Therefore, SolveDB proposes 3 specialized "conside yet powerful" operations on shared problem models: *instantiation*, *evaluation*, and *inlining*.

Model instantiation This operation instantiates a (generic) model into a (specific) problem model instance. This is done by allowing the user to redefine the input relation or any other CDTE in the problem model, along with their decision column list. For this, the operator << and another model are used, e.g.,

```
1 SELECT m << (SOLVEMODEL pars(b2) AS
2   (SELECT 0.995 AS a1, 0.001 AS b1, 0.2::float8 AS b2))
3 FROM model
```

In this example, a generic LTI model `m` is first selected from the table `model`. Then, `m` is instantiated using specifications of another model (say Δm) that is generated with **SOLVEMODEL** in the same query. Finally, the instantiation operator << replaces *pars* in m with *pars* in Δm while denoting $\{b2\}$ as a sole decision column with its initial value given in the table. The semantics of this operator is seen in Algorithm 1.

In general, as seen in Algorithm 1, model instantiation allows transferring an input relation, objective functions, constraint expressions, and any other CDTE expression from a *source model* to a *target model*. All entities that cannot be found using an *alias* in the target model are automatically added (instead of replaced) to the target model. This gives the possibility to inject data, different model parameters, objectives, constraints into a generic model.

Model Evaluation This operation allows accessing data from the input relation or any other CDTE inside the model. For this, SolveDB$^+$ introduces a new **MODELEVAL** clause:

```
MODELEVAL ( select_stmt ) IN ( select_stmt )
```

This clause retrieves a model instance by evaluating the 2^{nd} SELECT expression (`select_stmt`), then turns this model into a number of standard CTEs, and finally evaluates the 1^{st} SELECT expression in the context of these CTEs. Thus, the user can retrieve and inspect data specified by the model, e.g.,

```
MODELEVAL (SELECT a1, b1, b2 FROM pars)
       IN (SELECT m FROM model)
```

Model Inlining This operation allows embedding a model instance into another model instance – specified either by SOLVEMODEL or SOLVESELECT. To inline the model, the **INLINE** clause in **SOLVESELECT** or **SOLVEMODEL** is used, e.g.:

```
SOLVESELECT t(a1,b1,b2) AS
            (SELECT 0.5 AS a1,0 AS b1,0.5 AS b2)
     INLINE m AS (SELECT m <<
(SOLVEMODEL params AS (SELECT a1, b1, b2 FROM t)
       WITH  data0 AS (SELECT 25.0::float8 AS inTemp),
              data AS (SELECT * FROM input
              WHERE hload IS NOT NULL ))  FROM model)
MINIMIZE  (SELECT sum((x-inTemp)^2) FROM m_simulation)
SUBJECTTO (SELECT 0<=a1<=1, 0<=b1<=1, 0<=b2<=1 FROM t)
USING swarmops.sa()
```

This query specifies the problem of *least squares* to fit the LTI model parameters a_1, b_1, b_2 to the given data (Table 1). Here, the INLINE clause specifies that this problem depends on the shared problem model m from the table *model*. Before applying m to the outer problem, the model m has to be first instantiated with new LTI model parameters (line 4), a new initial value of the state variable (line 5), and new training dataset (line 6-7). Note, the decision columns (variables) from the outer problem (a1,b1,b2) are passed to the inner model during the instantiation, so their values can be used in computations defined by the inner model. Given this query, SolveDB$^+$ generates a new (outer) problem instance, making all internal model relations ($m.D$, $m.R$) available to the constraint expressions of the outer problem (lines 8-9) using the prefix $m\_$, where m is the assigned model alias (line 3).

The injection of the decision variables through model instantiation is not the only way to interconnect inner and outer problems in SolveDB$^+$. Another way is to declare that some of the inner model relations (CDTEs) contain decision columns. Consider the optimization/scheduling step of the PA process (P4 in Figure 1). To solve the cost minimization problem, SolveDB$^+$ allows defining the following query:

```
1  SOLVESELECT t(hload, iTemp) AS
2        (SELECT time, outTemp, inTemp, hLoad, pvSupply
3         FROM input WHERE hload IS NULL)
4  INLINE m AS (SELECT m << (SOLVEMODEL
5   data AS (SELECT time,outTemp,0 AS inTemp,hLoad FROM t)
6   WITH data0(inTemp) AS (SELECT NULL::float8 AS itemp))
7                FROM model)
8  MINIMIZE (SELECT sum((hload - pvsupply)*0.12) FROM t)
9  SUBJECTTO
10  -- Bind inner and outer problem variables
11  (SELECT t.inTemp = m_simul.x    FROM m_simul, t
12    WHERE t.time   = m_simul.time),
13  -- Initial conditions
14  (SELECT iTemp=20 FROM m_data0),
15   -- Comfort and HP power constraints
16  (SELECT 20<=intemp<=25, 0<=t.hpload<=17000 FROM t)
17 USING solverlp.cbc();
```

As seen here, model instantiation is used to declare that the attribute `inTemp` in the CDTE `data0` of the model m should be treated as decision column (line 6). Thus, a new decision variable(-s) will be introduced in the inner problem and made available to the specification of the outer problem (line 14).

Algorithm 2 elaborates the semantics of this **INLINE** clause. As seen in the algorithm, SolveDB$^+$ imports the input relation, CDTEs, and rule expressions from the inner model m into the outer model o. Each such expression receives a new prefixed alias for use in the outer problem to prevent naming collisions (lines 3,7). Further, table access scopes of these expressions are reworked such that the new relations (with new aliases) in the outer model can be accessed from the inner model expressions using the initial aliases, and without the need to modify the actual expressions (lines 5,9). In SolveDB$^+$, this is done by introducing additional CTEs in inner model expressions, e.g., **WITH** `data0` **AS** (**SELECT** * **FROM** `m_data0`), where m_data0 becomes a part of the outer model, but $data0$ is used in the inner model instead.

Algorithm 2: Problem model inlining

Input: o - a model instance before inlining; m - a model instance to be inlined; ma - a model alias;
Output: o' - a model instance after inlining

1 $prefix \leftarrow ma +' \_'$;
2 **for** $i \leftarrow 1 : |m_i.D|$ **do**
3 $\quad d^{prefix+a} \leftarrow \{d_i^a | d_i^a \in m.D\}$;
4 $\quad o.D \leftarrow o.D \cup \{d^{prefix+a}\}$;
5 $\quad scope(d^{prefix+a}) \leftarrow \{a_j \mapsto d^{prefix+a_j} | d_j^{a_j} \in m.D, j < i, d^{prefix+a_j} \in o.D\}$;
6 **for** $i \leftarrow 1 : |m_i.R|$ **do**
7 $\quad r^{prefix+a} \leftarrow \{r_i^a | r_i^a \in m.R\}$;
8 $\quad o.R \leftarrow o.R \cup \{r^{prefix+a}\}$;
9 $\quad scope(r^{prefix+a}) \leftarrow \{a_j \mapsto d^{prefix+a_j} | d_j^{a_j} \in m.D, d^{prefix+a_j} \in o.D\}$;
10 **return** *(o.D, o.R, o.s, o.m)*

Finally, as seen above, SolveDB$^+$ can "seamlessly integrate" the RC1-RC3 contributions of Sec. 3 and 4 and thus address RC4, allowing the user o specify a complete PA workflow as an extended SQL query. SolveDB$^+$ offers efficient in-DBMS processing by optimally using the DBMS query optimization and execution machinery for processing solver inputs and outputs, allowing for integrated (cache-aware) and optimized processing of PA workflows. The effects of using SolveDB$^+$ and its novel extensions are evaluated next.

5 EXPERIMENTAL EVALUATION

In this section, we first present results from a SolveDB$^+$ usability study involving a group of data scientists. To support the end user claims about SolveDB$^+$, we also evaluated SolveDB$^+$ on two typical PA use-cases from the fields of *energy* and *supply chain management*. Lastly, we used these use-cases to compare SolveDB$^+$ against SolveDB.

Table 6: Strong and Weak Points of SolveDB$^+$

Strong points
"*Syntax very SQL-like, queries feel natural, intuitive from a SQL users perspective. This also makes it **very** easy to pick up for anyone familiar with basic SQL.*"
"*I liked the syntax that makes you feel you are still working inside the database sphere while solving optimization problems without the need to jump between different solutions/languages*"
"*SolveDB+ is still a database system, meaning that it would be possible to to use it even in legacy systems...*"
"*I think SolveDB+ is a great tool! ... For any professionals I see this type of tool as the only tool for fast analytics.*"
"*... great idea and great tool. I have already suggested one of my students to check it out also...I am surprised how easy it was to implement and solve problems - definitely not the last time I will work with SolveDB+*"
"*Seems like a much more streamlined development experience.*"
"*Easy to use in a database-context*"
"*I do think python is more intuitive, but SolveDB+ is very close.*"
"*The simplicity, readability, easy to adapt and learn.*"
"*Fewer lines of code needed to solve the same problem...*"
"*SolveDB+ was faster than MADlib+pIPython*"
Weak points
"*...for some optimization problems, we need to put some "extra" effort to produce a good "representation" of the problem so it that can be handled by SolveDB+ (e.g. Sudoku solver). SolveDB+ needs a big community, and more detailed documentations and examples.*"
"*Needs to be updated on every PostgreSQL release*"
"*Due to relational nature of SQL syntax, some expressions are longer than they ideally should be*"

5.1 Usability Study

We conducted a study where the usability of SolveDB$^+$ was evaluated by a group of highly skilled data scientists, namely the 7 participants of the 2.5 day PhD course *Aspects of Advanced Analytics*, organized by Aalborg University in Dec. 2020. Each participant pre-reported strong competences in SQL, Python, PostgreSQL, and optimization problem solving. The participants used SolveDB$^+$ to solve their chosen subset of five simple optimization problems (Knapsack, production planning, Sudoku, curve fitting, and hypothetical DB deletes/inserts) and two more advanced PA problems (demand and supply balancing, heat-pump power optimization)[30]. In all cases, the initial data and the solution had to be stored in a database. For comparison, the participants had to use another in-DBMS analytics stack of their own choice for solving these problems. They agreed to use the stack based on PostgreSQL, the *PyMathProg* Python library for high-level optimization problem modeling, *PL/Python* language extension for in-DBMS Python programming, and the widely used PostgreSQL extension *MADlib*[5] for in-DBMS machine learning. Afterwards, the participants reflected on their experiences.

The study demonstrated that they solved their chosen problems with approx. 1.5-3.5 times less code and approx. 2 times faster SolveDB runtimes when using SolveDB$^+$. They identified a number of strong and weak points of SolveDB$^+$ - see Table 6. They also reflected on the new SolveDB$^+$ features, e.g., "*The SolveDB+ shared model concept is interesting...*", "*I think it [shared models] fits well with the rest of the system, ... can be incredibly useful in specific use cases...*" , "*...it is a great idea to incorporate the opportunity to do simulation models within the dbms... however, when doing this, my experience is that I need a lot of flexibility - and im not sure the compact style of solveDB+ will benefit me there. At least not yet*". In summary, the study confirmed our expectations that SolveDB$^+$ has good usability, explainability, developer productivity, and performance, even for new users. The next subsections dig deeper into these aspects.

5.2 Experimental Setup

To support the claims about SolveDB$^+$ (Section 5.1), we further evaluated SolveDB$^+$ in two typical PA use-cases from the fields of *energy* and *supply chain management*, covering the phases P1-P4 shown in Figure 1. For both use-cases, we implemented two PA technology stacks: 1) a stack consisting of a standard DBMS and relevant state-of-the-art PA tools and 2) a SolveDB$^+$ stack with a number of standard and specialized built-in solvers (used in place of the PA tools). In both configurations, input data is read from the database and the solution is stored back to the database. We compared these two technology stacks by measuring the Effective Lines of Code (eLOC)[24] (relevant since we are comparing high-level languages and eLoc is used in similar comparisons [28, 31]) of the full implementations and their inherent P1-P4 parts. We also compared them in terms of *execution time*, by encompassing database I/O time as well as prediction, model fitting, and optimization problem solving time. Lastly, these use-cases were used to compare SolveDB$^+$ against SolveDB by evaluating novel SolveDB$^+$ features, including *CDTEs*, *shared models*, and the *predictive framework*. In all experiments, we used SolveDB$^+$/SolveDB on top of PostgreSQL 11.2 in the default configuration and native SolveDB solvers for LP/MIP/Blackbox problems [31].

5.3 Energy Planning (UC1)

We evaluated the impact of using SolveDB$^+$ to solve the energy planning problem from the running example, denoted as UC1, using the NIST dataset [4] – containing 8737 hourly aggregates from PV, HVAC, temperature sensors, all from a high precision lab-home. We compared with two different PA technology stacks using either *specialized tools* or *general modeling tools*.

Specialized tools Here, we used standard PostgreSQL, Matlab R2015b, and three powerful specialized libraries, *Statistics and Machine Learning Toolbox*, *System Identification Toolbox*, and *Multi-Parametric Toolbox* (MPT), for *Linear Regression (LR) forecasting*, *state-space (SS) model fitting*, and *dynamical system optimization*, respectively. Specifically, we used a Matlab implementation that uses the following native library functions: `fitlm` to estimate the LR model coefficients, `predict` to produce PV supply forecasts, and `ssest` to fit HVAC state-space model parameters to the given data. The implementation uses the outputs of these functions to define an MPC (model-predictive control) controller with a number of constraints on the system input and state variables and the PV supply amounts used as a reference for minimizing electricity cost. The size of this implementation in eLOC is given in Figure 3(a) as **`Matlab-native`**. As this configuration is the most comprehensive, it is used as a *reference* for this comparison.

