An Approach to Optimize Data Processing in Business Processes

Marko Vrhovnik[1], Holger Schwarz[1], Oliver Suhre[2], Bernhard Mitschang[1], Volker Markl[3]
Albert Maier[2], Tobias Kraft[1]

[1]University of Stuttgart
Universitätsstrasse 38
70569 Stuttgart
Germany
firstname.lastname@
ipvs.uni-stuttgart.de

[2]IBM Deutschland
Entwicklung GmbH
Schönaicher Str. 220
71032 Böblingen
Germany
{suhre,amaier}@de.ibm.com

[3]IBM Almaden
Research Center
650 Harry Road
San Jose, CA 94120-6099
USA
markl@us.ibm.com

ABSTRACT

In order to optimize their revenues and profits, an increasing number of businesses organize their business activities in terms of business processes. Typically, they automate important business tasks by orchestrating a number of applications and data stores. Obviously, the performance of a business process is directly dependent on the efficiency of data access, data processing, and data management.

In this paper, we propose a framework for the optimization of data processing in business processes. We introduce a set of rewrite rules that transform a business process in such a way that an improved execution with respect to data management can be achieved without changing the semantics of the original process. These rewrite rules are based on a semi-procedural process graph model that externalizes data dependencies as well as control flow dependencies of a business process. Furthermore, we present a multi-stage control strategy for the optimization process. We illustrate the benefits and opportunities of our approach through a prototype implementation. Our experimental results demonstrate that independent of the underlying database system performance gains of orders of magnitude are achievable by reasoning about data and control in a unified framework.

1. INTRODUCTION

Enterprises use a multitude of heterogeneous applications and data stores. In a typical enterprise, the customer relationship management of a Siebel system will interact with a SAP sales and distribution system, as well as with business data warehouses and production planning systems from possibly different vendors. The grand challenge of information management nowadays is the integration and concerted operation of these different systems.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

Business process execution languages (also called workflow languages) like BPEL [14] together with service-oriented architectures and Web services integrate complex systems, and create a single execution framework for all the activities of an enterprise. Information integration systems unify the heterogeneous persistent storage systems (databases, file systems, content management systems) used by these applications.

As discussed in [4], the integration of databases and programming languages is one of the major challenges for research in the coming decade. We address this problem in the context of information integration and business wide workflows, by proposing a framework that makes data processing operations, e.g., SQL statements, first class citizens in a workflow language, and consequently in its optimization and execution. In analogy and as extension to QGM-like approaches [16], we define a Process Graph Model (PGM) to reason about workflows and their data flow and control flow issues. Furthermore, we show opportunities that arise for the rule-based optimization of data processing in business processes.

The main contributions of this paper are:

- We show how to exploit knowledge on the data flow as well as on the control flow of a business process for reasoning upon optimization decisions.
- We extend query optimization to the level of complex business tasks expressed as business processes.
- We develop a multi-step control strategy for comprehensive treatment of the optimization task.
- We demonstrate the opportunities, benefits and generality of our approach by means of a case study.

The remainder of this paper is organized as follows: After detailing on workflow languages in Section 2, we discuss related work in Section 3. In Section 4, we introduce our approach for rule-based optimization of business processes. We comment on the rewrite rules and the control strategy used in this approach in Section 5 and Section 6, respectively. In Section 7, we present our experimental results. Section 8 concludes and lists future work.

2. WORKFLOW LANGUAGES AND DATA MANAGEMENT

We focus on the business process execution language BPEL [14] that is broadly adopted by industry and, hence, can be seen as the de facto standard. BPEL exemplifies capabilities of typical workflow languages to express business processes[1]. Similar to other process specification approaches, BPEL fosters a two-level programming model. The *function layer* consists of executable software components in form of Web services that carry out the basic activities. The *choreography layer* specifies a process model defining the execution order of activities. By means of the Web service approach, BPEL achieves independence from the implementation level of activities and the execution platform.

BPEL offers various language constructs to express activities as well as control flow patterns. In the following, we describe the usage patterns of some activities that are relevant for the further discussion: (i) *invoke activities* call Web services during process execution, (ii) *assign activities* create the appropriate input for activities based on results from earlier service invocations, (iii) *sequence activities* sequentially execute activities, (iv) *ForEach activities* allow to iterate over datasets, and (v) *flow activities* concurrently execute activities.

In most workflow languages, users define process models in a graph-like fashion, supported by corresponding graphical design tools. From a modeling point of view, process design tools favor the description of control flow over data flow. This is also true for BPEL, where control flow is explicitly modeled, whereas data flow is only implicitly defined by specifying BPEL variables used as input and output data of activities. These variables are also used at the choreography layer, e.g., as part of conditions in the BPEL loop constructs.

The BPEL specification [14] comprises additional activity types. Furthermore, it supports elaborated compensation and error handling concepts. So far, the language specification is not finished. Proposed extensions comprise the support for human interaction, sub processes and the ability to embed Java code into the business process.

