SolveDF: Extending Spark DataFrames with support for constrained optimization

Frederik Madsen Halberg
fhalbe12@student.aau.dk

Sunday 11th June, 2017

Aalborg University
Faculty of Engineering and Science
Department of Computer Science
10th Semester project
Supervisor: Bent Thomsen

Preface

This project is a continuation of a 9th semester project titled *Prescriptive Analytics for Spark* by Freiberger et al. [1], supervised by Bent Thomsen and Torben Bach Pedersen. The Motivation and Background sections of this project are based largely on the report of the previous project.

Summary

Prescriptive Analytics (PA) is an emerging phase of Business Analytics (BA), which has traditionally consisted of Descriptive Analytics (DA) and Predictive Analytics (PR). Whereas DA and PR are concerned with understanding the past and the future, PA is concerned with providing direct support for decision making, by suggesting (prescribing) optimal decisions to make for a given business problem. However, existing PA solutions often consist of several specialized tools glued together in an improvised manner, which is cumbersome and ineffective. There is a need for more integrated solutions that support all the necessary steps for PA, including data management, prediction, and optimization problem solving.

This project details the design and implementation of SolveDF, a tool that extends Spark SQL with functionality that allows for declarative specification of constrained optimization problems through *solve queries*. SolveDF is heavily inspired by SolveDB, and allows for data management and constrained optimization to be performed seamlessly in a Big Data environment. SolveDF can leverage the distributed nature of Spark by splitting optimization problems into smaller independent subproblems that can be solved in parallel on a cluster. Like Spark SQL, SolveDF is not limited to a single type of data source, but can be used with many different types of data sources, including JSON-files, HDFS and any DBMS that supports JDBC.

The report also includes a brief overview of Spark and constrained optimization problem solving, as well as related work in the area of data management systems with integrated support for constrained optimization problem solving.

As a part of designing SolveDF, a small usability experiment of SolveDB is performed to evaluate how intuitive SolveDB is. The results suggest that SolveDB can be learned quickly with minimal guidance, and that the overall concept and structure of solve queries make sense. The experiment also identified a number of small problems encountered when using SolveDB, and some of these problems are addressed in SolveDF.

Performance experiments of SolveDF show that for certain types of problems, SolveDF has similar performance to SolveDB when running on a single machine. The results also show that when running SolveDF on a cluster, there appears to be a linear speedup relative to the amount of nodes in the cluster for certain problems, as shown by SolveDF being up to 6.85 times faster on a cluster of 8 nodes. In particular, optimization problems that are partitionable and have high complexity (e.g. Mixed integer programming problems) are ideal problems for SolveDF to solve. The results also show that SolveDF could still use more work, as SolveDF is relatively slow at constructing optimization problems compared to SolveDB.

Contents

Motivation

Traditionally, business analytics has consisted of two phases: Descriptive Analytics (DA) and Predictive Analytics (PR). Recently however, a third phase known as Prescriptive Analytics (PA) has started to emerge. Whereas DA answers the question of "what has happened?" and PR answers the question of "what will happen in the future?", PA answers the question of "what should we do about it?". As such, PA is concerned with automatically identifying and suggesting (prescribing) optimal decisions to make in a given business problem, usually through the use of mathematical optimization. It should be noted that PA also encompasses the tasks of DA and PR, i.e. you cannot perform PA effectively without DA and PR. This makes PA the hardest and most sophisticated type of business analytics, but it is also able to bring the most value to a business[2].

As an example of a PA application, consider a smart grid that is tasked with balancing energy production and consumption. Energy output from many Renewable Energy Sources (RESes) such as solar panels and wind turbines depends on weather conditions, and as such varies significantly over time. This can make balancing the grid challenging, as peaks in energy consumption might not coincide with high output from RESes. However, a lot of energy demand is flexible, meaning that it doesn't require that energy is consumed at one specific time[3]. For example, when using a dishwasher, one might not care about when exactly it runs, as long as the dishes are clean before the next morning. To make the most out of the RESes, we want to schedule the flexible demand at times where energy output of RESes is high. In order to do this effectively, we need some way to predict when the energy output of RESes is high, for example by using weather forecasts. As such, this problem requires analysis of existing data, predictions about the future (e.g. weather forecasts and forecasts about energy consumption), and automatically making optimal decisions regarding when to schedule energy consumption for flexible demand.

However, it is difficult to make effective PA solutions with current tools. Typically, you end up with a mishmash of several specialized and non-integrated tools glued together in an improvised manner (also known as the "hairball model"[4]). Such a solution could for example include Hadoop or an RDBMS for data collection and consolidation, MATLAB for predictions, and CPLEX for optimization. This is far from ideal, as using these suites of non-integrated tools tends to be labor-intensive, inefficient, error-prone, and requires expertise with several languages and technologies. Instead, it would make sense if all the needed functionality for PA was integrated into a single tool. Database systems with integrated support for optimization problem solving, such as SolveDB[5] and Tiresias[6], are examples of tools that try to integrate functionality for some or all

of the steps required for a PA workflow. For example, SolveDB allows users to specify optimization problems in an SQL-like language, allowing for seamless data management and optimization problem solving in a single tool.

At the same time, the volume, velocity and variety[7] of available data has seen an explosive growth. In fact, 90% of all the data we have today was produced over the course of the last two years[2], and a significant part of this data is either semi-structured or unstructured[8]. As a response to this, several Big Data technologies such as NoSQL databases and MapReduce frameworks have emerged to tackle the new sizes and types of data that traditional RDBMSes struggle with. However, the popular MapReduce frameworks are not without issues and limitations, such as being too low-level, I/O bound, and unsuitable for interactive analysis. In recent years, Apache Spark has appeared as a promising alternative to MapReduce. Spark was made specifically to address applications that MapReduce frameworks handle poorly, such as iterative algorithms and interactive data mining tools[9]. Not only does Spark boast orders of magnitude higher performance than Hadoop MapReduce[10], it also uses a more general and higher-level programming model based on Resilient Distributed Datasets (RDDs)[11].

The evolution of Big Data platforms is relevant to consider for PA, as some PA problems involve extremely large amounts of data. For example, if we wanted to adopt the earlier mentioned smart grid in all of Denmark, we would need data from up to 2.5 million households[12]. If we assume that we receive energy meter readings every 15 minutes from each of these households, we would get a total of 96 readings per household per day. Even if only 5% of danish households were part of this smart grid, we would still get 12 million readings every day from consumer data alone. In a more ambitious setting, where data is collected from all member countries of the European Union (220 million households [13]), 5% would correspond to over a billion readings per day, making it infeasible to store and query in traditional relational databases. To address these problems, there is a need for a solution capable of processing and querying such a vast amount of data for PA.

Problem Statement

The purpose of this project is to investigate the following hypothesis:

- Is it feasible to make a tool that allows for seamless integration of data management and constrained optimization problem solving in a Big Data context?

This investigation includes the following:

- A usability evaluation of SolveDB.
- Design and implementation of SolveDF, a tool that extends Spark SQL with SolveDB's concept of solve queries.
- Performance experiments of SolveDF and a comparison to SolveDB.

Part I.

Problem Analysis

1 Background

Before presenting SolveDF and related work, we will briefly look at some subjects related to the making of SolveDF, namely Apache Spark and constrained optimization. As this project is very focused on Spark, other cluster-computing frameworks are not covered in this report. If the reader is interested in a more thorough survey of existing cluster-computing frameworks, this can be found in the report of the previous project[1]. Likewise, if the reader is interested in more information regarding prescriptive analytics (and business analytics in general), this is also covered in the previous project.

1.1 Apache Spark

Apache Spark is a general-purpose cluster-computing framework based on the Resilient Distributed Dataset (RDD) data structure[11]. RDDs act as the primary abstraction in Spark, facilitating a restricted form of distributed shared memory. To the programmer, an RDD appears more or less like an ordinary collection of objects, but behind the scenes, it facilitates partitioning, distribution and fault-tolerance. RDD's are immutable, but a new RDD can be generated by applying coarse-grained *transformations* to an existing RDD. Examples of transformations could be the *map*, *filter* and *join* functions. Transformations are lazy evaluated, meaning that the result of a transformation is first computed when it is needed, which is when an *action* is applied to it. Examples of actions could be the *reduce*, *collect* and *foreach* functions. A Spark program is written by applying sequences of transformations and actions to RDDs, and although Spark is written in Scala, there is support for writing Spark programs in either Scala, Java, Python or R. Listing 1.1 shows a simple Spark program that loads the lines of a text file, capitalizes all the letters, and prints only the lines that start with the word "WARNING" and contain the word "ANALYTICS".

In terms of performance, Spark claims orders of magnitude faster computation than Hadoop MapReduce. This is primarily due to Spark's wide usage of in-memory computation compared to MapReduce, as RDDs can be stored in main memory (although

```
1 lines = spark.textFile("someTextFile.txt")
2 lines.map(line => line.toUpperCase())
3     .filter(line => line.startsWith("WARNING"))
4     .filter(line => line.contains("ANALYTICS"))
5     .foreach(line => println(line))
```

Listing 1.1: A simple Spark program written in Scala.

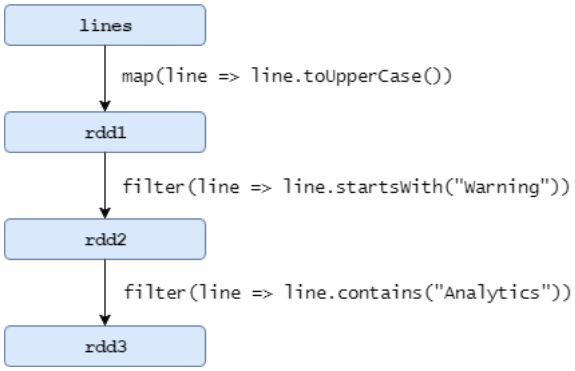

Figure 1.1.: Lineage graph of the RDDs used in Listing 1.1.

they might spill to disk if they are too large). Spark was designed specifically to deal with tasks that MapReduce struggles with. In particular, MapReduce frameworks tend to be very inefficient for applications that reuse intermediate results across nodes, which for example includes many iterative machine learning and graph algorithms[14]. This is because in most MapReduce frameworks, you have to write to external stable storage to reuse data between MapReduce jobs, which is heavy on disk I/O and requires replication.

Instead of using replication, RDDs support fault-tolerance by logging the lineage of the dataset. The lineage is the sequence of transformations (e.g. map, filter, join) that created the RDD. This means that if a partition of an RDD is lost, the RDD knows how it was derived from other datasets, and can therefore recompute the lost partition. Because of this, Spark can forgo the high cost of replicating RDDs. Figure 1.1 shows a lineage graph for the program in Listing 1.1.

Despite the fact that RDDs are immutable and can only be manipulated by coarse-grained transformations, they are actually quite expressive. In fact, not only does Spark's programming model generalize MapReduce, it can also efficiently express the programming models of cluster-computing frameworks such as DryadLINQ, Pregel, and HaLoop[11].

1.1.1 Spark SQL

In addition to RDDs, Spark offers functionality for combining procedural programming with declarative queries through the Spark SQL component[15]. Spark SQL provides a DataFrame API, which is inspired by the data frame concept from the R language. Conceptually, a DataFrame is more or less equivalent to a table in a database, and can be constructed from many different types of data, including tables from external data sources, JSON-files or existing RDDs. Like RDDs, DataFrames are immutable and distributed collections. Unlike RDDs, DataFrames organize data into named columns, which can be accessed by relational operations such as *select, where*, and *groupBy*. These operations are also known as *untyped[16]* operations, in contrast to the strongly-typed RDD operations. Furthermore, DataFrame queries are optimized by Spark SQL's built-in extensible optimizer called *Catalyst*. Listing 1.2 shows a simple Scala program that computes the number of employees with a salary below 5000 in each department, using the DataFrame API of Spark SQL. Listing 1.3 shows a corresponding SQL query. Alternatively, it is also possible to specify a query as an SQL-string in Spark SQL.

```
1 val poorEmps = employees
2     .where($"salary" < 5000)
3     .groupBy($"deptId")
4     .count()
```

Listing 1.2: Spark SQL query that returns the number of employees with a salary below 5000 in each department.

```
1 SELECT deptId, count(deptId)
2 FROM employees
3 WHERE salary < 5000
4 GROUP BY deptId
```

Listing 1.3: SQL query corresponding to the Spark SQL query in Listing 1.2 .

There are suprisingly many ways to refer to columns in a DataFrame query:

- Using the col function, i.e. by writing `col(columnName)`.
- Using the "dollar-sign"-syntax, i.e. by writing `$"columnName"`. This is the syntax used in Listing 1.2, and `$` is simply an alias to the `col()` function.
- Using Scala symbols, i.e. by writing `'columnName`.

An advantage of DataFrames over pure SQL is that they're integrated with a full programming language. This allows developers to break their code up into functions and use control structures (i.e. ifs and loops), which according to Armbrust et al. [15] makes it easier to structure and debug code compared to a purely SQL-based setting.

Catalyst

Catalyst is an extensible query optimizer based on functional programming constructs in Scala. Catalyst differentiates itself from other extensible optimizers by allowing developers to extend the optimizer without needing to specify rules in a separate domain specific language[15]. Instead, all rules can specified in Scala code, with Scala's pattern-matching feature being particularly effective at this.

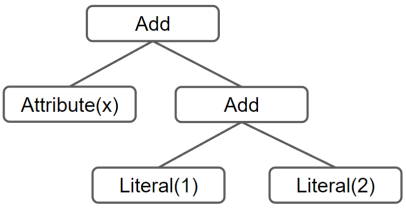

Figure 1.2.: Example of a simple tree representing the expression `x+(1+2)` in Catalyst. The example is taken fromArmbrust et al. [15].

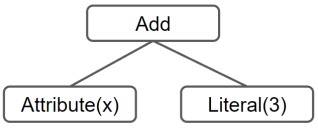

Figure 1.3.: The result of applying the rule in Listing 1.4 to the tree in Figure 1.2.

The foundation of Catalyst is two kinds of data types: *trees* and *rules*. A tree is simply a node object with zero or more child nodes, and every node has a node type, which must be a subclass of the *TreeNode* class. All nodes are immutable, but they can be manipulated by applying rules to them. Figure 1.2 shows an example of a simple tree representing the expression `x+(1+2)`.

A rule is simply a function that maps a tree to another tree. Rules are used to transform trees, and are usually specified by functions that use pattern-matching. Listing 1.4 shows an example of a simple rule in Catalyst that optimizes `Add`-expressions of literals (e.g. by turning 1+2 into 3) by using the pattern-matching feature of Scala. The result of applying this rule to the tree in Figure 1.2 can be seen in Figure 1.3.

```
1 tree.transform {
2     case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
3 }
```

Listing 1.4: Example of a simple rule that optimizes add-statements between literals in Catalyst. The example is taken fromArmbrust et al. [15].

User Defined Functions

Support for UDFs (**U**ser **D**efined **F**unctions) is not something new in the database world. However, UDFs in many database systems have to be specified in a separate programming environment (for example, MySQL requires UDFs to be written in C/C++[17]). In Spark SQL, there's no complicated packaging or registration process required to use UDFs, and they can be registered simply by providing a regular Scala function (or Java/Python functions in their corresponding APIs)[15]. Listing 1.5 shows an example of defining and using a simple UDF that computes the square of a number. UDFs are also important for accessing values of UDTs (User Defined Types). However, UDTs do not appear to

be fully developed yet, as the API for creating UDTs is currently (as of Spark version 2.1.1) private[18], although the API used to be public before Spark version 2.0[19].

```
1 val ss : SparkSession = ...
2
3 val squareUDF = udf( (x : Double) => x*x )
4 val table = ss.table("employees")
5
6 table.select('salary, squareUDF('salary))
```

Listing 1.5: Sample code for defining and using a UDF with Spark SQL. The result would show the salaries and squared salaries of all employees.

1.2 Constrained Optimization

Many real world problems such as energy trading or various kinds of scheduling or routing problems can be modelled as constrained optimization problems. Such problems are about optimizing (i.e. minimizing or maximizing) a given objective function while satisfiying a number of constraints. In this project, we only consider a subset of constrained optimization problem solving, namely **L**inear **P**rogramming (LP) and **M**ixed **I**nteger **P**rogramming (MIP) problems.

1.2.1 Linear Programming

A linear program (or linear optimization problem) is a constrained optimization problem where both the objective function and the constraints are linear. Linear programs are commonly written in *canonical form*, which looks like the following:

$$\begin{aligned} \textit{maximize} \quad & c^T x \\ \textit{subject to} \quad & Ax \leq b \\ \textit{and} \quad & x \geq 0 \end{aligned}$$

where c and b are vectors of known coefficients, A is a matrix of known coefficients, and x is a vector of unknown variables (also referred to as decision variables). Canonical form only allows for $\leq$ constraints and requires that all variables be non-negative, while the objective function must be maximized. However, these restrictions do not cause any loss of generality[20].

Activity Scheduling Example

As a simple example of a linear programming problem, imagine that we have a small software company with n profit-making activities $a_1, ..., a_n$ and m resources $r_1, ..., r_m$. Like most companies, it wants to maximize profits, which it does by scheduling the aforementioned activities in an optimal way without consuming more resources than what is available.

Let's say the company has 5 employees, each working 8 hours per day, meaning the company has a total of 40 man-hours to allocate to activities each day. Being that the employees are software developers, they require a certain amount of coffee to work effectively, so the company has a daily supply of 20 cups of coffee. As such, we can define the types of resources with the vector $r = [Hours, Coffee]^T$, and we let $b = [40, 20]^T$ denote the supply of resources, such that b_i is the daily supply of resource r_i.

In order to make money, the company does a combination of the following three activities:

- Produce new software
- Maintain legacy software
- Do consultancy work

These activities are represented by the vector $a = [Produce, Maintain, Consult]^T$. The company has data suggesting that on average, producing new software earns \$100 per hour, maintaining legacy software earns \$125 per hour, and consultancy work earns \$90 per hour. We represent this information with the vector $c = [100, 125, 90]^T$, such that c_j denotes the hourly profit of performing activity a_j.

Consultancy work does not require any coffee, whereas producing new software on average requires 1 cup of coffee per hour of work. Maintaining legacy software in the company involves working with ancient COBOL code, which is obviously very stressful, requiring 3 cups of coffee per hour of work. We represent this information with the matrix A, such that A_{ij} denotes the hourly amount of resource r_i used when operating activity a_j:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \end{bmatrix}$$

Note that the first row of the matrix A denotes how many man-hours are used per hour of work of each activity (which is obviously 1 regardless of the activity), which is why the row is filled with 1s.

Now, let x_i denote the intensity (in hours) of which we want to perform activity a_i, where $x = [x_1, x_2, x_3]^T$. The problem is now to find suitable values for x such that the profit is maximized, which we can represent as a linear programming problem in canonical form:

$$\begin{aligned} \textit{maximize} \quad & \begin{bmatrix} 100 & 125 & 90 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ \textit{subject to} \quad & \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 40 \\ 20 \end{bmatrix} \\ \textit{and} \quad & \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \geq 0 \end{aligned}$$

Using a linear programming solver on this problem, we get the following solution: $x = [0.0, 6.67, 33.33]$. According to this solution, it is never worth developing new software, and every day the company should instead spend 6.67 hours on maintaining software, and the rest of the time (33.33 hours) should be spent on consulting.

Energy Balancing Example

As a concrete example of an optimization problem related to prescriptive analytics, we consider a problem of balancing energy production and demand. In the area of smart grids, the flexibility of different types of energy producers (e.g. windturbines) and consumers (e.g. dishwashers) plays an important role in maximizing the effectiveness of renewable energy sources. A way to represent the flexibility in energy production and

demand is with so-called *flex-objects*, where a flex-object describes how much energy is needed, when it is needed, and how much flexibility is tolerated regarding time and amount. [21]. For this example, we use a simplified version of flex-objects that do not consider time-flexibility, such that a flex-object f consists of a sequence of m 2-tuples $f = \{n_1, n_2, ..., n_m\}$, where each tuple $n_i = (e_{min,i}, e_{max,i})$ represents the minimum and maximum energy values that the flex-object can produce/consume at a specific time, and m is the amount of time intervals covered by the flex-object. For a more detailed explanation of flex-objects and their relation to smart grids, see the report of the previous project[1].

Assume we have n flex-objects $f_1, ..., f_n$ representing the flexible demand and supply of energy producers/consumers, where each flex-object has m time intervals $t_1, ..., t_m$. We let $e\_min_{i,j}$ and $e\_max_{i,j}$ denote the minimum and maximum energy amount of a flex-object f_i in time interval t_j. We would like to schedule these flex-objects such that we balance the supply and demand optimally, i.e. minimize the difference between production and consumption at each time interval. We let $e_{i,j}$ denote the scheduled amount of energy to be produced/consumed (we say that energy is produced if $e_{i,j} < 0$, and consumed if $e_{i,j} \geq 0$) by flex-object f_i at time interval t_j.

To obtain the optimal balancing, we want the sum of $e_{i,j}$ for all production and consumption flex-objects to be as close to 0 as possible in each time interval, without scheduling anything outside the min/max bounds of any flex-object. This is equivalent to assigning values to all $e_{i,j}$, such that the total energy production and consumption are as close to each other as possible in each time interval. We can rephrase this as the following constrained optimization problem: (example taken from Šikšnys and Pedersen [5]):

$$\begin{aligned} &\textit{minimize} && \sum_{j=1...m} |\sum_{i=1...n} e_{i,j}| \\ &\textit{subject to} && e\_min_{i,j} \leq e_{i,j} \leq e\_max_{i,j}, i = 1, ..., n,\ j = 1, ..., m \end{aligned}$$

However, the objective function in this example makes use of absolute values, which makes it non-linear. Fortunately, by introducing additional variables and constraints, we can rewrite the problem into an equivalent problem that does not use absolute values[22]. An equivalent problem without absolute values could look like the following:

$$\begin{aligned} &\textit{minimize} && \sum_{j=1...m} t_j \\ &\textit{subject to} && e\_min_{i,j} \leq e_{i,j} \leq e\_max_{i,j}, i = 1, ..., n,\ j = 1, ..., m \\ & && \sum_{i=1...n} e_{i,j} \leq t_j,\ j = 1, ..., m \\ & && \sum_{i=1...n} e_{i,j} \geq -t_j,\ j = 1, ..., m \end{aligned}$$

As the objective function and all the constraints are now linear, the problem can be solved as a linear programming problem. We will revisit this example later in the experiments section.

Decomposition of Linear Programs

Many optimization problems exhibit special structure[23] that allows for "shortcuts" in solving the problem. In particular, some optimization problems can be seen as a combination of *independent subsystems*, meaning that the constraints of the problem can be divided into partitions, where the constraints in each partition only involve a subset of decision variables that is disjoint from every other partition's subset of variables. These partitions can be formulated into separate optimization problems that can be solved independently, and as the complexity of linear optimization problems generally grows polynomially with the size of the problem, this can provide significant performance benefits. Also, as the subproblems are completely independent, they can be solved in parallel. Section 4.4 shows how this decomposition method can be implemented.