General-purpose modeling tools In this configuration, we utilized a standard DBMS, Matlab R2015b, and YALMIP – a Matlab toolbox for rapid prototyping of optimization problems. Like SolveDB$^+$, YALMIP is provided with a variety of solvers for different problem classes. By using both YALMIP and SolveDB$^+$, we modeled *LR model estimation* (P2), *state-space model fitting*

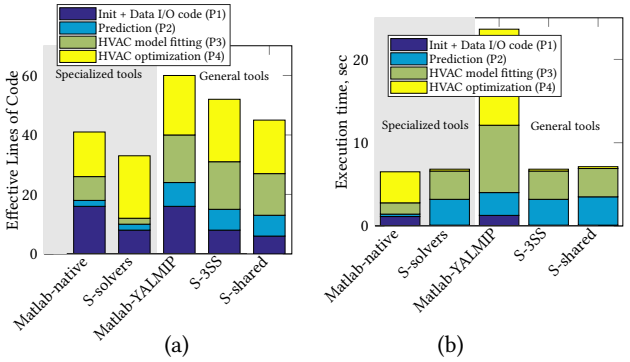

Figure 3: Implementation sizes (a) and run time (b) of UC1

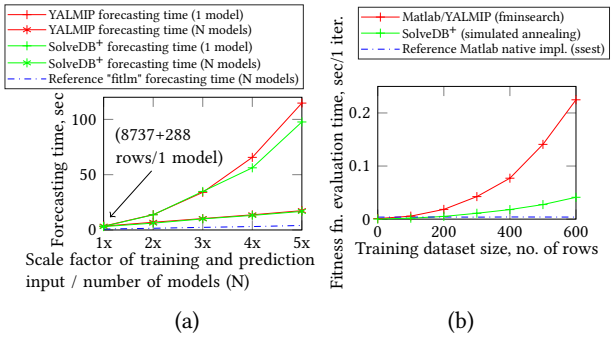

Figure 4: Scalability of prediction (P2) and model fitting (P3) using general-purpose (YALMIP, SolveDB$^+$) and specialized tools (P2: fitlm, P3: ssest)

(P3), and *dynamical system optimization* (P4) problems as explicit LP/nonlinear optimization problems using Matlab/YALMIP programs and SolveDB$^+$ queries, respectively. Specifically, P2 is modeled as an LP optimization problem by minimizing the forecasting error to compute regression model parameters. To solve this problem, SolveDB$^+$ and YALMIP use the Coin-OR CBC solver for the actual computations. Similarly, P3 is specified as a non-linear problem (NLP) of minimizing prediction error of a linear dynamical system using time domain data and *HVAC power levels* and *inside temperatures* as decision variables. To solve this problem, Matlab/YALMIP uses *fminsearch* and SolveDB$^+$ uses *simulated annealing*. These are two distinct NLP solvers that solve the problem in a non-deterministic way. Since they typically give different solutions each time, we only measure average time required for a single solving iteration (fitness function evaluation, Figure 4(b)). Lastly, P4 is modeled as a linear cost minimization problem, where the cost of electricity is minimized under a number of constraints on the HVAC system state and input, and by taking PV supply forecasts into account (based on the LR model). SolveDB$^+$ and YALMIP use CBC to solve this problem. The size of YALMIP implementation in eLOC is given in Figure 3(a) as **`Matlab-YALMIP`**. In SolveDB$^+$, the complete PA workflow, encompassing P2-P4, were implemented in 3 different ways:

-**`S-3SS`** P2-P4 were implemented as three independent **`SOLVESELECT`**s linked using temporary tables (P1).

-**`S-shared`** To be able to reuse the HVAC model parts repeating in P3 and P4, we defined the complete PA problem as a single **`SOLVESELECT`** using a SolveDB$^+$ *shared model*. The model captures indoor temperature dynamics, with P2 and P3 **`SOLVESELECT`** specifications embedded into the model. Note, the size of the model is equally shared by the respective parts in Figure 3(a).

-**`S-solvers`** To relieve the user from the need to specify detailed **`SOLVESELECT`** queries for P2 and P3, we implemented two *composite solvers* which hide respective problem specification details. As these solvers are conceptually similar to the library functions (`Matlab-native`), the overall PA workflow is simplified to a single **`SOLVESELECT`** invoking the composite solvers.

Comparison to specialized tools As seen in Figure 3, the complete PA problem can be specified in just 41 lines of Matlab code and solved in 6.5 secs using specialized tools (`Matlab-native`). Here, around 40% of code and 18% of time is used for initializing libraries and accessing the database, the rest is spent on forming required inputs for, and invoking, the black-box library functions (all considered as P1). As seen for `S-solvers`, this I/O overhead as well as optimization time can be reduced by more than one order of magnitude if all computations are pushed inside the DBMS. This also reduces the PA problem (code) size when SolveDB$^+$ with specialized (composite) solvers are used. As seen in Figure 5 (SolveDB$^+$ vs. MPT), this optimization (P4) performance improvement comes from the reduced model generation time – time spent by MPT to translate the problem to YALMIP, and for YALMIP to aggregate problem constraints and build an optimization (P4) model instance in the binary representation (required by CBC). However, as seen in Figure 3(b), native prediction (P2) and SS model fitting (P3) functions are hard to outperform using general-purpose solvers (`Matlab-native` v.s. `S-solvers`). Figure **??** hint that specialized SolveDB$^+$ solvers for prediction and model fitting are required for larger input datasets. Considering the prediction alone, LR model fitting (P2) using the general-purpose solvers scale *linearly* with respect to independent model count and *exponentially* with respect to training and prediction input size, and therefore might still be useful for some smaller PA cases.

Comparison to general-purpose tools Compared to the native tools (`Matlab-native`), general modeling tools (`Matlab-YALMIP`, `S-3SS` and `S-shared` – all using general-purpose solvers) offer a single language and the full control of how the three PA subproblems P2-P4 are specified. However, explicitly specifying these sub-problems requires up to 45% more code (see Figure 3(a)). Further, computations are up to 3.6 times slower (see Figure 3(b)) and they do not scale (linearly) as in the native case (see Figures 4–5). Comparing YALMIP to SolveDB$^+$, SolveDB$^+$ solves the complete PA problem 3.5 times faster due to significantly reduced data I/O and HVAC optimization time. This can also be seen in Figure 5, which shows that SolveDB$^+$ exhibits up to 2 order of magnitude less data I/O and up to 3 orders of magnitude less model generation time, which is spent translating high-level constraint and objective function specifications into the binary format required by CBC. Both YALMIP and SolveDB$^+$ exhibit somewhat comparable forecasting (P2) and model fitting performance (P3). In the P2 case, YALMIP model generation time is less significant as model constraints can be vectorized (defined without "'for"' loops) and, in the P3 case, just 3 decision variables (a_1, b_1, and b_2) are used. Still, as shown in Figure 4(a), SolveDB$^+$ implementation offers up to 18% lower forecasting time for larger input dataset due to more efficient processing of linear constraints. This difference is less evident when several independent forecasting models need to be estimated using smaller training datasets. Lastly, in addition to these performance benefits, SolveDB$^+$ offers up to 33% smaller implementation sizes as shown in Figure 3(a).

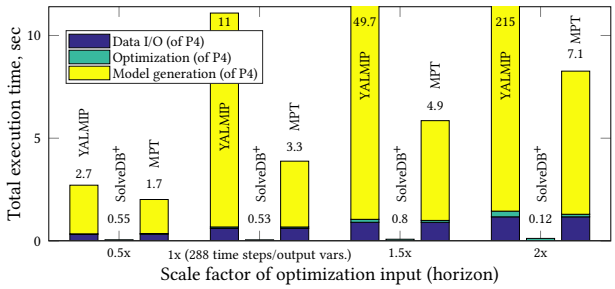

Figure 5: Scalability of HVAC energy optimization (P4)

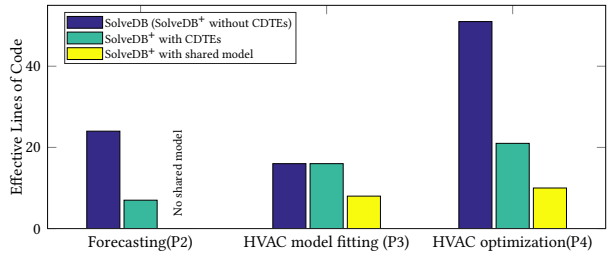

Figure 6: Sizes of SolveDB$^+$ implementations with and without CDTEs and Shared Models

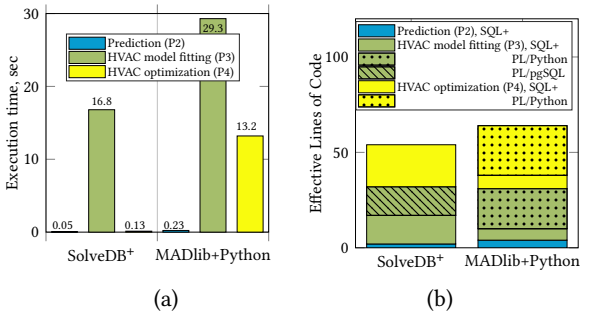

Figure 7: UC1 performance (a) and implementation sizes (b) when using SolveDB$^+$ and existing in-DBMS tools

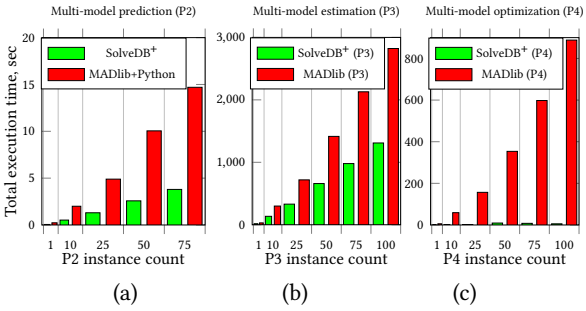

Figure 8: Scalability of In-DBMS UC1 implementations

Comparison to in-DBMS analytics tools Next, we compared SolveDB$^+$ against the in-DBMS analytics stack from the usability study (Section 5.1). We used MADlib's in-DBMS *linear regression* (*linregr_train* UDF) for P2. Since MADlib alone cannot be used to solve the HVAC model fitting and optimization sub-problems (P3-4), we implemented two in-DBMS Python (*PL/Python*) programs for HVAC model fitting (P3) and HVAC operation optimization (P4) by utilizing the *Swarmops* and *PyMathProg* Python libraries, respectively. These libraries offer high-level optimization problem modeling capabilities (required for P3-4) and, under the hood, invoke the low level solvers *Differential Evolution* and *GLPK*, respectively. A SolveDB$^+$ implementation uses three **SOLVESELECT** statements that define the P2-P4 sub-problems and invoke the (same) *linear regression*, *Swarmops*, *GLPK* low-level solvers using SolveDB$^+$'s high-level solvers (incl., *solverlp* and *swarmops* – see Section 3.2 and Section 4.1). The SolveDB$^+$ implementation also uses a *PL/pgSQL* UDF to compute prediction error (being minimized) given (solver-)supplied candidate values of the HVAC model parameters (P3). The goal of this experiment was to compare implementation sizes and runtimes of individual phases (P2-P4) when solving a number of UC1 instances using the same set of low-level solvers (i.e., linear regression, differential evolution, GLPK) running inside a DBMS. Thus, we aimed at comparing the two stacks in terms of *how P2-P4 are specified by the user, how well these (high-level) problem specifications are translated to (low-level) solver inputs*, and *how fast data, solver inputs and outputs are processed by the two in-DBMS stacks.*

As seen in Figure 7(b), *MADlib+Python* required 64 eLOC of mixed SQL and PL/Python code and SolveDB$^+$ required 47 eLOC of (extended) SQL and PL/pgSQL code. While implementation sizes are somewhat comparable, SolveDB$^+$ required very little non-SQL code (15 lines of PL/pgSQL only) to specify the iterative P3 computations. Note, we have also implemented UC1 using pure (extended) SQL (in total 42 lines) with a recursive CTE query for P3. However, this implementation with a recursive query for HVAC simulations might be less intuitive for inexperienced users. In terms of performance, as seen in Figure 7(a), a single instance of UC1 can be solved with SolveDB$^+$ more than twice as fast as with MADlib+Python (19.9 vs 42.7sec). Here, significant gains are observed primarily for P3 (16.8 vs 29.3sec) and P4 (0.13 vs 13.2 sec). For P3, SwarmOPS (in C++) was able to reevaluate the fitness function specified as a SELECT expression from **SOLVESELECT** (that calls a `PL/pgSQL` function) approx. 1.7 time faster that pure Python implementation, where both the solver (SwarmOPS) and the fitness function were implemented in Python. For P4, SolveDB$^+$ offers faster processing of P4 problem symbolic descriptors (*solverlp* vs *PyMathProg*), to be consumed by the same low-level solver (GLPK in C). As seen in Figure 8 (a-c), this gain is more significant when scaling the number of UC1 instances to be solved, i.e., scaling the number of parameters need to be estimated for P3, and predictions and optimization (P2, P4) need to be made for multiple independent HVAC installations. Here, SolveDB$^+$ offered 3.6x faster predictions (P2, Figure 8(a)) since it did not need to create intermediate tables for model parameters and summaries, unlike MADlib; 2.1x faster model parameter estimation, primarily, due to faster evaluation of the fitness function (P3, Figure 8(b)); and 161x faster optimization (P4, Figure 8(c)) primarily due to efficient manipulation of symbolic optimization models and automatic problem partitioning. All in all, SolveDB$^+$ had 2.8x faster execution of the complete PA workflow using less and less complex code, showing its clear advantage over MADlib+Python and confirming the claims about SolveDB$^+$ usability (and performance, see Section 5.1).

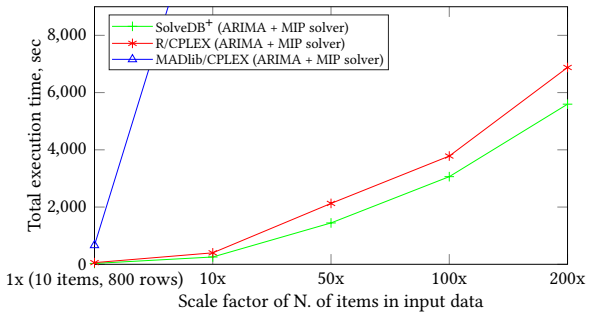

Figure 9: Scalability of combined P1-P4 for UC2

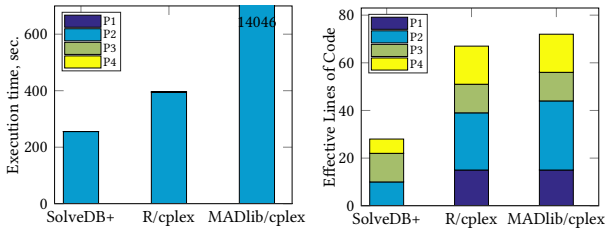

Figure 10: UC2 performance (a) and eLOC (b) comparison

5.4 Supply Chain Management (UC2)

As a second use case (UC2), we considered a common supply chain management scenario. We used the TPC-H dataset [26] containing production supply chain items with the information associated to these items, e.g., orders in the last months, parts needed to assemble the items, size of the parts, price, suppliers, etc. The objective in this use case is to increase revenue by producing in advance the items that will be the most profitable in the next month. The warehouse of the production facility has a limited *volume capacity*, so the decision on which items to produce and store has to be optimized subject to this constraint. This PA workflow requires predicting expected item demand for the next month (P2), modeling *expected profit* for the items by weighting item profit by the probability that the item is ordered in the next month (P3), and solving a variant of the *knapsack problem*, where the warehouse's capacity constraint is respected (P4).

We compared PA stacks with SolveDB$^+$ and both *standalone* and *integrated DBMS analytics* tools. For SolveDB$^+$, we used the predictive framework with a built-in ARIMA solver based on the Statsmodels 0.8.0 package [29] for P2, PL/pgSQL function for P3, and a pre-installed MIP solver from the GNU Linear Programming Kit (GLPK) v4.47 for P4. For standalone tools, we used a configuration with a standard PostgreSQL 9.6.1 (P1, P3), an ARIMA model in R 3.2.3 (P2), and a MIP solver in CPLEX 12.7.1 (P4). For the integrated DBMS analytics tools, we utilized PostgreSQL 9.6.1(P1, P3) with the MADlib [5] extension for in-DBMS machine learning using SQL (P2), and the same MIP solver in CPLEX 12.7.1 (P4). We used 5 different UC2 sizes, scaling the number of items in the dataset. Each item is associated with a time series containing 80 rows of monthly orders.