2.1 Data Management at the Choreography Level

In most business processes, accessing and processing data in enterprise data stores is a central task. Hence, the following primitive data usage patterns have to be supported for a comprehensive data provisioning: (i) Use business data to decide on the control flow. For example, one could use an inventory service to decide whether to call a supplier or not. (ii) Update business data. For example, the delivery time for an item may change based on information received from the supplier of the item. (iii) Retrieve business data and send it to partners, i.e., clients, services and humans. (iv) Store business data that was changed by partners.

The service-oriented approach allows to hide enterprise data stores behind Web services. This is achieved by so-called adapters that encapsulate SQL-specific functionality in Web services. They provide a well-documented, interoperable method for data access. Adapters represent proven and well-established technology and are provided by different vendors in a similar way. Nevertheless, database vendors pursue various additional approaches to support data management at the choreography level:

- The Oracle® BPEL Process Manager provides XPath extension functions that are embedded into assign activities [15]. The statement to be executed on a remote database is provided as parameter to the function. The functions support any valid SQL query, update operations, DDL statements as well as stored procedure calls. Data retrieved by an extension function is stored in a set-oriented process variable expressed in XML.
- The Microsoft Windows® Workflow Foundation [13] uses SQL activities to provide database processing as part of business processes. These activities are organized in so-called activity libraries and executed by a runtime engine. The workflow as well as its variables, activities, and parameters are for example described by XOML, an extensible object markup language.
- IBM's WebSphere® Process Server follows another approach to bring set-orientation to BPEL and its choreography layer [7, 8]. It introduces information service activities that for example allow to process data in a set-oriented manner expressed via SQL. Furthermore, it allows to pass data sets between activities by reference rather than by value. In this paper, we call such a direct integration approach BPEL/SQL.

The main common property of all three approaches is that data management tasks are explicitly modeled at the choreography layer. Hence, the basic concepts we present here, are applicable to all three approaches. In this paper, we focus on BPEL/SQL to explain these concepts. We will further discuss generality issues in Section 4 and Section 7.

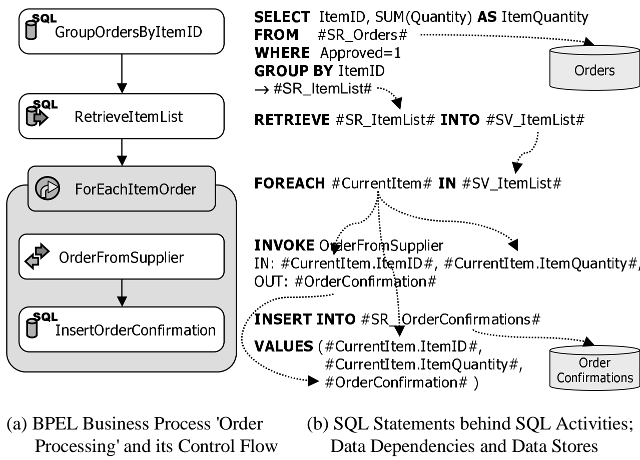
So-called *SQL activities* reflect the mentioned data usage patterns in BPEL/SQL. They provide read and write access to BPEL variables. Variables that refer to tables stored in a database system are called *set reference variables*. A lifecycle management facility is responsible for creating and removing these tables as needed. Variables of a set-oriented data structure representing a table that is materialized in the process space are called *set variables*. Furthermore, SQL activities may embed an SQL statement including several input parameters as host variables. An input parameter may be a scalar value or a set reference variable, i.e., a table reference. Input parameters are valid at any position in the SQL statement where in case of a scalar value a host variable or in case of a set reference variable a table name is allowed. Set reference variables as well as set variables may store the result set of SQL activities. A *retrieve set activity* is a specific SQL activity that allows to load data from a database system into the process space.

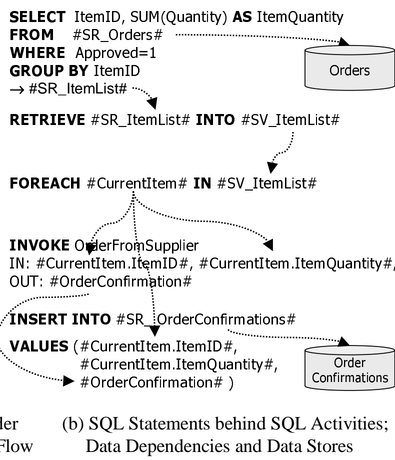
2.2 Sample Scenario

Our example is based on an order handling scenario. The process takes a set of orders and decides whether to process them directly or not. In the latter case, an employee has to approve an order according to some business policy. After this approval, the process sends notifications for all rejected orders. In parallel, it processes the remaining orders. This

[1] In compliance to the BPEL specification, we use the term *business process* to denote processes specified by BPEL.

® Oracle BPM is a trademark of Oracle Corporation. Windows Server 2003 is a trademark of Microsoft Corporation. WebSphere and DB2 are trademarks of IBM Corporation.