1.2.2 Mixed-Integer Linear Programming

In LP problems, the solutions we assign to the decision variables are allowed to be continuous. However, if we restrict 1 or more variables in an LP problem to be integers, the problem is called a **M**ixed-**I**nteger **P**rogramming (MIP) problem instead. Interestingly, imposing such restrictions makes the problem significantly harder to solve, as MIP problems are NP-hard in the general case[24], whereas LP problems can be solved in polynomial time[25]. If we again consider the example of scheduling activities in a software company (Section 1.2.1), but where the decision variables (i.e. how many hours to assign to each activity) are constrained to have integer values, the solution to the problem changes from $x = [0.0, 6.67, 33.33]$ to $x = [2, 6, 32]$.

2 Related Work

In this chapter, we will briefly look at some related work in the area of DBMSes with integrated support for constrained optimization.

2.1 SolveDB

Many PA applications require the solving of optimization problems based on data in relational databases, so it would be convenient if optimization problems could be defined and solved directly in SQL. Šikšnys and Pedersen [5] tackle this problem by proposing SolveDB, which is an RDBMS with integrated support for constrained optimization problems. SolveDB allows users to specify and solve optimization problems through so-called *solve queries*, which are written in an SQL-like language. As of the publication of the SolveDB article in 2016, only a PostgresSQL implementation exists.

An example of a solve query for solving the knapsack problem with a maximum allowed weight of 15 based on data in the *items* table can be seen in Listing 2.1. The rows of the *items* table can be seen in Table 2.1, and the result of running the query in Listing 2.1 can be seen in Table 2.2.

```
1 SOLVESELECT quantity IN (SELECT * FROM knapsack_items) AS
     r_in
2 MAXIMIZE (SELECT sum(quantity*profit) FROM r_in)
3 SUBJECTTO (SELECT sum(quantity*weight) <= 15 FROM r_in),
4           (SELECT quantity >= 0 FROM r_in)
5 USING solverlp();
```

Listing 2.1: SolveDB solve query for the knapsack problem with a maximum allowed weight of 15. The example is taken from a presentation by Laurynas Siksnys.

The query in Listing 2.1 can be divided into four parts, where each part is preceeded by a specific keyword:

item_name	weight	profit	quantity
'item 1'	6.0	10.0	NULL
'item 2'	4.0	7.0	NULL
'item 3'	4.5	8.0	NULL
'item 4'	8.0	13.0	NULL

Table 2.1.: Rows of the *items* table before running the solve query in Listing 2.1.

item_name	weight	profit	quantity
'item 1'	6.0	10.0	1
'item 2'	4.0	7.0	0
'item 3'	4.5	8.0	2
'item 4'	8.0	13.0	0

Table 2.2.: Rows returned by the solve query in Listing 2.1.

1. `SOLVESELECT` - Defines the beginning of the solve query, and specifies which columns hold decision variables (the *quantity* column) of the problem as well as the input relation (`SELECT * FROM items`).

2. `MAXIMIZE` (can also be MINIMIZE) - Defines the objective function of the optimization problem (maximize the sum of $quantity * profit$ in the *items* table).

3. `SUBJECTTO` - Defines the constraints of the optimization problem (the total weight must be smaller than or equal to 15, and *quantity* must be greater than or equal to 0).

4. `USING` - Defines what kind of solver is used to solve the optimization problem (`solverlp` refers to a linear programming solver). SolveDB comes bundled with a set of default solvers, but it is also possible to extend SolveDB with user-defined solvers.

SolveDB will automatically assign values to the columns that hold decision variables (these will be referred to as *decision columns* in the rest of the report) in the query, while conforming to the defined constraints and objective function. Furthermore, since the data type of the quantity column in Listing 2.1 is integer, SolveDB will only attempt to assign integer values to the quantity column, meaning that SolveDB will treat this as a MIP problem. If quantity was defined as a float instead, SolveDB would treat this as an ordinary LP problem.

2.1.1 Problem Partitioning

As not all solvers support partitioning of optimization problems, SolveDB employs built-in partitioning at the relational level, splitting a problem into a number of smaller sub-problems that can be solved independently. This partitioning method is the same method that is explained in Section 1.2.1, which partitions constraints based on disjoint sets of

variables. Although SolveDB does not solve these partitions in parallel, partitioning still provides very significant performance gains.

2.2 Tiresias

Tiresias[6] is a system that integrates support for constrained optimization into a DBMS through so-called *how-to queries*. These queries are also referred to as a type of *reverse data management* problem, since a how-to query specifies certain *rules* for the output of the query, and then the DBMS has to find new values for some of the data such that the rules are upheld. Conceptually, how-to queries are similar to SolveDB's solve queries, but whereas SolveDB uses an SQL-like syntax, how-to queries in Tiresias are written in a language called TiQL (**T**iresias **Q**uery **L**anguage), which is an extension of the Datalog language. However, whereas SolveDB's solve queries can be used easily alongside ordinary SQL-queries, it appears that you need to use both SQL and TiQL if you want to do how-to queries and ordinary queries (i.e. data management queries) in Tiresias[5].

A TiQL query is written as a set of rules, which define *hypothetical tables*. Together, these hypothetical tables form a *hypothetical database*. A hypothetical table has non-deterministic semantics, such that there are a number of "possible worlds" for the table defined by the constraints, where Tiresias chooses the possible world (i.e. possible configuration of data in the hypothetical table) that results in a specified objective function being maximized or minimized.

Attributes in a hypothetical table can be either known or unknown in Tiresias, with attributes being labelled as unknown by appending a '`?`' to the attribute name. Known attributes have values from the database, whereas unknown variables are non-deterministically assigned values (i.e. they are assigned values by the underlying MIP solver). Listing 2.2 shows an example of what a TiQL query looks like.

```
1  HTABLES:
2    HLineItem(ok,pk,sk,q?)        KEY:(ok,pk,sk)
3    HS(ok,pk,sk,qnt,q?)           KEY:(ok,pk,sk)
4    HOrderSum(ok,sk,c?)           KEY:(ok,sk)
5  RULES:
6    HLineItem(ok,pk,sk,q?)     :- LineItem(ok,pk,sk,qnt)
7    HS(ok,pk,sk,qnt,q?)        :- HLineItem(ok,pk,sk,q?)
8                                  & LineItem(ok,pk,sk,qnt)
9    [q? <= qnt]                <- HLineItem(ok,pk,sk,q?)
10                                 & LineItem(ok,pk,sk,qnt)
11   HOrderSum(ok,sk,sum(q?))   :- HLineItem(ok,pk,sk,q?)
12   [c? <= 50]                 <- HOrderSum(ok,sk, c?)
13 MINIMIZE(sum(qnt-q?)) :- HS(ok,pk,sk,qnt,q?)
```

Listing 2.2: TiQL query that decreases quantities of line items for a shipping company in order to achieve a desired KPI (**K**ey **P**erformance **I**ndicator). The example is taken directly from the Tiresias article[6].

A how-to query is translated into a MIP problem, which Tiresias can partition into smaller independent subproblems, where each subproblem is stored as a file. Since each subproblem is stored as a separate file, the system can become I/O bound for problems with a large amount of small partitions. Because of this, Tiresias utilizes *partition grouping*, resulting in some partitions consisting of more than 1 MIP problem. Tiresias also features other optimizations, such as variable elimination, where redundant variables are removed from the underlying optimization problem.

The first version of Tiresias is implemented in Java, and uses PostgreSQL as the RDBMS and GLPK (**G**NU **L**inear **P**rogramming **K**it) as the MIP solver. The authors of Tiresias claim that most parts of Tiresias are parallelizable, but the current implementation does not make use of this.

2.3 Searchlight

The authors of Searchlight[26] claim that existing tools for interactive search, exploration and mining of large datasets are insufficient, as traditional DBMSes lack support for optimization constructs and interactivity. This usually means that users have to "roll their own solutions" by gluing together several specialized libraries, scripts and databases in an ad-hoc manner, resulting in solutions that are difficult to scale and maintain. To rememdy this, the authors propose Searchlight, a system that combines the capabilities of array DBMSes with Constraint Programming (CP). Constraint programming is similar to constrained optimization, except there isn't necessarily an objective function to maximize/minimize, meaning that constraint programming looks for feasible solutions rather than optimal solutions[27]. Constraint programming problems are also known as *constraint satisfaction problems*.

Searchlight is implemented with SciDB as the underlying DBMS, and uses Google Or-Tools for CP solving. Searchlight allows CP solving to run efficiently inside the DBMS with so-called search queries, which consist of constraint programs that reference DBMS data.

2.3.1 Speculative Solving

Whereas many existing CP solvers make the assumption that all data is main-memory resident, Searchlight uses a novel method called *Speculative Solving* to operate on *synopses* that fit in main-memory instead of operating on the whole dataset. These Synopses are a type of lossy compression that allows for approximate answers known as *candidate solutions* to Searchlight API calls. These candidate solutions are guaranteed to include all correct solutions (meaning there are no false negatives), but there can be false positives which have to be filtered out later.

2.3.2 Distributed Execution

The search process of Searchlight can be executed in parallel on a cluster by using the built-in distribution capabilities of SciDB. The search process consists of 2 concurrent

phases: *solving* and *validation*. The solving phase performs speculative solving to find candidate solutions, while the validation phase filters out false positives in the candidate solutions by checking the solutions up against the real data. These phases are not required to be performed on the same nodes in the cluster, meaning that solvers can be put on CPU-optimized machines, while validators can be put on machines closer to the data.

To allow for distributed execution, Searchlight partitions the search space by viewing it as a hyper-rectangle, which can be sliced along one of its dimensions to produce evenly sized pieces, which can then be distributed to the solvers of the cluster.

Part II.

Technical Contribution

3 Design

This chapter details the design of SolveDF. The idea behind SolveDF is to extend Spark with functionality that allows for seamless integration of constrained optimization problem solving with data management. To attain this, Spark SQL is extended to support SolveDB's concept of solve queries. The primary advantages of implementing solve queries with Spark SQL are:

- **Scalability** - Some optimization problems can be decomposed into independent subproblems that can be solved in parallel, and Spark makes it easy to solve these subproblems in parallel on a cluster.
- **Integration with a powerful analytics engine** -You get to use solve queries in the same environment as many other analytics tasks. Spark is a powerful analytics tool, having libraries for tasks such as machine learning, stream processing and graph processing.
- **Data source independence** - Spark SQL supports several different types of data sources, including any JDBC compliant database, JSON files or HDFS (e.g. through Apache Hive).

3.1 Requirements

This section specifies the requirements for SolveDF.

3.1.1 Declarative Specification

It should be possible for users to define and solve optimization problems with declarative queries (referred to as solve queries), using syntax that is similar to ordinary Spark SQL queries. Specifically, the user should be able to specify constraints and objective functions as DataFrame queries. A solve query should consist of the following parts:

- An input DataFrame (known as an input relation in SolveDB)
- A set of decision columns (i.e. columns whose values represent unknown variables) in the input DataFrame
- 0 or 1 objective queries
- A number of constraint queries

The result of evaluating (i.e. solving) a solve query should be the original input DataFrame, but where the values of the decision columns have been replaced by values that satisfy the constraints (specified by the constraint queries) and where the value of the objective (specified by the objective query) is optimal (i.e. minimal or maximal).

3.1.2 Solver Independence

SolveDF should be able to solve general LP and MIP problems. Although Spark has some built-in optimization functionality ([1] goes into more detail with this) from MLlib, this is mostly tailored to machine learning. Instead, we will be relying on external solvers for optimization problem solving, but the implementation should not be tied to a single solver. As Spark is written in Scala, we can use solvers that are written in Scala or Java, or solvers that have Java/Scala bindings. However, if we use a solver written in a non-JVM langauge (e.g. C or C++) with Java/Scala-bindings, we need to make sure the native libraries are available on all nodes in the Spark cluster.

We treat the external solvers as black-boxes, and we want the system to be independent of specific solvers, such that it is easy to integrate multiple solvers if desired. This way, the system can also more easily be extended to support other classes of optimization problems than LP and MIP problems (e.g. quadratic programming problems) in the future. Therefore, the solution should offer an interface for integrating solvers.

Ideally, the system should be able to automatically choose a suitable solver to use when given an optimization problem, but it is deemed sufficient that this can be specified manually by the user.

3.1.3 Decomposition of Problems

As shown by both Tiresias and SolveDB, decomposition of LP/MIP problems into smaller subproblems can improve performance significantly, even when the subproblems are solved serially. Moreover, as the subproblems can be solved independently from each other, we can easily parallelize the solving with Spark. Therefore, the system should be able to decompose optimization problems, solve the subproblems in parallel, and combine the solutions to the subproblems into a solution to the original problem. This neccesitates that the external solvers are able to solve separate optimization problem instances in parallel (though they do not need to be able to parallelize the solving of a single problem instance), so that one machine in the Spark cluster can solve multiple subproblems at the same time.

3.1.4 No Modification of Spark

Spark is a very actively developed project, so it is important that the implementation of SolveDF doesn't change the source code of Spark, but rather builds on top of Spark. Not only would it be very inconvenient if users had to install a seperate version of Spark to use SolveDF, but it would be very hard to keep SolveDF updated as new features are added to Spark.

3.2 SolveDB Usability Evaluation

As SolveDB is the main inspiration for SolveDF, it makes sense to gain insight into whether SolveDB is intuitive to use, and to identify ideas and concepts of SolveDB that would make sense to use in the design of SolveDF, as well as what parts could be improved on. For this reason, a usability experiment of SolveDB is conducted. This experiment is not supposed to be an extensive or thorough study of SolveDB, but rather a quick way to get an idea of whether SolveDB is intuitive.

3.2.1 Method

The experiment is based on the Discount Method for Evaluating Programming Languages[28] (referred to as DMEPL in the rest of the report). This is a work in progress method inspired by the Discount Usability Evaluation (DUE) method[29] and Instant Data Analysis (IDA) method[30]. According to the authors, DUE is best applied when evaluating a full language with a corresponding IDE and compiler, but less effective for evaluating languages in the early design phases. This is primarily because it is hard to separate feedback on the IDE and the language itself. On the other hand, DMEPL is intended to be method that doesn't require the tested language to be fully designed or implemented, making the method applicable in the early stages of programming language design. Like DUE, DMEPL recommends no more than five participants for the test, making it a low-cost, lightweight method. The specific setup of the experiment is reletively flexible, and can vary from pen and paper to a full-blown usability lab.

The procedure of DMEPL can roughly be summarized as the following:

1. Create tasks specific to the language being tested. These are the tasks that the participants of the experiment should solve.

2. Create a short sample sheet of code examples in the language being tested, which the participants can use as a guideline for solving the tasks.

3. Perform the test on each participant, i.e. make them solve the tasks defined in step 1.

4. Interview each participant briefly after the test, where the language and the tasks can be discussed.

5. Analyze the resulting data to produce a list of problems.

During the test, it is important that it is made clear to the participants that it is the language that is being tested, and not the participants themselves. The participants should also be encouraged to think-aloud while trying to solve the tasks. The faciliator of the experiment is allowed to answer any questions related to the language, and he is also allowed to discuss the solutions with the participants, as it is not the partcipants' ability to formulate solutions we are testing.

3.2.2 Setup

As prescribed by the DMEPL method, a task sheet and sample sheet was provided to the participants. The sheets can be seen in Appendix A.1 and Appendix A.2, respectively. A discrepancy from the original method is that the sample sheet in this experiment had more than just code examples. Specifically, the sample sheet included a very brief introduction (~3 lines of text) to SolveDB and a simplified syntax description of SolveDB. There was also included a short description (~5 lines of text) of what a constrained optimization problem is, and a very small example of a linear programming problem. This was to give the participants an intuition of what tasks they were to solve, as the participants were not expected to be familiar with constrained optimization problems prior to the experiment.

The participants were asked to solve 3 different tasks, which were estimated to take approximately an hour to solve in total. The tasks were designed to be rather simple (e.g. none of them required the involvement of joins in the solution), as the test was aimed at users that were familiar with SQL, but unfamiliar with constrained optimization problems.

3.2.3 Results

The experiment was conducted on three different participants, each having to solve the same tasks. The experiment is divided into 3 test cases, where each test case consists of one participant solving the tasks. All of the participants were familiar with SQL, and none of them had any prior experience with SolveDB. None of the participants were familiar with linear programming or constrained optimization problem solving, although the participant in test case 1 remarked that he had "tried putting a sequence of linear equations into a solver once". The participants in test cases 2 and 3 were both 10th semester computer science students, while the participant in test case 1 was a computer science Phd-student. In the following, a summary of each test case is given, and then a categorization of the problems encountered is presented.

Test Case 1

It should be noted that this test case functioned as a "pilot" for the experiment, meaning that the task sheet and sample sheet for this case were slightly different from the ones listed in the Appendix. These were primarily minor changes to make the exercises a little more clear for test cases 1 and 2.

Task 1 The participant mostly grasped the basic syntax within a few minutes by looking at the first example query on the sample sheet. However, while formulating a solution to the first task, the meaning of the first part of the query (the part that specifies decision columns and the input relation) was unclear, resulting in the test person mixing up decision columns and input columns.

Task 2 Most parts of the second task were solved without issues, but formulating the objective query proved challenging, and was easily the most time-consuming part of solving the task. Eventually, the participant figured out how to formulate the query by looking at some other examples in the sample sheet. There also occurred a number of confusing error messages from the compiler during this task. It was also not initially clear that the objective query must always return only 1 row, but this quickly became clear to the test person. The participant also hadn't considered that he could use constants in the objective query.

Task 3 For the third task, there were a significant amount of confusing error messages from the compiler, mostly because many of the error messages referred to internal SolveDB types (e.g. `lp_ctr` and `lp_functional`). The participant remarked that the error messages seemed directed more towards the person who made the compiler than the user. Other than that, there weren't any significant issues in formulating the query.

Follow-up interview During the follow-up interview, the participant said that the syntax was generally understandable, and that the structure of the query made sense, as it was clear how the syntactical constructs corresponded to the mathematical definition of a constrained optimization problem. The participant also felt that the language could use some more syntactical sugar, as a lot of the syntax seemed superfluous for the tasks in the experiment. In particular, each constraint query and objective query in all cases used data from only a single table, so having to write "FROM r_in" after every constraint seemed unnecessary. Likewise, the SELECT keyword could perhaps be omitted as well, as it seemed to make the language unnecessarily verbose, and thereby harder to read. The participant acknowledged that having all these keywords would make sense in bigger queries that involve more than a single table, but for queries that only make use of 1 table, the extra syntax seemed superfluous.

The participant also remarked that the `SOLVESELECT` keyword was confusing. The participant expected that the definition of columns in the input relation should immediately follow the keyword, due to "`SELECT`" being part of the keyword. The participant said that it might be more clear if it was just called `SOLVE` instead.

Test Case 2

In general, the participant didn't seem to properly understand what the point of the `SELECT` statement after the `IN` keyword was. This was an issue for all three tasks.

Task 1 For the first task, the participant had some issues formulating the objective query, as it was not clear that the objective query had to return only a single row (e.g. by using an aggregate function). Initially, the participant mistook the `SOLVESELECT` keyword for a `SELECT` keyword. When solving Task 1b, it was immediately clear to the participant that he just had to insert a simple WHERE statement in the previous query.

Task 2 In the second task, the primary issue was formulating the objective query, which took a lot of time to get right.

Task 3 The participant didn't run into any issues when solving the third task, other than having to rethink some of his constraint/objective function defintions, but in most cases he could quickly figure out what was wrong on his own.

Follow-up interview In the follow-up interview, the participant thought that having to write "`FROM <table>`" after every constraint and objective query seemed annoying and unnecessary. Likewise, many of the `SELECT`s and the "`WITH <solver()>`" seemed unnecessary. The `SOLVESELECT` keyword was also slightly confusing, as the participant first thought it was just an ordinary `SELECT`-statement.

It was not intuitive that AND/OR could not be used in constraints (e.g. "quantity = 0 OR quantity = 1" is not valid syntax for defining constraints). The participant also acknowledged that some of the things that seemed unintuitive to him may be caused by his inexperience with constrained optimization problems.

Test Case 3

Task 1 The participant had some issues understanding the task. It took some time to understand that he only had to define a solve query to solve the task, and didn't need to do any INSERT statements or JOINS to obtain the desired result. It also wasn't initially clear why the objective function should return a single row.

Task 2 In the second task, there were no issues specifying the constraints. However, specifying the objective function took a very long time, with one of the big reasons for this being that he hadn't considered the ABS-function. As the solving of this task was dragging on for a long time, it was decided to skip the rest of task 2 (The only thing missing from his solution was inserting an ABS-function).

It should also be noted that there were some techincal difficulties towards the end of task 2 with the virtual machine running SolveDB, so the rest of the experiment was conducted using Notepad++ instead of pgadmin. This did not seem to have any significant impact on solving the rest of the tasks.

Task 3 In contrast to the previous two tasks, the participant solved the third task very easily and quickly. The participant only used about 5 minutes in total (out of the approximately 60 minutes that the experiment took in total) for task 3a, and his first attempt to write the solve query was almost 100% correct, having only a single error (there was a missing SUM in one of the constraint queries), but he figured this out in less than a minute. In task 3b, the participant tried to use OR incorrectly in a constraint query ("SELECT quantity = 0 OR quantity = 1"), but otherwise he had no issues.

Follow-up interview The participant expressed that the concepts and different parts of a solve query were confusing at first, but once he had tried them and had "gotten the hang of it", they made a lot of sense to him, and he said that if he was given another similar task, he could probably solve it relatively easily (which was exemplified by how easily he solved the third task compared to the first two tasks). Because of this, he also explicitly said that he thinks SolveDB has a minimal learning curve.

The participant remarked that he was overthinking the solutions to the first two tasks, and said that he was thinking too much in a procedural manner rather than declarative when solving task 2 (He specifically mentioned how he wanted to insert an if-statement at one point). In regards to the syntax, he didn't think that the name of the SOLVESELECT keyword made sense. Either "`SOLVE`" or "`SOLVE FOR`" seemed more appropriate. He didn't have any other issues with the syntax, and he mentioned that it was nice how the syntax "looks like SQL".

Another small thing was that the syntax-description of a solve query (from the sample sheet) showed that defining an alias for the input relation was optional, when it is in fact required.

Summary of Test Cases

Table 3.1 shows a table of problems encountered in the test cases, categorized according to their severity. For this categorization, the following definitions of *Cosmetic*, *Serious* and *Critical* problems are used:

Cosmetic problems Problems such as typos and small character deviations, i.e. problems that can be easily fixed by replacing a single wrong part.

Serious problems Problems that can be fixed with a few changes, such as minor structural errors or misunderstandings of a single language construct.

Critical problems Problems that require a revision of the code, e.g. large structural errors or fundamental misunderstandings of how to structure code in the language.

