Bernhard Jaecksch, Franz Faerber, Wolfgang Lehner

Cherry Picking in Database Languages

Erstveröffentlichung in / First published in:

IDEAS '10: Fourteenth International Database Engineering & Applications, Montreal 16.-18.08.2010. ACM Digital Library, S. 117–122. ISBN 978-1-60558-900-8.

DOI: https://doi.org/10.1145/1866480.1866498

Cherry Picking in Database Languages

Bernhard Jaecksch
SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
b.jaecksch@sap.com

Franz Faerber
SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
franz.faerber@sap.com

Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
01062 Dresden
wolfgang.lehner@tu-dresden.de

ABSTRACT

To avoid expensive round-trips between the application layer and the database layer it is crucial that data-intensive processing and calculations happen close to where the data resides – ideally within the database engine. However, each application has its own domain and provides domain-specific languages (DSL) as a user interface to keep interactions confined within the well-known metaphors of the respective domain. Revealing the innards of the underlying data layer by forcing users to formulate problems in terms of a general database language is often not an option. To bridge that gap, we propose an approach to transform and directly compile a DSL into a general database execution plan using graph transformations. We identify the commonalities and mismatches between different models and show which parts can be cherry-picked for direct translation. Finally, we argue that graph transformations can be used in general to translate a DSL into an executable plan for a database.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Database (persistent) programming languages*

1. INTRODUCTION

Today it is often not enough to transform and process precalculated or aggregated data sets that are copied in and out of the application layer. Typically, a user wants to formulate the analysis and queries on the most granular level of data. Each application area provides a specific view to the underlying data and has its own domain-specific metaphors that agree with the user's perspective of the problem space. In the same manner, domain-specific languages (DSL) are often part of such applications allowing a user to express custom logic with an easy-to-learn and very confined language that is tailored to the specific problem domain. The application side provides the DSL interface to the user and translates between domain-specific metaphors and a general language that is used on the database side. As a result, the following steps are necessary: fetching data from the database into the application, processing it and pushing changes back into the database. Obviously, this round-trip becomes more and more expensive if the amount of transferred data increases.

Ideally, most of the data-intensive parts of the user calculations are executed within the database engine and only the results are shipped to the application layer. Then, however, the database engine has to be capable to interpret custom code written in a specific DSL. Obviously, it is not feasible that a database "understands" all these dialects and subsets of languages natively. However, general models or languages that can be natively executed by the database are a means to address this problem. Therefore, a translation between the DSL and the general model glues both together – see Figure 1. This can be either done within the application – as it is most often today – or in the database, which is what we propose. Unfortunately, the general database model is often declarative and data-flow-driven, while many DSLs are procedural and control-flow-driven. Our goal is to use graph transformations to translate as much as possible from the DSL directly to the declarative database model and keep the set of constructs that are expressed using procedural fragments as small as possible. We want to cherry-pick as many statements of a procedural script as possible and translate them into an equivalent declarative plan. This is schematically represented in Figure 2, where the goal is to push the bar between declarative and procedural translation as high as possible. Throughout the rest of the paper, we explain our graph transformation approach with the help

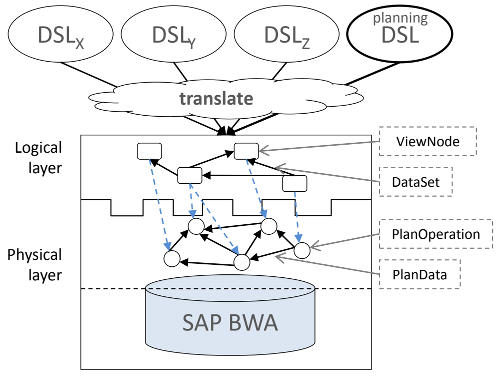


Figure 1: Translating different DSLs into a semantic plan

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 117–122, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866498

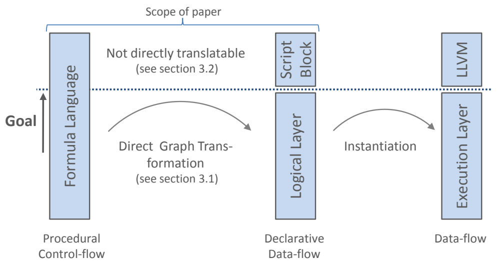


Figure 2: Translate as many parts of a DSL as possible into a semantic plan

of an example. The example is based on a formula scripting language for business planning, and the target model is the so-called semantic layer of the SAP Business Warehouse Accelerator (BWA) [2]. In section 2, we describe the challenges that are inherent to the translation approach. Then we introduce a set of graph transformation rules to map a procedural script to a declarative graph. In section 4, an example for the DSL is introduced that is the source of the transformation. In section 5, the target model is explained. Section 6 gives a short evaluation of the advantages of such a translation, and section 7 provides a conclusion.