(a) BPEL Business Process 'Order Processing' and its Control Flow

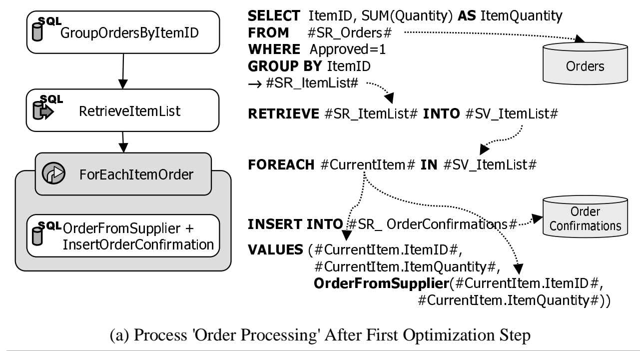
(b) SQL Statements behind SQL Activities; Data Dependencies and Data Stores

Figure 1: Sample Process

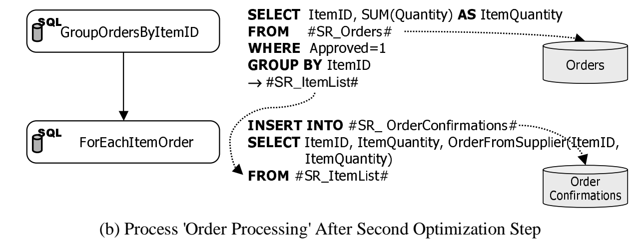
part of the overall scenario, i.e., the automatic processing of approved orders, is depicted in Figure 1(a). Modeling this process in BPEL/SQL is quite straightforward. Activity *GroupOrdersByItemIDs* creates a set of orders each containing an itemID and the required quantity. *ForEachItemOrder* now iterates over these orders. A Web service call (*OrderFromSupplier*) submits the orders to suppliers, which return confirmation information whether they can supply the requested items or not. Activity *InsertOrderConfirmation* makes this confirmation persistent.

We show SQL statements and additional information for the sample process in Figure 1(b). For clarity we mark BPEL variables by surrounding hash marks (#). Set reference variable *SR_Orders* refers to table *Orders* containing all orders. Activity *GroupOrdersByItemID* contains a SELECT statement that extracts items from approved orders and writes them to set reference variable *SR_ItemList*. *RetrieveItemList* reads these orders into set variable *SV_ItemList*. The succeeding *ForEachItemOrder* activity iterates over *SV_ItemList* and binds a tuple of this set to the variable *CurrentItem* in each iteration. The attributes *ItemID* and *ItemQuantity* of *CurrentItem* serve as input for the *OrderFromSupplier* and *InsertOrderConfirmation* activities. The latter specifies an INSERT statement, which writes the result of the Web service call into an *OrderConfirmations* table that is referenced by a set reference variable *SR_OrderConfirmations*.

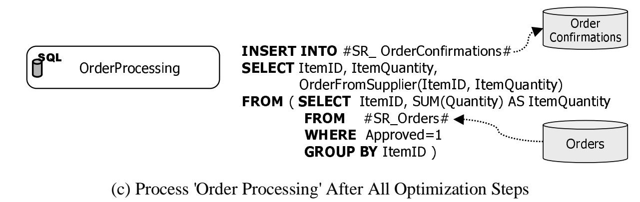
This sample business process contains potential for optimization. Figure 2 shows the processes resulting from three subsequent optimization steps. In a first step, it is possible to merge the Web service invocation *OrderFromSupplier* into the SQL statement of activity *InsertOrderConfirmation*. This is achieved by the user-defined function *OrderFromSupplier* (Figure 2(a)). Based on the resulting process, we are further able to rewrite the tuple-oriented cursor loop into a single set-oriented SQL activity. Hence, the entire processing of the *ForEachItemOrder* activity can be replaced by one SQL activity (Figure 2(b)). In a last step, we could merge this activity with the *GroupOrdersByItemID* activity (Figure 2(c)). In Section 4, we elaborate on the set of rewrite rules for such optimizations. Logically, the resulting process combines in a single SQL activity the same processing as the original process. The important dif-



(a) Process 'Order Processing' After First Optimization Step



(b) Process 'Order Processing' After Second Optimization Step



(c) Process 'Order Processing' After All Optimization Steps

Figure 2: Optimization of Sample Process

ference is that all set-oriented data management activities are performed by the underlying database system in a single SQL statement. In Section 7, we show that this results in a remarkable performance improvement independent from the underlying DBMS. The main reasons are:

- Compared to the original process, there are more optimization opportunities for the database system.
- The optimization leads to a reduction of data volume transferred between the database level and the workflow processing level.
- Overhead that is associated with the processing of each SQL statement is saved because only a single statement is left.

Although the optimization of the sample process seems to be straightforward, several conditions on control flow dependencies and data dependencies have to hold in order to preserve the workflow semantics. Here are some examples for slight modifications of the process shown in Figure 1 (a) and how they affect or inhibit the optimization process shown in Figure 2 (more details are given in Section 5.1):

- Assume a process where SQL activity *InsertOrderConfirmation* does not access the result of the Web service call. In this case, there is no data dependency between both activities in the ForEach loop. Hence, the Web service invocation cannot be merged into the SQL activity and no further optimizations of the ForEach loop are possible.