3.2.4 Discussion

While there are a number of problems categorized as serious, some of these problems are arguably caused more by the participants' inexperience with constrained optimization concepts (e.g. decision variables, constraints, objective function) than the language itself, such as problems B1 and B5. Furthermore, for the most part, these problems only happened once or twice (problem B2 being a notable exception, as this was a particularly consistent problem in test case 2), which could indicate that learning to avoid these errors doesn't require a lot of effort. This is especially supported by the dramatic improvement shown by the participant in test case 3 when solving task 3. Likewise, all participants remarked that they generally thought the structure and logic of solve queries made sense after they made it through all the tasks.

Critical	Serious	Cosmetic
	B1: Not understanding what columns to write directly after a `SOLVESELECT` keyword.	**C1**: Mistaking the `SOLVESELECT` keyword for a `SELECT` keyword
	B2: Not understanding what columns to select from the input relation.	**C2**: Forgetting to write `SELECT` at the beginning of a constraint, or `FROM <table>` at the end of a constraint
	B3: Not aggregating an expression with `SUM` in constraint queries or objective queries	
	B4: Trying to use `AND` or `OR` keywords in constraint queries	
	B5: Writing an objective query that returns more than 1 row	
	B6: Not realizing that the `ABS` function can be used in an objective query	
	B7: Not considering that constants could be used in an objective query	

Table 3.1.: Categorization of problems encountered from the SolveDB test cases. Problems related to error messages given by the compiler are not considered here, as these are problems that are related to the implementation rather than the language itself.

Although problem C2 might seem like only a minor annoyance, as it is easy to recognize and correct, the participants in test cases 1 and 2 both mentioned this problem in the follow-up interview and during the tasks. They had to remind themselves several times to write `SELECT` and `FROM` between constraints, indicating that the language could be more intuitive if it wasn't required to always write `SELECT` and `FROM` between constraints.

It is important to keep in mind that the tasks in this experiment only required the writing of small queries that use data from a single table. Realistically, solve queries can be much larger and involve more complicated objectives and constraints (constraint queries can be significantly more complex if they for example involve multiple joins and subqueries). However, even though the tasks are examples of easy problems to solve with SolveDB, the results still look promising for SolveDB, as the participants did not have prior experience with constrained optimization or SolveDB, yet could still figure out how to solve these problems with minimal guidance.

Finally, it should be mentioned that although all the participants spent a significant amount of time specifying the objective function in task 2, this was not unexpected, as that task was intentionally made to test if the partcipants could formulate more complex objective functions in SolveDB. In hindsight, this task should arguably have been made more clear in its definition of the objective, as figuring out how to express the objective mathematically seemed to require more effort than writing the query itself. As soon as the participants realized that they needed to use a sum of absolute values in the objective, the task was solved quickly.

3.2.5 Conclusion

It appears that the basics of SolveDB can be learned relatively easily with minimal guidance, and that the overall concept of solve queries makes sense. As a result of this, it is decided that SolveDF will use similar syntax and the same query-structure as SolveDB. However, unlike SolveDB, I will address problem C2 by making it optional to write the '`SELECT`' and '`FROM`' part of a constraint/objective query whenever the input relation is selected from directly (i.e. without a `WHERE`-statement or join), as this problem came up a lot and seems relatively easy to address. Likewise, I will attempt to find a more intuitive name for the `SOLVESELECT` keyword.

3.3 Architecture

Figure 3.1 shows an overview of the primary components of SolveDF. In this design, most of the logic is located on the Spark driver program, which is responsible for translating solve queries into optimization problems, partitioning the generated optimization problems into smaller subproblems, and combining the solutions to the subproblems into a final result. The nodes in the cluster only have one responsibility, which is to solve subproblems sent to them by the driver. The responsibilities of the primary components are explained in the following.

3.3.1 Query Processor

This component is responsible for reading a given solve query as input, and formulate the given query into a constrained optimization problem by using data extracted with Spark SQL.

3.3.2 Decomposer

This component partitions a given optimization problem into smaller, independent subproblems that can be solved in parallel. These partitions are distributed across the cluster for parallel solving. The decomposition process runs in the Spark driver program, meaning that all the data (i.e. constraints and objective) has to be collected on a single node (the Driver node) before it can be partitioned.

3.3.3 Solver Adapter

The Solver Adapter is responsible for providing a uniform interface to the external solvers. For this to work, it is required that the external solvers are installed on all nodes in the cluster.

3.3.4 Solution Combiner

When solutions have been found for the subproblems, the Solution Combiner collects these solutions. Afterwards, the solutions are inserted into the decision columns of the input DataFrame from the original solve query, yielding the final result.

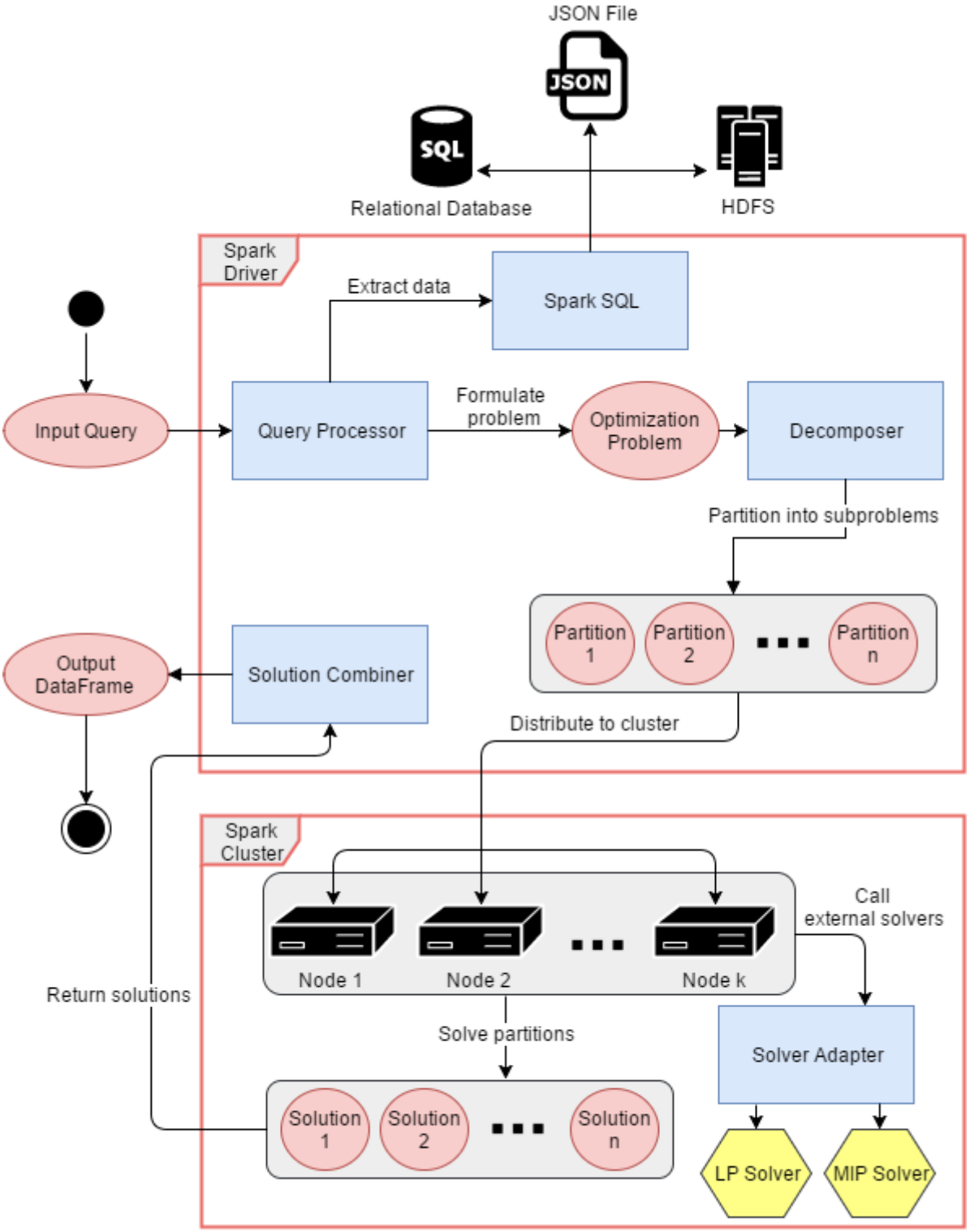

Figure 3.1.: Architecture diagram of SolveDF.

4 Implementation

This chapter documents the result of implementing a prototype of SolveDF as specified by the design in Chapter 3. The general structure of solve queries in SolveDF is presented, and examples of solve queries are shown and compared to equivalent queries in SolveDB. Solutions to some of the major challenges encountered during the implementation will also be presented.

4.1 Structure of a SolveDF Query

The structure of a solve query in SolveDF is modelled closely to SolveDB's structure, but adapted to fit the syntax of Spark's DataFrame API. Listing 4.1 shows an example of a simple solve query.

```
1 val inputDF = spark.table("someTable") // Input DataFrame
2
3 inputDF.solveFor('x, 'y)    // The decision columns
4 .maximize(in => in.select(sum('x)) )    // The objective
5 .subjectTo( // The constraints:
6     in => in.select(sum('y) >= sum('x)),
7     in => in.select('x + 'y <= 10,
8                     'x >= 0,
9                     'y >= 0))
```

Listing 4.1: An example of a solve query in SolveDF.

Objectives and constraints are specified by queries on instances of the *SolveDataFrame* class. A `SolveDataFrame` just represents a `DataFrame` where some of the columns are decision columns (i.e. they contain decision variables), and it supports most of the same operations (e.g. `select`, `where`, `join`) as a normal `DataFrame`, so for the most part it is indistinguishable from a `DataFrame`. The exact details of why this class is used instead of regular `DataFrames` is explained in Section 4.3.

A solve query is initialized by calling the `solveFor` method on an existing `DataFrame`, which returns an instance of the `SolveQuery` class. The objective and the constraints of the `SolveQuery` can then be defined with a builder-like pattern, using the `minimize`, `maximize` and `subjectTo` methods. Similar to SolveDB's equivalent keywords, the `subjectTo`, `maximize` and `minimize` methods take queries (i.e. `SELECT`-statements) as input. In SolveDF, such a query is defined by a lambda of type (`SolveDataFrame`) => `SolveDataFrame`, where the parameter of the lambda represents the input `DataFrame` of the solve query, and the body of the lambda is a query on the input `DataFrame`. The point of this lambda is to hide that the input `DataFrame` is wrapped into a `SolveDataFrame`, and to provide a way to reference this `SolveDataFrame` without storing it in a variable outside the query.

Although the general query structure of SolveDF is very similar to SolveDB, there are some notable discrepancies from SolveDB's syntax:

1. SolveDF's equivalent of SolveDB's `SOLVESELECT` keyword is called `solveFor` instead.

2. In SolveDB, the objective must be specified before the constraints. In SolveDF, the constraints and objectives can be specified in any order (you can even specify constraints first, then the objective, and then more constraints after that).

3. Constraint- and objective queries can be expressed purely as column expressions, i.e. without specifying a source table or `SELECT`-statement. In this case, the expressions are implicitly selected from the input `DataFrame` of the solve query. For example, the following constraint query:
   `.subjectTo(input => input.select(sum('salary) <= 1000))`
   can be written with the following shorthand notation instead:
   `.subjectTo(sum('salary) <= 1000)`
   This shorthand notation was made as a direct consequence of the feedback from the SolveDB usability evaluation in Section 3.2. Listing 4.2 shows an equivalent query to the one in Listing 4.1, but where the objective query and all the constraint queries make use of this shorthand notation instead.

```
1 val inputDF = spark.table("someTable") // Input DataFrame
2
3 inputDF.solveFor('x, 'y)     // The decision columns
4 .maximize( sum('x) )     // The objective
5 .subjectTo( // The constraints:
6     sum('y) >= sum('x),
7     'x + 'y <= 10,
8     'x >= 0,
9     'y >= 0)
```

Listing 4.2: An alternative version of the query in Listing 4.1 that uses shorthand notation for constraint- and objective queries.

4.1.1 Alternative Solve Query Formulation

Instead of using the `solveFor` method on an existing `DataFrame` to create a `SolveQuery`, you can explicitly invoke the constructor of the `SolveQuery` class as shown in Line 4 of Listing 4.3. The `SolveQuery` constructor takes a `SolveDataFrame` (representing the input `DataFrame` of the `SolveQuery`) as an argument, meaning that you also have to explicitly create a `SolveDataFrame` by calling the `withDecisionColumns` method on a `DataFrame`, as seen on Line 2 of Listing 4.3.

```
1 val inputDF = spark.table("someTable")
2 val solveDF = inputDF.withDecisionColumns('x, 'y)
3
4 SolveQuery(solveDF)
5   .maximize( solveDF.select(sum('x + 'y)) )
6   .subjectTo(
7     solveDF.select('x + 'y >= 0),
8     solveDF.select('x + 'y <= 10))
```

Listing 4.3: A Scala program that explicitly creates a `SolveDataFrame` outside of the solve query.

As you explicitly create a SolveDataFrame from the input `DataFrame` with this approach, you can store that `SolveDataFrame` in a variable, which allows you to reference it directly in a constraint- or objective query (i.e. you do not need to provide a lambda as parameter) as shown in Lines 5-8 of Listing 4.3.

4.1.2 Using Multiple Tables in a Solve Query

In SolveDB, you can include additional tables that have decision columns in a solve query by using CTEs (`WITH`-statements). In SolveDF, you can accomplish the same thing by creating `SolveDataFrames` for any additional `DataFrames` (using the `withDecisionColumns` method), and simply reference them in the constraints/objective. Listing 4.4 shows an example of this, where an additional `SolveDataFrame` is used in a join in the solve query. This is also used in the flex-object scheduling query presented in Section 4.2.

```
1 val inputDF = spark.table("someTable")
2 val extraSolveDF =
     spark.table("someOtherTable").withDecisionColumns('a)
3
4 inputDF.solveFor('x, 'y)
5   .maximize( sum('x) )
6   .subjectTo(
7     in => in.join(extraSolveDF, in("someColumn") ===
        extraSolveDF("someOtherColumn"))
8           .select('x <= 'a))
```

Listing 4.4: Example showcasing the use of additional `DataFrames` with decision columns in a solve query.

4.1.3 Retrieving the Solution to a Solve Query

To solve a `SolveQuery`, you can explicitly invoke the `solve` method of the `SolveQuery`, which will return a `DataFrame` with the result. The result is the original input `DataFrame`, but where the columns designated as decision columns have been assigned new values that satisfy the specified constraints and maximize/minimize the objective function.

`SolveQuery` also supports implicit conversion to a `DataFrame` (using Scala's concept of implicit classes[31]), meaning that you don't even need to call the `solve` method to get the result. To define which solver is used for solving the query, the `withSolver` method can be called, as illustrated on Line 8 of Listing 4.5. If no solver is explicitly defined for the query, it will use the default solver (Currently, this is `GlpkSolver`).

```
1 val inputDF = spark.table("someTable")
2
3 inputDF.solveFor('x, 'y)
4   .maximize( sum('x) )
5   .subjectTo(
6     'x + 'y >= 0,
7     'x + 'y <= 10)
8   .withSolver(ClpSolver)
9 .solve() // The solve() call is optional
```

Listing 4.5: Example of a solve query where the external solver is specified explicitly.

4.1.4 Discrete and Continuous Variables

The decision columns of a `SolveDataFrame` can either be discrete (i.e. the values of the column must be assigned integer values) or continuous. To determine whether a decision column is discrete or continuous, SolveDF looks in the schema of the orginal `DataFrame`. If the original data type of a decision column is `Integer` or `Long`, SolveDF will treat the underlying optimization problem as a MIP problem, and will only assign discrete values to the decision variables in the column. This is the same approach that SolveDB uses, although SolveDB also supports binary decision columns when the original data type of a decision column is `boolean`.

4.2 Example Queries

A series of different optimization problems will now be presented, along with solve queries for solving each problem with both SolveDB and SolveDF.

4.2.1 Knapsack Problem

The well known knapsack problem[32] can be formulated as a MIP problem, and we have already shown a SolveDB query that solves this in 2.1, which is also shown here in Listing 4.6. The queries use data from a table with the following relational schema:

`knapsack_items (item_name, weight, profit, quantity)`

2.1 shows a SolveDF query for solving the same problem, and 4.6 shows an equivalent, but more compact query using the shorthand notation for constraints and objectives described in Section 4.1.

```
1 SOLVESELECT quantity IN (SELECT * FROM knapsack_items) AS
     r_in
2 MAXIMIZE (SELECT sum(quantity*profit) FROM r_in)
3 SUBJECTTO (SELECT sum(quantity*weight) <= 15 FROM r_in),
4           (SELECT quantity >= 0 FROM r_in)
5 USING solverlp();
```

Listing 4.6: SolveDB solve query for the knapsack problem with a maximum allowed weight of 15.

```
1 knapsack_items.solveFor('quantity)
2 .maximize(r_in => r_in.select(sum('profit * 'quantity)))
3 .subjectTo(
4     r_in => r_in.select(sum('weight * 'quantity) <= 15),
5     r_in => r_in.select('quantity >= 0))
```

Listing 4.7: SolveDF solve query for the knapsack problem with a maximum allowed weight of 15.

```
1 knapsack_items.solveFor('quantity)
2 .maximize(sum('profit * 'quantity))
3 .subjectTo(
4     sum('weight * 'quantity) <= 15,
5     'quantity >= 0)
```

Listing 4.8: An alternative SolveDF solve query for solving the knapsack problem with a maximum allowed weight of 15, using more compact syntax.

4.2.2 Flex-object Scheduling

Section 1.2.1 introduced an energy balancing problem where we have to schedule flex-objects. We can formulate this problem as a solve query, where we store the flex-objects as rows with the following relational schema:

`flexobjects (f_id, timeindex, e_min, e_max)`

where each row represents the energy bounds in one time interval (denoted by `timeindex`) of a flex-object (denoted by `f_id`). Listing 4.9 shows a SolveDB query for solving the problem, and Listing 4.10 shows an equivalent SolveDF query. It is worth pointing out that the SolveDB query has to define the column `e` (which represents the scheduled amount of energy for a given interval in a flex-object) explicitly inside the input relation, whereas this column is implicitly created in the SolveDF query by the `solveFor('e)`-call. For a detailed explanation of the optimization problem, see Section 1.2.1.

```
1
```

```
SOLVESELECT e IN ( SELECT f_id, timeindex, e_min, e_max,
   NULL :: float8 AS e
            FROM flexobjects) AS r_in
    WITH t IN (SELECT DISTINCT timeindex AS ttid, NULL ::
       int AS t FROM flexobjects) as temp
MINIMIZE (SELECT sum(t) FROM temp)
SUBJECTTO (SELECT e_min <= e <= e_max FROM r_in),
          (SELECT -1 * t <= e_sum, e_sum <= t
          FROM (SELECT sum(e) as e_sum, timeindex FROM r_in
             GROUP BY timeindex) as a
          JOIN temp ON temp.ttid = a.timeindex)
USING solverlp);
```

Listing 4.9: SolveDB solve query for flexobject scheduling.

```
val temp = flexobjects
            .select('timeindex.as("ttid"))
            .distinct()
            .withDecisionColumns('t)
temp.cache()
df.solveFor('e)
.minimize(in => temp.select(sum('t)))
.subjectTo('e >= 'e_min, 'e <= 'e_max)
.subjectTo(in => in.groupBy('timeindex)
    .agg(sum('e) as 'e_sum, 'timeindex)
    .join(temp, 'ttid === 'timeindex)
    .select('e_sum <= 't, 'e_sum >= 't * -1.0))
```

Listing 4.10: SolveDF solve query for flexobject scheduling.

4.3 Extending the DataFrame API

A significant part of SolveDF's functionality is implemented through the use of UDTs (User Defined Types) in Spark SQL. Specifically, there is a need for UDTs that represent linear expressions (e.g. $2x_1 + x_2 - 3x_3$ or $x_1 - 2x_2 + x_3$) and linear constraints (e.g. $x_2+2x_4 \geq 7$ or $2x_1-x_3 = 0$). These are implemented with the classes `LinearExpression` and `LinearConstraint`. The `LinearExpression` class has definitions for many algebraic operators (e.g. $+, -, *, /, \geq, \leq$) on numeric types and other `LinearExpression`s, which is fairly easy to implement in Scala. However, it is a bit more difficult to make Spark SQL use these operators. For example, consider the following `DataFrame` in Listing 4.11:

```
// the salary column is of type LinearExpression
employees.select('salary + 100) // Type mismatch error!
```

Listing 4.11: A simple DataFrame query.

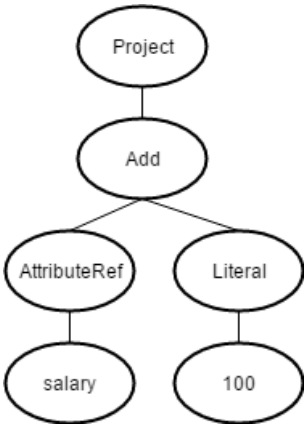

Figure 4.1.: Simplified representation of an expression tree created from the query in Listing 4.11.

The `DataFrame` in Listing 4.11 will cause a type mismatch error, as the + operator in Spark SQL Spark always creates a tree node (Spark SQL query plans are represented as trees) that only accepts children of numeric types. The same problem happens for the other operators as well.

A simple solution to this problem would be to define UDFs (User Defined Functions) for all the operators, but this results in ugly and needlessly verbose syntax (I would argue that "`'salary + 100`" looks much cleaner than "`plus('salary, 100)`", and even more so for more complex expressions). Furthermore, UDFs are generally slow, as they require serialization and deserialization for each row, and are harder to optimize in the Spark SQL execution plan[33]. Instead, I decided to make a wrapper class for `DataFrame` called `SolveDataFrame`, which supports most of the same operations as a `DataFrame`, e.g. `select`, `where` and `groupBy`. However, the `SolveDataFrame` versions of these operations make a substitution in the parameters before applying the operation to the underlying `DataFrame`. For example, the expression used in the query from Listing 4.11 would create an expression tree with an `Add` node, which would look like somewhat like what is shown in Figure 4.1.

SolveDF will substitute the `Add` node of this tree with a node of type `CustomAdd`, which is a node type I implemented that supports addition between `LinearExpressions` and numeric types. The substitution is very straightforward to perform, as Spark SQL offers a `transform` method that makes it easy to recursively manipulate trees through pattern matching. Specifically, the substitution is carried out by calling the `transform` method of the expression with the partial function defined in Listing 4.12 as parameter.

```
1 def transformExprs = {
2   case Add(l,r) => CustomAdd(l, r)
3   case Subtract(l,r) => CustomSubtract(l, r)
4   case Multiply(l,r) => CustomMultiply(l, r)
5   case GreaterThanOrEqual(l,r) =>
      CustomGreaterThanOrEqual(l,r)
```

```
6   case LessThanOrEqual(l,r) => CustomLessThanOrEqual(l,r)
7   case AggregateExpression(aggFunc,a,b,c) =>
8     aggFunc match {
9       case Sum(x) => ScalaUDF(sumLinearExpressions,
            LinearExpression.sparkSqlType,
            Seq(AggregateExpression(CollectList(x),a,b,c)),
            Seq(ArrayType(LinearExpression.sparkSqlType)))
10      case default => AggregateExpression(default,a,b,c)
11    }
12  case default => default
13} : PartialFunction[Expression, Expression]
```

Listing 4.12: Partial function used for replacing nodes in Spark SQL with custom nodes that support `LinearExpressions`.