2. CHALLENGES

After we have outlined the goal of the translation process, we want to enumerate the challenges that are posed to the translation process. One major difference between the declarative model and the procedural model is that the target model supports only row-wise expressions. Declarative operators do not carry any external state when operating on each row within their input sets. Thus, when calculating an expression, it is necessary to have every value used within the expression in one single tuple. Obviously, this is not always possible when the operators process sets of tuples. An example for such a constellation is, if the result of an expression at iteration $i + 1$ depends on the result of previous iterations. In that case, no translation into a declarative equivalent is possible, since there is no explicit loop operator in the declarative model that can carry the state between the processing of two tuples in its input set. To incorporate such parts of the procedural script into the declarative model anyway, we propose the use of a custom plan operator that can execute procedural code fragments. The fragments contain parts of the procedural model written, for example, in an interpretable language. However, if the loop is free of such side-effects, the declarative model provides implicit loop unrolling mechanisms as every operation performs a loop over its input data set(s) during operator execution and there are many cases where a direct translation into a declarative plan is possible. Throughout the rest of the paper, we will show how far we can get and provide graphical transformation rules that describe these mappings from a procedural DSL to a declarative execution graph.

As an example, used and refined throughout the paper, serves a *rolling plan* from the context of business planning. Such a *rolling plan* is created multiple times a year. Essentially, sales values for a company have been planned at the beginning of the year for the next 12 months. As the year proceeds and actual sales results arrive, the planned values have to be refined for the remaining months, based on the new knowledge. If the results so far are below the plan, the target for the remaining months is raised to keep track with the overall goal of the year. If the results are better than planned, the target is lowered. Figure 3 shows a value example.

3. LANGUAGE TRANSLATION VIA GRAPH TRANSFORMATION

A recent work by Göres et al. [3] introduced a very flexible Graph Transformation Language (GTL) for schema integration, which is based on feature-rich graph transformation rules. We exploit the same basic mechanism here. Ultimately, the source graph should be translated into the target graph, preserving the semantics of the source but capturing procedural logic in a declarative form. Graph transformation rules are a powerful means to express such a translation. To illustrate our overall idea, we want to introduce a set of transformation rules that translate statements of a procedural data-driven DSL into a declarative data flow graph (described in section 5). We then refine the general idea by applying these rules to a script that captures the business logic of the rolling plan example.

But what qualifies a graph transformation approach to tackle the problem of compiling a procedural DSL into a declarative plan? Graph transformations are very flexible and within a single transformation rule, multiple aspects of a transformation step can be captured. Furthermore, they are very generic, capture complex transformations in a visual and understandable way and are suitable to describe very different structures in a common framework. Another strong point is that it allows for easy extension with different variants and, assigned with costs, can be used in a cost-based selection strategy. For these reasons, graph transformations seem very suitable for our problem, and indeed, they are widely accepted as the means of choice in compiler design [1], model transformations and the like. We use the following basic graphical notation: nodes are labeled with a name and a type in the form *name:type*, and edges express the data-flow, control-flow or membership with some other node. Edges can also be labeled and nodes can have arbitrary properties that are prefixed by an @. Nodes might occur inside another node such that the parent node contains a subgraph of the complete graph, for example, to encapsulate statement blocks.