- Assume an additional SQL activity SQL_1 in between the activities *GroupOrderByItemID* and *ForEachItem-Order*. Further assume that SQL_1 does neither access table *Orders* nor table *OrderConfirmation*, i.e., there is no data dependency. The optimization would basically work as shown in Figure 2. The final workflow would contain activity SQL_1 followed by activity *OrderProcessing* shown in Figure 2 (c).
- Now, assume the same additional activity SQL_1 as before. This time it is placed before the *OrderFrom-Supplier* activity in the loop. Then, the Web service call of activity *OrderFromSupplier* can still be merged into SQL activity *InsertOrderConfirmation* as shown in Figure 2 (a). But further optimization steps are inhibited because two separate SQL activities remain in the ForEach loop. Hence, it is not possible to rewrite the cursor loop into a single set-oriented SQL activity.

Similar considerations apply if the ForEach loop would be replaced by other control flow constructs, e.g., the flow activity in BPEL that allows to define concurrent execution. For efficient reasoning upon such optimization decisions, we need an appropriate internal representation of processes that explicitly models control flow dependencies as well as data dependencies in a single graph structure (see Section 4).

3. RELATED WORK

Optimization of data management and data processing in business processes covers several optimization levels as well as optimization subjects. Obviously, there is process optimization at the level of the business model and there is query optimization at the data and database level.

With a business-oriented perspective, optimization on the process level is often termed business process re-engineering [12]. At this level, the business model and the involved processes are analyzed from a business point of view and the control flow is exploited for optimization. We do not address optimization on this level. We rather assume that process designers conduct process re-engineering activities before the rule-based transformation of business processes is accomplished.

At the data level, the database community has focused on optimizing data flow based on graph models, e.g., the query graph model (QGM) [16]. This comprises the construction of optimized query execution plans for individual queries as well as multi-query optimization. Multi-query optimization detects common inter- and intra-query subexpressions and avoids redundant computation [10, 3, 18, 19]. Our work builds on this paradigm. Moreover, our approach borrows well-known concepts for rule-based optimization, all grounded on graph-transformations [16, 17]. In contrast to QGM, our process graph model refers to control flow issues as well as to data dependencies.

Federated database systems integrate data sources and provide a homogeneous database schema for heterogeneous sources. In analogy to federated database systems [9], our framework contains an optimizer, complementing the query optimizers of the database management systems that are responsible for executing the data management activities of a process.

In contrast to pure data flow and control flow optimization, there are several approaches that exploit in addition to data flow knowledge some kind of control flow knowledge.

Coarse-grained optimization (CGO) [11] is an approach for optimizing sequences of data processing tasks. Similar to CGO, our optimization approach is based on rewrite rules. However, our work is more general in two aspects: First, we do not restrict the optimization to data processing activities, e.g., we also consider Web service calls. Secondly, we do not restrict the flow definition to sequences.

An interesting piece of work is shown in [22], where DBMS-like capabilities are expressed as Select-Project-Join queries over one or more Web services. The query concept introduced implicitly reflects a notion of set-orientation and it further enables optimization towards an optimal pipelined query processing over the involved Web services. The capabilities at the choreography layer are far from BPEL expressiveness and the capabilities for data processing are restricted to retrieval only. A similar approach is described in [21]. There, a language for modeling workflows is sketched that is tightly integrated with SQL. The concept of an active table/view is used to represent the workflow and to provide its data. It is a proprietary and restricted approach that does not reflect available standards like BPEL.

ACTIVE XML [1] is an approach where XML documents allow for embedded Web service calls. Optimization work in Active XML refers to deciding which of the embedded Web service calls need to be processed for efficiently answering a query posed over the XML document. This interesting work is related to [22], and similarly restricted with respect to control flow issues and data management operations.

The microprocessor, compiler and programming language community considers optimization of control flow [2]. Well-known optimization approaches cover keyhole optimization, instruction reordering, speculative execution, and branch prediction. They are usually studied in a context where basic operations are very simple, e.g., assignments. We apply these ideas to a workflow scenario where basic operations are as complex as SQL statements and Web services.

A first approach to unify data and control flow is presented in [20]. However, they only consider database-internal processing and address the optimization of loops and sequences containing database commands. We extend this work as we want to support the whole range of control flow constructs provided by workflow languages. Furthermore, we focus on an integrated execution framework that covers the database and the choreography level.

The seamless integration of operations on business data into the workflow languages opens the space for considering control flow as well as data flow issues together. New optimization potential could be exploited. Hence, a comprehensive approach that enables data-level optimization of business processes has to cover data flow as well as control flow issues. Therefore, we propose a graph model that explicitly exposes activities, control flow, and data flow. Furthermore, it is independent from the workflow language. Existing process-related models like Petri nets and Event-Driven Process Chains are based on graphs or automatons as well. In contrast to our graph model, they focus on control flow and are used to investigate properties of a process and its activities [6]. Our work is the first to combine several concepts from the programming language and database communities, and that presents an integrated approach.