Figure 10(a) shows the results on the UC2 instance with 100 items. In all implementations, the prediction process accounted almost exclusively for the total execution time, as up to 10000 ARIMA models are trained: 100 per item in R and MADlib, 10 particles with 10 iterations per item in SolveDB$^+$. However, the SolveDB$^+$ implementation was approximately 30% faster than R, and 2 orders of magnitude faster than MADlib, thanks to the efficient use of particle swarm optimization solver for cross validation of the model parameters. Specifically, MADlib does not provide efficient support for cross-validating the forecasting models (ARIMA), with multiple write/read operations accounting for as much as 60% of the total execution time. Figure 10(b) shows the size for the three implementations (implementation size is identical across instances), with SolveDB$^+$ being approximately 50% smaller than the R/MADlib and CPLEX implementations.

The performance results for the different UC2 instances in Figure 9, together with Figure 10(b), show that SolveDB$^+$ allows for a more compact problem definition and execution times that are between 20% and 30% faster than the R configuration, and orders of magnitude faster than the MADlib setup. SolveDB$^+$ outperforms the other two systems thanks to a reduced number of I/O operations and the use of the native local search solvers for hyper-parameters optimization in the model training phase. All in all, UC2 also confirms the end-user claims about SolveDB$^+$ usability (and performance) (Section 5.1).

5.5 SolveDB$^+$ Feature Evaluation (Comparison to SolveDB)

SolveDB$^+$ inherits features and advantages from SolveDB [31]. Specifically, both offer wider applicability and significantly increased tool productivity and usability (order of magnitude less code), while in most cases providing much (up to > 2 orders of magnitude) better performance than systems such as LogicBlox or Tiresias (seeSection 2). We now evaluated the novel SolveDB$^+$ features that distinguish SolveDB$^+$ from SolveDB using the energy and supply chain management use-cases, UC1 and UC2.

Common Decision Table Expressions (CDTEs) As explained in Section 4, CDTEs extend the SOLVESELECT clause like Common Table Expressions (CTEs) extend the simple SELECT in standard SQL. In contrast to CTEs, CDTEs allow annotating some table attributes as *decision columns*, the values of which are evaluated as part of a (much better organized) single SOLVESELECT problem. As seen in Figure 6, CDTEs have a major impact on SolveDB$^+$ usability. Specifying LR model estimation/prediction problems and HVAC optimization problems from the energy planning use-case without CDTEs (SolveDB) requires up to 3 times more **SOLVESELECT** code compared to using CDTEs (SolveDB$^+$). In this case, the HVAC model fitting problem does not benefit from CDTEs, as it uses just a single collection of decision variables, which can be well arranged in a single table. Our experiments also showed that CDTEs do not introduce significant performance overhead to the overall PA workflow.

Shared Optimization Models As explained in Section 4, shared optimization models allow reusing data, objective, and constraint specifications across several optimization problems. UC1 can benefit from such models, by reducing the amount of **SOLVESELECT** code 2 times (Figure 6) for HVAC model fitting and optimization sub-problems alone, and 16% for the complete PA application (see `S-3SS` and `S-shared` in Figure 3(a)), which also includes the shared model specifications. As can be seen in Figure 3(b), shared models do not introduce significant performance overhead to the overall PA workflow.

Predictive Framework As discussed in Section 4, the predictive framework of SolveDB$^+$ offers two ways to integrate new

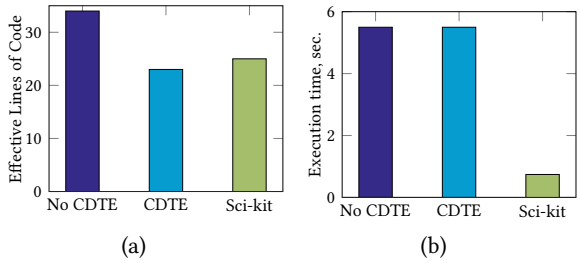

Figure 11: LR code size (a) and execution time (b)

forecasting models. The user can either manually specify forecasting models as SOLVESELECT queries and/or specialized solvers, or install them as "wrappers" over third-party general purpose forecasting libraries. We now compare these two approaches.

For this experiment, we developed the linear regression model as a SOLVESELECT query a) with CDTEs, and b) with no CDTEs wrapped into the respective solvers within the predictive framework. Additionally, we c) installed a general purpose linear regression model from the Sci-kit learn library [3] as a wrapper in SolveDB$^+$. Figure 11(a) shows the implementation size for these three cases. While the size of the Sci-kit implementation is approximately the same as the CDTE implementation, the no CDTE implementation is approximately 30% larger than the other two. Still, the Sci-kit solver implementation is conceptually simpler as it just uses a library function. Furthermore, Figure 11(b) shows that the specialized SolveDB$^+$ implementation is almost 8 times faster than the manual SOLVESELECT implementation (CDTEs do not affect performance), as it combines both in-DBMS execution and a highly specialized machine learning library.

6 CONCLUSION AND FUTURE WORK

This paper presented SolveDB$^+$, the first SQL-based DBMS to provide an extensible and efficient eco-system for all Prescriptive Analytics (PA) phases. SolveDB$^+$ reduces the complexities and inefficiencies of existing PA application stacks, which consist of many specialized, independent, poorly connected systems with different APIs and languages. SolveDB$^+$ acts as a "*swiss-army knife*" system for PA, effectively supporting all 5 phases of PA development: *P1: data management*, *P2: prediction/forecasting*, *P3: system modeling*, *P4: optimization problem solving*, and *P5: solution analysis*. SolveDB$^+$ provides extensibility, allowing developers to add new custom functionalities for specialized PA cases. SolveDB$^+$'s common SQL-based language can express an entire PA workflow in a single SQL-based query. SolveDB$^+$ offers faster PA workflow execution due to its in-DBMS PA algorithms.

Compared to the earlier (SolveDB) tool, SolveDB$^+$ provides a number of novel modeling features, including *common decision table expressions* and *shared optimization models*, enabling a significant size reduction of complex PA problem specifications. It also introduces a new *predictive framework*, which is a generic and extensible in-DBMS platform for the use and development of time series forecasting methods. With all its features, SolveDB$^+$ offers convenient and efficient ways to use and extend the eco-system of forecasting models and optimization problem solvers, thus adapting the system to virtually unlimited PA scenarios.

Our experiments showed that the new SolveDB$^+$ features yield up to 5 times smaller problem specifications (better productivity and explainability) and up to 6 times faster forecasting time, compared to SolveDB. Overall, SolveDB$^+$ offers up to three orders of magnitude better performance for individual PA steps, and up to 3.5 times faster execution times and 3 times smaller implementation sizes for the full PA workflow, compared to state-of-the-art baselines. SolveDB$^+$ scales well in its chosen in-DBMS setting.

Future work will redesign SolveDB$^+$ for distributed Big Data processing and integrate What-If analysis for hypothetical scenarios, and support more data formats, operators on shared models, and further ML models.

REFERENCES

[1] A. Ghoting et al. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*.
[2] A. Raj et al. 2020. From Ad-Hoc Data Analytics to DataOps. In *ICSSP*.
[3] F. Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12 (2011).
[4] H. William et al. 2017. Net Zero Energy Residential Test Facility Instrumented Data; Year 2. (2017). https://doi.org/doi.org/10.18434/T46W2X
[5] J. M. Hellerstein et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *PVLDB* 5, 12 (2012).
[6] M. Aref et al. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*.
[7] M. Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.
[8] M. Brucato et al. 2016. Scalable package queries in relational database systems. *PVLDB* 9, 7 (2016).
[9] M. Hall et al. 2009. The WEKA data mining software: an update. *SIGKDD Explor.* 11, 1 (2009).
[10] M. Jasny et al. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD*.
[11] M. Stonebraker et al. 2013. SciDB: A database management system for applications with complex analytics. *CiS&E* 15, 3 (2013).
[12] M. Schule et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW*.
[13] S. Sanda et al. 2019. In-database Distributed Machine Learning: Demonstration using Teradata SQL Engine. *PVLDB* 12(12) (2019).
[14] T. Kraska et al. 2013. MLbase: A Distributed Machine-learning System.. In *CIDR*.
[15] U. Fischer, F. Rosenthal, and W. Lehner. 2012. F2DB: The Flash-Forward Database System. In *ICDE*.
[16] D. Frazzetto, T. D. Nielsen, T. B. Pedersen, and L. Siksnys. 2020. Prescriptive Analytics: A Survey of Emerging Trends And Technologies. *VLDBJ* 28(4) (2020).
[17] Clyde W. Holsapple, Anita Lee-Post, and Ramakrishnan Pakath. 2014. A unified foundation for business analytics. *DSS* 64, C (2014).
[18] K. Hu, D. Orghian, and C. Hidalgo. [n.d.]. DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. In *HILDA*.
[19] A. Kalinin, U. Cetintemel, and S. Zdonik. 2015. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB* 8, 10 (2015).
[20] James Kennedy. 2011. Particle swarm optimization. In *Encyclopedia of machine learning*. Springer, 760–766.
[21] P. Manolios, V. Papavasileiou, and M. Riedewald. 2014. Ilp modulo data. In *FMCAD*.
[22] MATLAB. 2020. *MATLAB API for Python.* Available at se.mathworks.com/help/matlab/matlab-engine-for-python.html.
[23] A. Meliou and D. Suciu. 2012. Tiresias: the database oracle for how-to queries. In *SIGMOD*.
[24] E. Morozoff. 2010. Using a line of code metric to understand software rework. *IEEE software* 27, 1 (2010).
[25] B. Omidvar-Tehrani, S. Amer-Yahia, E. Simon, and et al. [n.d.]. UserDEV: A Mixed-Initiative System for User Group Analytics. In *ILDA*.
[26] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.
[27] Mark Rittman. 2012. *Oracle Business Intelligence 11g Developers Guide.* McGraw-Hill Osborne Media.
[28] O. Rybnytska, L. Siksnys, T. B. Pedersen, and Bijay Neupane. 2020. pgFMU: Integrating Data Management with Physical System Modelling. In *EDBT*.
[29] S. Seabold and J. Perktold. 2010. Statsmodels: Econometric and statistical modeling with python. In *PiSC*.
[30] L. Siksnys. 2020. *Phd Exercises.* Available at https://www.daisy.aau.dk/wp-content/uploads/2020/12/Advanced-Analytics-Exercises.pdf.
[31] Laurynas Šikšnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *Proc. of SSDBM*. 14.
[32] Z. Tang and J. Maclennan. 2005. *Data mining with SQL Server 2005.* Wiley.
[33] I. Xanthopoulos, I. Tsamardinos, V. Christophides, E. Simon, and A. Salinger. 2020. Putting the Human Back in the AutoML Loop. In *ETLMP*.

CONTENTS

SolveDB$^+$: SQL-Based Prescriptive Analytics

Laurynas Siksnys, Torben Bach Pedersen, Thomas Dyhre Nielsen, Davide Frazzetto
Department of Computer Science, Aalborg University, Denmark
{ siksnys, tbp, tdn }@cs.aau.dk david.frazzetto@gmail.com

ABSTRACT

Today, advanced data analysts make use of both predictive models and optimization problem solving to build data-driven decision making applications, a combination of technologies recently termed *Prescriptive Analytics* (PA). Current PA applications typically have multiple layers of poorly integrated components: a relational DBMS for data storage/management, ML tools for prediction, and specialized software packages for problem modeling and optimization problem solving. This complex stack leads to inefficient, labor-intensive, and error-prone PA workflows, blocking wider adoption of PA. In this paper, we present SolveDB$^+$ – an RDBMS for PA applications which supports all PA steps with *modeling*, *predictive*, and *optimization* functionalities, and integrates these in a common SQL-based framework. Major SolveDB$^+$ novelties are 1) a powerful SQL-based approach for PA problem specification and solving, 2) an extensible in-DBMS infrastructure for prediction and optimization solvers, and 3) in-DBMS modeling and management of PA models. SolveDB$^+$ significantly improves both PA developer productivity and performance.

1 INTRODUCTION

As the next step after Predictive Analytics, *Prescriptive Analytics* (PA) has recently emerged as a new frontier in analytics, combining data management, predictive analytics and ML, and operations research [17]. PA provide a specific course of action for questions such as "How should we maximize our sales in Europe?" PA systems are still in their infancy, typically glued together in an ad-hoc system with separate analytics and optimization tools on top of an RDBMS. There are no integrated PA platforms that combine *data management*, *predictive*, and *optimization* functionalities using a single language, e.g., the frequently used in-DBMS analytics engines only support the first two.

As a running PA example, we consider renewable energy optimization. In a building, PV panels produce intermittent, varying electricity, to run its Heating, Ventilating, and Air Conditioning (HVAC) system. We want to reduce energy costs by using more PV electricity, which requires aligning HVAC operation to PV supply ahead of time, taking forecasted prices and user comfort into account. Table 1 shows a dataset for this case. Input data is a multivariate time series of outdoor (`OutTemp`)/indoor (`inTemp`) temperatures, HVAC consumption (`hLoad`), and PV production (`pvSupply`) per hour. Rows 07:00 - 11:00 are historical data from sensors. Rows 12:00 - 16:00 define future states: `outTemp` contains forecasted outside temperatures; the unknown values of `inTemp`, `hLoad` and `pvSupply` in 12:00 - 16:00 represent decision variables for which PA should compute values by aligning `hLoad` with `pvSupply` at the next 5 hours such that `inTemp` remains within the 20–24°C comfort range and HVAC power limits (0–17kW) are respected. The workflow below exemplifies the 5 overall phases of PA seen in Figure 1.

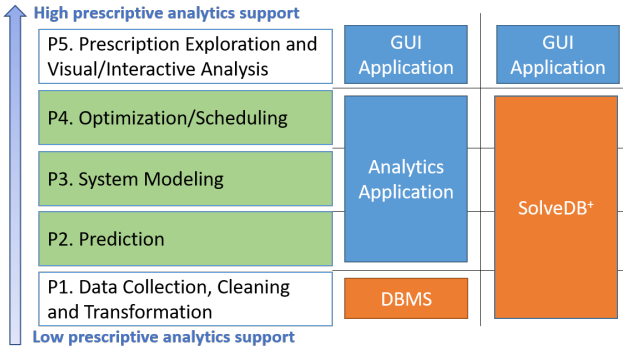

Figure 1: The 5 PA phases and the used software stacks

P1: Collect, clean, validate, and transform the input data.
P2: Predict PV supply given `pvSupply` and `outTemp`.
P3: Model `inTemp` dynamics in relation to `inTemp` and `hLoad`, which requires tuning parameter values specific to this building.
P4: Find optimal `hLoad` values by minimizing electricity cost subject to initial conditions, `pvSupply`, `hLoad`, and comfort constraints, applied over the calibrated model (P3).
P5: Analyze, visualize, and validate the results.