4.4 Decomposition of Optimization Problems

To obtain better performance and make use of the parallelization provided by Spark, SolveDF can decompose optimizaton problems into smaller subproblems. Specifically, SolveDF can decompose problems with the special structure known as *independent subsystems*, which is mentioned in Section 1.2.1. It should be noted that the current implementation of the decomposition algorithm requires that the objective and all constraints of the optimization problem is collected on a single machine.

The decomposition algorithm is implemented by using a *union-find[34]* data structure (also known as a disjoint-set data structure), which is a data structure that allows for efficient partitioning of elements into disjoint subsets. The union-find structure consists of a forest, where each tree in the forest represents a subset (i.e. partition). The union-find structure supports two fundamental operations:

- `Find(x)`: Returns the subset `x` belongs to. Specifically, a representative element (i.e. the root of the set's tree) of the set is returned by this function.
- `Union(x,y)`: Merges the subsets that `x` and `y` belong to into a single subset.

Listing 4.13 shows SolveDF's implementation of the union-find structure. Optimization problems are partitioned in SolveDF by first extracting the sets of decision variables used by each constraint. These sets are then combined into disjoint subsets with the following code:

```
varSets.foreach(vars => vars.reduce((l, r) => unionFind.union(l, r)))
```

After this call, the partition that a variable belongs to can be found simply by calling the `find` function of the union-find structure.

```
1 class UnionFind(values : Seq[Long]) {
2   private case class Node(var parent: Option[Long], var
      rank: Int = 0)
```

```
3   private val nodes =  values.map(i => (i, new
        Node(None))).toMap
4
5   def union(x: Long, y: Long): Long = {
6     if (x == y) return x
7
8     val xRoot = find(x)
9     val yRoot = find(y)
10    if (xRoot == yRoot) return xRoot
11
12    val xRootNode = nodes(xRoot)
13    val yRootNode = nodes(yRoot)
14    if (xRootNode.rank < yRootNode.rank) {
15      xRootNode.parent = Some(yRoot)
16      return yRoot
17    }
18    else if (xRootNode.rank > yRootNode.rank) {
19      yRootNode.parent = Some(xRoot)
20      return xRoot
21    }
22    else {
23      yRootNode.parent = Some(xRoot)
24      xRootNode.rank += 1
25      return xRoot
26     }
27   }
28
29  @tailrec
30  final def find(t: Long): Long = nodes(t).parent match {
31    case None => t
32    case Some(p) => find(p)
33  }
34 }
```

Listing 4.13: SolveDF implementation of a union-find data structure. Note that Long is used instead of Int, as decision variables use a Long identifier in SolveDF.

4.5 Supported Solvers

For the most part, integrating new solvers into SolveDF is relatively easy. Each external solver simply needs a class that implements the `Solver` trait (Scala traits are similar to Java's interfaces) which has one method that needs to be overwritten. However, if the solver requires native libraries (e.g. if it is a solver written in C/C++), things can

get more tricky, as you need to use a Java binding to the solver and make sure the native libraries are installed and accesible on all nodes in the cluster. SolveDF currently supports 2 external solvers: Clp and GLPK.

4.5.1 Clp

Clp (**C**oin-or **l**inear **p**rogramming)[35] is an open-source linear programming library written in C++ as part of COIN-OR (The Computational Infrastructure for Operations Research)[36]. Clp does not support integer variables (i.e. it is not a MIP-solver). As the solver is written in C++, it cannot be called directly from Scala, but there fortunately exists an open source Java interface for Clp called clp-java[37]. clp-java is not just a simple interface to the native code, but actually provides rather high-level abstraction to the native library. Furthermore, installation is very straightforward, as you can download an "all-in-one jar file" that contains the native libraries for the most common systems.

4.5.2 GLPK

GLPK (**G**NU **L**inear **P**rogramming **K**it)[38] is an open-source LP and MIP solving library written in C. Like Clp, it cannot be called directly from Scala, so a Java binding called GLPK for Java[39] is used. This binding gives a more or less direct interface to the C-functions, meaning that you have to create and manipulate C-arrays (and clean up after them) in Java. GLPK for Java does not include binaries for the native libraries, so they have to be installed separately.

4.6 Known Issues and Limitations

As SolveDF is not fully developed yet, there are a number of bugs and unimplemented features. The following list shows some of the known issues and limitations of the current implementation:

- The code generated for the `>=` and `<=` operators is wrong if the first operand isn't a `LinearExpression`.
- Unary minus is not supported.
- Optimization problems without an objective function (i.e. constraint satisfaction problems) are not properly supported yet.
- The schema of the output `DataFrame` from solving a `SolveQuery` does not conform properly to the original schema of the input `DataFrame`, as the decision columns are always converted to `Double`.
- Spark SQL's `between` operator is not supported yet, although the same logic can be expressed with the `>=` and `<=` operators.

- `SolveDataFrame` does not yet support all operations that a normal `DataFrame` has. `SolveDataFrame` currently supports the following methods from `DataFrame`: select, apply, cache, withColumn, join, crossJoin, where, groupBy, limit.
- The only currently supported aggregate function for `LinearExpressions` is `sum`. In particular, `abs` could be a very useful aggregate function to support as well.

5 Experiments

To evaluate the performance of SolveDF and compare it to SolveDB, a number of benchmarks of SolveDF and SolveDB are performed. The scalability of SolveDF will also be evaluated by running these benchmarks on clusters of different sizes.

5.1 Hardware and Software

The experiments were run on Amazon Web Services EC2[40] r3.xlarge machines (4 virtual CPU cores, 30.5 GiB memory, 80 GB SSD storage). Table 5.1 shows an overview of the software versions used in the experiments.

Software	Version
Ubuntu	16.04.2
PostgreSQL	9.6
SolveDB	9.6
Spark	2.1.1
Scala	2.11.8
SBT	0.13.15
Java (OpenJDK)	1.8.0_131
GLPK	4.61
GLPK for Java	1.8.0
Clp	1.16.10
Clp-java	1.16.10

Table 5.1.: Versions of software used for the experiments.

5.1.1 Spark settings

Spark is run with the following settings in the spark-defaults.conf file:

```
spark.executor.memory 10g
spark.driver.memory 12g
spark.driver.maxResultSize 0
spark.sql.retainGroupColumns false
```

5.2 Measurements

5.2.1 SolveDB

For SolveDB, the following time values are measured in each experiment:

- **Partitioning time**: The time spent partitioning the optimization problem into indepenedent subproblems.
- **Solving time**: The time spent solving problem.
- **Total solving time**: The total time of the solving procedure, including I/O time and partitioning.
- **Total query time**: The total time it takes to execute the solve query.

The partitioning time, solving time, and total solving time are all printed out by SolveDB automatically when solving a solve query. The total query time is measured by running the query with the bash command *time*, i.e. '`time psql -d databaseName -f "fileName.sql"`'. Also, as the `psql` command prints the output rows of the given queries, all solve queries are wrapped in a "`SELECT WHERE EXISTS (solveQuery)`" statement to avoid printing the result rows of a solve query, as this can take up extra time (and clog the result files).

5.2.2 SolveDF

For SolveDF, the following time values are measured in each experiment:

- **Building query time**: The time it takes to initialize a solve query. This includes everything up untill the `solve` method is called on the `SolveQuery` object.
- **Materializing query time**: The time it takes to generate the objective and constraints of a solve query, and move it to the Driver node. This includes the time spent extracting data from the input `DataFrame` and running Spark SQL operations on UDTs.
- **Partitioning time**: The time it takes to partition the optimization problem into independent subproblems.

- **Solving time**: The time it takes to solve all of the subproblems. This includes time spent distributing the partitions to the cluster.
- **Total query time**: The total time it takes to build and execute the solve query, i.e. the time it takes to produce the output `DataFrame`.

5.3 Single Machine Experiments

First, we compare the performance of SolveDF with SolveDB when run on a single machine. SolveDF will be running with Spark in local mode, and each test case is run in a separate JVM-process. To get insight into how much performance is gained by parallelizing the solving process on a single machine (the machine has 4 cores), each test case will be run twice for SolveDF; once with Spark using all of its cores, and once with Spark being limited to using only 1 core (this is done by setting the master of the `SparkSession` to "`local[n]`", where `n` is the amount of cores).

5.3.1 Flex-object Scheduling Experiment

In this experiment, we solve the energy balancing problem introduced in 1.2.1, where flex-objects have to be scheduled.

Setup

The data used for this experiment is generated by a script shown in Appendix B.1.1. The data is stored in a PostgreSQL database prior to running the experiments, and we use the same schema that was used in Section 4.2.2 for representing the flex-objects:

```
flexobjects (f_id, timeindex, e_min, e_max)
```

Each row represents the energy bounds in one time interval (denoted by `timeindex`) of a flex-object (denoted by `f_id`). The amount of flex-objects and time intervals per flex-object will be varied throughout the test cases, as shown in Table 5.2. Note that this optimization problem can be partitioned across the time intervals into independent subproblems, meaning that an instance of the flex-object scheduling problem with n time intervals per flex-object can be partitioned into n subproblems. Because of this, it is easy to tweak how partitionable the problem is by varying the amount of time intervals per flex-object. For SolveDF, each test case will be run twice to test both solvers supported by SolveDF (GLPK and Clp).

Queries

The same queries that were presented in Section 4.2.2 are used for this experiment. Listing 5.1 Shows the SolveDB query for solving the problem, and Listing 5.2 shows the SolveDF query for solving the problem.

Table 5.2.: Test cases for the flex-object scheduling experiment.

Test case	Flex-objects	Time intervals per flex-object	Total rows
1	1000	3	3000
2	1000	5	5000
3	1000	10	10000
4	1000	20	20000
5	5000	3	15000
6	5000	5	25000
7	5000	10	50000
8	5000	20	100000
9	10000	3	30000
10	10000	5	50000
11	10000	10	100000
12	10000	20	200000
13	25000	3	75000
14	25000	5	125000
15	25000	10	250000
16	25000	20	500000
17	50000	3	150000
18	50000	5	250000
19	50000	10	500000
20	50000	20	1000000
21	100000	3	300000
22	100000	5	500000
23	100000	10	1000000
24	100000	20	2000000

```
1
2  SOLVESELECT e IN ( SELECT f_id, timeindex, e_min, e_max,
       NULL :: float8 AS e
3                FROM flexobjects) AS r_in
4      WITH t IN (SELECT DISTINCT timeindex AS ttid, NULL ::
          float8 AS t FROM flexobjects) as temp
5  MINIMIZE (SELECT sum(t) FROM temp)
6  SUBJECTTO (SELECT e_min <= e <= e_max FROM r_in),
7            (SELECT -1 * t <= e_sum, e_sum <= t
8            FROM (SELECT sum(e) as e_sum, timeindex FROM r_in
                GROUP BY timeindex) as a
9            JOIN temp ON temp.ttid = a.timeindex)
10 USING solverlp);
```

Listing 5.1: SolveDB solve query for flexobject scheduling.

```
1  val temp = flexobjects
2              .select('timeindex.as("ttid"))
3              .distinct()
4              .withDecisionColumns('t)
5  temp.cache()
6  df.solveFor('e)
7  .minimize(in => temp.select(sum('t)))
8  .subjectTo('e >= 'e_min, 'e <= 'e_max)
9  .subjectTo(in => in.groupBy('timeindex)
10     .agg(sum('e) as 'e_sum, 'timeindex)
11     .join(temp, 'ttid === 'timeindex)
12     .select('e_sum <= 't, 'e_sum >= 't * -1.0))
```

Listing 5.2: SolveDF solve query for flexobject scheduling.

Results

Tables with the complete results can be found in Appendix B.

It appears that the difference between using 1 or 4 cores on a single machine for SolveDF is not as big as one might expect. Although SolveDF in this case can solve 4 subproblems in parallel, it solves each subproblem significantly slower than if it was solving 1 at a time. Figure 5.1 shows the relative speedups in solving time when going from 1 to 4 cores for SolveDF when solving the flex-object scheduling problem.

When looking at the time it takes to solve the partitions, the speedup gained by going from 1 core to 4 cores appears to vary significantly between test cases, although there's a general trend for the speedup to be higher for larger problem sizes. When using GLPK as the solver, the speedup in solving time is higher, peaking at a relative speedup of 2.09 in test case 24, whereas CLP peaks at a relative speedup of 1.31. However, as solving the partitions is only a part of executing a solve query, the total speedup in terms of total query time is smaller than what is indicated by Figure 5.1. Figure 5.2 shows the relative speedup of the total query times. It is also worth pointing out that there are a few outliers where the speedup is actually slightly lower than 1 (meaning the solving is slower with 4 cores).

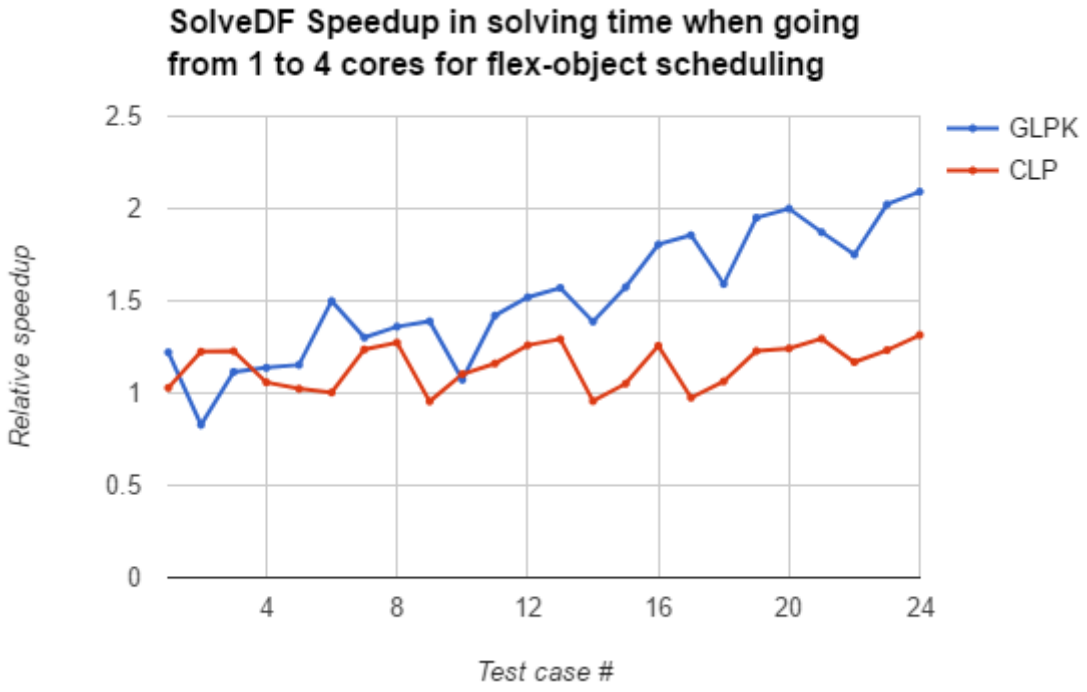

Figure 5.1.: Graph showing the speedup in solving time when going from 1 to 4 cores for SolveDF when solving the flex-object scheduling problem across all test cases. Note that this speedup is for the *solving time*, not the *total query time*.

For all test cases, SolveDB is faster than SolveDF. Figure 5.3 shows a comparison between the total query times of SolveDB and SolveDF across all test cases. The results

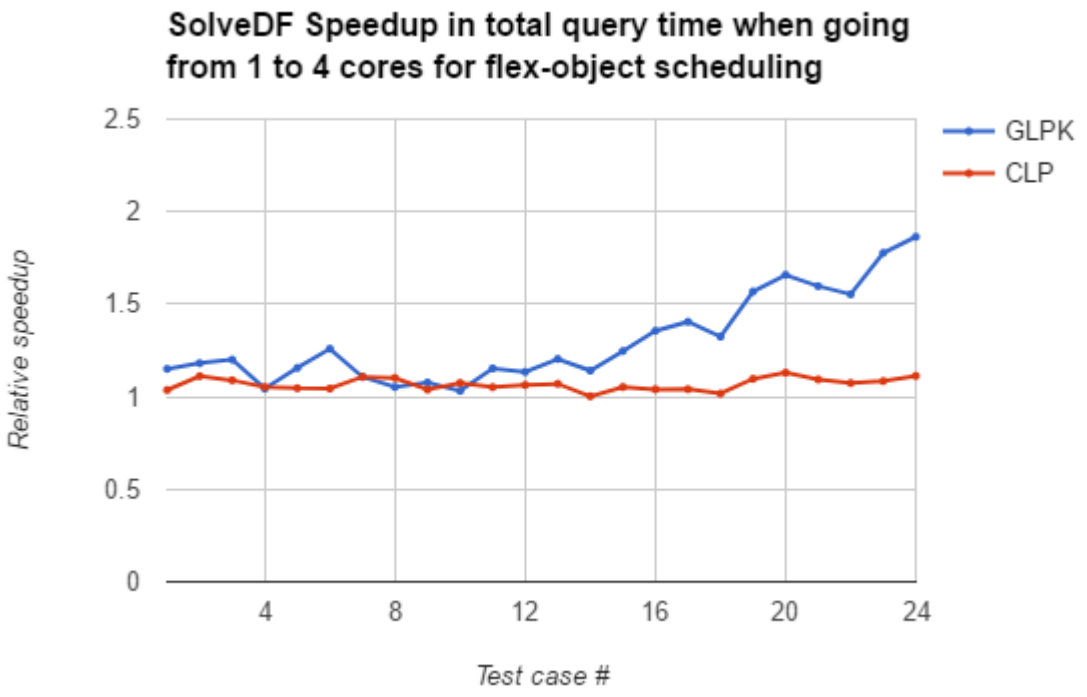

Figure 5.2.: Graph showing the speedup in total query time when going from 1 to 4 cores for SolveDF when solving the flex-object scheduling problem across all test cases.

show that for SolveDF, using Clp as the solver is significantly faster than GLPK, as Clp becomes several times faster than GLPK as the problem size grows (e.g. in test case 24, Clp has a 5.95 times faster solving time than GLPK). However, even though Clp has a significantly faster solving time than GLPK, the time it takes to materialize constraints and partition the problem ends up dominating the solving time. This is even more of a problem for smaller problem sizes, where the solving time is only a small fraction of the total query time for SolveDF, as shown in Figure 5.4. This appears to be a much smaller problem for SolveDB, as it can solve test cases 1-6 in less than a second, whereas SolveDF requires 7-10 seconds for these cases. However, although there is more than an order of magnitude difference in total query time between SolveDB and SolveDF for some of the smaller test cases (i.e. test cases 1-7), the difference becomes smaller as the problem size grows. In fact, for test case 24, SolveDB is only 2.38 times faster than SolveDF when using Clp and 6.69 times faster than SolveDF when using GLPK, as shown in Figure 5.3.

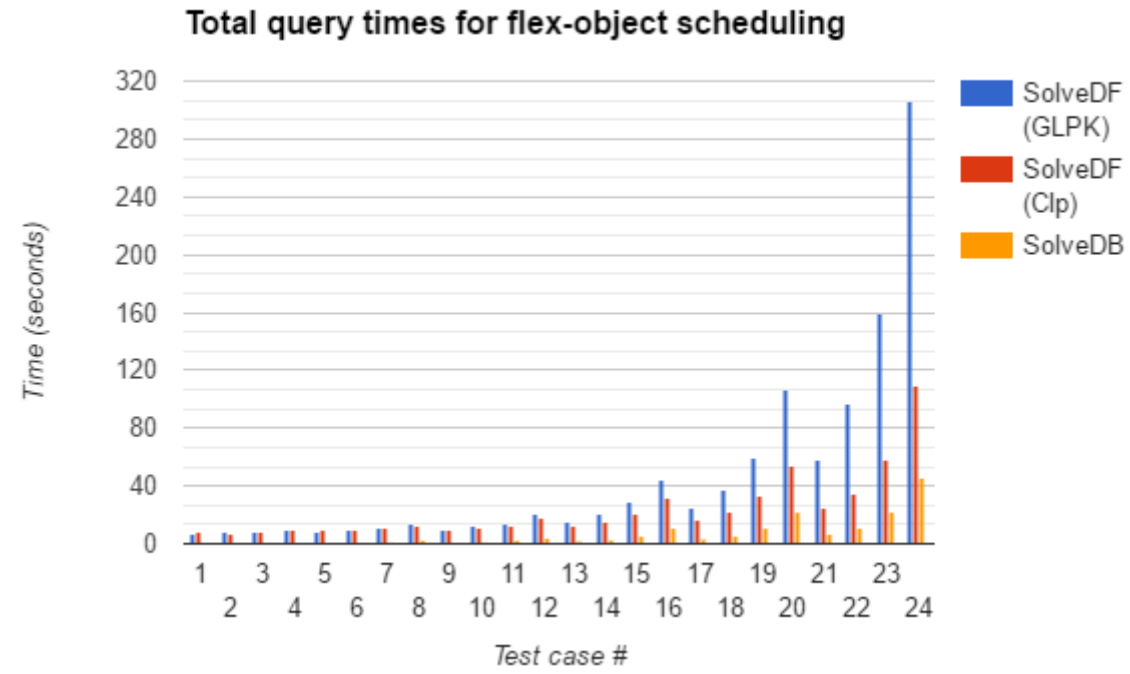

Figure 5.3.: Comparison of total query times between SolveDB and SolveDF for flex-object scheduling. The results for SolveDF are with 4 cores used.

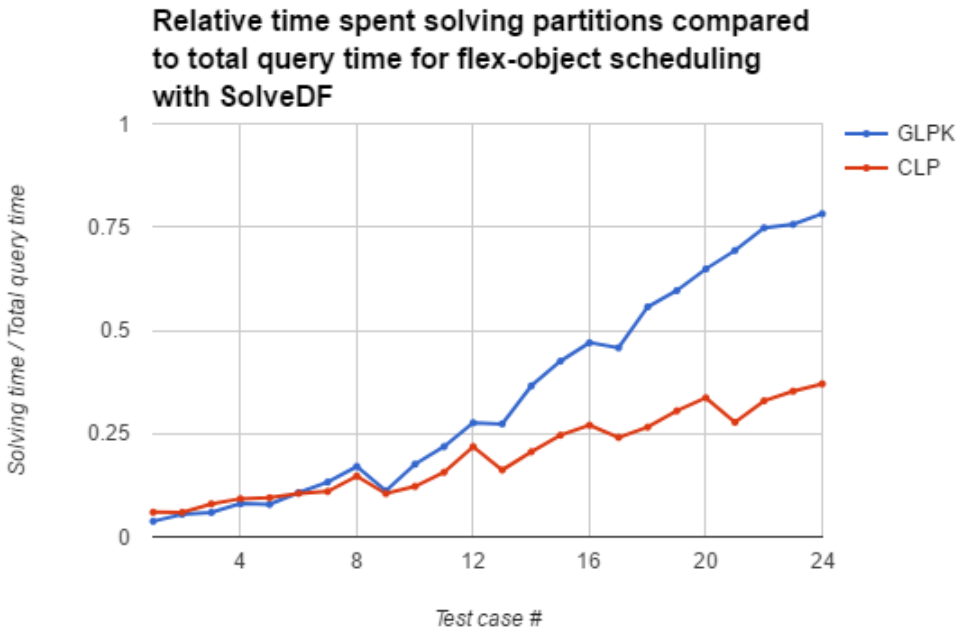

Figure 5.4.: Graph showing how much time of a solve query is spent on solving partitions relative to the total query time across all 24 test cases for SolveDF using 4 cores in the flex-object experiment.