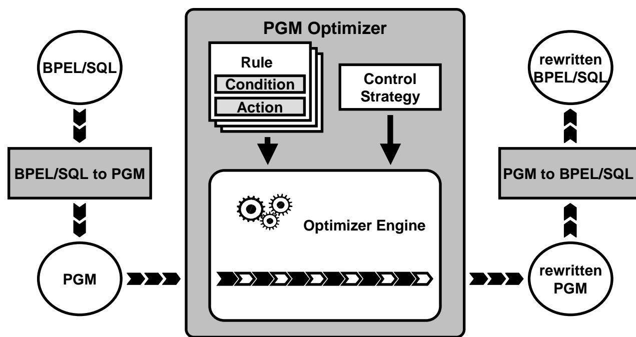


Figure 3: Processing Model

4. RULE-BASED OPTIMIZATION OF BUSINESS PROCESSES

The main idea behind our optimization approach is to apply rewrite rules to BPEL/SQL processes. While these rules keep the semantics of the original process, they change its structure as well as its constituent parts in such a way that the resulting process shows improved performance.

Figure 3 presents the processing model behind our optimization approach. The core component is an *optimizer engine* that operates on the internal representation of a process. This engine is configured by a set of *rewrite rules* and a *control strategy* for rule application. Each rewrite rule consists of two parts: a *condition* and an *action* part. The condition part defines what conditions have to hold for a rule application in order to preserve the process semantics. It refers to the control flow dependencies as well as to the data dependencies of a process. Additionally, it considers detailed information of process activities. The type of SQL statements is a good example, as the condition part of some rewrite rules states that they are applicable to certain combinations of INSERT and DELETE statements only. The action part of a rewrite rule defines the transformations applied to a process provided that the corresponding condition part is fulfilled. In Section 5, we further elaborate on an appropriate ruleset and give examples for the condition and action part of a rewrite rule.

So far, we described that rewrite rules come along with a set of conditions that allow to identify rules being applicable to a given process. In addition to that, the optimizer has to decide where on the process structure and in which order rewrite rules are applied. This is the main task of the *control strategy.* One of its functions is to identify so-called optimization spheres, i.e., parts of a process for which applicable rewrite rules should be identified. Determining such spheres is necessary, because if one applies rewrite rules across spheres, the semantics of a process may change. Another function of the control strategy is to define the order in which rule conditions are checked for applicability and the order in which rules are finally applied. In Section 6, we provide more details on these issues.

4.1 Process Graph Model

As already mentioned, the optimizer engine works on an internal representation of processes. We developed a *Process Graph Model* (PGM) for this purpose. Formally, PGM defines a process as a tuple (A, E_c, E_d, V, P) where A represents the set of process activities, E_c represents directed control flow edges, E_d is a set of directed data flow edges, V is a set of typed variables used in the process, and P covers partners, i.e., external systems the process interacts with. This model is similar to well-known query graph models like QGM [16]. QGM defines a query as a tuple (B, E_d, Q, E_p), where B (called boxes) is a set of operations, Q represents the set of quantified tuple variables, called quantifiers, E_d is a set of data flow edges between quantifiers and boxes, and E_p is a set of predicate edges connecting quantifiers. PGM extends the scope of such models by adding control flow edges and links to external systems. Like QGM, our process graph model allows to reason about an optimization approach, e.g., to show that a lossless mapping from and to the internal representation is possible and that the termination of the optimization process is guaranteed. In favor of a concise presentation we do not elaborate on these aspects here and present the set of rewrite rules on the BPEL/SQL level in Section 5.

PGM turns out to be the appropriate basis for rule-based transformations as the condition part as well as the action part of rewrite rules can directly be expressed as graph conditions and graph transformations, respectively. PGM supports this by explicitly modeling control flow dependencies as well as data dependencies in a single graph structure. In workflow languages like BPEL/SQL, only the control flow part is explicitly modeled whereas the data dependencies are implicitly covered by variables and variable references. Remember that the conditions of our rewrite rules refer to both, control flow and data dependencies. Hence, phrasing these conditions directly on BPEL/SQL would make them more complex and thus more error-prone.

4.2 Generality Issues

For the discussion of generality, we first want to emphasize two important preconditions for our optimization approach: (i) The optimizer engine needs to know the exact statements that are used in data management tasks. This is important because several rewrite rules transform these statements. (ii) The optimizer engine needs to know control flow dependencies as well as data dependencies in order to check rule conditions. Control flow dependencies are provided by the workflow language. Data dependencies have to be extracted from variables and variable references. Control flow dependencies as well as data dependencies are externalized by the PGM representation of a business process.

These preconditions are fulfilled by all approaches mentioned in Section 2.1 as they all explicitly define data management tasks at the choreography layer. Microsoft's Workflow Foundation and IBM's WebSphere Process Server provide specific SQL activities. The definition of such activities comprises the SQL statement to be executed. Oracle provides XPath extension functions that receive the SQL statements as parameters.