Graph transformation rules consist of a left-hand-side pattern, which matches a set of nodes in the host graph, and a right-hand-side pattern, which determines how the graph is transformed when the pattern has matched. Transformations include node deletion and node creation as well as creation, deletion or changes of node properties. Also, edges will be deleted automatically if both the start and end node of the edge are deleted. Edges are preserved if one or both of the start and end nodes change their type or are replaced by another node. Additionally, the following semantics are part of our rule definitions: nodes with double-line borders match at least one node of the specified type. Dashed borders symbolize that this node is optional in the pattern and will be matched in the host graph if existing. Both multiplicity notations can be combined, that is, a node can be matched zero or multiple times. The same notations are allowed on the right hand side and correspond to zero, one or multiple nodes, depending on the specific pattern instance

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Version 0	*1486*	*1326*	*1745*	*1732*	-	-	-	-	-	-	-	-
Version 1 *(before)*	*1486*	*1326*	*1745*	*1790*	*1900*	*2000*	*1950*	*2050*	*1600*	*1700*	*1900*	*2100*
Version 1 *(after)*	*1486*	*1326*	*1745*	*1732*	*1907.25*	*2007.63*	*1957.44*	*2057.82*	*1606.11*	*1706.49*	*1907.25*	*2108.01*

Figure 3: Example table of *rolling plan* for actual period May

that has been matched in the host graph.

3.1 Direct transformations

Direct transformation yields a declarative plan that is semantically equivalent to the source and uses no executable code blocks, that is, only native operators of the declarative plan layer. As will be shown in our evaluation, a direct translation is desirable since a declarative plan can be executed much more efficiently on large data sets than their procedural counterparts. A number of patterns can be identified that allow for direct translation, some of which we will describe in the following section.

The starting point of each translation process is to set up the base data set that contains all the tuples that are going to be used throughout the formula. This base set is the first node in the resulting graph, and all subsequent nodes are based on it. For all rules, we assume the availability of the base set node within the declarative graph. Furthermore, rules can modify the base set, as is the case when new values are assigned or variables are added. The assignment is a basic non-control-flow element of a procedural DSL. The left-hand-side of an assignment can either be a variable or an element of a tuple. Within our rolling plan example, from the time of assignment, the variable *actper* is available in all subsequent statements of the script so it is added as an attribute to the base set. The rule to assign a value to a variable and to add it to the context is given in figure 4(a). Note that the translation of the expression is done by another rule shown later. In our example, we need to calculate the sum of revenue for all periods after the actual period. This would involve three types of statements: a data-driven FOREACH loop, an IF condition inside and the assignment of an expression to a variable value. In declarative terms, this is an aggregation of the revenue over all fiscal periods larger than the actual period. This special combination of statements reflects a pattern that is recognized with the rules in figure 4(b). Using an arithmetic operation in the expression other than an addition would still lead to an aggregation but with a different aggregation function. In general, this situation results in an aggregation with a custom aggregation function. Any other FOREACH loop that does not exhibit the aggregation pattern is mapped to a node that selects all possible value combinations of the looping variables from the current base set; we call the translated node the *loop driver*. The IF statement takes the Boolean expression and adds it as an attribute to a new base set that evaluates either to true or false, depending on the condition. For each branch of the IF statement, a node is added to filter on the very same attribute. All statements within the branches then refer to these filtered views as their base set. For the aggregation in our example, this means that the referenced base set is the base view created by the TRUE branch of the IF statement, and only tuples where the expression evaluates to true are aggregated. The rule to translate an IF statement within the context of a FOREACH statement is shown in Figure 4(c).

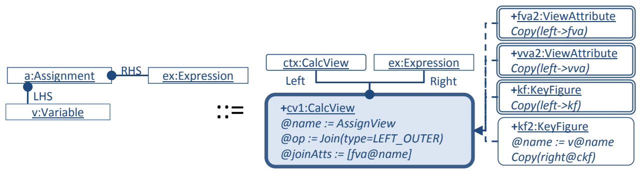
Two of the previous rules – variable assignment and the IF-statement – contain expressions. An expression consists of tuples and variables. Variables are attributes of the base set, and tuples map to a selection of tuples from the base set. For all constant values in a tuple, a filter predicate is added. To evaluate an expression, all contained tuples need to be joined together. The actual expression is calculated and the result of the join and can then be used during the translation of other rules. The translation of an expression is shown in Figure 4(d). Rule 4(e) shows how the assignment of an expression result to a tuple is translated. In this case the assignment is part of a FOREACH loop. The operand is joined with the result of the expression, and the attribute holding the expression result is mapped to the name of the left-hand-side operand key-figure. One step that has not been shown in the previous assignment rule is to merge the result of the assignment with the base set. The merge is necessary for the following reason: the assignment view contains a set of all tuples that have been assigned a new value for the respective key-figure; however, there can be tuples in the current base set that have not been assigned a new value. Then, the tuples must be merged, overwriting all values that have changed during the assignment. Using rules similar to the rules shown in this section large portions of a procedural script can be translated into a declarative plan. However, as mentioned earlier, a complete translation is not always possible. The next section will briefly describe how we will deal with these cases to fit into the declarative graph model.