Traditionally, this PA workflow requires a complex software stack with different tools for data management, forecasting, system modeling, and optimization, leading to several *problems*: **Steep Learning Curve:** Different tools have different usage and modeling methodologies, making the learning curve for building PA applications much steeper, which, in turn, leads to more *errors* and *misuse*. **Poor Developer Productivity:** The tools are based on different programming/query languages and have to be glued together in ad-hoc ways to realize PA workflows, leading to *poor developer productivity*, *tool incompatibilities*, and even more *errors* [2]. **Bad performance:** Large amounts of data have to be shipped back and forth between the many tools, leading to high *I/O and memory costs* and *long runtimes* (see Sec. 5). To remedy these problems,these *research challenges* (RCs) must be met:
RC1: Provide a concise yet powerful SQL-based syntax for PA decision problems, supporting efficient query processing.

Table 1: Input dataset for campus energy management.

time	outTemp	inTemp	hLoad	pvSupply
2017/07/02 07:00	05	21	100	0
2017/07/02 08:00	06	20.5	250	0
2017/07/02 09:00	06	21	150	200
2017/07/02 10:00	07	23	120	254
2017/07/02 11:00	08	23	80	320
2017/07/02 12:00	09	?	?	?
2017/07/02 13:00	11	?	?	?
2017/07/02 14:00	12	?	?	?
2017/07/02 15:00	11	?	?	?
2017/07/02 16:00	11	?	?	?

RC2: Provide a concise yet powerful way to share optimization models across sub-problems of the overall PA problem.
RC3: Provide a powerful, easy-to-use, and extensible way of transparently integrating external prediction functionality into PA workflows.
RC4: Seamlessly integrate RC1–RC3 in a SQL-based system.

To meet these challenges, we present SolveDB$^+$. The fact that most PA systems use an RDBMS for data storage [16, 17] combined with the huge popularity of in-DBMS analytics (see Sec. 2), motivates us to propose the first SQL-based in-DBMS platform for PA applications, with these features (www.daisy.aau.dk/solvedb):

Supporting all PA phases: SolveDB$^+$ integrates data management, prediction, system modeling, and optimization in a single tool, yielding better PA productivity. **Extensibility:** SolveDB$^+$ allows developers to add new functionalities for custom PA applications. **Unified SQL-Based PA language:** SolveDB$^+$ extends SQL with new declarative constructs for unified PA problem modeling and analytical functionalities. An entire PA workflow, including forecasting, simulation, and optimization models, can be expressed in a single extended SQL query. **High performance:** The built-in PA algorithms (and user extensions) run in-DBMS, yielding more efficient execution and data exchange. Our experiments show that SolveDB$^+$ yields up to three orders of magnitude better performance for individual PA steps, and up to 3.5 times faster execution and 3 times smaller implementations for complete PA workflows, compared to state-of-the-art baselines, thus combining performance with usability/productivity.

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 describes SolveDB$^+$'s prediction framework. Section 4 presents its new PA problem modeling features. Section 5 provides the experimental evaluation. Finally, Section 6 concludes and points out future work.

2 RELATED WORK

A recent extensive survey [16] identifies major emerging trends, remaining challenges, and available technology in the field of PA. In the classification used in this survey, SolveDB$^+$ falls in the category of analytical DBMSes, where analytical functionality is integrated directly within the DBMS back-end. Efforts within this category can be classified into *prediction* DBMSes, for forecasting and probabilistic analysis, and *optimization* DBMSes, for optimization problem solving. Table 2 summarizes and compares essential relevant systems in these sub-categories. The systems are compared in terms of: 1. *What primary language is used for data management* (Data QL); 2. *What primary language is used to specify analytics (incl., prediction and optimization) tasks* (Anl. QL); 3. *Does the system offer native support for predictions?* (Pred); 4. *Does the system offer native support for physical system models and estimating their parameters?* (Est), 5. *Does the system support optimization problem solving?* (Opt); 6. *Does the system support optimization (sub-)models that can be stored natively and manipulated as first-class citizens in the database, and re-used to forms more complex models?* (Mod). We now review these systems.

In-DBMS analytics is a major trend. Among prediction DBMSes, forecasting and in-database ML is supported by the major commercial DBMSes, Oracle [27], SQL Server [32], and TeraData [13]. Recently, HyPEr [12] and DB4ML [10] provide in-DBMS ML for main memory DBMSes. These systems provide efficient in-DBMS forecasting/ML functions, but lack automatic forecasting model selection, parameter estimation, optimization problem solving, and model management, unlike SolveDB$^+$. The

Table 2: Comparison between relevant tools and SolveDB$^+$

System	Data QL	Anl. QL	Pred	Est	Opt	Mod
Oracle	SQL	SQL	✓	✗	✗	✗
SQL Server	SQL	SQL	✓	✗	✗	✗
TeraData[13]	SQL	SQL	✓	✗	✗	✗
DB4ML[10]	SQL	SQL	✓	✗	✗	✗
HyPer[12]	SQL	SQL	✓	✗	✗	✗
MADlib[5]	SQL	SQL+UDF	✓	✗	✗	✗
F^{2}DB[15]	SQL	ext.SQL	✓	✗	✗	✗
SystemML[1]	R-like	R-like	✓	✗	✗	✗
MLbase[14]	R-like	R-like	✓	✗	✗	✗
SciDB [11]	SQL-like	SQL-like	✓	✗	✗	✗
pgFMU[28]	SQL	SQL+UDF	✓	✓	✗	✗
Searchlight[19]	SQL-like	SQL-like	✗	✗	✗	✗
PaQL[8]	ext.SQL	ext.SQL	✗	✗	✗	✗
InezDB[21]	ext.OCaml	ext.OCaml	✗	✗	✓	✗
Tiresias[23]	SQL	ext.Datalog	✗	✗	✓	✗
LogicBlox[6]	LogiQL	LogiQL	✓	✗	✓	✗
SolveDB[31]	SQL	ext.SQL	✗	✗	✓	✗
SolveDB$^+$	SQL	ext.SQL	✓	✓	✓	✓

open source alternative MADlib [5] extends PostgreSQL with UDFs specialized for ML tasks like clustering, classification, and forecasting. Similar to MADlib, pgFMU [28] offers PostgreSQL UDFs for in-DBMS simulation and parameter estimation of *Functional Mock-up Units* (FMUs). These are interoperable simulation models that can define dynamic behaviour of complex physical systems. While FMUs are often used for predictions (P2, see Figure 1), pgFMU does not support including FMUs into user-defined optimization problems (P4). In comparison, SolveDB$^+$ supports (less detailed) so-called *grey-box* models that can be both simulated and optimized in the same environment. Among stand-alone DBMSes, F^{2}DB [15] focuses on time series forecasting in an SQL-based environment. While F^{2}DB specializes in, and is highly optimized for, time series forecasting tasks and employing specific model reuse and maintenance techniques, it does not support the development and integration of user-defined "do-it-yourself" models and generic library models, unlike SolveDB$^+$. In the Big Data context, systems such as SystemML [1], MLBase [14], and SciDB [11] integrate general-purpose declarative machine learning tools that offer scalable distributed computations. In the context of PA, *all* systems (except pgFMU) in this category *only offer support for the predictive analytics phase (P2).*

The optimization DBMSes have focused on advanced what-if scenarios, in-DBMS optimization problem solving, and search under advanced forms of constraints. Systems such as Searchlight [19] and PaQL [8] exploit powerful constraint solvers when processing advanced data search queries. InezDB [21] proposes a formal logic for the symbolic manipulation of optimization models inside a DBMS. Tiresias [23] and LogicBlox [6] provide users a Datalog-based language for what-if scenario analysis. Being the predecessor of SolveDB$^+$, SolveDB [31] is an extension of PostgreSQL for in-DBMS optimization problem solving and solver integration. SolveDB$^+$ extends SolveDB in the directions covering the highlighted PA phases in Figure 1. These new features in SolveDB$^+$, together with their impact (to be observed in Section 5), are highlighted in Table 3. These correspond 1-1 to the research challenges RC1–RC3 mentioned in Section 1, while the integrated SolveDB$^+$ system corresponds to RC4.

Table 3: New features of SolveDB$^+$ compared to SolveDB.

Feature	Description	Impact
In-DBMS Predictive Framework	Specialized forecasting models that are easy to install, (auto)select, and use.	Forecasting easier to use and up to 6 times faster.
Shared Optimization Models	Allow defining reusable optimization (sub-)models stored in-database with their objective functions, constraints, and data specs.	Up to: 2X less code for P3-P4, 16% less code for P1-P4, similar performance.
New Language Features	Asterisk notation, common decision table expressions, model inlining allow specifying PA problems more concisely/efficiently.	Up to 5X less code for P2-P4, similar performance.

In summary, Table 2 shows that while predictive and optimization DBMSes offer some level of in-DBMS analytics support, they do it only for *some* PA phases and do not offer "SQL for all PA phases" like SolveDB$^+$. In comparison, SolveDB$^+$ is the only system to combine and unify predictions and optimization problem solving within a single SQL-based system.

Explainability, also called interpretability, of ML pipelines has received much attention in recent years. It has been considered both for specific categories of ML pipelines, e.g., user group analytics [25] or data exploration [18], and more generally in a survey of AutoML pipelines [33]. In comparison, SolveDB$^+$ focuses on another category, PA pipelines, and supports explainability in PA phases P1-P4. For P1, we do not claim any new contributions, but simply offer the time-honored explainability of SQL. For (external) Prediction methods (P2), we inherit their existing explainability and add to it by declaratively specifying input and output in the solver specs. For System Modeling (P3) and for overall integration of the phases, our high-level declarative SQL-based syntax and shared models allow a higher level of abstraction which is more compact and explainable than a traditional imperative-style ML pipeline. For Optimization (P4), the declarative specifications of objective functions are immediately explainable. Section 5 provides more details.

Another key aspect of ML pipelines is their connectivity to other components/frameworks [33]. As for the "inbound" connectivity, external components are integrated for use in SolveDB$^+$ in two ways. Like other in-DBMS analytics tools (see Tab. 2), SolveDB$^+$ uses UDFs to wrap external functions for direct use in SQL queries, Specifically to SolveDB$^+$, the solver concept is used to integrate external prediction components in a seamless way (see Sec. 3). As for the "outbound" connectivity, SolveDB$^+$ can be integrated in larger pipelines just like other SQL-based in-DBMS analytics tools.

3 PREDICTION

The first phase in Figure 1 *P1: Data Collection, Cleaning, and Transformation* is well supported by the SQL queries, built-in functions, and UDFs of traditional RDBMSes [16], including SolveDB$^+$. Since PA applications need to look ahead in time, effectively supporting the next phase *P2: Prediction* is a key research challenge (RC3). This section describes how we meet RC3. While SolveDB$^+$ can accommodate different models and algorithms for prediction (using both built-in and external tools), it offers dedicated support

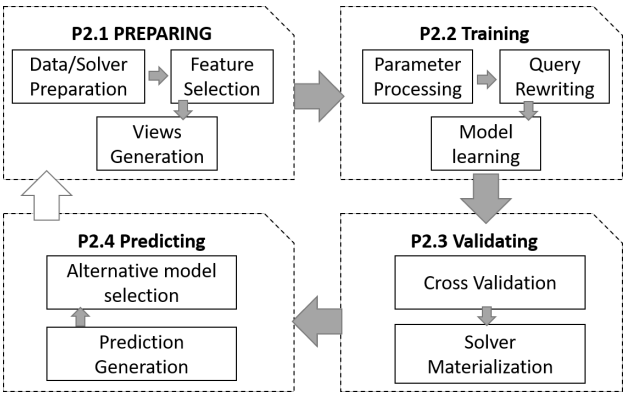

Figure 2: Prediction process + SolveDB$^+$ implementation.

for *time series forecasting* methods. These are widely used for data-driven prediction based on current and historical data.

3.1 Time series forecasting in SolveDB$^+$

Following the energy planning example, the input to the prediction phase is the time series shown in Table 1. The objective is to predict the PV supply for the next 5 hours, by filling in the missing `pvSupply` values in Table 1. This is accomplished by a specific time series forecasting method (e.g., regression) involving a number of steps, as shown in Figure 2: *preparing* – extracting and formatting the data to fit forecasting models, *training* – fitting the forecasting models on the dataset, *validating* – validating the models using cross validation or other evaluation procedures, and *predicting* – forecasting new values.

To support the user in using these methods, SolveDB$^+$ provides its in-DBMS *Predictive Framework*, which (1) exposes various time-series forecasting methods through SQL ("transparently integrating" in RC3), (2) hides the complexity ("easy-to-use" in RC3) of choosing and using these methods (the *preparation*, *training*, *validation*, and *prediction* steps), and (3) offers different extensibility options when a new forecasting method needs to be integrated ("extensible" in RC3). For example, the prediction problem above can be solved in two different ways, using:

Specific forecasting method The following example query invokes the specific forecasting method ARIMA:

```
1 SOLVESELECT t(pvSupply) AS (SELECT * FROM input)
2 USING arima_solver(predictions := 5, time_window := 5,
3                    features := outTemp)
```

To expose the method, SolveDB$^+$ uses the specialized **`SOLVESELECT`** statement (extending the one from SolveDB [31]), to be described in detail in Section 4. It invokes a SolveDB$^+$-native *solver* (*arima_solver*) to derive a so-called *output relation* (a database table) from a so-called *input relation* (**`SELECT * FROM`** `input`) by adding/deleting rows or filling in values in the specified *decision columns*. In this example, the decision column is `pvSupply`, the values of which are requested to be populated by *arima_solver*. The output relation has the same schema as the input relation, but with the `pvSupply` column filled as shown in Table 4. To derive the output relation from the input relation, `arima_solver` additionally takes solver parameters: the number of predictions (*predictions := 5*), the number of time steps to use for training (*time_window:=5*), and the column (*features:=outTemp*) to use as a feature attribute. The solver then performs the steps of *preparation*, *training*, *validation*, and *prediction* (see Figure 2) using the ARIMA model

Table 4: Output of the *Prediction* phase for the example.

time	outTemp	inTemp	hLoad	pvSupply
2017/07/02 07:00	05	21	100	0
2017/07/02 08:00	06	20.5	250	0
2017/07/02 09:00	06	21	150	200
2017/07/02 10:00	07	23	120	254
2017/07/02 11:00	08	23	80	320
2017/07/02 12:00	09	?	?	**200**
2017/07/02 13:00	11	?	?	**220**
2017/07/02 14:00	12	?	?	**260**
2017/07/02 15:00	11	?	?	**140**
2017/07/02 16:00	11	?	?	**0**

trained on data from the input relation with the given parameters. Thus, **SOLVESELECT** allows the user to invoke any specific predictive solver installed in SolveDB$^+$, including solvers for Linear Regression, Logistic Regression, ARIMA, or the powerful *Predictive Advisor* described next. The carefully designed use of the solver ensures the transparency mentioned in RC3.