A key advantage of PGM is that it makes the optimization process depicted in Figure 3 independent from a specific workflow language. PGM comprises a set of basic activity types that are common to major workflow languages. Hence, for a particular language, only a mapping to PGM and vice versa has to be provided. This is achievable for the three approaches mentioned in Section 2.1. Furthermore, PGM can easily be extended which provides the flexibility to adjust it to future extensions of BPEL as well as to the constructs of other workflow languages. For BPEL extensions, a mod-

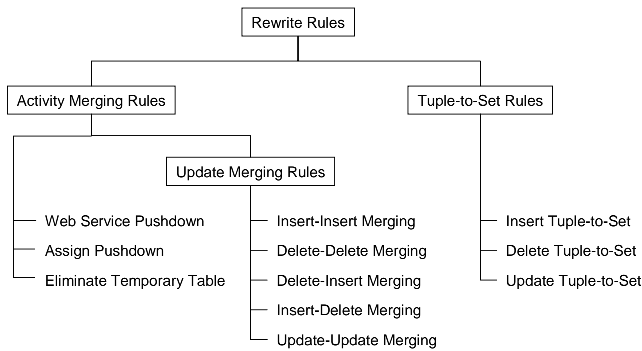


Figure 4: Classification of Rewrite Rules

ification of the mapping components *BPEL/SQL to PGM* and *PGM to BPEL/SQL* is necessary. For other workflow languages, individual mapping components have to be provided. If a certain workflow language allows for additional optimization rules, they can easily be added to the ruleset of the PGM Optimizer, and, based on the optimization sphere concept, be latched into the control strategy.

Our optimization approach supports different database languages as well as different underlying DBMSs. A language dialect is reflected in the condition and action part of a rule. Thus, new dialects can be incorporated simply by adding the corresponding rules and, if necessary, by adapting the control strategy. In Section 7, we show that the performance improvements are independent from the underlying DBMS.

In this paper, we assume that all data management activities of a process are issued against a single database system. However, in a practical setting, heterogeneity with respect to data sources is the normalcy. We can use existing federated database technology [5] that homogenizes heterogeneous data source to extend our optimization approach to this general case and to carry all findings and results over.

5. REWRITE RULES

In this section, we introduce rewrite rules for the rule-based optimization of processes. Figure 4 shows the set of rules we focus on in this paper. Rule classes are shown in rectangles whereas their instances are given as plain text. Based on the similarity of rules, we distinguish two major classes of rewrite rules: *Activity Merging Rules* and *Tuple-to-Set Rules*.

Remark that according to our processing model, we apply these rules to PGM processes. In favor of a concise presentation in this section, we introduce the rules based on the set of BPEL/SQL activities and omit details at PGM level.

The purpose of *Activity Merging Rules* is to resolve data dependencies by merging a source activity and a consecutive destination activity that depends on data delivered by the source activity. The destination activity is always an SQL activity while the source activity may be an assign activity, an invoke activity or an SQL activity. In the following, we explain some *Activity Merging Rules* in more detail.

The *Web Service Pushdown* rule pushes an invoke activity into the SQL activity that depends on the Web service invocation. As a result, the Web service becomes part of the SQL statement. This pushdown is only applicable if the underlying DBMS supports Web service calls, e.g., as user-defined function calls.

The *Assign Pushdown* rule directly integrates an assign activity into an SQL activity. It requires an assign activity writing a variable that serves as input for a consecutive SQL activity. We push the assign operation into the SQL statement replacing the considered variable through its definition. This allows to omit the assign activity.

The *Eliminate Temporary Table* rule removes the usage of temporary tables within SQL statements of SQL activities. BPEL/SQL allows to load a query result set into a table that is referenced by a result set reference [8]. If such a table is created for each single process instance at process start up time, and if it is dropped as soon as the process instance has finished, we call it a temporary table. Within an SQL activity, this rule replaces the usage of a result set reference that refers to a temporary table directly by its definition statement. This reduces the costs for the lifecycle management of temporary tables as well as for SQL processing.

Update Merging Rules merge two consecutive SQL activities executing updates on the same database tables into a single SQL activity. As an example, consider two successive INSERT statements both updating the same database table. By using the SQL UNION ALL operation, the *Insert-Insert Merging* rule merges these INSERT statements, thereby reducing the number of SQL activities in the BPEL/SQL process. As Figure 4 shows, there are similar rewrite rules for other combinations of updates.

The class of *Tuple-to-Set Rules* addresses loops iterating over a given set and executing an SQL activity for each tuple of the set. These rules replace such a loop and the SQL activity in the loop body by a single SQL activity, which covers the semantics of the entire loop. By transforming the tuple-oriented SQL statement into a set-oriented SQL statement, the iterative execution of the SQL activitiy becomes needless. Thus, we can remove the ForEach activity from the process logic. In the following section, we discuss the *Insert Tuple-to-Set* rule in more details.