3.2 Transformation into executables

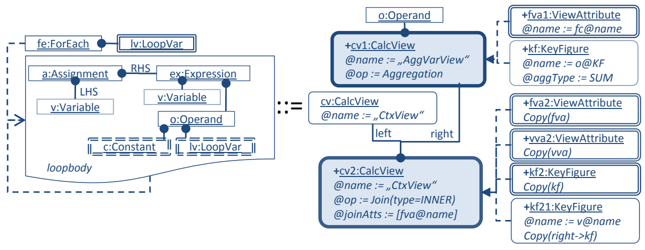
To translate parts of a script that cannot be transformed directly into a declarative plan, the idea is to keep them in procedural form and embed them into a node that can execute a code block. Within the semantic plan, these operators are black boxes acting like any other data-set operator within the plan. Procedural elements like variables and control-flow statements are translated in straight-forward fashion into the procedural language of the code operator. However, the data-driven FOREACH loop deserves a special notion: there are different possibilities to translate it, e.g. to have the set of looping values as another input to the view node and loop over this input set within the script block. The other possibility would be to construct the set of distinct value combinations within the executable block itself.

4. A DSL FOR BUSINESS PLANNING

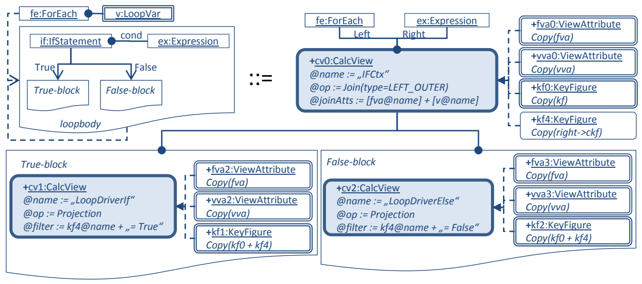
As an example of a domain-specific language that can be translated with our transformation rules, we introduce a formula language (FL) that is used in the domain of business planning. In business planning, either historical data in a warehouse is transformed and modified or new data is entered into the warehouse to create plan data for future periods. The plan data is used as a means of measuring actual



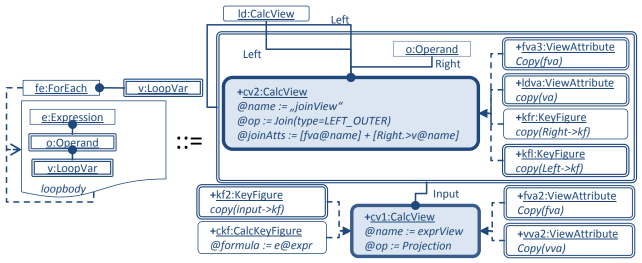
(a) Rule 1: transform a variable assignment



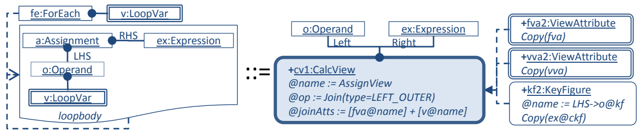
(b) Rule 2: special case of sum aggregation



(c) Rule 3: translate an IF statement within a FOREACH statement



(d) Rule 4: translate an "'expression"' within a FOREACH statement



(e) Rule 5: translate an "'operand assignment"' within a FOREACH statement

performance against the defined targets. Aside from a set of frequently used functions, like copying data or distributing values from aggregated levels to more fine-granular levels, planning functions contain custom logic that is specifically tailored to the individual business needs.