Predictive Advisor Users can get automated model selection and configuration by using the *Predictive Advisor*, exposed as `predictive_solver`. This solver hides *model selection*, *feature selection*, and *parameter fitting* from the user, and transparently performs *preparation*, *training*, *validation*, and *prediction* and fills in the missing values in the input relation, thus ensuring "easy-of-use" in RC3. Now, the prediction query above can be rewritten as the following simpler query:

```
1 SOLVESELECT t(pvSupply) AS (SELECT * FROM input)
2 USING predictive_solver()
```

The extensibility offered by SolveDB$^+$ also allows for alternative automated predictive frameworks to be integrated as part of the SolveDB$^+$ predictive advisor ("extensible" in RC3).

3.2 Steps of the Predictive Framework

In SolveDB$^+$, the underlying steps of *preparation*, *training*, *validation*, and *prediction* are standardized and their common routines are shared among different forecasting methods, (ensuring "easy-of-use" in RC3.

P2.1 Preparing When the predictive solver (e.g., `arima_solver`) is invoked, the input relation is first analyzed. The framework extracts *decision* (i.e., to be populated with values) and *feature* (to be used as features) columns specified by the user. After recognizing the types of the input columns, it selects candidate solvers from the pool of predictive solvers by comparing the set of decision and features columns to those supported by the solvers. The framework logically partitions the input relation into the training, test, and validation segments by matching the schema for each candidate solver. The selected solver(s) are then used for the training step.

P2.2 Training Next, the model-specific parameters of the candidate solvers are tuned on the training segment of the input relation. The predictive framework automatically generates a **SOLVESELECT** query that specifies an optimization problem with model parameters as decision variables to optimize. This optimization problem is solved by utilizing the solving capabilities of SolveDB$^+$ (Section 4). For example, the ARIMA solver is installed with the standard ARIMA parameters `ar`, `i`, and `ma`, each associated to the domain $[0, 5]$. Therefore, `predictive_solver` described earlier automatically and transparently invokes the following parameter estimation query:

```
1 SOLVESELECT p(ar, i, ma) AS
2   (SELECT NULL::int AS ar,NULL::int AS i,NULL::int AS ma)
3 MINIMIZE(SELECT arima_rmse(
4          ar:=SELECT ar FROM p,
5          i := SELECT i FROM p,
6          ma := SELECT ma FROM p))
7 SUBJECTTO (
8          SELECT 0 <= ar <= 5, 0 <= i <= 5, 0 <= ma <= 5
9          FROM p)
10 USING swarmops.pso()
```

The above **SOLVESELECT** query specifies a global black-box optimization problem, where the values of the parameters `ar`, `i`, and `ma` are found by minimizing the RMSE between the training set and the ARIMA predictions, computed by the function `arima_rmse` in the **MINIMIZE** clause (line 3). The **SUBJECTTO** clause specifies the range in which the parameters can vary. The optimization solver `swarmops` uses a built-in particle swarm optimization method [20] to iteratively attempt to improve a candidate solution with regards to RMSE.

P2.3 Validating Next, the candidate predictive solvers are compared using cross validation. The solver/model leading to the lowest error is selected. As a side effect, the calibrated model instances are stored in a database as user-defined type (UDT) entities for fast reuse of the solver result later.

P2.4 Predicting Finally, predictions are generated by the selected best candidate solver and returned to the user in the form of an output relation of **SOLVESELECT** (Table 4). As **SOLVESELECT** expresses a view over the input relation (Table 1), no user tables are modified in the database.

3.3 Developer Interface

SolveDB$^+$ addresses the "extensible" in RC3 by providing the user with a developer interface to install new in-DBMS predictive solvers. There exists two categories of solvers: *black box* and *white box*. *Black box* solvers are expected to manually handle the steps of data preparation, feature selection, cross-validation, etc., thus *overriding* the predictive framework functionalities. In contrast, *white box* solvers expose the model specifics (e.g., model parameters, their types, etc.) as well as model training and prediction logic to the predictive framework. This way, the solvers may use the functionalities (e.g., **SOLVESELECT**) provided by SolveDB$^+$ for preparing, training, and validating. Such solvers use the solver extensibility capabilities already present in SolveDB [31]. This allows the developers to easily expand the system by taking advantage of existing SolveDB$^+$ solvers/functionality and integrating new prediction models from existing frameworks, e.g., Scikit-Learn [3], Weka [9], MATLAB [22], Statsmodels [29], and TensorFlow[7].

As we will show in Section 5, SolveDB$^+$ is able to offer reduced PA application development efforts and improved overall performance after the integration of desired solvers, yielding up to 5 times more compact problem specifications and up to 6 times reduced forecasting time, compared to SolveDB and commonly used predictive frameworks.

4 OPTIMIZATIONS AND SYSTEM MODELING

Optimization problem solving is essential in 3 of the 5 PA phases (P2, P3, P4), and it therefore plays an essential role in SolveDB$^+$. To deal with optimization problems, SolveDB$^+$ borrows a number of solvers from SolveDB for the different classes of optimization problems, including *linear programming* (LP), *mixed-integer programming* (MIP), and blackbox global optimization (GO), some of

Table 5: LR problem variable layout and a new c_mask column introduced during the CDTE rewrite

id	*pOTemp*	*pMonth*	*pEps*	*error*	**c_mask**
1	*pOTemp*	*pMonth*	*pEps*	e_1	B'11'
2				e_2	B'01'
...				...	B'01'
M				e_M	B'01'

which were already demonstrated in Section 3. To address RC1, SolveDB$^+$ further extends the query syntax used for accessing these solvers. We now elaborate on these new language features.

4.1 Model Specification Syntax

SolveDB$^+$ uses the following syntax to interact with various (e.g., LP/MIP) solvers registered in the active database:

```
1 {SOLVESELECT | SOLVEMODEL}
2         [alias[(col_name[,...])] AS](select_stmt)
3 [INLINE [alias                   AS](select_stmt) [,...]]
4 [WITH   [alias[(col_name[,...])] AS](select_stmt) [,...]]
5 [MINIMIZE (select_stmt) [MAXIMIZE (select_stmt)] |
6  MAXIMIZE (select_stmt) [MINIMIZE (select_stmt)]]
7 [SUBJECTTO [alias AS] (select_stmt) [,...]]
8 [USING solver_name[.method_name][(param[:=expr][,...])]]
```

As shown earlier, the user can use **SOLVESELECT** to define a *model* and pass it to SolveDB$^+$-compliant solver `solver_name` for evaluation using an optionally specified solving method, `method_name`, all defined as follows.

A *problem model* m is defined as a 4-tuple (D, R, s, m). D is the specification of *data and decision variable columns* (lines 2,4). R is the specification of rules that define how the values of the decision variable columns should be instantiated (lines 5-7). s is the name of the solver (`solver_name`) that should evaluate the rules R on the given D using some method m (`method_name`, line 8). Both D and R define two separate sets of specially annotated database relations. Specifically, $D = (D_1^{a_1}, D^{a_2}, \ldots, D_N^{a_N})$ where, $\forall i \in 1 : N, D_i^{a_i} = (c_1, \ldots, c_k, \overline{c_1}, \ldots, \overline{c_l})$ is a SELECT statement (`select_stmt`) defining a database relation with the alias a_i (`alias`) assigned and defined by k *data columns* $c_1, \ldots, c_k$ and l so-called *decision columns* $\overline{c_1}, \ldots, \overline{c_l}$ (`col_name`). Decision columns denote that their rows are decision variables, the values of which should be computed by s. Here, $D_1^{a_1}$ (line 2) is denoted as *input relation*. In a similar way, $R = (R_1^{min}, R_2^{max}, R_3^{a_3}, \ldots, R_M^{a_M})$ is a set of relations that contain s-specific representations of rules defining how decision column values in D should be computed. For convenience, the aliases of R_1^{min} and R_2^{max} are fixed and they are specified in the **MINIMIZE** and **MAXIMIZE** clauses, respectively (line 5-6). The remaining $R_3^{a_3}$ to $R_M^{a_M}$ are specified in the **SUBJECTTO** block along with their respective aliases (line 7). This provides powerful yet concise model specs for RC1.

A *solver* in SolveDB is a user-defined function (UDF) capable of producing (a query for) a so-called *output relation* O in the schema of the *input relation* $D_1^{a_1}$ from a given problem model instance (D, R, s, m) and additionally supplied solver parameters `param` (line 8). SolveDB$^+$ assumes the following standard scoping rules within SOLVESELECT. Each $d_i^{a_i} \in D$ may access a relation $d_j^{a_j} \in D$ using the alias a_j if $j < i$, i.e., $\forall d_i^{a_i} \in D : scope(d_i^{a_i}) = \{(a_j \mapsto d_j^{a_j} | d_j^{a_j} \in D, j < i\}$. Each $r_i^{a_i} \in R$ may access all data and decision variable tables, i.e., $\forall r_i^{a_i} \in R : scope(r_i^{a_i}) = \{(a \mapsto d^a | d^a \in D\}$.

For example, consider a predictive solver (for P2) based on linear regression (LR). In SolveDB$^+$, LR model parameter estimation is specified using the following **SOLVESELECT**:

```
1 SOLVESELECT p(pOTemp, pMonth, pEps) AS (SELECT * FROM pars)
2 WITH e(error) AS (SELECT *, NULL::float8 AS error
3                   FROM input)
4 MINIMIZE (SELECT sum(error) FROM e)
5 SUBJECTTO(SELECT -1*error <=
6   (pOTemp*outTemp + pMonth*month(time) +
7   pEps - pvSupply) <= error FROM e, p)
8 USING solverlp.cbc()
```

Here, lines 1-3 specify model *data and decision columns*. Lines 4-7 specify *rules* that define an objective function and constraints that involve decision variables from the tables p and e. Finally, line 8 specifies *solverlp* and *cbc* as a SolveDB$^+$-compatible solver and a solving method, respectively.

This general **SOLVESELECT** syntax based on standard SQL SELECTs allows exposing different kinds of models and solvers to user queries in a powerful yet concise way (RC1). Compared to SolveDB, SolveDB$^+$ uses a number of novel modeling features unavailable in SolveDB. These are outlined in the remainder of this section.

4.2 Asterisk notation

To support RC1's need for concise and powerful syntax , SolveDB$^+$ proposes the asterisk (*) notation for decision variable column specification (`col_name`). Like SELECT * in the standard SQL, this allows declaring all table columns as decision variables, thus offering more compact problem specifications. Using asterisks, Line 1 in the above optimization problem can be concisely specified as **SOLVESELECT** `p(*)` **AS** (**SELECT** * **FROM** `pars`).

4.3 Common Decision Table Expressions

In SolveDB, the **WITH** clause within **SOLVESELECT** is not supported. Consequently, decision columns (variables) are only allowed in a single (input) relation $D_1^{a_1}$ (i.e., $N = 1$). Therefore, objective and constraint (SELECT) expressions in the **MINIMIZE**/ **MAXIMIZE** and **SUBJECTTO** blocks may become unnecessarily large and complex. Consider the LR model fitting example. This problem uses 2 collections of decision variables: *pOTemp*, *pMonth*, *pEps* as model parameters and e_1, e_2, ..., e_M ($M >> 3$) as prediction errors. One of the most convenient ways to arrange these variables in a single input relation in SolveDB is depicted in Table 5. Here, *pOTemp*, *pMonth*, *pEps* are contained within a single row and e_1, ..., e_M contained within a single column, with many "empty cells" representing *unbound* decision variables. When not referenced within **MINIMIZE**/**MAXIMIZE** and **SUBJECTTO** expressions, such unbound variables are automatically excluded from computations by SolveDB$^+$. Still, referencing *pOTemp*, *pMonth*, *pEps* in the objective and constraint expressions is quite cumbersome - the user is required to supply the predicate **WHERE** `id=1` in all relevant **MINIMIZE**/ **MAXIMIZE**, and **SUBJECTTO** expressions. This makes problem specifications complex and less readable, especially when more than two variable collections are modeled.

Again meeting RC1's need for concise and powerful syntax, SolveDB$^+$ proposes to extend the SOLVESELECT clause with so-called *Common Decision Table Expressions* (CDTEs). As an extension of Common Table Expressions (CTEs, i.e. WITH queries), these allow specifying additional temporary relations, $D_2^{a_2}$, ..., $D_N^{a_N}$, with or without decision columns, where each relation $D_i^{a_i}$ can be accessed from SELECTs of $D_j^{a_j}$, $j > i$, and in the

MINIMIZE/**MAXIMIZE** and **SUBJECTTO** blocks ($R_1^{min}, \ldots, R_M^{ctr_M}$) using the alias a_i. All decision variables of $D_1^{a_1}, \ldots, D_N^{a_N}$ are solved together in a single optimization problem. Note, when the list of the decision columns is empty ($|\{\overline{c} \in D_i^{a_i}\}| = 0$), the CDTE has the semantics of the standard CTE. As demonstrated earlier, CDTEs in SolveDB$^+$ allow conveniently modeling two or more collections of decision variables, unlike SolveDB.

Efficient CDTE query evaluation ("efficient query processing" in RC1): SolveDB$^+$ efficiently evaluates SOLVESELECT queries with CDTEs in two different ways. SolveDB$^+$ either rewrites the CDTEs to a single input relation and standard CTEs, or passes them to a solver for specialized processing. The first approach is preferred, as it is transparent and applicable to all registered SolveDB$^+$ solvers. Here, SolveDB$^+$ first generates a new input relation ($D_1'^{a_1}$) by joining all CDTEs with decision variables and adding a special bit string attribute `c_mask` (see Table 5) to denote CDTEs relevant to specific rows. Then, SolveDB$^+$ generates and processes a new SOLVESELECT *without* decision variables in CDTEs, by using different projections over the new input relation:

```
1 SOLVESELECT l(pOTemp,pMonth,pEps,error) AS
2             (SELECT * FROM input)
3 WITH p AS (SELECT pOTemp, pMonth, pEps FROM l
4            WHERE (c_mask & b'10') <> b'00'),
5      e AS (SELECT error FROM l
6            WHERE (c_mask & b'01') <> b'00')
7 MINIMIZE(SELECT sum(error) FROM e) ...
```

This syntactical extension does not increase the expressive power of SOLVESELECT as the WITH sub-expressions can always be combined into a joint input relation. Instead, CDTEs allow a more intuitive and comcise organization of decision variables in a SOLVESELECT query ("powerful yet concise" in RC1), which is particularly useful when dealing with many auxiliary variables in complex PA cases.