5.1 The Insert Tuple-to-Set Rule

The *Insert Tuple-to-Set* rule requires a situation as shown on the left hand side of Figure 5. A ForEach activity iterates over a set of rows that is provided by a preceding SQL activity. The body of the ForEach loop consists of a single SQL activity containing an INSERT statement. By applying the *Insert Tuple-to-Set* rule, we replace the ForEach activity by a single SQL activity covering a set-oriented INSERT statement and the SQL statement that produced the set (shown in Figure 5 right hand side).

The rational behind this rule is to avoid calling a database system in each loop iteration because this causes overhead for transmitting and processing SQL statements. Another motivation is that the set-oriented SQL statement resulting from this rewrite rule offers additional optimization opportunities at the database level. This rewrite rule is an example for optimizations that are not achievable by the underlying DBMS alone. The DBMS receives only one INSERT statement per iteration of the ForEach activity. Hence, it has no means to combine these statements into a single one as done by the *Tuple-to-Set* rule.

In the following, we show how to apply this rule to a BPEL/SQL process. The motiviation for going into details here is twofold. First, we want to show that it is not straight-

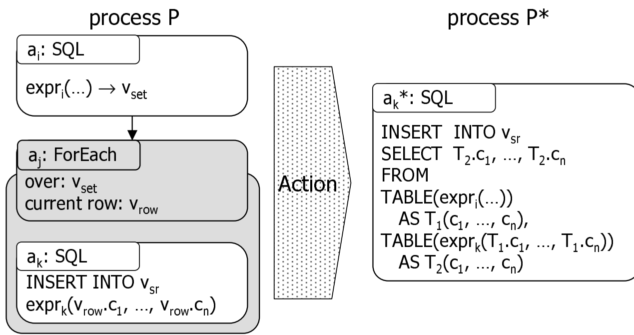


Figure 5: Insert Tuple-to-Set Rule

forward to define rule conditions in such a way that the process semantics is preserved when applying a rewrite rule. Secondly, the example demonstrates that for the efficient evaluation of rule conditions, a process representation that explicitly defines control flow dependencies as well as data dependencies is mandatory.

In analogy to conventional optimization, we consider activities that have no side effects and that are defined on a single data source. To exclude non-determinism at the choreography level, we assume processes without parallel activities referencing the same variable. We call processes that comply with these assumptions *well-formed processes*.

5.1.1 The Rule Conditions

As shown in Figure 5, the Insert Tuple-to-Set rule considers a BPEL/SQL process P that is transformed into process $P*$. $V = \{v_{set}, v_{row}, v_{sr}\}$ is the set of variables, v_{set} being a set variable, v_{row} providing a row of a materialization set, and v_{sr} being a set reference variable referring to a table. $A = \{a_1, a_2, ..., a_n\}$ is the set of activities of P, where the index uniquely numbers all activities in A.

A first set of conditions characterizes the activities the *Insert Tuple-to-Set* rule builds on:

- Activity Condition A1: Activity a_i is of type SQL providing the results of query expression $expr_i$ in a set variable v_{set}.
- Activity Condition A2: ForEach activity a_j iterates over v_{set} and provides the current row in variable v_{row}. The value of column c is denoted by $v_{row}.c_l$ with $l = 1...n$.
- Activity Condition A3: Activity a_k is of type SQL being the only activity within the loop body of a_j. Activity a_k executes an INSERT statement $expr_k$ that is either a *query expression* or a *VALUES expression* taking the tuple values provided by v_{row} as input.

Before we discuss further conditions based on data dependencies and interfering control flow dependencies in Section 5.1.3, we demonstrate the rule's action part.

5.1.2 The Rule Action

The following transformation steps convert P into $P*$:

- Transform a_k to a_k* by rewriting the SQL statement of a_k as it is shown in Figure 5. Intuitively speaking, we "pull up" the INSERT statement by joining $expr_i$ with a correlated table reference containing the results of expression $expr_k$ for each row. Due to the correlation between the joined tables within the FROM clause, we add the keyword TABLE to the table reference. Remark that this is the generalized version of the *Tuple-to-Set* rule. In Figure 2, we applied a simplified rule that is appropriate for INSERT statements with a VALUES clause.
- Replace a_j including a_k by a_{k*}.
- Remove a_i and adapt the control flow accordingly, i.e., connect all direct preceding activities with all direct succeeding activities of a_i.

This transformation converts the former correlation between the ForEach activity and the SQL activity in its body into a correlation between two tables in a_{k*}. This opens up optimization at the database level and thus leads to performance improvements.

5.1.3 Further Data and Control Flow Dependencies

Now, we extend the set of conditions by adding further conditions based on data dependencies and interfering control flow dependencies:

- Data Dependency Condition D1: A *single* write-read data dependency based on v_{set} does exist between a_i and a_j, such that a_i writes v_{set} before a_j reads v_{set}.
- Data Dependency Condition D2: There is a *single* write-read data dependency based on v_{row} between a_j and a_k, such that a_j writes v_{row} before a_k reads it.
- Value Stability Condition S1: v_{set} is stable, i.e., it does not change between its definition and its usage.
- Value Stability Condition S2: In each iteration of a_j, a_k reads that value of v_{row} that is provided by a_j.