5.3.2 Knapsack Experiment

In this experiment, we look to solve a problem that is both computationally complex and partitionable by SolveDB and SolveDF. For this, we use a variation of the 0-1 knapsack problem that is partitionable, which will be referred to as the *categorized knapsack problem*. The problem is defined as follows:

We are given a set of n items, where each item belongs to one of m categories (each category is denoted by a number). We are also given m knapsacks (one for each category) to fill with items, and each knapsack has the same weight limit denoted by w. Each item has a value and a weight, and we need to find quantity values for each item, such that the total value of all items put into the knapsacks is maximized, while not putting items with a total weight exceeding w into any knapsack. Essentially, solving this problem is equivalent to solving m ordinary 0-1 knapsack problems separately. Like the flex-object scheduling experiment, it is easy to control how partitionable this optimization problem is by tweaking the value of m (the amount of possible partitions is equal to m).

Setup

All data used for the experiment is generated by a script shown in Appendix B.1.2. The data is stored in a PostgreSQL database prior to running the experiments with the following schema:

```
knapsack_items (item_name, weight, profit, category, quantity)
```

Each row represents an item that can be put into a knapsack. Note that the data type of quantity is defined as integer, meaning that SolveDB and SolveDF will treat this as a MIP problem. The amount of items in each category and the amount of categories will be varied across the test cases, as shown in Table 5.3.

Queries

The queries for solving this problem are very similar to the queries for solving an ordinary knapsack problem as shown in Section 4.2. Listing 5.3 Shows the SolveDB query used in the experiment, and Listing 5.4 shows the SolveDF query used. Both SolveDB and SolveDF use GLPK for the solving of the problem. We do not run SolveDF with Clp as the solver, as Clp doesn't support integer variables.

Table 5.3.: Test cases for the knapsack experiment.

Test case	Items per category	Amount of categories	Max weight	Total rows
1	1000	1	1000	1000
2	1000	5	1000	5000
3	1000	10	1000	10000
4	1000	20	1000	20000
5	10000	1	10000	10000
6	10000	5	10000	50000
7	10000	10	10000	100000
8	10000	20	10000	200000
9	20000	1	20000	20000
10	20000	5	20000	100000
11	20000	10	20000	200000
12	20000	20	20000	400000
13	30000	1	30000	30000
14	30000	5	30000	150000
15	30000	10	30000	300000
16	30000	20	30000	600000
17	40000	1	40000	40000
18	40000	5	40000	200000
19	40000	10	40000	400000
20	40000	20	40000	800000
21	50000	1	50000	50000
22	50000	5	50000	250000
23	50000	10	50000	500000
24	50000	20	50000	1000000

```
1 SOLVESELECT quantity IN (SELECT * knapsack_items) as u
2 MAXIMIZE  (SELECT SUM(quantity * profit) FROM u)
3 SUBJECTTO (SELECT SUM(quantity * weight) <= MAX_WEIGHT
4         FROM u GROUP BY category),
5          (SELECT 0 <= quantity <= 1 FROM u)
6 USING solverlp;
```

Listing 5.3: SolveDB solve query for the knapsack experiment. The maximum allowed weight per knapsack is denoted by `MAX_WEIGHT`.

```
knapsack_items.solveFor('quantity)
  .maximize(sum('profit * 'quantity))
  .subjectTo(in => in.groupBy('category)
    .agg(sum('weight * 'quantity) <= maxWeight))
  .subjectTo('quantity >= 0.0, 'quantity <= 1.0)
.withSolver(GlpkSolver)
```

Listing 5.4: SolveDF solve query for the knapsack experiment. The maximum allowed weight per knapsack is denoted by `maxWeight`.

Results

Tables with the complete results can found in Appendix B.

Like in the flex-object experiment, going from 1 to 4 cores allows SolveDF to solve 4 subproblems at the same time in parallel, but each subproblem is solved more slowly than when using 1 core. The resulting speedups in solving time and total query time are shown in Figure 5.5 and Figure 5.6. Overall, the problem is solved faster with 4 cores, but it varies significantly between test cases. For all test cases with only 1 partition (cases 1, 5, 9, 13, 17 and 21) the speedup is very close to 1 (i.e. there's basically no change), which makes sense as there's only 1 subproblem that can be solved. The highest speedup is achieved in Test case 8, which has a speedup of 2.71 for the solving time (which results in a 2.50 speedup in total query time). In general, it appears that test cases with higher amounts of categories (and thereby partitions) experience a higher speedup, while test cases with higher amounts of items seem to have a lower speedup.

For test cases 1-5, SolveDB is significantly (2 to 50 times) faster than SolveDF, as SolveDF spends several seconds building the query and optimization problem, wheras actually solving the optimization problem only takes up a fraction of the time, as shown in Figure 5.9. However, from test case 6 and onwards, SolveDF and SolveDB are generally close to each other in performance, with SolveDF performing better in all but 4 test cases. The total query times for SolveDB and SolveDF can be seen in Figure 5.7 and Figure 5.8. There also appears to be a tendency for SolveDF to perform better in test cases with more than 1 partition (i.e. test cases with more than 1 category).

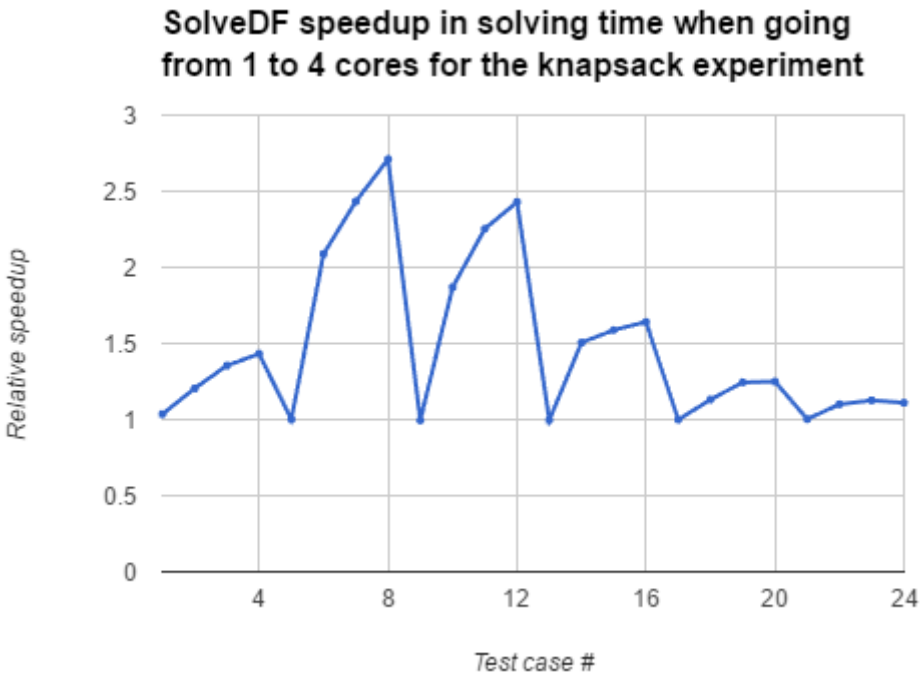

Figure 5.5.: Graph showing the speedup in solving time when going from 1 to 4 cores for SolveDF in the knapsack experiment across all test cases. Note that this speedup is for the *solving time*, not the *total query time*.

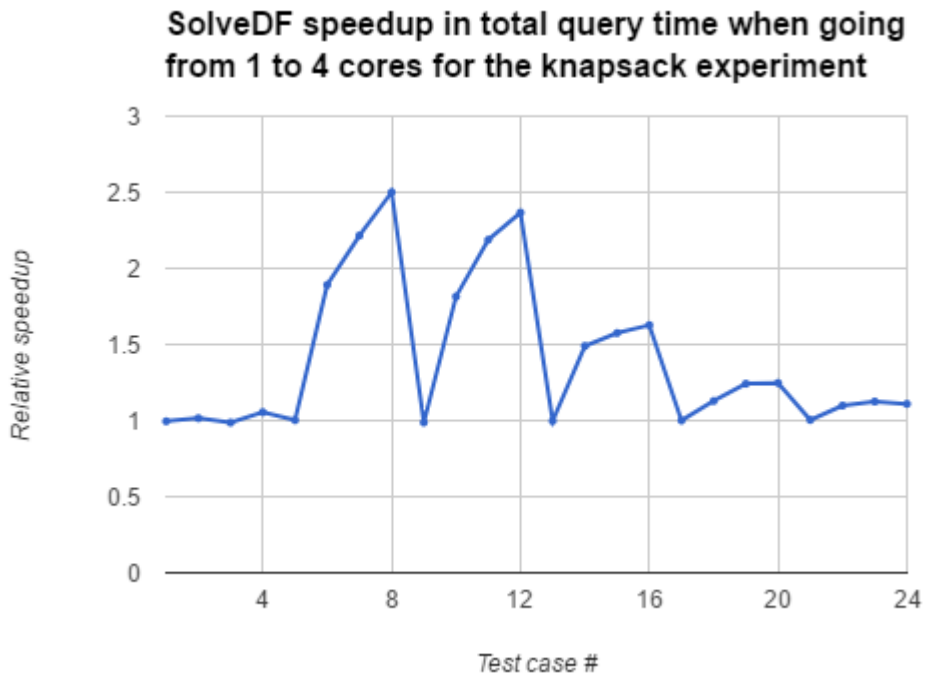

Figure 5.6.: Graph showing the speedup in total query time when going from 1 to 4 cores for SolveDF in the knapsack experiment across all test cases.

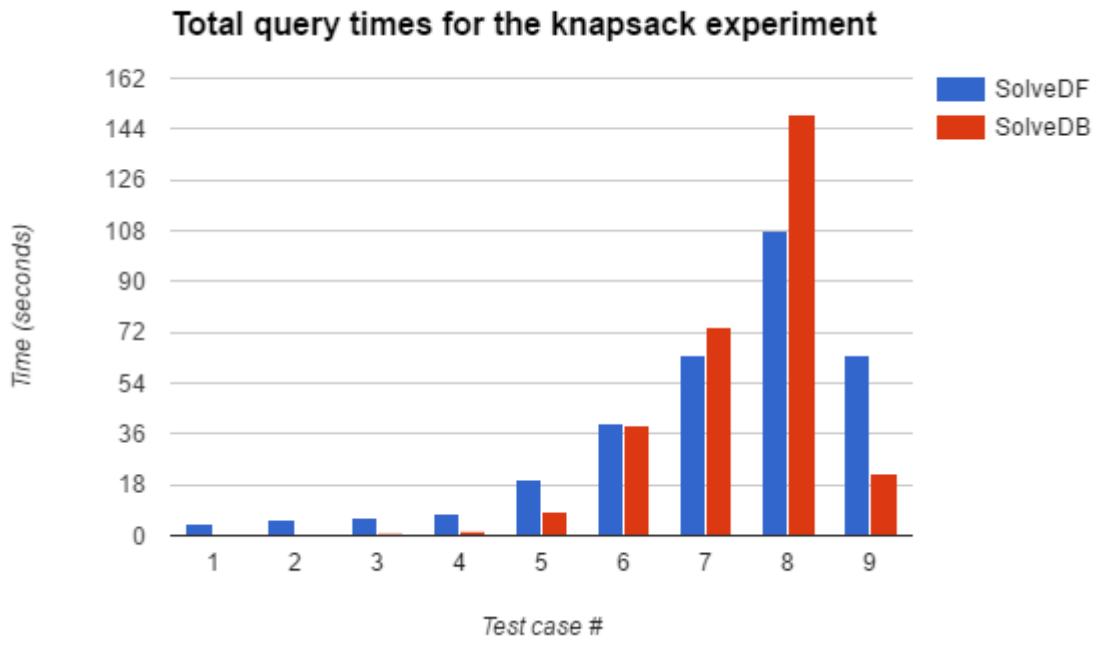

Figure 5.7.: Comparison of total query times between SolveDB and SolveDF for the knapsack experiment in test cases 1-9. The results for SolveDF are with 4 cores used.

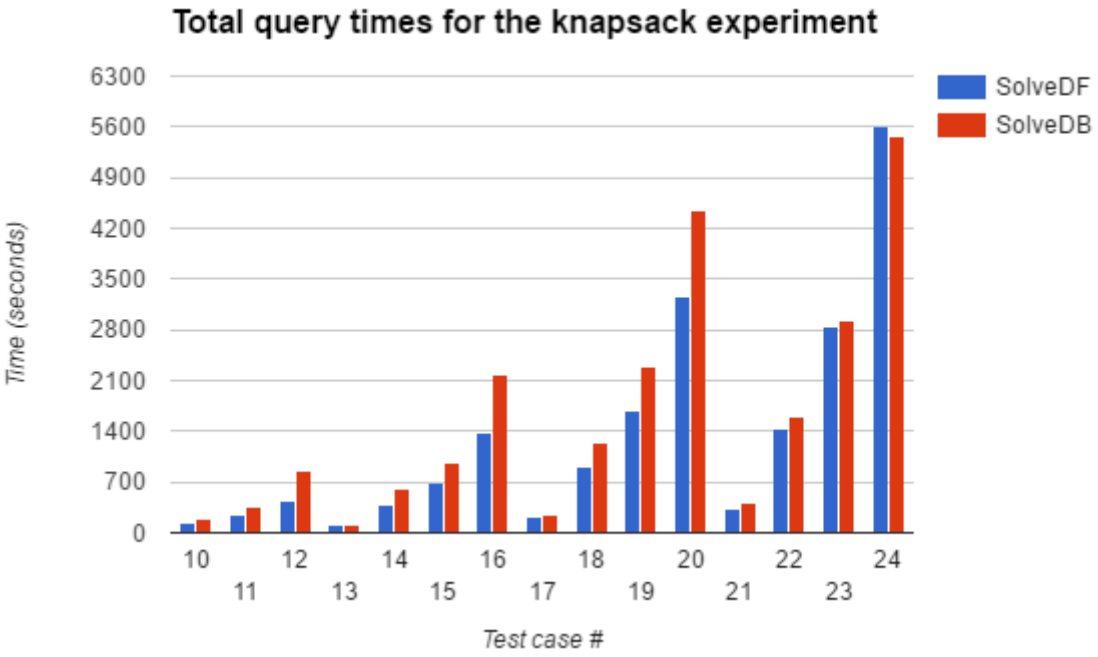

Figure 5.8.: Comparison of total query times between SolveDB and SolveDF for the knapsack experiment in test cases 10-24. The results for SolveDF are with 4 cores used.

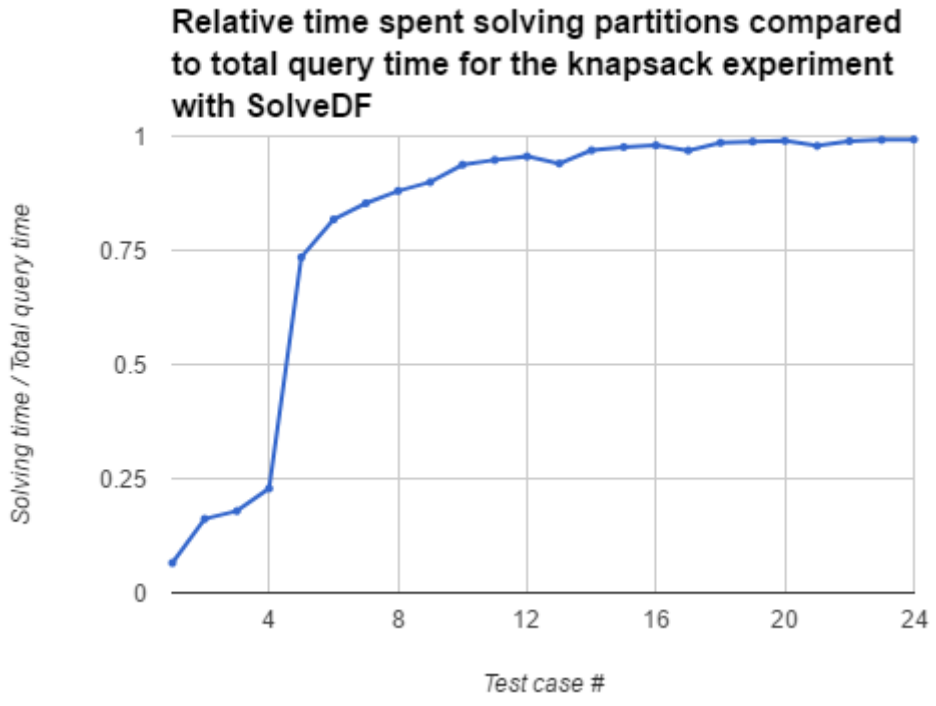

Figure 5.9.: Graph showing how much time of a solve query is spent on solving partitions relative to the total query time across all 24 test cases for SolveDF using 4 cores in the knapsack experiment.

5.4 Cluster Experiments

Now that we have an idea of how SolveDF performs in comparison to SolveDB on a single machine, we will test the scalability of SolveDF by running the test cases of the previous experiments on Spark clusters of varying sizes. Specifically, we will be running SolveDF on clusters of sizes 2,4 and 8. All machines in the cluster are of the same type (EC2 r3.xlarge) as the previous experiment. Spark is run using Spark Standalone mode, and each test case is run as a separate application in client mode on the master node. The program submitted to the cluster is a "fat JAR", i.e. a JAR-file containing the SolveDF code along with all of its external dependencies (except Spark, as this is provided automatically by the cluster). The master node will have both a master- and worker process running, meaning that the master node also participates in the solving of submitted queries.

5.4.1 Results

Tables with the complete results can be found in Appendix B.

For the knapsack experiment, there is no speedup gained when solving test cases with only 1 partition (i.e. cases 1, 5, 9, 13, 17, 21), which is to be expected. For the smaller test cases (cases 1-4), there is in fact an increase in total query time, as it apparantly takes longer to build the query when running on a cluster. However, things quickly change in later test cases with more than 1 partition, as shown in Figure 5.10. On a cluster of 2 nodes, most test cases with more than 1 partition show a speedup between 1.5 to 2, with test case 18 actually having a speedup of 2.23. For a cluster of size 4, the maximum speedup in total query time achieved is 3.66 in test case 18, and for clusters of size 8 the maximum speedup is 6.85 achieved in test case 20. In general, for clusters of sizes 4 and 8, test cases with more partitions tend to experience a greater speedup.

For the most part, the speedups in solving time are only slighty higher than the speedups in total query time (e.g. for a cluster size of 8, test case 20 has a speedup in solving time of 7.63 compared to 6.85 in total query time), which makes sense as the solving time is by far the most time consuming part of the solve query in the knapsack experiment as shown earlier in Figure 5.9. Figure 5.11 shows the relative speedups in solving time.

For the flex-object experiment, the speedups are significantly smaller, especially when using Clp as the solver. This is not surprising, as the solving time is only a fraction of the total query time when using Clp. The highest speedup in solving time with Clp on a cluster of size 8 is in test case 23, where the speedup is 1.7, but the speedup in total query time is only 1.05. In fact, most test cases with Clp take slightly longer when you increase the cluster size, as the partitions are so quickly solved that the speedup in solving time is overshadowed by the increased time in constructing the optimization problem. Like in the knapsack experiment, the time it takes to build the query is higher by a few seconds when running on a cluster. Graphs over the speedups in total query time and solving time for Clp can be found in Figure 5.12 and Figure 5.13.

When using GLPK as the solver in the flex-object experiment, the speedup is higher,

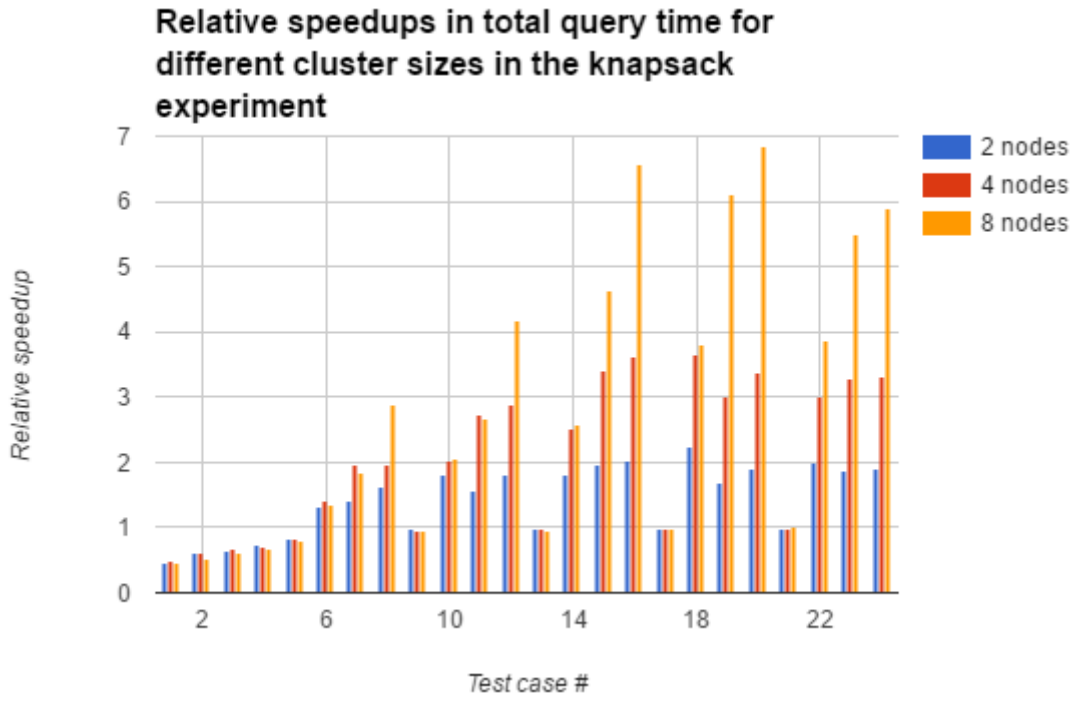

Figure 5.10.: Graph showing the relative speedups in total query time for SolveDF with different cluster sizes.