4.1 Language Example

FL provides a small set of language constructs to express business logic on multidimensional data. FL keeps the metaphor of the multidimensional cube and provides the user with simple statements to access the values of individual cube cells. It abstracts away from how the cube

Listing 1: Planning script example: rolling plan

```
1  DATA actper TYPE fiscper;
2  DATA fiscper TYPE fiscper;
3  DATA sum TYPE FLOAT;
4  DATA delta TYPE FLOAT;
5  // Read value from application variable period
6  actper = varv( 'PERIOD' );
7  // Calculate the sum
8  FOREACH fiscper:
9    IF fiscper > actper:
10     sum = sum + { revenue, 1, fiscper };
11   ENDIF;
12 ENDFOR;
13 // Delta between planned and actual
14 delta = {revenue, 1, actper}
15       − {revenue, 0, actper};
16 // Set planned to actual
17 {revenue, 1, actper} = {revenue, 0, actper};
18 // Distribute delta by weight
19 FOREACH fiscper:
20   IF fiscper > actper:
21     {revenue, 1, fiscper} =
22               {revenue, 1, fiscper} + delta ∗
23               {revenue, 1, fiscper} / sum;
24   ENDIF;
25 ENDFOR;
```

is stored and concentrates on the key parts that are necessary to express business logic. Listing 1 describes the *rolling plan* logic from our example as an FL script: the variable *actper* holds the value for the current month (line 6); by setting the value for this variable, the script can be reused each month. In the first block (lines 1 - 4) the variables are declared. Variables can have simple types like *INTEGER*, *FLOAT* or *STRING*, or they reference the type of a dimension in the underlying cube. The sum of revenue in version 1 from the beginning of the year up to the current period is calculated in lines 8 - 12 in Listing 1. The version dimension is used to distinguish planned values from actual values. Version 1 contains the planned and version 0 contains the actual values. A *FOREACH* loop is used to iterate over all distinct values of the *fiscper* dimension. Furthermore, the IF statement is used to check whether the *fiscper* value of the current iteration is after the current period in *actper*. The value *sum* is then accumulated from the revenue values of the tuples that satisfy this condition. This shows the use of a tuple (enclosed in curly brackets) within an expression. Next, the *delta* between the current sum of revenue in version 0 and the planned sum of revenue in version 1 up to the actual month is calculated (line 14). In the final *FOREACH* loop, the value of *delta* is distributed to the revenue values in version 1 for the remaining months, such that the planned overall amount of revenue remains constant. The result is assigned to a tuple that again references a specific cell in the underlying cube for each loop iteration.

4.2 Graphical representation of the FL

It follows a graphical notation for the FL script language. Each statement is represented by a rectangle node. Figure 4 shows the complete graphical representation of the rolling-

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 117–122, ISBN 978-1-60558-900-8
https://doi.org/10.1145/1866480.1866498

plan script from our example. Each node has as type the statement it represents and a unique name. For example, the part numbered with ① in Figure 4 shows a node of type assignment named *a1*. The assignment node *a1* has one tuple node as child representing the left-hand side and an expression node *ex1* in Figure 4 representing the right-hand side of the assignment. The child nodes are connected to the parent node via containment edges, meaning that they belong to the parent node. The detail marked with ① in Figure 4 corresponds to line 6 in Listing 1. Alternatively, if the child nodes deliver a value to their parent, they are connected with a data-flow edge, which also captures containment (see *ex2* at annotation ② in Figure 4), and data flows to the end with the circle. Edges can be labeled with a role, for example LHS and RHS for left- and right-hand side, respectively, for the assignment at ①. Keeping the example of the assignment node, there is also a data-flow edge between the expression node and the operand node. This expresses that the result value of the expression flows into the node and replaces the key figure value of the operand. Consecutive statements in the script, represented by different nodes in the graph, are linked via control-flow edges representing the order of execution, as in ③ in Figure 4. A list of statements can be encapsulated in a statement block, e.g, if there are a number of statements within the True branch of a conditional node or within the body of a loop, as marked with ④ in the FL graph.

5. THE SEMANTIC PLAN

The goal is to translate an FL script into something that the underlying database engine can execute. In our case, this will be a logical execution plan for the SAP BWA execution engine. While the nature of the FL script is procedural, the logical plan is a data-flow graph with declarative semantics. The SAP BWA execution engine provides a distributed and parallel execution environment with excellent scaling behavior. Furthermore, it provides a logical execution model, which offers most of the basic operations of relational algebra; it also provides rich expression semantics and the fast aggregation capabilities of the SAP BWA. The goal of the logical plan is to allow much more flexible queries than standard OLAP queries and to formulate complex data processing logic using a declarative data-flow graph. For example, in [4], it has been discussed how the logical plan can be used to express a set of standard planning functions.