4.4 Shared Models and Model Management

PA applications often build (optimization) models by combining several existing models, e.g. for P3 in our use-case we want to use a generic linear time-invariant *state-space model* (LTI) for capturing temperature dynamics of the HVAC-equipped campus building, and then apply this model in two optimization problems – *LTI model parameter estimation* and *electricity cost optimization* – P3 and P4 in Figure 1. For the first problem, we want to use our input data to estimate the parameters a_1, b_1, and b_2 of the following discrete LTI model for this specific building:

$$x[n+1] = \begin{bmatrix} a_1 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} b_1, b_2 \end{bmatrix} \mathbf{u}[n]$$
$$y[n] = \begin{bmatrix} 1 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 0, 0 \end{bmatrix} \mathbf{u}[n]$$

Here, $\mathbf{x}$ is the system 1×1 *state vector* denoting the *inside temperature* of the building; $\mathbf{u}$ is the system 2×1 *input vector* denoting *outside temperature* and applied *HVAC load*, and $\mathbf{y}$ is the 1×1 *output vector* which just "feeds forward" the inside temperature.

In the second problem, we want to use this LTI model with instantiated parameters a_1, b_1, and b_2 inside the cost optimization problem with additionally specified constraints on state variables (inside temperature bounds) and input variables (HVAC power bounds). Obviously, these two problems share the common specification of the generic LTI model (i.e., equations above). However, the LTI model constraints have to be redefined in each of the problems when using SolveDB, as there is no way to reuse them.

Algorithm 1: Problem model instantiation

Input: m - a generic model; Δm - instantiation model
Output: m' - an instantiated model

1 $D \leftarrow \{d^{alias} \in m.D | alias \notin \{alias | d^{alias} \in \Delta m.D\}\} \cup \Delta m.D$
2 $R \leftarrow \{r^{alias} \in m.R | alias \notin \{alias | r^{alias} \in \Delta m.R\}\} \cup \Delta m.R$
3 **return** *(D, R, m.s, m.m)*

To address RC2, SolveDB$^+$ proposes the concept of a *shared problem model*. The shared problem model is a special user-defined data type (UDT), which can be created via the **SOLVEMODEL** clause sharing the same syntax as **SOLVESELECT** (see above). Instead of returning an output relation, this new clause returns the UDT with the complete problem specification inside, i.e., (D, R, s, m). In SolveDB$^+$, such UDTs can be *transformed*, *used in computations*, or *stored* in a database using SolveDB$^+$ queries. The shared LTI model of the building inside temperature can be specified, for example, as:

```
1 SELECT (SOLVEMODEL
2   pars  AS (SELECT 0.0 AS a1,0.0 AS b1,0.0 AS b2)
3 WITH
4   data0 AS (SELECT 21.0 AS inTemp),
5   data  AS (SELECT time,outTemp,inTemp,hLoad FROM input),
6   simul AS (
7     WITH RECURSIVE t(time, x, inTemp) AS (
8       -- Initial data, for step 0
9       SELECT (SELECT min(ts) FROM data) AS time,
10             (SELECT x0 FROM data0) AS x,
11             (SELECT intemp0 FROM data0) AS inTemp
12    UNION ALL
13      -- Computed data, for steps > 0
14      SELECT (SELECT time+interval '1 hour'),
15             (SELECT a1*x+b1*outTemp+b2*hLoad FROM pars),
16              n.inTemp
17      FROM t LEFT JOIN LATERAL
18               (SELECT time, inTemp, outTemp, hLoad
19                FROM data) AS n
20             ON t.time = n.time - interval '1 hour'
21      WHERE (time < (SELECT max(time) FROM data))
22    SELECT time, x, intemp FROM t)))
```

As seen in the example, this model is, essentially, a placeholder with (dummy) relations for LTI model parameters (*pars*), initial values of the state variables (*data*0), and system inputs to be used for model training or predictions (*data*); and relations that represent simulated system states and outputs (*simul*). This model is fairly useless without actual model parameters and data being specified. Therefore, SolveDB proposes 3 specialized "conside yet powerful" operations on shared problem models: *instantiation*, *evaluation*, and *inlining*.

Model instantiation This operation instantiates a (generic) model into a (specific) problem model instance. This is done by allowing the user to redefine the input relation or any other CDTE in the problem model, along with their decision column list. For this, the operator << and another model are used, e.g.,

```
1 SELECT m << (SOLVEMODEL pars(b2) AS
2   (SELECT 0.995 AS a1, 0.001 AS b1, 0.2::float8 AS b2))
3 FROM model
```

In this example, a generic LTI model `m` is first selected from the table `model`. Then, `m` is instantiated using specifications of another model (say Δm) that is generated with **SOLVEMODEL** in the same query. Finally, the instantiation operator << replaces *pars* in m with *pars* in Δm while denoting $\{b2\}$ as a sole decision column with its initial value given in the table. The semantics of this operator is seen in Algorithm 1.

In general, as seen in Algorithm 1, model instantiation allows transferring an input relation, objective functions, constraint expressions, and any other CDTE expression from a *source model* to a *target model*. All entities that cannot be found using an *alias* in the target model are automatically added (instead of replaced) to the target model. This gives the possibility to inject data, different model parameters, objectives, constraints into a generic model.

Model Evaluation This operation allows accessing data from the input relation or any other CDTE inside the model. For this, SolveDB$^+$ introduces a new **MODELEVAL** clause:

```
MODELEVAL ( select_stmt ) IN ( select_stmt )
```

This clause retrieves a model instance by evaluating the 2^{nd} SELECT expression (`select_stmt`), then turns this model into a number of standard CTEs, and finally evaluates the 1^{st} SELECT expression in the context of these CTEs. Thus, the user can retrieve and inspect data specified by the model, e.g.,

```
MODELEVAL (SELECT a1, b1, b2 FROM pars)
       IN (SELECT m FROM model)
```

Model Inlining This operation allows embedding a model instance into another model instance – specified either by SOLVEMODEL or SOLVESELECT. To inline the model, the **INLINE** clause in **SOLVESELECT** or **SOLVEMODEL** is used, e.g.:

```
SOLVESELECT t(a1,b1,b2) AS
            (SELECT 0.5 AS a1,0 AS b1,0.5 AS b2)
     INLINE m AS (SELECT m <<
(SOLVEMODEL params AS (SELECT a1, b1, b2 FROM t)
       WITH  data0 AS (SELECT 25.0::float8 AS inTemp),
              data AS (SELECT * FROM input
              WHERE hload IS NOT NULL ))  FROM model)
MINIMIZE  (SELECT sum((x-inTemp)^2) FROM m_simulation)
SUBJECTTO (SELECT 0<=a1<=1, 0<=b1<=1, 0<=b2<=1 FROM t)
USING swarmops.sa()
```

This query specifies the problem of *least squares* to fit the LTI model parameters a_1, b_1, b_2 to the given data (Table 1). Here, the INLINE clause specifies that this problem depends on the shared problem model m from the table *model*. Before applying m to the outer problem, the model m has to be first instantiated with new LTI model parameters (line 4), a new initial value of the state variable (line 5), and new training dataset (line 6-7). Note, the decision columns (variables) from the outer problem (a1,b1,b2) are passed to the inner model during the instantiation, so their values can be used in computations defined by the inner model. Given this query, SolveDB$^+$ generates a new (outer) problem instance, making all internal model relations ($m.D$, $m.R$) available to the constraint expressions of the outer problem (lines 8-9) using the prefix $m\_$, where m is the assigned model alias (line 3).

The injection of the decision variables through model instantiation is not the only way to interconnect inner and outer problems in SolveDB$^+$. Another way is to declare that some of the inner model relations (CDTEs) contain decision columns. Consider the optimization/scheduling step of the PA process (P4 in Figure 1). To solve the cost minimization problem, SolveDB$^+$ allows defining the following query:

```
1  SOLVESELECT t(hload, iTemp) AS
2        (SELECT time, outTemp, inTemp, hLoad, pvSupply
3         FROM input WHERE hload IS NULL)
4  INLINE m AS (SELECT m << (SOLVEMODEL
5   data AS (SELECT time,outTemp,0 AS inTemp,hLoad FROM t)
6   WITH data0(inTemp) AS (SELECT NULL::float8 AS itemp))
7                FROM model)
8  MINIMIZE (SELECT sum((hload - pvsupply)*0.12) FROM t)
9  SUBJECTTO
10  -- Bind inner and outer problem variables
11  (SELECT t.inTemp = m_simul.x    FROM m_simul, t
12    WHERE t.time   = m_simul.time),
13  -- Initial conditions
14  (SELECT iTemp=20 FROM m_data0),
15   -- Comfort and HP power constraints
16  (SELECT 20<=intemp<=25, 0<=t.hpload<=17000 FROM t)
17 USING solverlp.cbc();
```

As seen here, model instantiation is used to declare that the attribute `inTemp` in the CDTE `data0` of the model m should be treated as decision column (line 6). Thus, a new decision variable(-s) will be introduced in the inner problem and made available to the specification of the outer problem (line 14).

Algorithm 2 elaborates the semantics of this **INLINE** clause. As seen in the algorithm, SolveDB$^+$ imports the input relation, CDTEs, and rule expressions from the inner model m into the outer model o. Each such expression receives a new prefixed alias for use in the outer problem to prevent naming collisions (lines 3,7). Further, table access scopes of these expressions are reworked such that the new relations (with new aliases) in the outer model can be accessed from the inner model expressions using the initial aliases, and without the need to modify the actual expressions (lines 5,9). In SolveDB$^+$, this is done by introducing additional CTEs in inner model expressions, e.g., **WITH** `data0` **AS** (**SELECT** * **FROM** `m_data0`), where m_data0 becomes a part of the outer model, but $data0$ is used in the inner model instead.

Algorithm 2: Problem model inlining

Input: o - a model instance before inlining; m - a model instance to be inlined; ma - a model alias;
Output: o' - a model instance after inlining

1 $prefix \leftarrow ma +' \_'$;
2 **for** $i \leftarrow 1 : |m_i.D|$ **do**
3 $\quad d^{prefix+a} \leftarrow \{d_i^a | d_i^a \in m.D\}$;
4 $\quad o.D \leftarrow o.D \cup \{d^{prefix+a}\}$;
5 $\quad scope(d^{prefix+a}) \leftarrow \{a_j \mapsto d^{prefix+a_j} | d_j^{a_j} \in m.D, j < i, d^{prefix+a_j} \in o.D\}$;
6 **for** $i \leftarrow 1 : |m_i.R|$ **do**
7 $\quad r^{prefix+a} \leftarrow \{r_i^a | r_i^a \in m.R\}$;
8 $\quad o.R \leftarrow o.R \cup \{r^{prefix+a}\}$;
9 $\quad scope(r^{prefix+a}) \leftarrow \{a_j \mapsto d^{prefix+a_j} | d_j^{a_j} \in m.D, d^{prefix+a_j} \in o.D\}$;
10 **return** *(o.D, o.R, o.s, o.m)*

Finally, as seen above, SolveDB$^+$ can "seamlessly integrate" the RC1-RC3 contributions of Sec. 3 and 4 and thus address RC4, allowing the user o specify a complete PA workflow as an extended SQL query. SolveDB$^+$ offers efficient in-DBMS processing by optimally using the DBMS query optimization and execution machinery for processing solver inputs and outputs, allowing for integrated (cache-aware) and optimized processing of PA workflows. The effects of using SolveDB$^+$ and its novel extensions are evaluated next.

5 EXPERIMENTAL EVALUATION

In this section, we first present results from a SolveDB$^+$ usability study involving a group of data scientists. To support the end user claims about SolveDB$^+$, we also evaluated SolveDB$^+$ on two typical PA use-cases from the fields of *energy* and *supply chain management*. Lastly, we used these use-cases to compare SolveDB$^+$ against SolveDB.

Table 6: Strong and Weak Points of SolveDB$^+$

Strong points
"*Syntax very SQL-like, queries feel natural, intuitive from a SQL users perspective. This also makes it **very** easy to pick up for anyone familiar with basic SQL.*"
"*I liked the syntax that makes you feel you are still working inside the database sphere while solving optimization problems without the need to jump between different solutions/languages*"
"*SolveDB+ is still a database system, meaning that it would be possible to to use it even in legacy systems...*"
"*I think SolveDB+ is a great tool! ... For any professionals I see this type of tool as the only tool for fast analytics.*"
"*... great idea and great tool. I have already suggested one of my students to check it out also...I am surprised how easy it was to implement and solve problems - definitely not the last time I will work with SolveDB+*"
"*Seems like a much more streamlined development experience.*"
"*Easy to use in a database-context*"
"*I do think python is more intuitive, but SolveDB+ is very close.*"
"*The simplicity, readability, easy to adapt and learn.*"
"*Fewer lines of code needed to solve the same problem...*"
"*SolveDB+ was faster than MADlib+pIPython*"
Weak points
"*...for some optimization problems, we need to put some "extra" effort to produce a good "representation" of the problem so it that can be handled by SolveDB+ (e.g. Sudoku solver). SolveDB+ needs a big community, and more detailed documentations and examples.*"
"*Needs to be updated on every PostgreSQL release*"
"*Due to relational nature of SQL syntax, some expressions are longer than they ideally should be*"

5.1 Usability Study

We conducted a study where the usability of SolveDB$^+$ was evaluated by a group of highly skilled data scientists, namely the 7 participants of the 2.5 day PhD course *Aspects of Advanced Analytics*, organized by Aalborg University in Dec. 2020. Each participant pre-reported strong competences in SQL, Python, PostgreSQL, and optimization problem solving. The participants used SolveDB$^+$ to solve their chosen subset of five simple optimization problems (Knapsack, production planning, Sudoku, curve fitting, and hypothetical DB deletes/inserts) and two more advanced PA problems (demand and supply balancing, heat-pump power optimization)[30]. In all cases, the initial data and the solution had to be stored in a database. For comparison, the participants had to use another in-DBMS analytics stack of their own choice for solving these problems. They agreed to use the stack based on PostgreSQL, the *PyMathProg* Python library for high-level optimization problem modeling, *PL/Python* language extension for in-DBMS Python programming, and the widely used PostgreSQL extension *MADlib*[5] for in-DBMS machine learning. Afterwards, the participants reflected on their experiences.

The study demonstrated that they solved their chosen problems with approx. 1.5-3.5 times less code and approx. 2 times faster SolveDB runtimes when using SolveDB$^+$. They identified a number of strong and weak points of SolveDB$^+$ - see Table 6. They also reflected on the new SolveDB$^+$ features, e.g., "*The SolveDB+ shared model concept is interesting...*", "*I think it [shared models] fits well with the rest of the system, ... can be incredibly useful in specific use cases...*" , "*...it is a great idea to incorporate the opportunity to do simulation models within the dbms... however, when doing this, my experience is that I need a lot of flexibility - and im not sure the compact style of solveDB+ will benefit me there. At least not yet*". In summary, the study confirmed our expectations that SolveDB$^+$ has good usability, explainability, developer productivity, and performance, even for new users. The next subsections dig deeper into these aspects.