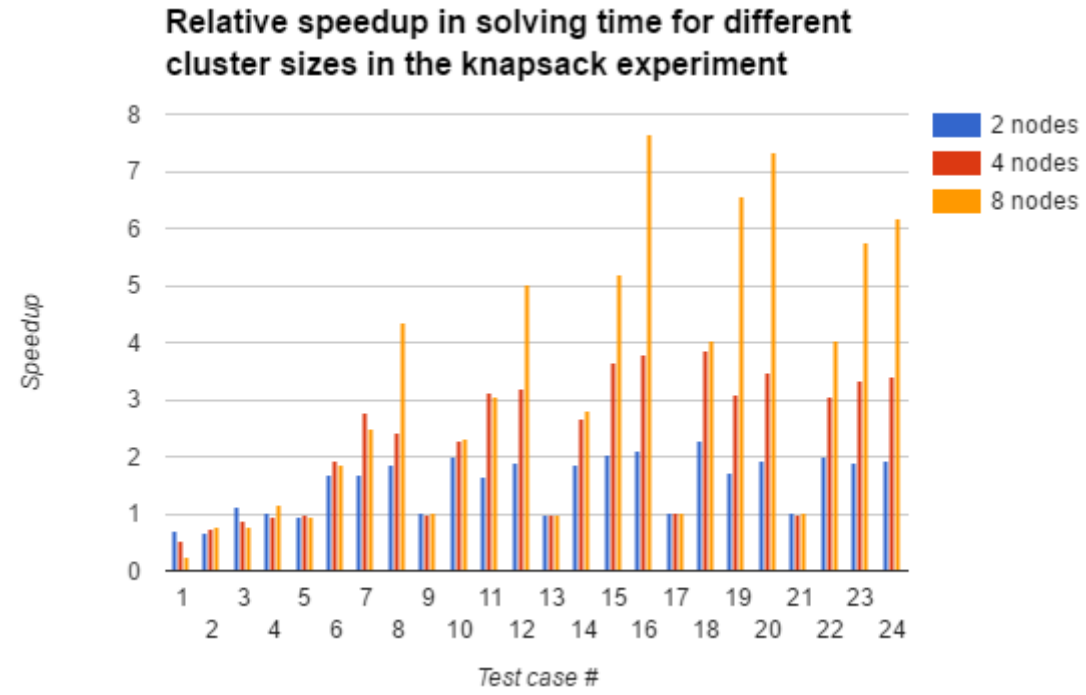

Figure 5.11.: Graph showing the relative speedups in solving time for SolveDF with different cluster sizes.

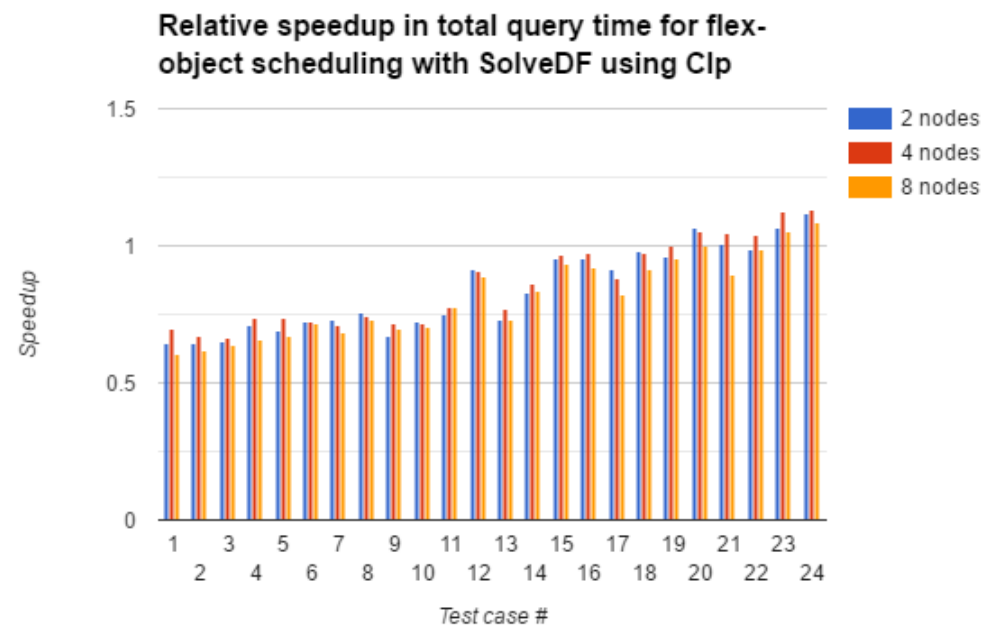

Figure 5.12.: Graph showing the relative speedups in total query time for SolveDF with different cluster sizes for the flex-object experiment with SolveDF using Clp.

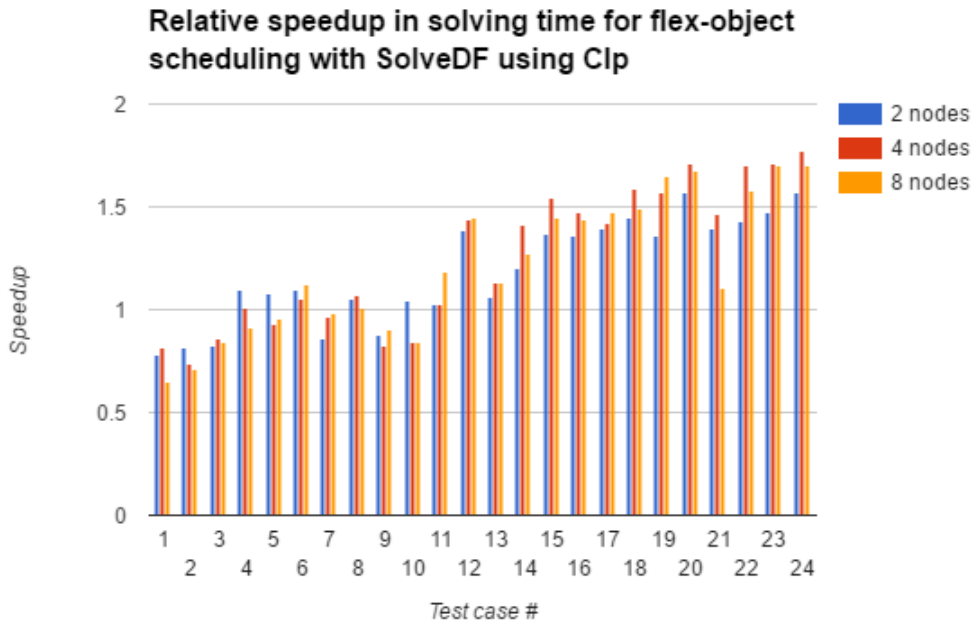

Figure 5.13.: Graph showing the relative speedups in solving time for SolveDF with different cluster sizes for the flex-object experiment with SolveDF using Clp.

but still very low compared to the knapsack experiment. The highest speedup on a cluster of size 8 with GLPK is achieved in test case 24, where the speedup in solving time is 3.79, resulting in a speedup of 2.15 in total query time. In general, the speedup is higher for larger problem sizes, with most of the lower-half of the test cases being slower when run on a cluster. Graphs over the speedups in total query time and solving time for GLPK can be found in Figure 5.14 and Figure 5.15.

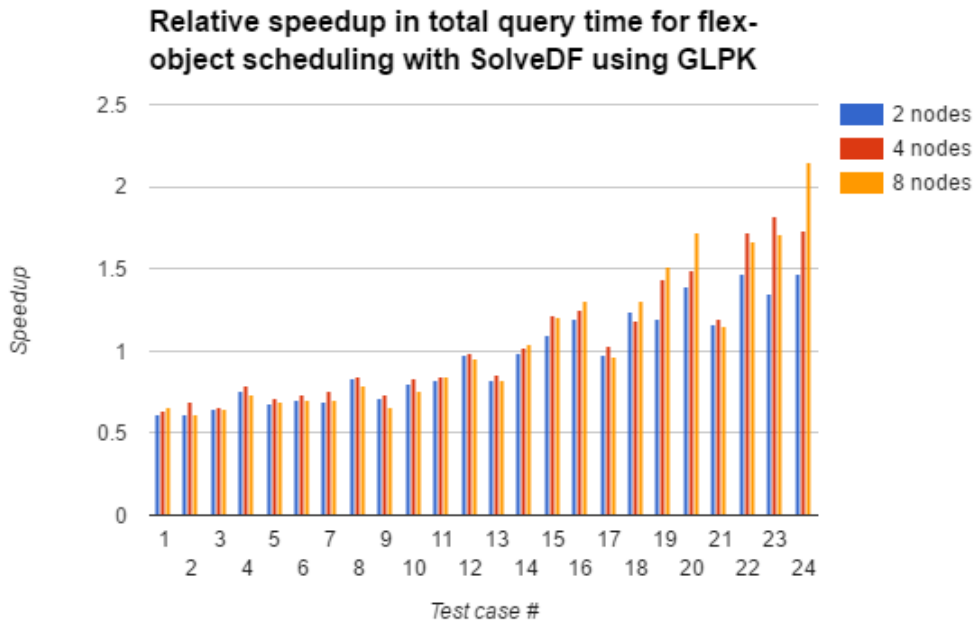

Figure 5.14.: Graph showing the relative speedups in total query time for SolveDF with different cluster sizes for the flex-object experiment with SolveDF using GLPK.

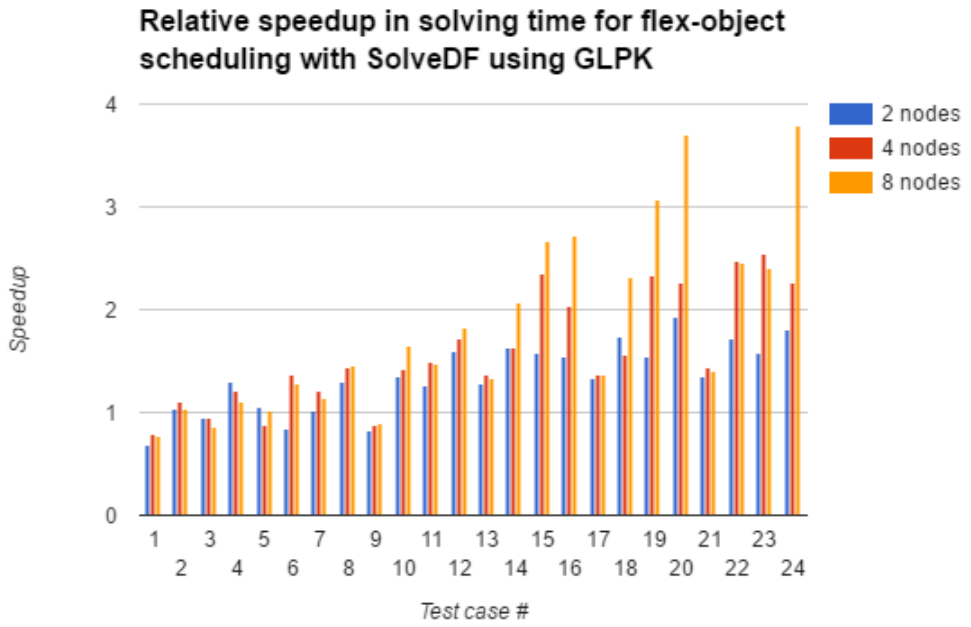

Figure 5.15.: Graph showing the relative speedups in solving time for SolveDF with different cluster sizes for the flex-object experiment with SolveDF using GLPK.

5.5 Discussion

As shown by the results, SolveDF appears to spend a lot of time on constructing optimization problems compared to SolveDB. However, this appears to become less of an issue as the size and especially the complexity of the problem grows, as exemplified by the knapsack experiment, where the solving time ends up dominating all other parts of executing the solve query. This type of problem has two important characteristics that seem to make it ideal for SolveDF: It is partitionable (and thereby parallelizable) and has high complexity. This means that time spent partitioning, moving data around between nodes, and constructing optimization problems becomes negligible compared to the solving time. Even when using only 1 machine, SolveDF has very similar performance to SolveDB in the knapsack experiment, and with the exception of some of the smallest test cases, SolveDF actually performs better in most test cases of the knapsack experiment. When SolveDF runs the knapsack experiment on a cluster, the solving time seems to grow approximately linearly with the number of nodes in the cluster, exemplified by SolveDF in one case having a 7.63 times faster solving time (and thereby 6.85 times faster total query time) on a cluster with 8 machines compared to 1 machine. This suggests that at least for complex problems with enough partitions, it is easy to scale up SolveDF's performance by using more machines.

On the other hand, the flex-object scheduling problem is a far less ideal problem for SolveDF. Even for cases with large amounts of data, SolveDF was slower than SolveDB in all cases, and adding more machines to the cluster had little effect when using Clp

as the solver (in fact, it is faster with only 1 machine in many cases). Although the flex-object scheduling problem is partitionable, it appears that it is not very complex, meaning that solving the actual optimization problem is very quick. Even for the largest test case, SolveDF ends up spending only 37% of its time solving partitions when using Clp.

Interestingly, solving the flex-object scheduling problem with SolveDF using GLPK was surprisingly slow compared to SolveDB. This is interesting, as SolveDB also uses GLPK as the underlying solver, so one could except the solving times to be similar, especially since the solving times are similar in the knapsack experiment. However, although SolveDF and SolveDB use the same solver, there are some possible explanations for why the performance differs so much here. It could be caused by the fact that SolveDF and SolveDF use different version of GLPK (SolveDF uses version 4.61, whereas SolveDB uses version 4.47), but it seems weird that a newer version of GLPK would perform far slower than an older version. I think a more likely explanation is that either the optimization problem is formulated differently in SolveDB, e.g. SolveDB might perform some presolving before handing the problem to GLPK, or perhaps SolveDB uses different configurations for GLPK.

Although using 4 cores to solve partitions in parallel on a single machine compared to only using 1 core provides some speedup, it seems to vary a lot, and it tends to become smaller as the problem size grows. Even in the knapsack experiment, where adding more nodes to the cluster seems to provide a linear speedup in solving time, using more cores on a single machine did not have much effect on the larger test cases. SolveDF is able to solve 4 problems in parallel with 4 cores, but each problem is solved more slowly this way. This could suggest that these problems become memory-bound as we add more cores.

Another interesting result is how the "building query time" for SolveDF is very high for small problem sizes, but doesn't change much as the problem size grows. For example, when running on 1 machine in the knapsack experiment, SolveDF uses 1.83 seconds to build the query in test case 1, which involves only 1000 rows, whereas test case 24 only takes 2.35 seconds to build the query despite involving 1000 times as many rows as test case 1. I have reason to believe that the high building time in test case 1 is caused by Spark having to "warm up", as this only seems to happen when every test case is run as separate JVM processes. If I instead run the test cases one by one in the same JVM process, the first case that is run takes approximately 1.8-2.4 seconds to build the query, while all the other test cases take less than a second (in particular, test case 1 only takes 0.1 seconds). In fact, whereas the total query time for test case 1 is 4.10 seconds in the results, the total query time for test case 1 is only 0.9 seconds if it is run after any other solve query. This could partially explain why the results show SolveDF as being very slow for smaller test cases.

6

Conclusion and Future Work

6.1 Conclusion

In this project, I set out to test the following hypothesis:

- Is it feasible to make a tool that allows for seamless integration of data management and constrained optimization problem solving in a Big Data context?

I conclude that i have proven this hypothesis through the design and implementation of SolveDF. SolveDF is a tool that extends Spark SQL with SolveDB's concept of solve queries. These solve queries allow users to delcaratively specify and solve constrained optimization problems through a familiar SQL-like interface, allowing for data management and constrained optimization to be performed seamlessly in the same environment. The syntax and structure of SolveDF is heavily inspired by SolveDB, and i have evaluated the usability of SolveDB through the Discount Method for Evaluating Programming Languages. This evaluation suggests that SolveDB has an intuitive syntax that can be learned in a short time by people who are familar with SQL, and that the overall concept and structure of solve queries is intuitive.

In Chapter 3, four requirements were specified for the design of SolveDF:

1. Declarative specification
2. Solver independence
3. Decomposition of problems
4. No modification of Spark

I conclude that all of these requirements are fulfilled. Requirement 1 is fulfilled because SolveDF allows for optimization problems to be specified declaratively as DataFrame queries in Spark SQL. Requirement 2 is fulfilled, as support for additional external solvers can be implemented simply by writing a simple class with 1 method, and SolveDF already supports 2 different external solvers. Requirement 3 is fulfilled through the implementation of a decomposition algorithm using a union-find data structure, and SolveDF is able to solve the resulting partitions in parallel on a cluster. Requirement 4 is also fulfilled, as SolveDF does not modify the source code of Spark, but rather builds on top of Spark SQL by using user-defined types and custom tree-node types.

Finally, i have evaluated the performance of SolveDF and compared it to SolveDB in a number of performance experiments. The results show that even when running on only a single machine, SolveDF has similar performance to SolveDB for certain problems, and in some cases outperforms SolveDB. The results also suggest that for problems of high complexity that are partitionable, the performance of SolveDF scales well when running on a cluster, exemplified by SolveDF being up to 6.85 times faster when run on a cluster of 8 machines. However, the results also indicate that SolveDF is generally significantly slower than SolveDB when constructing optimization problems, which results in poorer performance for some problems, especially smaller and less complex problems.

6.2 Future Work

Although the current implementation of SolveDF may look promising, there are still many areas that could be improved on.

6.2.1 Performance Optimizations

As shown by the experiments, SolveDF is slow at constructing optimization problems, and it would be worth looking into ways to improve this. One way to do this could be to optimize the code generation of SolveDF's custom `TreeNode` types (e.g. the `TreeNode`s representing operators such as $\geq, \leq, +, -, *$), as the generated code currently performs some needless conversions between Spark SQL internal types and UDTs for every operation. Likewise, the `TreeNode` for the `sum`-aggregate function could probably be implemented in a more effective way, as it is currently implemented with a `CollectList` node followed by a call to a UDF.

It might also be worth implementing presolving functionality to SolveDF, e.g. by making SolveDF able to identify and remove redundant constraints in an optimization problem before forwarding it to an external solver.

6.2.2 Support for DataSets

Currently, SolveDF is designed to work with Spark SQL's `DataFrame`s. However, Spark SQL also provides a very similar data structure called `DataSet`, which is basically a strongly-typed `DataFrame`. In fact, the `DataFrame` type is actually just an alias for the type `DataSet[Row]` in newer Spark versions. It might be worth making SolveDF work on `DataSet`s instead of `DataFrame`s, as `DataSet` is a more general type than `DataFrame`.

6.2.3 Spark SQL Strings in Solve Queries

Currently, SolveDF allows solve queries to be defined with the syntax of Spark SQL's DataFrame API. However, Spark SQL queries can also specified with regular SQL-strings, and it might be an interesting addition to allow constraint- and objective queries to be specified with SQL-strings as well, as this might be more natural to write in some cases, especially since the DataFrame API currently seems to have poor support for subqueries.

6.2.4 Follow-up Usability Test

The usability evaluation of SolveDB suggests that it is generally intuitive and easy to learn, and although SolveDF is heaviliy inspired by SolveDB, SolveDF's usability hasn't been evaluated experimentally. In particular, it might be interesting to test the usability of SolveDF on the same participants that the SolveDB usability evaluation was performed on, to see if the reaction to SolveDF would be similar.

6.2.5 Alternative Methods for Parallelization

Currently, SolveDF leverages the parallel capabilities of Spark through decomposition of optimization problems. However, the current decomposition algorithm can only find partitions for optimization problems with a specific type of structure (independent subsystems), but there exists other types of special structure[23] in optimization problems that can be exploited. In particular, it could be worth looking into Dantzig-Wolfe decomposition for SolveDF, as existing work[41] already exists on using Dantzig-Wolfe decomposition for distributed computation of optimization problems.

Furthermore, the current implementation of the decomposition algorithm requires that all data is collected on a single node. It would be worth looking into ways to perform the decomposition efficiently without having to gather all data on a single node.

6.2.6 Solver Configurations

Currently, the user can define what solver to use for a solve query in SolveDF, but the user doesn't have control over any of the configurations/parameters of the solvers. It would be a nice addition if solver-specific parameters could be specified directly as a part of the solve query. Specifically, it could be very useful if users could specify whether an exact solution is needed, or whether an approximate solution with certain tolerance values would be sufficient.

Perhaps there could even be a configuration for running a solve query in "interactive mode", where the solver outputs feasible solutions while searching for an optimal solution. In this case, the user can stop the solving process early if the solver finds a solution that the user deems to be good enough.

Bibliography

[1] Philipp Daniel Freiberger, Frederik Madsen Halberg, and Christian Slot. Prescriptive analytics for spark.

[2] Davide Frazzetto, Torben Bach Pedersen, Thomas Dyhre Nielsen, and Laurynas Siksnys. Prescriptive Analytics: Emerging Trends and Technologies.

[3] Matthias Boehm, Lars Dannecker, Andreas Doms, Erik Dovgan, Bogdan Filipič, Ulrike Fischer, Wolfgang Lehner, Torben Bach Pedersen, Yoann Pitarch, Laurynas Šikšnys, and Tea Tušar. Data Management in the MIRABEL Smart Grid System. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 95–102, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1143-4. doi: 10.1145/2320765.2320797. URL `http://doi.acm.org/10.1145/2320765.2320797`.

[4] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. *LogicBlox, Platform and Language: A Tutorial*, pages 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32925-8. doi: 10.1007/978-3-642-32925-8_1. URL `http://dx.doi.org/10.1007/978-3-642-32925-8_1`.

[5] Laurynas Šikšnys and Torben Bach Pedersen. Solvedb: Integrating Optimization Problem Solvers Into SQL Databases. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, SSDBM '16, pages 14:1–14:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4215-5. doi: 10.1145/2949689.2949693. URL `http://doi.acm.org/10.1145/2949689.2949693`.

[6] Alexandra Meliou and Dan Suciu. Tiresias: The Database Oracle for How-to Queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 337–348, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213875. URL `c`.

[7] Avita Katal, Mohammad Wazid, and RH Goudar. Big Data: Issues, Challenges, Tools and Good Practices. In *Contemporary Computing (IC3), 2013 Sixth International Conference on*, pages 404–409. IEEE, 2013.

[8] Stephen Kaisler, Frank Armour, J Alberto Espinosa, and William Money. Big Data: Issues and Challenges Moving Forward. In *46th Hawaii International Conference on System Sciences (HICSS), 2013*, pages 995–1004. IEEE, 2013.

[9] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[10] Apache Software Foundation. Apache spark, . URL `http://spark.apache.org/`.

[11] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[12] Danmarks Statistik. Husstande 1. januar efter område og tid. `http://www.statistikbanken.dk/`, November 2016.

[13] eurostat. Houshold composition statistics, 2016. URL `http://ec.europa.eu/eurostat/statistics-explained/index.php/Household_composition_statistics`. Accessed: 2016-11-14.

[14] Apache Software Foundation. Apache MLlib, . URL `http://spark.apache.org/mllib/`. Accessed: 2016-10-24.

[15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742797. URL `http://doi.acm.org/10.1145/2723372.2742797`.

[16] Jules Damji. A tale of three apache spark apis: Rdds, dataframes, and datasets. URL `https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html`.

[17] Adding a new user-defined function. URL `https://dev.mysql.com/doc/refman/5.7/en/adding-udf.html`.

[18] Make user-defined type (udt) api public, . URL `https://issues.apache.org/jira/browse/SPARK-7768`. Accessed: 2017-05-21.