5.1 A data-flow graph execution model

Within the SAP BWA, the logical plan is represented as a directed acyclic graph (DAG) and consists of so-called view nodes (see Figure 1). Edges represent data flow between nodes. Each node describes how many inputs it expects and how the inputs are transformed to produce the output. Inputs can be the result of the output of another node or tables of the underlying database. Each view node has an operation that transforms the inputs. Available operations are equal to relational algebra: projection, aggregation/-grouping, join and union. Sorting and selection are further operations, but they are part of every view in addition to the main operation. Selection and sort are always applied after the main operation has been done. Logical plans are stored as views and at the time of query, the plan is instantiated

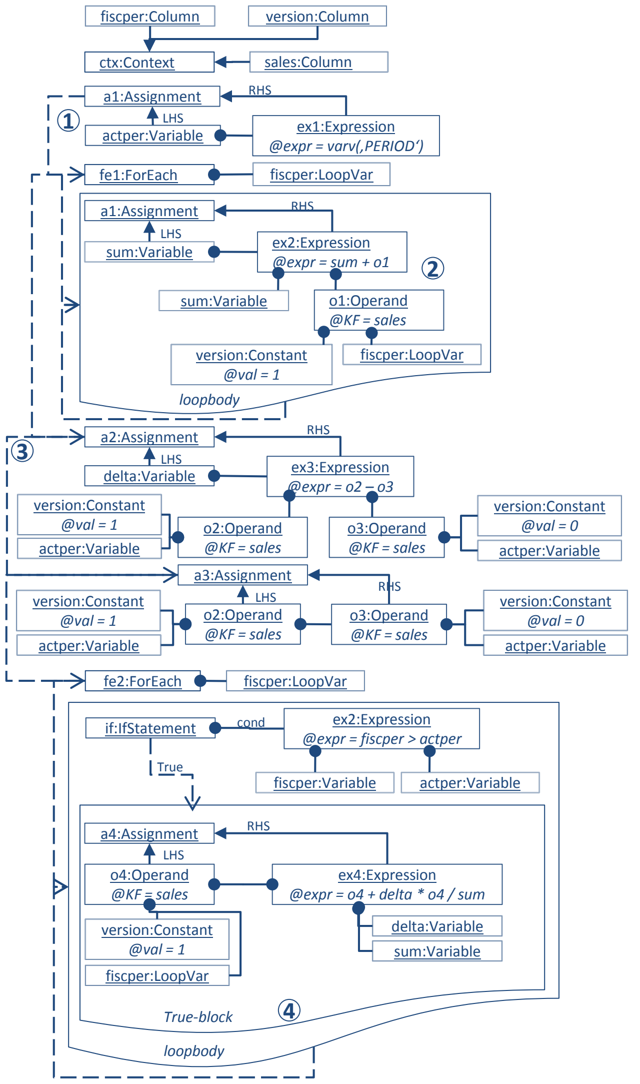


Figure 4: Graphical representation of rolling plan FL script

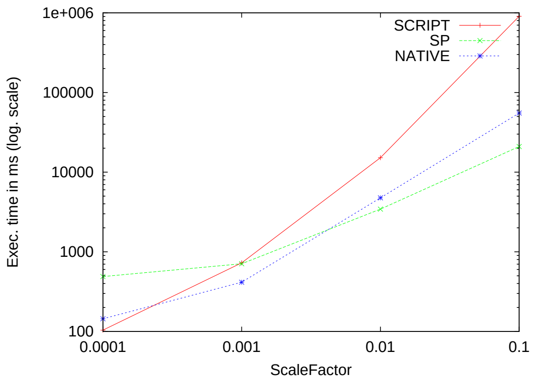
and translated into a physical execution plan.

5.2 Graphical representation of the semantic plan

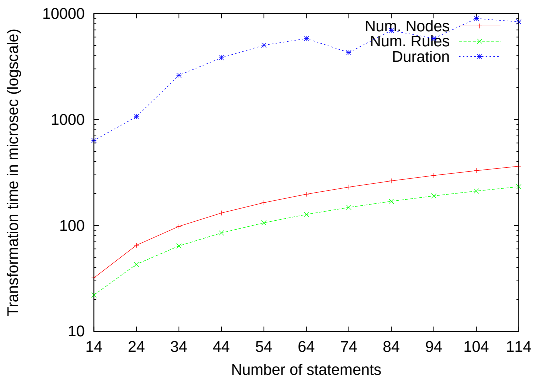
The representation of the logical plan shares many commonalities with the representation of the FL. Nodes of the semantic plan are represented by rectangles with rounded corners. They have a unique name and are of type view. Attributes are also represented by rectangle nodes with rounded corners and their respective type: view attribute or keyfigure. They also have properties that can be addressed with *@<property-name>*. Since the logical plan is a pure dataflow graph, there are only data-flow edges and edges that represent containment. Aside from view nodes and their children being the main entity of a plan graph, there are data source nodes that reference the actual source data.