5.2 Experimental Setup

To support the claims about SolveDB$^+$ (Section 5.1), we further evaluated SolveDB$^+$ in two typical PA use-cases from the fields of *energy* and *supply chain management*, covering the phases P1-P4 shown in Figure 1. For both use-cases, we implemented two PA technology stacks: 1) a stack consisting of a standard DBMS and relevant state-of-the-art PA tools and 2) a SolveDB$^+$ stack with a number of standard and specialized built-in solvers (used in place of the PA tools). In both configurations, input data is read from the database and the solution is stored back to the database. We compared these two technology stacks by measuring the Effective Lines of Code (eLOC)[24] (relevant since we are comparing high-level languages and eLoc is used in similar comparisons [28, 31]) of the full implementations and their inherent P1-P4 parts. We also compared them in terms of *execution time*, by encompassing database I/O time as well as prediction, model fitting, and optimization problem solving time. Lastly, these use-cases were used to compare SolveDB$^+$ against SolveDB by evaluating novel SolveDB$^+$ features, including *CDTEs*, *shared models*, and the *predictive framework*. In all experiments, we used SolveDB$^+$/SolveDB on top of PostgreSQL 11.2 in the default configuration and native SolveDB solvers for LP/MIP/Blackbox problems [31].

5.3 Energy Planning (UC1)

We evaluated the impact of using SolveDB$^+$ to solve the energy planning problem from the running example, denoted as UC1, using the NIST dataset [4] – containing 8737 hourly aggregates from PV, HVAC, temperature sensors, all from a high precision lab-home. We compared with two different PA technology stacks using either *specialized tools* or *general modeling tools*.

Specialized tools Here, we used standard PostgreSQL, Matlab R2015b, and three powerful specialized libraries, *Statistics and Machine Learning Toolbox*, *System Identification Toolbox*, and *Multi-Parametric Toolbox* (MPT), for *Linear Regression (LR) forecasting*, *state-space (SS) model fitting*, and *dynamical system optimization*, respectively. Specifically, we used a Matlab implementation that uses the following native library functions: `fitlm` to estimate the LR model coefficients, `predict` to produce PV supply forecasts, and `ssest` to fit HVAC state-space model parameters to the given data. The implementation uses the outputs of these functions to define an MPC (model-predictive control) controller with a number of constraints on the system input and state variables and the PV supply amounts used as a reference for minimizing electricity cost. The size of this implementation in eLOC is given in Figure 3(a) as **`Matlab-native`**. As this configuration is the most comprehensive, it is used as a *reference* for this comparison.

General-purpose modeling tools In this configuration, we utilized a standard DBMS, Matlab R2015b, and YALMIP – a Matlab toolbox for rapid prototyping of optimization problems. Like SolveDB$^+$, YALMIP is provided with a variety of solvers for different problem classes. By using both YALMIP and SolveDB$^+$, we modeled *LR model estimation* (P2), *state-space model fitting*

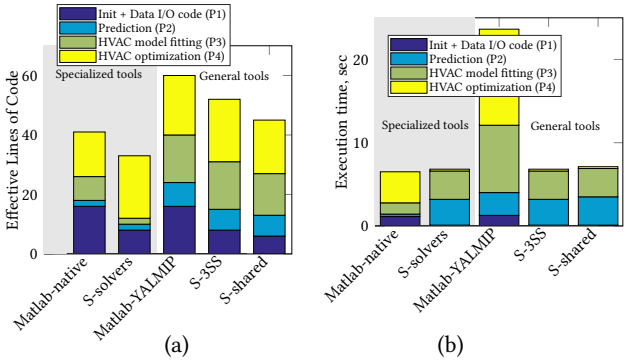

Figure 3: Implementation sizes (a) and run time (b) of UC1

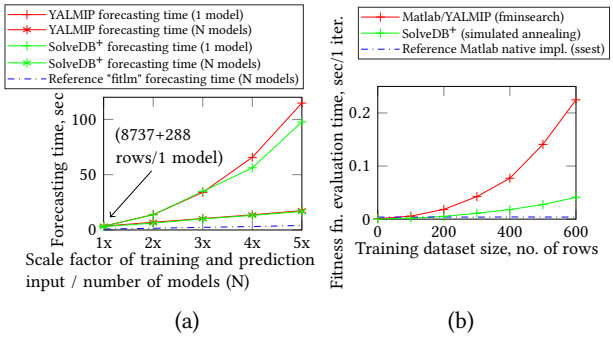

Figure 4: Scalability of prediction (P2) and model fitting (P3) using general-purpose (YALMIP, SolveDB$^+$) and specialized tools (P2: fitlm, P3: ssest)

(P3), and *dynamical system optimization* (P4) problems as explicit LP/nonlinear optimization problems using Matlab/YALMIP programs and SolveDB$^+$ queries, respectively. Specifically, P2 is modeled as an LP optimization problem by minimizing the forecasting error to compute regression model parameters. To solve this problem, SolveDB$^+$ and YALMIP use the Coin-OR CBC solver for the actual computations. Similarly, P3 is specified as a non-linear problem (NLP) of minimizing prediction error of a linear dynamical system using time domain data and *HVAC power levels* and *inside temperatures* as decision variables. To solve this problem, Matlab/YALMIP uses *fminsearch* and SolveDB$^+$ uses *simulated annealing*. These are two distinct NLP solvers that solve the problem in a non-deterministic way. Since they typically give different solutions each time, we only measure average time required for a single solving iteration (fitness function evaluation, Figure 4(b)). Lastly, P4 is modeled as a linear cost minimization problem, where the cost of electricity is minimized under a number of constraints on the HVAC system state and input, and by taking PV supply forecasts into account (based on the LR model). SolveDB$^+$ and YALMIP use CBC to solve this problem. The size of YALMIP implementation in eLOC is given in Figure 3(a) as **`Matlab-YALMIP`**. In SolveDB$^+$, the complete PA workflow, encompassing P2-P4, were implemented in 3 different ways:

-**`S-3SS`** P2-P4 were implemented as three independent **`SOLVESELECT`**s linked using temporary tables (P1).

-**`S-shared`** To be able to reuse the HVAC model parts repeating in P3 and P4, we defined the complete PA problem as a single **`SOLVESELECT`** using a SolveDB$^+$ *shared model*. The model captures indoor temperature dynamics, with P2 and P3 **`SOLVESELECT`** specifications embedded into the model. Note, the size of the model is equally shared by the respective parts in Figure 3(a).

-**`S-solvers`** To relieve the user from the need to specify detailed **`SOLVESELECT`** queries for P2 and P3, we implemented two *composite solvers* which hide respective problem specification details. As these solvers are conceptually similar to the library functions (`Matlab-native`), the overall PA workflow is simplified to a single **`SOLVESELECT`** invoking the composite solvers.

Comparison to specialized tools As seen in Figure 3, the complete PA problem can be specified in just 41 lines of Matlab code and solved in 6.5 secs using specialized tools (`Matlab-native`). Here, around 40% of code and 18% of time is used for initializing libraries and accessing the database, the rest is spent on forming required inputs for, and invoking, the black-box library functions (all considered as P1). As seen for `S-solvers`, this I/O overhead as well as optimization time can be reduced by more than one order of magnitude if all computations are pushed inside the DBMS. This also reduces the PA problem (code) size when SolveDB$^+$ with specialized (composite) solvers are used. As seen in Figure 5 (SolveDB$^+$ vs. MPT), this optimization (P4) performance improvement comes from the reduced model generation time – time spent by MPT to translate the problem to YALMIP, and for YALMIP to aggregate problem constraints and build an optimization (P4) model instance in the binary representation (required by CBC). However, as seen in Figure 3(b), native prediction (P2) and SS model fitting (P3) functions are hard to outperform using general-purpose solvers (`Matlab-native` v.s. `S-solvers`). Figure **??** hint that specialized SolveDB$^+$ solvers for prediction and model fitting are required for larger input datasets. Considering the prediction alone, LR model fitting (P2) using the general-purpose solvers scale *linearly* with respect to independent model count and *exponentially* with respect to training and prediction input size, and therefore might still be useful for some smaller PA cases.

Comparison to general-purpose tools Compared to the native tools (`Matlab-native`), general modeling tools (`Matlab-YALMIP`, `S-3SS` and `S-shared` – all using general-purpose solvers) offer a single language and the full control of how the three PA subproblems P2-P4 are specified. However, explicitly specifying these sub-problems requires up to 45% more code (see Figure 3(a)). Further, computations are up to 3.6 times slower (see Figure 3(b)) and they do not scale (linearly) as in the native case (see Figures 4–5). Comparing YALMIP to SolveDB$^+$, SolveDB$^+$ solves the complete PA problem 3.5 times faster due to significantly reduced data I/O and HVAC optimization time. This can also be seen in Figure 5, which shows that SolveDB$^+$ exhibits up to 2 order of magnitude less data I/O and up to 3 orders of magnitude less model generation time, which is spent translating high-level constraint and objective function specifications into the binary format required by CBC. Both YALMIP and SolveDB$^+$ exhibit somewhat comparable forecasting (P2) and model fitting performance (P3). In the P2 case, YALMIP model generation time is less significant as model constraints can be vectorized (defined without "'for"' loops) and, in the P3 case, just 3 decision variables (a_1, b_1, and b_2) are used. Still, as shown in Figure 4(a), SolveDB$^+$ implementation offers up to 18% lower forecasting time for larger input dataset due to more efficient processing of linear constraints. This difference is less evident when several independent forecasting models need to be estimated using smaller training datasets. Lastly, in addition to these performance benefits, SolveDB$^+$ offers up to 33% smaller implementation sizes as shown in Figure 3(a).

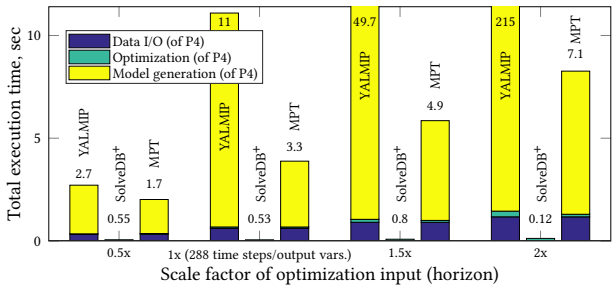

Figure 5: Scalability of HVAC energy optimization (P4)

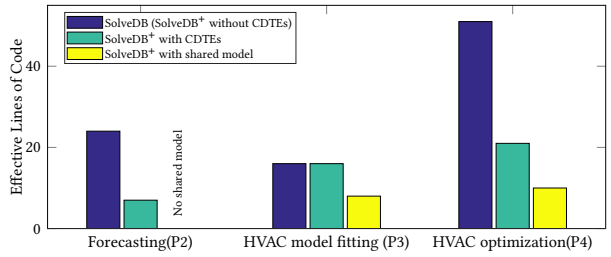

Figure 6: Sizes of SolveDB$^+$ implementations with and without CDTEs and Shared Models

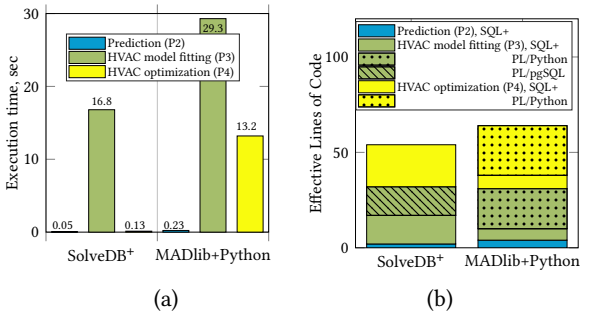

Figure 7: UC1 performance (a) and implementation sizes (b) when using SolveDB$^+$ and existing in-DBMS tools

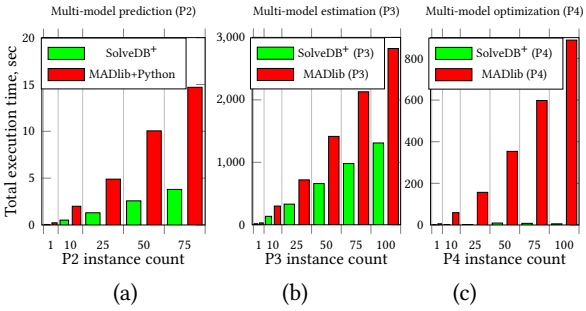

Figure 8: Scalability of In-DBMS UC1 implementations

Comparison to in-DBMS analytics tools Next, we compared SolveDB$^+$ against the in-DBMS analytics stack from the usability study (Section 5.1). We used MADlib's in-DBMS *linear regression* (*linregr_train* UDF) for P2. Since MADlib alone cannot be used to solve the HVAC model fitting and optimization sub-problems (P3-4), we implemented two in-DBMS Python (*PL/Python*) programs for HVAC model fitting (P3) and HVAC operation optimization (P4) by utilizing the *Swarmops* and *PyMathProg* Python libraries, respectively. These libraries offer high-level optimization problem modeling capabilities (required for P3-4) and, under the hood, invoke the low level solvers *Differential Evolution* and *GLPK*, respectively. A SolveDB$^+$ implementation uses three **SOLVESELECT** statements that define the P2-P4 sub-problems and invoke the (same) *linear regression*, *Swarmops*, *GLPK* low-level solvers using SolveDB$^+$'s high-level solvers (incl., *solverlp* and *swarmops* – see Section 3.2 and Section 4.1). The SolveDB$^+$ implementation also uses a *PL/pgSQL* UDF to compute prediction error (being minimized) given (solver-)supplied candidate values of the HVAC model parameters (P3). The goal of this experiment was to compare implementation sizes and runtimes of individual phases (P2-P4) when solving a number of UC1 instances using the same set of low-level solvers (i.e., linear regression, differential evolution, GLPK) running inside a DBMS. Thus, we aimed at comparing the two stacks in terms of *how P2-P4 are specified by the user, how well these (high-level) problem specifications are translated to (low-level) solver inputs*, and *how fast data, solver inputs and outputs are processed by the two in-DBMS stacks.*

As seen in Figure 7(b), *MADlib+Python* required 64 eLOC of mixed SQL and PL/Python code and SolveDB$^+$ required 47 eLOC of (extended) SQL and PL/pgSQL code. While implementation sizes are somewhat comparable, SolveDB$^+$ required very little non-SQL code (15 lines of PL/pgSQL only) to specify the iterative P3 computations. Note, we have also implemented UC1 using pure (extended) SQL (in total 42 lines) with a recursive CTE query for P3. However, this implementation with a recursive query for HVAC simulations might be less intuitive for inexperienced users. In terms of performance, as seen in Figure 7(a), a single instance of UC1 can be solved with SolveDB$^+$ more than twice as fast as with MADlib+Python (19.9 vs 42.7sec). Here, significant gains are observed primarily for P3 (16.8 vs 29.3sec) and P4 (0.13 vs 13.2 sec). For P3, SwarmOPS (in C++) was able to reevaluate the fitness function specified as a SELECT expression from **SOLVESELECT** (that calls a `PL/pgSQL` function) approx. 1.7 time faster that pure Python implementation, where both the solver (SwarmOPS) and the fitness function were implemented in Python. For P4, SolveDB$^+$ offers faster processing of P4 problem symbolic descriptors (*solverlp* vs *PyMathProg*), to be consumed by the same low-level solver (GLPK in C). As seen in Figure 8 (a-c), this gain is more significant when scaling the number of UC1 instances to be solved, i.e., scaling the number of parameters need to be estimated for P3, and predictions and optimization (P2, P4) need to be made for multiple independent HVAC installations. Here, SolveDB$^+$ offered 3.6x faster predictions (P2, Figure 8(a)) since it did not need to create intermediate tables for model parameters and summaries, unlike MADlib; 2.1x faster model parameter estimation, primarily, due to faster evaluation of the fitness function (P3, Figure 8(b)); and 161x faster optimization (P4, Figure 8(c)) primarily due to efficient manipulation of symbolic optimization models and automatic problem partitioning. All in all, SolveDB$^+$ had 2.8x faster execution of the complete PA workflow using less and less complex code, showing its clear advantage over MADlib+Python and confirming the claims about SolveDB$^+$ usability (and performance, see Section 5.1).