[19] Hide userdefinedtype in spark 2.0, . URL `https://issues.apache.org/jira/browse/SPARK-14155`. Accessed: 2017-05-21.

[20] Linearprogramming, . URL `http://www.cs.yale.edu/homes/aspnes/pinewiki/LinearProgramming.html`.

[21] L. Šikšnys, E. Valsomatzis, K. Hose, and T. B. Pedersen. Aggregating and Disaggregating Flexibility Objects. *IEEE Transactions on Knowledge and Data Engineering*, 27(11):2893–2906, Nov 2015. ISSN 1041-4347. doi: 10.1109/TKDE.2015.2445755.

[22] lp_solve group at Yahoo.com. Absolute values. URL http://lpsolve.sourceforge.net/5.1/absolute.htm.

[23] Large-scale systems. URL http://web.mit.edu/15.053/www/AMP-Chapter-12.pdf.

[24] Integer programming. URL https://www.cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/integer_programming.pdf.

[25] . URL http://mathworld.wolfram.com/LinearProgramming.html.

[26] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *Proceedings of the VLDB Endowment*, 8(10):1094–1105, 2015.

[27] Constraint optimization. URL https://developers.google.com/optimization/cp/.

[28] Svetomir Kurtev, Tommy Aagaard Christensen, and Bent Thomsen. Discount method for programming language evaluation. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (plateau 2016)*. Association for Computing Machinery, 2016.

[29] David Benyon. *Designing Interactive Systems - A comprehensive guide to HCI and interaction design*. Pearson Education Limited, 2010.

[30] Jesper Kjeldskov, Mikael B Skov, and Jan Stage. Instant data analysis: conducting usability evaluations in a day. In *Proceedings of the third Nordic conference on Human-computer interaction*, pages 233–240. ACM, 2004.

[31] Implicit classes. URL http://docs.scala-lang.org/overviews/core/implicit-classes.html.

[32] Knapsack problem. URL https://en.wikipedia.org/wiki/Knapsack_problem. Accessed: 2017-05-21.

[33] Jacek Laskowski. Udfs - user-defined functions. URL https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sql-udfs.html.

[34] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46. ACM, 2007.

[35] Clp. URL https://projects.coin-or.org/Clp. Accessed: 2017-05-21.

[36] Coin-or. URL https://www.coin-or.org/. Accessed: 2017-05-21.

[37] Nils Löhndorf. clp-java. Accessed: 2017-05-21.

[38] Glpk, . URL https://www.gnu.org/software/glpk/. Accessed: 2017-05-21.

[39] Glpk for java, . URL http://glpk-java.sourceforge.net/. Accessed: 2017-05-21.

[40] Amazon ec2 instance types. URL https://aws.amazon.com/ec2/instance-types/.

[41] James Richard Tebboth. *A Computational Study of Dantzig-Wolfe Decomposition*. PhD thesis, University of Buckingham, 2001. URL http://eaton.math.rpi.edu/CourseMaterials/Spring08/JM6640/tebboth.pdf.

[42] Daisy - solvedb. URL http://www.daisy.aau.dk/?page_id=1198.

Part III.

Appendix

Appendix A

SolveDB Usability Experiment

A.1 Task Sheet

Task 1: Inserting Money

You are given a table of bank accounts with the following schema:

account (id, interest, min_balance, balance)

Currently, all these accounts have a `balance` of 0. Your task is to determine how much money should be put into the accounts. Each account has minimum `balance` requirement which must be fulfilled, i.e. `balance` must be greater than or equal to `min_balance`. If the `min_balance` of an account is less than $80, there must be at least $80 on the account instead.

At the same time, we want to minimize the total amount of money inserted into all accounts. We can formulate this as a constrained optimization problem:

Unknown variables: The balance column
Objective function: Minimize the sum of the balance values
Constraints: balance >= min_balance and balance >= 80

a)
Solve this problem with a SolveDB query using data from the *account* table.

b)
Management has decided that accounts with an `interest` value greater than 0.05 should always have a balance of exactly $300. Change your query to reflect this new constraint.

Task 2: Bucket Filling

You are given a table of buckets with the following schema:

```
bucket (id, min_amount, max_amount, amount)
```

As an employee in a bucket-filling agency, your task is to fill all these buckets with an amount of water that is within the allowed bounds of the bucket. In other words, find values for the amount column, such that `min_amount` <= `amount` <= max_amount for each bucket.

At the same time, management has decided that 3 liters is the ideal amount of water to have in a bucket, so each bucket should be filled with an amount that is as close to 3 liters as possible.

Solve this problem with a SolveDB query using data from the bucket table. (Hint: Try to identify the unknown variables, the objective function, and the constraints of the problem first)

Task 3: The Knapsack Problem

Given a set of items, each with a weight and profit value, determine the number of each item to include in a collection, such that the total weight is less than or equal to a given limit, and the total profit is as large as possible.

a)
Solve the knapsack problem with a SolveDB query using data from the knapsack table (shown below) with a maximum allowed weight of 15. In other words, replace the 0's in the quantity column with values such that the total profit is maximized, without the total weight exceeding 15. Note that the assigned quantity values are not allowed to be negative.

knapsack (item_name, weight, profit, quantity)

item_name	**weight**	**profit**	**quantity**
item 1	6.0	10.0	0
item 2	4.0	7.0	0
item 3	4.5	8.0	0
item 4	8.0	13.0	0

b)
Change your solution to solve the 0-1 knapsack problem instead. This imposes the restriction that the assigned quantity values must be either 0 or 1.

A.2 Sample Sheet

The following is the sample sheet used for the experiment. Note that the syntax description of SolveDB shown is a simplified version, and the constrained optimization problem example as well as the solution query were taken from the Daisy page on SolveDB[42]. The flex-object scheduling query is based on the flex-object query shown in the SolveDB article[5].

SolveDB Cheat Sheet

SolveDB integrates constrained optimization problem solving directly into SQL-queries. SolveDB allows for so-called solve queries by introducing the SOLVESELECT clause. A solve query has the following syntax:

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

A constrained optimization problem consists of an objective function and a number of constraints. The goal is to find values for a set of unknown variables such that the objective function is either minimized or maximized, where the values of the unknown variables uphold the constraints. Below is a simple example of an optimization problem with two unknown variables and two constraints:

Maximize: $0.6x_1 + 0.5x_2$
Subject to:
$x_1 + 2x_2 \leq 1$
$3x_1 + x_2 \leq 2$

This optimization problem can be solved by the following query in SolveDB:

```
1 SOLVESELECT x1, x2 IN (SELECT x1, x2 FROM data) AS u
2 MAXIMIZE (SELECT 0.6*x1 + 0.5*x2 FROM u)
3 SUBJECTTO (SELECT x1+2*x2 <=1 FROM u),
4           (SELECT 3*x1+x2 <= 2 FROM u)
5 WITH solverlp();
```

The solution in this case is $x_1 = 1$, $x_2 = -1$

Other examples of solve queries

Giving raises to employees:

```
SOLVESELECT new_salary IN
    (SELECT id, name, age, current_salary, new_salary
     FROM employee) as r_in
MINIMIZE (SELECT sum(new_salary - current_salary) FROM r_in)
SUBJECTTO
    (SELECT new_salary >= 1.10 * current_salary
     FROM r_in
     WHERE age > 40),
    (SELECT new_salary >= (SELECT avg(currrent_salary) FROM
       r_in)
     FROM r_in)
WITH solverlp();
```

Flex-object scheduling query:

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

Activity scheduling query:

```
SOLVESELECT hours IN
    (SELECT a_id, a_name, a_profit, NULL::FLOAT4 AS hours
     FROM activities) AS t
MAXIMIZE
    (SELECT sum(hours * a_profit) FROM t)
SUBJECTTO
    (SELECT hours >= 0 FROM t),
    (SELECT sum(hours * a.r_cost) <=
        (SELECT r.r_amount FROM resources r WHERE r.r_id =
           a.r_id)
```

```
10      FROM t INNER JOIN act_res a ON t.a_id = a.a_id
11      GROUP BY a.r_id)
12  WITH solverlp();
```

Appendix B

Performance Experiments

B.1 Data Generation Scripts

The following Scala scripts were used to generate the test data used for the experiments.

B.1.1 Flexobject Experiment

```
1  def generateFlexObjects(amount: Int, sliceAmount: Int, seed
       : Int = 0): List[(Int, Int, Double, Double)] = {
2    var r = new scala.util.Random
3    if (seed != 0)
4      r = new scala.util.Random(seed)
5      val result = (0 to amount - 1).flatMap(i => (0 to
         sliceAmount - 1)
6        .map(j => {
7          val min = r.nextInt(10).toDouble
8          val max = min + r.nextInt(10).toDouble
9          if (i % 2 == 0)
10           (i, j, min, max)
11         else
12           (i, j, -max, -min)
13       }))
14   result.toList
15 }
```

B.1.2 Knapsack Experiment

```
def generateKnapsackProblem(items: Int, categories: Int =
    1): Seq[(String, Double, Double, Int, Int)] = {
  val r = new scala.util.Random(8)
  var i = 0
  (1 to categories).flatMap(category =>
  {
    (1 to items).map(item => {
      val weight = 1 + r.nextInt(7).toDouble
      val profit = 1 + r.nextInt(11).toDouble
      (s"item ${category}_$item", weight, profit, category,
          0)
    })
  })
}
```

B.2 Performance Results

Table B.1.: SolveDB results for the knapsack experiment.

Test case	Items per category	Categories	Partitioning (s)	Solving (s)	Total solving time (s)	Total query time (s)
1	1000	1	0.000197	0.034735	0.036413	0.076
2	1000	5	0.002833	0.239689	0.247060	0.332
3	1000	10	0.006027	0.573048	0.587914	0.733
4	1000	20	0.011793	1.250665	1.280465	1.546
5	10000	1	0.001892	8.700339	8.718208	8.860
6	10000	5	0.028448	38.173300	38.258963	38.914
7	10000	10	0.063910	72.333700	72.504322	73.794
8	10000	20	0.127880	146.410104	146.743293	149.346
9	20000	1	0.003837	21.782770	21.823697	22.088
10	20000	5	0.051334	197.054578	197.267551	198.549
11	20000	10	0.103605	354.313020	354.776651	357.385
12	20000	20	0.209928	823.228627	824.117938	849.263
13	30000	1	0.006166	109.493241	109.570802	109.960
14	30000	5	0.079939	614.011008	614.395631	616.359
15	30000	10	0.149665	959.354045	959.993860	963.846
16	30000	20	0.306307	2176.636156	2177.938338	2185.625
17	40000	1	0.008125	243.863668	243.969899	244.479
18	40000	5	0.103911	1247.217043	1247.747884	1250.359
19	40000	10	0.202089	2273.935586	2274.934365	2280.089
20	40000	20	0.420395	4434.017902	4436.034668	4446.413
21	50000	1	0.011307	418.45404	418.596917	419.244
22	50000	5	0.126202	1605.964286	1606.622000	1609.87
23	50000	10	0.262116	2909.682967	2910.998416	2917.506
24	50000	20	0.519682	5463.076890	5465.730893	5478.660

Table B.2.: SolveDFresults for the knapsack experiment on 1 machine using all of its cores.

Test case	Items per category	Categories	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	1	1.83184	1.29559	0.05271	0.26817	3.44833
2	1000	5	1.86116	1.76385	0.14258	0.86678	4.63436
3	1000	10	2.11940	2.36188	0.21658	1.19112	5.88900
4	1000	20	1.93256	3.11462	0.52519	1.85922	7.43162
5	10000	1	192548	2.43359	0.19702	14.5769	19.1330
6	10000	5	1.94600	3.92162	0.65603	32.3350	38.8587
7	10000	10	1.86531	5.76653	1.09641	54.7168	63.4451
8	10000	20	1.98431	8.09862	2.1977	95.0986	107.379
9	20000	1	2.08725	3.08441	0.51295	57.2786	62.9632
10	20000	5	2.00972	5.63132	1.14672	142.416	151.204
11	20000	10	2.03759	8.16278	2.22239	238.081	250.504
12	20000	20	2.05396	12.9353	4.28366	429.670	448.943
13	30000	1	1.91782	3.66648	0.43618	104.853	110.87
14	30000	5	2.02903	7.01539	1.95913	365.046	376.049
15	30000	10	1.99359	10.6430	3.40936	686.182	702.228
16	30000	20	2.23586	18.0513	6.58685	1351.28	1378.1
17	40000	1	1.89297	3.76199	0.55130	214.012	220.21
18	40000	5	1.99651	8.22364	2.35332	907.120	919.693
19	40000	10	2.18670	12.7011	4.39288	1673.62	1692.90
20	40000	20	2.17868	21.0728	9.05302	3228.45	3260.75
21	50000	1	1.88023	4.10221	0.60487	333.526	340.113
22	50000	5	1.97917	10.1649	2.93841	1430.90	1445.98
23	50000	10	2.12284	14.9473	5.37411	2830.12	2852.56
24	50000	20	2.35437	26.0828	12.2698	5573.82	5614.53

Table B.3.: SolveDFresults for the knapsack experiment with Spark restricted to using only 1 core.

Test case	Items per category	Categories	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	1	1.793498846	1.250139958	0.051235641	0.277747127	4.083263766
2	1000	5	1.799558411	1.778027242	0.134435546	1.044416174	5.437982698
3	1000	10	1.850839782	2.225807003	0.192987242	1.613107622	6.561690906
4	1000	20	1.94895647	2.876518753	0.392780099	2.662646771	8.580235544
5	10000	1	1.930837787	2.293077063	0.297847001	14.59459103	19.8952817
6	10000	5	1.974886774	4.089188493	0.602030537	67.51023515	74.81306558
7	10000	10	1.888689801	5.3922989	1.023364521	133.147552	142.1129383
8	10000	20	2.091030665	7.588350771	2.100117751	257.7392485	270.2097576
9	20000	1	2.010657365	2.938488562	0.340084014	56.93010978	62.88215143
10	20000	5	2.167876597	5.58994391	1.111524768	266.1590879	275.7231679
11	20000	10	2.071365979	8.019183362	2.177545502	536.4883105	549.4231064
12	20000	20	2.355353693	12.57294421	4.746734164	1043.655041	1064.040165
13	30000	1	2.132300799	3.524584682	0.485307946	104.4773153	111.3123371
14	30000	5	2.231152747	6.904874316	1.7066054	549.7409991	561.3161027
15	30000	10	2.337665632	10.5182359	3.788324157	1089.968553	1107.296471
16	30000	20	2.079559384	16.86295164	6.291272097	2217.689149	2243.619042
17	40000	1	2.063112384	3.839996365	0.501930267	213.8917506	221.0295439
18	40000	5	1.964249964	7.955149138	2.204489327	1026.120961	1038.929484
19	40000	10	2.285773784	12.43086838	4.488843595	2083.630669	2103.555141
20	40000	20	2.449216013	21.18401089	8.632441544	4034.306377	4067.321747
21	50000	1	1.876340225	4.524157865	0.576021278	334.5849277	342.2244753
22	50000	5	1.941504593	9.650400829	2.749478309	1574.33116	1589.358402
23	50000	10	2.039450548	15.44316421	5.707534055	3186.931566	3210.782198
24	50000	20	2.506555368	26.26091845	11.44948562	6189.382674	6230.278378

Table B.4.: SolveDFresults for the knapsack experiment on a cluster of 2 machines.

Test case	Items per category	Categories	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	1	2.764339488	4.771588434	0.081968338	0.375327064	8.641844732
2	1000	5	2.871038768	3.76269752	0.174927737	1.307756542	8.766361123
3	1000	10	2.773950761	5.602444497	0.245999538	1.047785298	10.26409562
4	1000	20	2.847766816	5.220783749	0.406023705	1.825142693	10.92442936
5	10000	1	2.836184411	4.769586325	0.287827539	15.28709561	23.78031148
6	10000	5	3.197393999	6.14899562	0.737019219	19.1897906	29.94857432
7	10000	10	2.894779908	8.354186499	1.057790239	32.65404294	45.573425
8	10000	20	2.892990037	9.30380729	1.883148509	51.53256364	66.25141762
9	20000	1	2.889164243	4.709130705	0.303312072	57.19843582	65.71643895
10	20000	5	3.112960489	7.318389619	1.11182142	71.59487843	83.76035765
11	20000	10	2.979818519	10.17183911	1.963657713	143.3576421	159.1383857
12	20000	20	3.072507402	12.23222937	3.591353487	227.2984866	246.8455612
13	30000	1	2.799432165	5.339118437	0.471241161	105.5718451	114.7693868
14	30000	5	3.076652322	8.061555263	1.496613471	194.7856819	207.9884111
15	30000	10	2.967161938	12.13550914	2.964676221	338.2599553	356.9709105
16	30000	20	3.283090259	17.26184047	7.850018315	646.4595441	675.4919997
17	40000	1	3.038122302	5.877708351	0.589654214	212.78973	222.9235351
18	40000	5	3.247127476	8.994943387	1.978889371	396.4897146	411.3314238
19	40000	10	3.175018353	14.86062545	3.590537882	982.8180678	1005.012841
20	40000	20	3.410436014	22.37394836	10.23963136	1679.856545	1716.522146
21	50000	1	2.936940888	7.343996257	0.650524947	333.1711	344.7651346
22	50000	5	3.247736593	10.99294459	2.392510289	712.314732	729.6243217
23	50000	10	3.102317498	15.19922216	7.02133726	1504.418633	1530.390983
24	50000	20	3.740208426	28.03165873	12.07708726	2897.333715	2941.823681

Table B.5.: SolveDFresults for the knapsack experiment on a cluster of 4 machines.

Test case	Items per category	Categories	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	1	2.838999249	4.412304036	0.066916954	0.520340465	8.489570581
2	1000	5	2.765757625	3.845728559	0.1630185	1.164778165	8.549093765
3	1000	10	2.863642142	4.583384482	0.294900645	1.371798541	9.736133879
4	1000	20	2.781010731	6.010089316	0.299668671	1.945240536	11.68589503
5	10000	1	2.754613938	5.420561626	0.25057849	14.74438148	23.77861714
6	10000	5	2.815271789	7.092681763	0.647143863	16.84967238	28.01574344
7	10000	10	2.898667062	8.173279317	0.9830232	19.84691063	32.52882345
8	10000	20	2.838440671	10.42895634	1.887138085	39.13578429	54.93929687
9	20000	1	3.111199425	5.873218848	0.374306033	57.54446336	67.48259843
10	20000	5	3.097333304	8.385186627	1.030289156	62.11686824	75.23996127
11	20000	10	2.920909502	9.996125372	1.987706705	76.5737519	92.06293006
12	20000	20	3.223418329	13.63421668	3.576430577	134.6781355	155.7936153
13	30000	1	2.785606257	5.328151692	0.39057471	105.1867433	114.3052086
14	30000	5	2.8528272	7.955078857	1.455093959	136.1846604	149.0514157
15	30000	10	2.97982841	11.61733639	2.823439377	187.5533837	205.5534513
16	30000	20	3.101231064	16.54613205	5.781741809	355.5010005	381.5512192
17	40000	1	2.794023934	6.711555863	0.579736912	212.4930087	223.2077164
18	40000	5	2.928819683	10.03748977	1.906173987	235.5910484	251.0644871
19	40000	10	3.153958137	13.20724627	3.734998191	543.3096641	564.0989089
20	40000	20	3.010855712	20.2632161	10.74909594	932.825782	967.5037793
21	50000	1	3.090208978	6.770206623	0.701727409	333.5298709	344.7339695
22	50000	5	2.925490487	10.10903176	2.582429107	466.1665337	482.3754069
23	50000	10	2.995998071	15.79063765	7.000011675	845.1977447	871.6090325
24	50000	20	3.198091622	27.15355079	12.24896427	1643.837991	1687.016664

Table B.6.: SolveDFresults for the knapsack experiment on a cluster of 8 machines.

Test case	Items per category	Categories	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	1	2.874069013	4.426668459	0.070535354	1.055755285	9.060928016
2	1000	5	3.003195765	5.085196277	0.170157539	1.130065725	10.00408268
3	1000	10	3.023588893	5.364895739	0.265774827	1.537895137	10.82621149
4	1000	20	2.908353715	6.690695344	0.406051279	1.583302577	12.22543613
5	10000	1	2.826714087	5.43440468	0.246315977	15.32900998	24.46193862
6	10000	5	2.960945791	7.095356644	0.78233076	17.53582622	29.00079321
7	10000	10	3.034848168	7.953822094	1.200235989	21.93370448	34.76165879
8	10000	20	3.008675625	9.889928006	2.262767737	21.78187152	37.60379976
9	20000	1	3.073919315	4.747206469	0.349375074	57.10141068	65.95509714
10	20000	5	2.909576283	8.081846719	1.318598805	61.07302895	73.98677347
11	20000	10	2.991686698	10.48257819	2.50789215	77.73540139	94.35236068
12	20000	20	3.19359983	14.11712138	4.42453006	85.50712612	107.9587289
13	30000	1	3.000314978	6.598046889	0.600706886	105.3156456	116.1494012
14	30000	5	3.306062391	9.385600932	1.811564364	130.6737435	145.791299
15	30000	10	3.239641175	11.86719104	3.729235973	131.8564802	151.4050981
16	30000	20	3.526857327	19.07979276	9.379746699	176.9261245	209.7366833
17	40000	1	3.003887931	6.634905338	0.601430827	212.4532739	223.3017958
18	40000	5	3.068239953	10.21979892	2.386177098	225.2814512	241.5817053
19	40000	10	3.382379784	13.6546512	4.787167408	254.6300229	277.0964946
20	40000	20	3.439122757	22.31475462	9.395504629	439.8222098	475.6697813
21	50000	1	2.950313166	6.312396938	0.784250386	327.2279772	337.8992983
22	50000	5	3.008347259	10.35446111	3.314834784	355.6892529	373.017167
23	50000	10	3.45021714	15.69958711	8.031117147	490.8219508	518.6340589
24	50000	20	3.603727643	29.54087618	15.62783056	901.621019	951.0659186

Table B.7.: SolveDB results for the flexobject experiment.