6. EVALUATION

The logical plan of the compiled FL script of our running



(a) Comparison of declarative plan vs. procedural plan



(b) Translation time and number of rules/nodes for different number of statements

Figure 5: Evaluation results

example is used for our following evaluation. As discussed in the previous section about transformation rules and also shown in Figure 2 from the introduction, there exist two extremes. One is to translate the DSL completely into a declarative plan. The contrary is a plan with one code block node that contains the complete procedural script. We argue that the execution within the SAP BWA engine benefits from a declarative plan that consists of typical set operations like selection, union, join and aggregation. The following results underpin this claim and show the benefits that can be gained when executing data-intensive tasks written in a user-friendly DSL directly within the database. We implemented a prototype that translates an AST representation of an FL script into a declarative plan using the set of transformation rules explained throughout the paper. The execution of the logical plan (SP) of our rolling plan example is compared to a variant that only contains one code block node, which wraps a procedural translation of the FL script (SCRIPT). Furthermore, we compared another version, but this time, the script is translated into native C++ code. The dataset is the SSB OLAP benchmark data [5], which is a customized version of the TPC-H benchmark [6] specifically tailored to OLAP scenarios. The size of the generated data contains a fact table with 1,200,000 line items for the largest scale factor (SF) 0.1 we used. This is twice the size of the original SSB dataset with scale factor 0.1, because we duplicated the entries of the fact table with slightly changed measure values to provide for the plan and the actual version of the data as required in our example. The rolling plan is calculated for every combination of customers and parts within the SSB benchmark cube with the current period set to May. The experiments have been executed on a 2-CPU Intel® Core® i7 machine with 12GB RAM running Windows® 7, 64 Bit. As can be seen in Figure 5(a), for very small datasets – SF 0.0001 – the execution of the plans with the procedural nodes is faster than the semantic plan; however, with growing size of the data set, the fully declarative semantic plan performs orders of magnitudes better than the procedural versions. The second experiment in Figure 5(b) shows linear scaling of the number of rule applications and created nodes with respect to the number of statements. We measured the rolling plan example again and multiplied the FL script code. The result is as expected, since the left-hand sides of our rule patterns are disjoint and hence, for every statement, only one rule applies. Furthermore, after the application of a rule, no other rule matches the resulting subgraph, thus the translation process is deterministic.

7. CONCLUSION AND OUTLOOK

There are several parts of a procedural DSL that are good candidates for translation into a declarative, data-driven execution model of a database engine. To drive this translation, we propose graph transformations: models are expressed using a graphical representation and graph transformation rules are powerful enough to express an automatic translation from one representation into the other. Although there is a mismatch between the procedural DSL and the declarative execution plan, this gap can be bridged with code blocks that are embedded into the plan. Furthermore, we showed in our evaluation that the declarative translation of a script can leverage the underlying database execution engine and performs much better on large datasets than the procedural variant. Further work includes assembling new rules that capture the translation into the executable code blocks mentioned before. Also, the graph transformation technique can be used to express optimizations of the logical plan, which can be applied iteratively until a minimal number of view nodes is reached or as much as possible is directly translated.

8. REFERENCES

[1] U. Assmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, 2000.

[2] R. Burns and R. Dorin. The SAP NetWeaver BI Accelerator - Transforming Business Intelligence, White Paper, Winter Corporation, 2006, http://www.wintercorp.com/whitepapers/whitepapers.asp.

[3] J. Göres. *A Model Management Framework for Information Integration*. Phd, Technische Universität Kaiserslautern, 2009.

[4] B. Jaecksch, W. Lehner, and F. Faerber. A plan for OLAP. In *EDBT*, pages 681–686, 2010.

[5] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark (ssb). *Pat*, (January):1–10, 2007.

[6] Transaction Processing Performance Council. TPC-H Benchmark Specification.