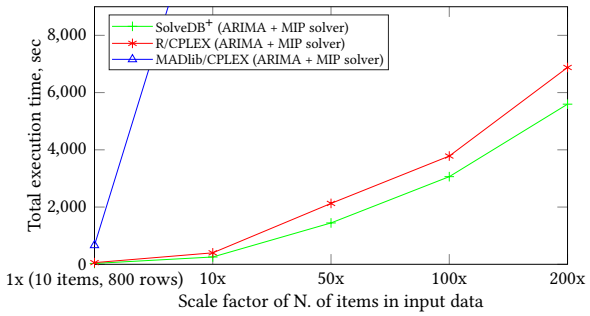

Figure 9: Scalability of combined P1-P4 for UC2

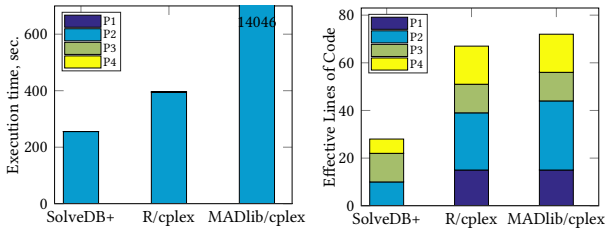

Figure 10: UC2 performance (a) and eLOC (b) comparison

5.4 Supply Chain Management (UC2)

As a second use case (UC2), we considered a common supply chain management scenario. We used the TPC-H dataset [26] containing production supply chain items with the information associated to these items, e.g., orders in the last months, parts needed to assemble the items, size of the parts, price, suppliers, etc. The objective in this use case is to increase revenue by producing in advance the items that will be the most profitable in the next month. The warehouse of the production facility has a limited *volume capacity*, so the decision on which items to produce and store has to be optimized subject to this constraint. This PA workflow requires predicting expected item demand for the next month (P2), modeling *expected profit* for the items by weighting item profit by the probability that the item is ordered in the next month (P3), and solving a variant of the *knapsack problem*, where the warehouse's capacity constraint is respected (P4).

We compared PA stacks with SolveDB$^+$ and both *standalone* and *integrated DBMS analytics* tools. For SolveDB$^+$, we used the predictive framework with a built-in ARIMA solver based on the Statsmodels 0.8.0 package [29] for P2, PL/pgSQL function for P3, and a pre-installed MIP solver from the GNU Linear Programming Kit (GLPK) v4.47 for P4. For standalone tools, we used a configuration with a standard PostgreSQL 9.6.1 (P1, P3), an ARIMA model in R 3.2.3 (P2), and a MIP solver in CPLEX 12.7.1 (P4). For the integrated DBMS analytics tools, we utilized PostgreSQL 9.6.1(P1, P3) with the MADlib [5] extension for in-DBMS machine learning using SQL (P2), and the same MIP solver in CPLEX 12.7.1 (P4). We used 5 different UC2 sizes, scaling the number of items in the dataset. Each item is associated with a time series containing 80 rows of monthly orders.

Figure 10(a) shows the results on the UC2 instance with 100 items. In all implementations, the prediction process accounted almost exclusively for the total execution time, as up to 10000 ARIMA models are trained: 100 per item in R and MADlib, 10 particles with 10 iterations per item in SolveDB$^+$. However, the SolveDB$^+$ implementation was approximately 30% faster than R, and 2 orders of magnitude faster than MADlib, thanks to the efficient use of particle swarm optimization solver for cross validation of the model parameters. Specifically, MADlib does not provide efficient support for cross-validating the forecasting models (ARIMA), with multiple write/read operations accounting for as much as 60% of the total execution time. Figure 10(b) shows the size for the three implementations (implementation size is identical across instances), with SolveDB$^+$ being approximately 50% smaller than the R/MADlib and CPLEX implementations.

The performance results for the different UC2 instances in Figure 9, together with Figure 10(b), show that SolveDB$^+$ allows for a more compact problem definition and execution times that are between 20% and 30% faster than the R configuration, and orders of magnitude faster than the MADlib setup. SolveDB$^+$ outperforms the other two systems thanks to a reduced number of I/O operations and the use of the native local search solvers for hyper-parameters optimization in the model training phase. All in all, UC2 also confirms the end-user claims about SolveDB$^+$ usability (and performance) (Section 5.1).

5.5 SolveDB$^+$ Feature Evaluation (Comparison to SolveDB)

SolveDB$^+$ inherits features and advantages from SolveDB [31]. Specifically, both offer wider applicability and significantly increased tool productivity and usability (order of magnitude less code), while in most cases providing much (up to > 2 orders of magnitude) better performance than systems such as LogicBlox or Tiresias (seeSection 2). We now evaluated the novel SolveDB$^+$ features that distinguish SolveDB$^+$ from SolveDB using the energy and supply chain management use-cases, UC1 and UC2.

Common Decision Table Expressions (CDTEs) As explained in Section 4, CDTEs extend the SOLVESELECT clause like Common Table Expressions (CTEs) extend the simple SELECT in standard SQL. In contrast to CTEs, CDTEs allow annotating some table attributes as *decision columns*, the values of which are evaluated as part of a (much better organized) single SOLVESELECT problem. As seen in Figure 6, CDTEs have a major impact on SolveDB$^+$ usability. Specifying LR model estimation/prediction problems and HVAC optimization problems from the energy planning use-case without CDTEs (SolveDB) requires up to 3 times more **SOLVESELECT** code compared to using CDTEs (SolveDB$^+$). In this case, the HVAC model fitting problem does not benefit from CDTEs, as it uses just a single collection of decision variables, which can be well arranged in a single table. Our experiments also showed that CDTEs do not introduce significant performance overhead to the overall PA workflow.

Shared Optimization Models As explained in Section 4, shared optimization models allow reusing data, objective, and constraint specifications across several optimization problems. UC1 can benefit from such models, by reducing the amount of **SOLVESELECT** code 2 times (Figure 6) for HVAC model fitting and optimization sub-problems alone, and 16% for the complete PA application (see `S-3SS` and `S-shared` in Figure 3(a)), which also includes the shared model specifications. As can be seen in Figure 3(b), shared models do not introduce significant performance overhead to the overall PA workflow.

Predictive Framework As discussed in Section 4, the predictive framework of SolveDB$^+$ offers two ways to integrate new

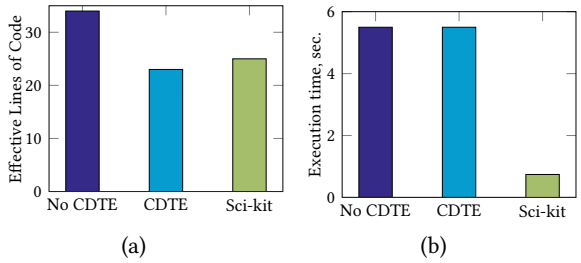

Figure 11: LR code size (a) and execution time (b)

forecasting models. The user can either manually specify forecasting models as SOLVESELECT queries and/or specialized solvers, or install them as "wrappers" over third-party general purpose forecasting libraries. We now compare these two approaches.

For this experiment, we developed the linear regression model as a SOLVESELECT query a) with CDTEs, and b) with no CDTEs wrapped into the respective solvers within the predictive framework. Additionally, we c) installed a general purpose linear regression model from the Sci-kit learn library [3] as a wrapper in SolveDB$^+$. Figure 11(a) shows the implementation size for these three cases. While the size of the Sci-kit implementation is approximately the same as the CDTE implementation, the no CDTE implementation is approximately 30% larger than the other two. Still, the Sci-kit solver implementation is conceptually simpler as it just uses a library function. Furthermore, Figure 11(b) shows that the specialized SolveDB$^+$ implementation is almost 8 times faster than the manual SOLVESELECT implementation (CDTEs do not affect performance), as it combines both in-DBMS execution and a highly specialized machine learning library.

6 CONCLUSION AND FUTURE WORK

This paper presented SolveDB$^+$, the first SQL-based DBMS to provide an extensible and efficient eco-system for all Prescriptive Analytics (PA) phases. SolveDB$^+$ reduces the complexities and inefficiencies of existing PA application stacks, which consist of many specialized, independent, poorly connected systems with different APIs and languages. SolveDB$^+$ acts as a "*swiss-army knife*" system for PA, effectively supporting all 5 phases of PA development: *P1: data management*, *P2: prediction/forecasting*, *P3: system modeling*, *P4: optimization problem solving*, and *P5: solution analysis*. SolveDB$^+$ provides extensibility, allowing developers to add new custom functionalities for specialized PA cases. SolveDB$^+$'s common SQL-based language can express an entire PA workflow in a single SQL-based query. SolveDB$^+$ offers faster PA workflow execution due to its in-DBMS PA algorithms.

Compared to the earlier (SolveDB) tool, SolveDB$^+$ provides a number of novel modeling features, including *common decision table expressions* and *shared optimization models*, enabling a significant size reduction of complex PA problem specifications. It also introduces a new *predictive framework*, which is a generic and extensible in-DBMS platform for the use and development of time series forecasting methods. With all its features, SolveDB$^+$ offers convenient and efficient ways to use and extend the eco-system of forecasting models and optimization problem solvers, thus adapting the system to virtually unlimited PA scenarios.

Our experiments showed that the new SolveDB$^+$ features yield up to 5 times smaller problem specifications (better productivity and explainability) and up to 6 times faster forecasting time, compared to SolveDB. Overall, SolveDB$^+$ offers up to three orders of magnitude better performance for individual PA steps, and up to 3.5 times faster execution times and 3 times smaller implementation sizes for the full PA workflow, compared to state-of-the-art baselines. SolveDB$^+$ scales well in its chosen in-DBMS setting.

Future work will redesign SolveDB$^+$ for distributed Big Data processing and integrate What-If analysis for hypothetical scenarios, and support more data formats, operators on shared models, and further ML models.

REFERENCES

[1] A. Ghoting et al. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE*.
[2] A. Raj et al. 2020. From Ad-Hoc Data Analytics to DataOps. In *ICSSP*.
[3] F. Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12 (2011).
[4] H. William et al. 2017. Net Zero Energy Residential Test Facility Instrumented Data; Year 2. (2017). https://doi.org/doi.org/10.18434/T46W2X
[5] J. M. Hellerstein et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *PVLDB* 5, 12 (2012).
[6] M. Aref et al. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*.
[7] M. Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.
[8] M. Brucato et al. 2016. Scalable package queries in relational database systems. *PVLDB* 9, 7 (2016).
[9] M. Hall et al. 2009. The WEKA data mining software: an update. *SIGKDD Explor.* 11, 1 (2009).
[10] M. Jasny et al. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD*.
[11] M. Stonebraker et al. 2013. SciDB: A database management system for applications with complex analytics. *CiS&E* 15, 3 (2013).
[12] M. Schule et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems. In *BTW*.
[13] S. Sanda et al. 2019. In-database Distributed Machine Learning: Demonstration using Teradata SQL Engine. *PVLDB* 12(12) (2019).
[14] T. Kraska et al. 2013. MLbase: A Distributed Machine-learning System.. In *CIDR*.
[15] U. Fischer, F. Rosenthal, and W. Lehner. 2012. F2DB: The Flash-Forward Database System. In *ICDE*.
[16] D. Frazzetto, T. D. Nielsen, T. B. Pedersen, and L. Siksnys. 2020. Prescriptive Analytics: A Survey of Emerging Trends And Technologies. *VLDBJ* 28(4) (2020).
[17] Clyde W. Holsapple, Anita Lee-Post, and Ramakrishnan Pakath. 2014. A unified foundation for business analytics. *DSS* 64, C (2014).
[18] K. Hu, D. Orghian, and C. Hidalgo. [n.d.]. DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. In *HILDA*.
[19] A. Kalinin, U. Cetintemel, and S. Zdonik. 2015. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB* 8, 10 (2015).
[20] James Kennedy. 2011. Particle swarm optimization. In *Encyclopedia of machine learning*. Springer, 760–766.
[21] P. Manolios, V. Papavasileiou, and M. Riedewald. 2014. Ilp modulo data. In *FMCAD*.
[22] MATLAB. 2020. *MATLAB API for Python.* Available at se.mathworks.com/help/matlab/matlab-engine-for-python.html.
[23] A. Meliou and D. Suciu. 2012. Tiresias: the database oracle for how-to queries. In *SIGMOD*.
[24] E. Morozoff. 2010. Using a line of code metric to understand software rework. *IEEE software* 27, 1 (2010).
[25] B. Omidvar-Tehrani, S. Amer-Yahia, E. Simon, and et al. [n.d.]. UserDEV: A Mixed-Initiative System for User Group Analytics. In *ILDA*.
[26] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.
[27] Mark Rittman. 2012. *Oracle Business Intelligence 11g Developers Guide.* McGraw-Hill Osborne Media.
[28] O. Rybnytska, L. Siksnys, T. B. Pedersen, and Bijay Neupane. 2020. pgFMU: Integrating Data Management with Physical System Modelling. In *EDBT*.
[29] S. Seabold and J. Perktold. 2010. Statsmodels: Econometric and statistical modeling with python. In *PiSC*.
[30] L. Siksnys. 2020. *Phd Exercises.* Available at https://www.daisy.aau.dk/wp-content/uploads/2020/12/Advanced-Analytics-Exercises.pdf.
[31] Laurynas Šikšnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *Proc. of SSDBM*. 14.
[32] Z. Tang and J. Maclennan. 2005. *Data mining with SQL Server 2005.* Wiley.
[33] I. Xanthopoulos, I. Tsamardinos, V. Christophides, E. Simon, and A. Salinger. 2020. Putting the Human Back in the AutoML Loop. In *ETLMP*.