Test case	Flex-objects	Time intervals	Partitioning (s)	Solving (s)	Total solving time (s)	Total query time (s)
1	1000	3	0.001231	0.003705	0.00788	0.09
2	1000	5	0.002045	0.005811	0.012774	0.126
3	1000	10	0.004129	0.011384	0.026169	0.221
4	1000	20	0.008702	0.022573	0.053362	0.418
5	5000	3	0.006294	0.020504	0.04432	0.33
6	5000	5	0.011177	0.032004	0.07435	0.524
7	5000	10	0.022026	0.064732	0.152853	1.053
8	5000	20	0.044735	0.128235	0.313489	2.089
9	10000	3	0.012704	0.04056	0.090762	0.632
10	10000	5	0.022727	0.069515	0.162805	1.058
11	10000	10	0.044492	0.137368	0.335974	2.113
12	10000	20	0.100414	0.275763	0.690089	4.219
13	25000	3	0.033443	0.131321	0.296953	1.623
14	25000	5	0.059538	0.228984	0.541585	2.767
15	25000	10	0.118708	0.436807	1.076982	5.468
16	25000	20	0.252738	0.820591	2.089813	10.821
17	50000	3	0.067891	0.287114	0.686903	3.331
18	50000	5	0.118507	0.510516	1.214833	5.587
19	50000	10	0.221757	0.98178	2.400821	11.19
20	50000	20	0.467269	1.973622	4.895806	22.334
21	100000	3	0.132937	0.729039	1.552951	6.774
22	100000	5	0.223669	1.193978	2.61712	11.313
23	100000	10	0.495333	2.21673	5.238106	22.507
24	100000	20	1.08152	4.43679	10.68769	45.849

Table B.8.: SolveDF results for the flex-object experiment with GLPK as the solver, and using all 4 cores of a single machine.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	3.94987	2.04626	0.15040	0.26940	6.41595
2	1000	5	3.99333	2.44113	0.17526	0.42411	7.03384
3	1000	10	4.07134	2.77392	0.19203	0.48254	7.51984
4	1000	20	4.27337	3.56261	0.47647	0.78680	9.09926
5	5000	3	4.06819	2.96537	0.35771	0.68650	8.07780
6	5000	5	4.06538	3.40278	0.51854	1.03805	9.02476
7	5000	10	3.91706	4.08980	0.83019	1.45270	10.2897
8	5000	20	4.38159	5.35966	1.27815	2.38501	13.4044
9	10000	3	4.16735	3.44724	0.47523	1.08955	9.17939
10	10000	5	4.28507	4.12981	0.83059	2.12295	11.3684
11	10000	10	4.14873	5.13349	1.22751	3.13503	13.6447
12	10000	20	4.51582	7.32894	2.13152	5.58295	19.5592
13	25000	3	4.20815	4.70281	0.92078	4.00481	13.8365
14	25000	5	4.41676	6.40381	1.30708	7.42151	19.5491
15	25000	10	4.56460	8.49336	2.77904	12.2009	28.0379
16	25000	20	4.95646	12.5487	5.73010	21.1301	44.3654
17	50000	3	4.26414	6.58752	1.65829	11.2182	23.7282
18	50000	5	4.28234	8.28066	3.35441	20.9277	36.8451
19	50000	10	4.68807	13.1229	5.60599	35.4323	58.8494
20	50000	20	4.96907	20.4072	11.2996	68.7613	105.437
21	100000	3	4.31637	9.81192	3.04043	40.3814	57.5501
22	100000	5	4.69077	13.6899	5.56684	72.9560	96.9036
23	100000	10	5.37417	20.9774	11.8531	120.582	158.786
24	100000	20	6.00104	36.5736	23.7124	239.921	306.208

Table B.9.: SolveDF results for the flex-object experiment with GLPK as the solver, and using 1 core of a single machine.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	4.817232668	2.309073143	0.123155606	0.328936853	8.226830501
2	1000	5	4.717116501	3.130696524	0.17642493	0.351351223	9.153361177
3	1000	10	4.75294075	3.466620729	0.300656776	0.538067845	9.81905606
4	1000	20	4.528786152	3.522587969	0.615825875	0.894846464	10.23319844
5	5000	3	4.705518038	3.624601301	0.398544371	0.791612851	10.11980007
6	5000	5	4.681147671	4.330284653	0.838694687	1.556401279	12.2165098
7	5000	10	4.349730163	4.424989458	0.837479	1.890359036	12.12945811
8	5000	20	4.572540975	5.254539802	1.045741426	3.244259333	14.73907363
9	10000	3	4.1321	3.798200279	0.498826077	1.513597838	10.50666978
10	10000	5	4.145588615	4.687526811	0.674349397	2.278000293	12.45482584
11	10000	10	4.545181926	5.668633811	1.291146106	4.453656562	16.55130247
12	10000	20	4.644543347	7.284544897	1.954099059	8.481697364	22.94139645
13	25000	3	4.377350127	5.145805725	1.190894058	6.288862433	17.63719577
14	25000	5	4.60137368	6.208023939	1.478429163	10.29780526	23.16895133
15	25000	10	4.44343616	8.689926519	2.783917757	19.21350612	35.77179456
16	25000	20	4.758315556	12.37393956	5.143910029	38.14904527	60.97851067
17	50000	3	4.600230882	6.713481303	1.729451092	20.81827359	34.43676095
18	50000	5	4.522812351	8.464311777	2.921838604	33.30909842	49.80556231
19	50000	10	5.055347285	12.89472939	5.582149002	69.1068617	93.23979928
20	50000	20	5.317862665	20.95940897	11.48358289	137.5091618	175.8826795
21	100000	3	4.328394778	9.283836931	3.214689721	75.58934858	93.05298381
22	100000	5	4.788981045	13.01848247	5.411326806	127.7300951	151.5263232
23	100000	10	5.307487562	21.30798645	12.28347661	243.8709104	283.3840832
24	100000	20	6.471663508	38.96461923	24.01788428	501.8251207	571.9040741

Table B.10.: SolveDF results for the flex-object experiment with GLPK as the solver run on a cluster of 2 nodes.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	6.453303728	4.00839088	0.121796718	0.394149175	11.55084025
2	1000	5	6.447078435	5.065114693	0.13528432	0.407883228	12.65189908
3	1000	10	5.67104583	5.599641507	0.278105229	0.511170353	12.66747799
4	1000	20	6.352442445	5.222795804	0.307219918	0.609649798	13.0505222
5	5000	3	5.913792637	5.391691319	0.357735693	0.651898038	12.848441
6	5000	5	6.072631769	5.493235509	0.441010795	1.22070149	13.85456837
7	5000	10	6.830928783	6.386600677	0.669366703	1.433451877	15.91266976
8	5000	20	6.287127549	7.043055178	1.083399658	1.825249433	16.76874836
9	10000	3	6.141600415	5.289288133	0.434121668	1.320967155	13.7649127
10	10000	5	6.43608313	5.946749792	0.539594265	1.576397453	15.00968525
11	10000	10	6.063174235	7.192175378	1.2245532	2.487201035	17.52837937
12	10000	20	6.093702789	8.760613439	1.869688278	3.503729975	20.77496197
13	25000	3	6.245657718	7.018896372	0.888188055	3.112506979	17.7997278
14	25000	5	6.471199227	7.630233875	1.348604711	4.533375984	20.50048662
15	25000	10	6.884265833	8.531021412	2.530868484	7.708151469	26.18292368
16	25000	20	6.609198304	12.22256547	4.595048258	13.64586003	37.60346429
17	50000	3	6.576530885	8.029532839	1.511142213	8.435688314	25.15317553
18	50000	5	6.333111837	9.146232339	2.367489087	12.06217165	30.46617271
19	50000	10	6.900364511	12.33266587	7.054871743	23.02219366	49.8250082
20	50000	20	7.080472141	20.3744154	12.2688272	35.72897436	75.96424275
21	100000	3	6.544129007	10.09037177	2.953818727	30.00240641	50.09605065
22	100000	5	6.849442171	11.9311027	4.651002873	42.21875478	66.16434353
23	100000	10	7.816714286	20.32100532	12.37172131	76.78296127	117.8098343
24	100000	20	7.841586733	35.50093174	32.4078332	133.3217755	209.5937009

Table B.11.: SolveDF results for the flex-object experiment with GLPK as the solver run on a cluster of 4 nodes.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	5.546171408	4.722149548	0.13663527	0.339811839	11.31158749
2	1000	5	5.965503418	4.142039496	0.153768971	0.380794993	11.19779977
3	1000	10	5.737588801	5.373278774	0.271661527	0.508445244	12.46867924
4	1000	20	5.696388358	5.227567531	0.330440763	0.647950186	12.45350817
5	5000	3	6.013503038	4.64884821	0.2733321	0.786807628	12.27201017
6	5000	5	5.751431144	5.741788177	0.37334366	0.756302014	13.1734532
7	5000	10	6.114203389	6.009090247	0.684741087	1.203871916	14.5629923
8	5000	20	5.937887005	7.32077137	1.066136496	1.656669397	16.48164973
9	10000	3	5.808269265	5.374174635	0.345936968	1.243246098	13.28933978
10	10000	5	6.156053395	5.649036261	0.658032349	1.497440594	14.48293595
11	10000	10	6.284268235	6.93665874	1.035332365	2.10979931	16.91225416
12	10000	20	6.389650921	8.433352166	1.936256661	3.252517412	20.56324994
13	25000	3	6.254349887	6.51503598	0.817644069	2.939309173	17.05378497
14	25000	5	6.146268028	7.497301229	1.190047778	4.546754388	19.89802946
15	25000	10	6.352114322	9.007038688	2.447574751	5.182989252	23.49957781
16	25000	20	6.680012347	12.21393778	6.300598012	10.34534349	36.07726139
17	50000	3	6.116718467	7.601453167	1.390377518	8.146809065	23.77808563
18	50000	5	6.37607979	9.191602081	2.341936169	13.46726097	31.86710585
19	50000	10	6.555654789	12.14818201	7.043636602	15.18158509	41.43775258
20	50000	20	7.350417763	20.81183462	11.81278074	30.49110156	70.98927485
21	100000	3	6.097458408	10.50331058	3.175033538	28.23357154	48.54715376
22	100000	5	7.063002621	12.03820739	7.633271827	29.55128284	56.83374
23	100000	10	7.278147699	20.33067721	12.06143765	47.3688362	87.59206486
24	100000	20	8.126481221	37.43357423	25.4870456	105.8351997	177.3924994

Table B.12.: SolveDF results for the flex-object experiment with GLPK as the solver run on a cluster of 8 nodes.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	5.83348273	4.150612258	0.080024089	0.349277879	10.95621981
2	1000	5	6.265806093	5.134791739	0.151867767	0.410334178	12.51280567
3	1000	10	6.141867647	5.039088422	0.248156765	0.564669149	12.559532
4	1000	20	6.399305851	5.281617198	0.305968513	0.707853449	13.28966681
5	5000	3	6.049483635	5.167443085	0.31065149	0.678378703	12.75531965
6	5000	5	5.974236956	6.027909467	0.370678348	0.809294341	13.74025681
7	5000	10	6.575157726	6.443109564	0.626989035	1.271845591	15.52397524
8	5000	20	6.295315788	8.119715501	1.208966722	1.64759276	17.82196691
9	10000	3	6.515882854	6.10308791	0.513564892	1.219729621	14.89649622
10	10000	5	6.566801265	6.843259712	0.769625415	1.281324249	16.01001241
11	10000	10	6.187748528	7.067037484	1.175919186	2.123119721	17.09210962
12	10000	20	6.589483268	8.933172363	2.106016085	3.066489536	21.24922621
13	25000	3	6.270187706	6.941894155	1.063270751	3.022650132	17.82242921
14	25000	5	6.398151676	7.560767646	1.482466777	3.582342324	19.5717873
15	25000	10	6.431074797	9.429627819	2.825307094	4.580742022	23.80014749
16	25000	20	7.05748922	13.12468955	5.883562373	7.770915538	34.40069793
17	50000	3	6.859632756	8.109833773	1.785486132	8.22393208	25.52520835
18	50000	5	6.836176806	9.402221203	2.861798518	9.069556544	28.7132155
19	50000	10	6.681548137	13.06815866	7.411750032	11.56181678	39.2745656
20	50000	20	7.705499602	20.37311441	14.28846745	18.59889615	61.54106053
21	100000	3	6.686644917	11.00264528	3.277468353	28.91122991	50.42782635
22	100000	5	6.932051124	12.93758022	8.231327304	29.69271387	58.36237429
23	100000	10	7.282432692	20.49521935	14.62568778	50.29102173	93.21275902
24	100000	20	8.778870641	39.8160996	30.23659855	63.21719314	142.5841847

Table B.13.: SolveDF results for the flex-object experiment with Clp as the solver, and using all 4 cores of a single machine.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	4.218749599	2.046354612	0.113988322	0.457015919	7.670234729
2	1000	5	3.7159783	2.522961355	0.14837371	0.448021697	7.542632232
3	1000	10	3.935315007	2.623947648	0.272035275	0.66166631	8.293492324
4	1000	20	4.06463029	3.401639142	0.411489842	0.874923691	9.489278459
5	5000	3	4.020081303	3.153398137	0.271388776	0.852481199	9.017388181
6	5000	5	4.100298025	3.611402959	0.555767312	1.044950189	9.927863821
7	5000	10	3.924683239	4.275600808	0.840809283	1.198503904	10.91171128
8	5000	20	4.135499237	5.041851675	1.059163289	1.870595715	12.70447361
9	10000	3	3.910231719	3.739442315	0.529568902	1.039087412	9.910131864
10	10000	5	3.955017538	4.256968754	0.749761467	1.336194208	10.94669462
11	10000	10	4.123914047	4.885228416	1.393314009	2.031302821	12.99304472
12	10000	20	4.356060352	7.27160188	2.201318526	4.042236047	18.49911703
13	25000	3	3.941157112	4.959715956	0.959298736	2.032349446	12.55783772
14	25000	5	4.096794333	6.07887508	1.37413412	3.165969637	15.37159343
15	25000	10	4.365169462	8.064454848	2.720268689	5.147888792	20.92700186
16	25000	20	4.601615739	12.21145473	5.432781525	8.452961766	31.28906311
17	50000	3	4.169804665	6.30618098	1.588417813	4.047989864	16.81310423
18	50000	5	4.665819925	8.144492921	2.841169674	5.877075082	22.1271264
19	50000	10	4.579371977	12.57665757	5.184744766	10.06398997	33.00368918
20	50000	20	4.998459152	19.63356755	10.60175452	18.24598055	54.20319446
21	100000	3	4.308125278	10.02140472	2.950973729	6.867496998	24.75661447
22	100000	5	4.379349368	12.99109121	5.367281612	11.4644678	34.81869465
23	100000	10	5.127819584	20.66229721	11.33564949	20.54840622	58.28496551
24	100000	20	6.21382865	37.21307473	24.5065139	40.31718603	108.8795686

Table B.14.: SolveDF results for the flex-object experiment with Clp as the solver, and using 1 core of a single machine.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	4.346369811	2.362143245	0.103449422	0.47018477	7.950218542
2	1000	5	4.154960543	2.762048169	0.160045791	0.54851488	8.382141049
3	1000	10	4.187238244	3.043544443	0.354082385	0.812174554	9.030774978
4	1000	20	4.427314273	3.46461886	0.565219202	0.926101119	9.991014387
5	5000	3	4.205262617	3.221478575	0.390319778	0.872520916	9.433640714
6	5000	5	4.225415898	3.706753555	0.786202972	1.04860907	10.36579777
7	5000	10	4.681418497	4.512315677	0.830147389	1.481767076	12.0867529
8	5000	20	4.463432616	5.475447202	1.074823456	2.381685825	13.99775993
9	10000	3	4.312376691	3.891367077	0.516466365	0.992394685	10.29415053
10	10000	5	4.322240947	4.572951456	0.801733465	1.473316431	11.76213726
11	10000	10	4.259194644	5.391118879	1.064041437	2.35744822	13.66577825
12	10000	20	4.769547044	7.037571964	2.192532715	5.091260174	19.67262693
13	25000	3	4.366717715	4.933262125	0.820569003	2.627321109	13.4340932
14	25000	5	4.351874081	6.182718173	1.210328979	3.030108537	15.41311905
15	25000	10	4.777510688	8.375433558	2.882923001	5.412478512	22.02796945
16	25000	20	4.68925268	11.57572767	5.079956944	10.61962426	32.54228574
17	50000	3	4.35298662	6.826424922	1.76964016	3.947459315	17.49640104
18	50000	5	4.395984225	8.389983191	2.893280652	6.250144554	22.50355019
19	50000	10	4.883840621	12.97434143	5.372112371	12.36189656	36.19992114
20	50000	20	5.373296472	21.46599044	11.15774187	22.66512172	61.24941486
21	100000	3	4.683472077	9.550431702	3.283027309	8.891371667	27.0704511
22	100000	5	5.123161603	12.68066432	5.517543738	13.39355278	37.40008868
23	100000	10	5.110031364	20.93859091	11.22855669	25.33461971	63.22039113
24	100000	20	6.337139541	37.41275763	23.79748977	53.00493463	121.1544888

Table B.15.: SolveDF results for the flex-object experiment with Clp as the solver run on a cluster of 2 nodes.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	5.791399079	4.697865776	0.202918451	0.585866826	11.89657502
2	1000	5	5.966473784	4.4691293	0.151356455	0.550861121	11.70159481
3	1000	10	6.257310094	4.858967467	0.158698265	0.802799726	12.73519205
4	1000	20	6.495183982	5.071143246	0.31891948	0.800114934	13.3283744
5	5000	3	6.079051803	5.288251671	0.333471264	0.788764371	13.02487481
6	5000	5	5.936646144	5.803632327	0.463344923	0.954747609	13.69142503
7	5000	10	6.000688734	6.333900978	0.56684582	1.396967799	14.90304615
8	5000	20	6.335499907	7.079570131	1.086168505	1.774097471	16.82726445
9	10000	3	6.192025032	6.047314782	0.670880615	1.179292676	14.69642126
10	10000	5	6.293681687	6.344041483	0.625346837	1.275781182	15.07612894
11	10000	10	6.149658739	7.507011258	1.109987527	1.987314573	17.26012543
12	10000	20	6.502218682	8.43418409	1.891511236	2.92019947	20.23710198
13	25000	3	6.322961011	7.363699727	1.031621473	1.912438196	17.1689535
14	25000	5	6.290048182	7.707402847	1.380439032	2.638300228	18.54811792
15	25000	10	6.280015138	8.852576926	2.520235261	3.762275339	21.98709584
16	25000	20	6.689714439	11.82848609	7.51290479	6.202835395	32.76884856
17	50000	3	6.199599391	7.367401532	1.414639295	2.899983976	18.41678837
18	50000	5	6.265146713	9.098076672	2.517527831	4.059177548	22.50738539
19	50000	10	7.190867438	12.00491886	7.163333701	7.399278129	34.27888367
20	50000	20	7.150304305	19.47853763	11.97021324	11.59919412	50.71634844
21	100000	3	6.419295109	10.05759542	2.637567529	4.921342626	24.56665442
22	100000	5	6.718431923	13.20385361	6.924036755	8.006104205	35.37649708
23	100000	10	7.312484439	20.40039634	12.3493691	13.94065956	54.5208473
24	100000	20	8.359743102	37.77676788	25.10042909	25.62728817	97.4054009

Table B.16.: SolveDF results for the flex-object experiment with Clp as the solver run on a cluster of 4 nodes.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	5.782649802	3.981911522	0.113992858	0.560282139	10.9774878
2	1000	5	5.788385948	4.153292895	0.135354053	0.604443303	11.23011219
3	1000	10	5.86991936	5.037291326	0.277217272	0.765849775	12.49222849
4	1000	20	6.006103957	5.115711043	0.369063369	0.86727789	12.89605512
5	5000	3	6.039514027	4.551202627	0.214097507	0.914601404	12.2509649
6	5000	5	6.16536435	5.659362112	0.305567992	0.991031206	13.66438315
7	5000	10	5.970822701	6.940839425	0.705473189	1.236761527	15.41001939
8	5000	20	6.334289039	7.527741796	0.993875074	1.751286439	17.15042497
9	10000	3	5.930124448	5.607948003	0.46138865	1.261663389	13.77967721
10	10000	5	6.364867013	6.152099668	0.630398179	1.592847201	15.24226121
11	10000	10	6.361398543	6.879514844	1.009263216	1.984537782	16.79940324
12	10000	20	6.40604746	8.777515784	1.854331497	2.816752613	20.36153615
13	25000	3	6.241262213	6.702314233	1.041761934	1.798736386	16.29778098
14	25000	5	6.21821355	7.653273005	1.178331149	2.241427428	17.86604878
15	25000	10	6.575321767	8.709399367	2.508396569	3.329819714	21.65466509
16	25000	20	6.746773767	12.01905721	7.075880064	5.740826966	32.1193429
17	50000	3	6.129108979	8.150964518	1.487498403	2.848703748	19.12060874
18	50000	5	6.394166668	9.543491048	2.658536092	3.693721558	22.80208416
19	50000	10	6.841492172	12.23987057	7.08838202	6.412791446	33.10530372
20	50000	20	7.250810183	20.52974035	12.49660384	10.6579386	51.47654093
21	100000	3	6.30293684	9.577774943	2.637534657	4.676377451	23.71980305
22	100000	5	7.131378053	12.34785185	6.833885108	6.729274551	33.55358965
23	100000	10	6.953394391	19.96738377	12.47287964	12.03291509	51.95301815
24	100000	20	8.038535534	39.2897232	25.50308614	22.70250288	96.05924978

Table B.17.: SolveDF results for the flex-object experiment with Clp as the solver run on a cluster of 8 nodes.

Test case	Flex-objects	Time intervals	Building query (s)	Materializing query (s)	Partitioning (s)	Solving (s)	Total query time (s)
1	1000	3	6.431379084	4.83651274	0.103902439	0.708218654	12.63152038
2	1000	5	6.342835868	4.485847277	0.106604961	0.634233309	12.13832173
3	1000	10	5.989553073	5.528559072	0.154004753	0.783841284	13.01645872
4	1000	20	6.634165887	5.874510588	0.372269964	0.955503105	14.38253275
5	5000	3	6.333649359	5.343029556	0.331103371	0.888873877	13.4306974
6	5000	5	6.07185071	5.862091427	0.366089503	0.93308883	13.80207156
7	5000	10	6.552790504	6.895117486	0.665999996	1.222712704	15.87995131
8	5000	20	6.165038913	7.515596397	1.148494605	1.850238399	17.33030791
9	10000	3	6.306902427	5.724400396	0.467672509	1.15324503	14.17382848
10	10000	5	6.009163172	6.788745696	0.638874462	1.58186715	15.5770122
11	10000	10	6.354139417	6.971342438	1.085205199	1.720694941	16.74516804
12	10000	20	6.372147405	8.977667977	2.129424387	2.800654158	20.8284848
13	25000	3	6.130232486	7.710328448	1.025689732	1.793249888	17.1862095
14	25000	5	6.325440704	7.635684525	1.39290629	2.488514595	18.37148199
15	25000	10	6.432112097	9.183667696	2.66048417	3.547411524	22.35096363
16	25000	20	6.999253163	12.41625997	8.219883461	5.865980466	34.0298038
17	50000	3	6.86331442	8.550502055	1.723721065	2.748166647	20.41979706
18	50000	5	6.760818541	10.15590598	2.89640364	3.937177648	24.27544201
19	50000	10	6.949898711	12.61525792	8.382417544	6.113214617	34.63520411
20	50000	20	7.578448237	20.02720328	15.00357516	10.91797262	54.12005117
21	100000	3	6.602777692	10.729836	3.66801406	6.23158387	27.76859073
22	100000	5	6.625922496	12.47996972	8.46375701	7.259109783	35.38506878
23	100000	10	7.682328134	20.43134294	14.57380097	12.07744774	55.34502637
24	100000	20	8.834290883	38.69092866	28.57348968	23.71369457	100.3613654