In the following discussion, we demonstrate that it is not sufficient to consider only data dependencies. Rather, we also have to take into account control flow dependencies that interfer with data dependencies.

The focus of *Condition D1* is on activities providing a materialization set that is afterwards iteratively read by a ForEach activity. Remark that the existence of a write-read data dependency does not only depend on the fact that two activities access the same variable. We also have to take into account control flow dependencies. Consider the situation depicted in Figure 6(a) where a_i and a_j are executed on alternative paths. In BPEL, this is modeled by switch or pick activities. Although both activities process the same variable, there is no write-read data dependency since at most one of the activities is executed. As a consequence, we can not apply the *Insert Tuple-to-Set* rule. Remark that Figure 6 shows several variations in the control flow of process P from Figure 5. Hence, activities a_i and a_j are exactly the same as in Figure 5. Especially, the ForEach activity a_j contains a_k as before.

The same problem arises, if only one of the two activities is executed on an alternative path. Assume, as shown in Figure 6(b), that a_j is part of an alternative path, but a_i is not. Since we can not determine, whether a_j will execute at all, we can not guarantee that a write-read data dependency between both activities exists at runtime.

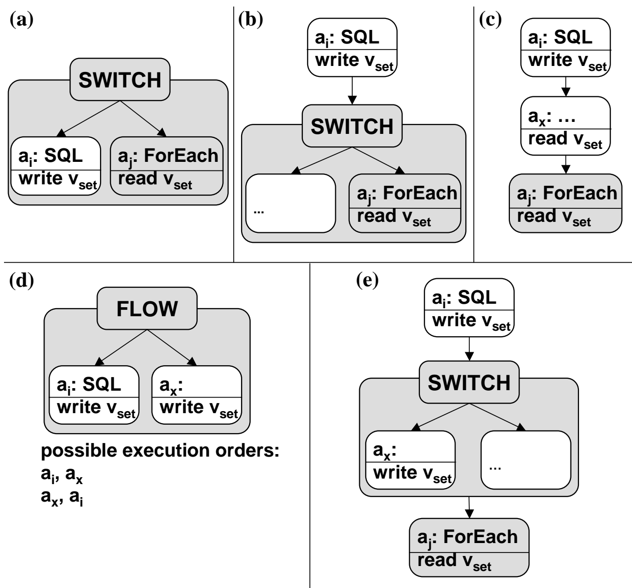


Figure 6: Critical interferences among data flow and control flow dependencies

Condition D1 also requires that there is a *single* write-read data dependency between a_i and a_j based on v_{set}. Hence, there must not be another activity that reads v_{set} and is executed after a_i. In Figure 6(c), we assume that such an activity a_x does exist, i.e., there is a write-read data dependency based on v_{set} between a_i and a_j as well as a write-read data dependency based on v_{set} between a_i and any activity a_x that is executed after a_i. Thus, a_j as well as a_x depend on the value of v_{set} written by a_i. If we would apply the *Tuple-to-Set* rule, we would remove a_i, and thereby destroy the write-read data dependency between a_i and a_x. As a result, the runtime behavior of $P*$ and P may differ, since, in $P*$, a_x reads a different value of v_{set} than in P. Hence, we only can apply this rule, if there is no activity other than a_j relying on the value of v_{set} written by a_i.

Condition D2 claims the existence of a single write-read data dependency between a_j and a_k based on v_{row}. Different to D1, the ForEach activity and the SQL activity a_k express a well-defined control flow dependency, because the definition of ForEach activities ensures that a row of v_{set} is provided by a_j before it is processed by a_k. Hence, there is a write-read dependency between both activities. Finally, from condition A3 follows that a_k is the only reader of v_{row}, since it is the only activity executed in the loop body.

According to *Condition S1*, the rule's action can only be applied if v_{set} does not change during its usage. Hence, we have to ensure that there is no other activity than a_i writing variable v_{set}. In other words, there is no write-write data dependency based on v_{set} between a_i and any activity $a_x \in A$. Such data dependencies exist, if a_x writes v_{set} between the execution of a_i and a_k and one of the following control flow dependencies holds:

1. Activity a_i and a_x are executed in parallel (see Figure 6(d)). As discussed above, our assumption on well-formed processes prohibits this situation. The same argument holds if activity a_j and the enclosed activity a_k are executed in parallel to a_x.
2. Activity a_x is defined on an alternative path between a_i and a_k (see Figure 6(e)). If a_x writes v_{set} there is a write-write dependency between a_i and a_x. Condition S1 ensures that the *Insert Tuple-to-Set* is not applied in this situation.

The purpose of *condition S2* is to make sure that in each iteration of a_j, a_k reads the value of v_{row} that was previously written by a_j. Therefore, we have to avoid activities that change v_{row} and run in parallel to the ForEach activity. This is excluded because we assume well-formed processes.

5.2 Properties of the Ruleset

The rewrite rules presented in the previous sections show the following properties: They preserve the semantics of the original process and they likely improve its performance.

The *Semantics Preserving Property* is individually guaranteed for each rule by its composition. Each rule consists of a condition and an action part. A rule action may only be executed if the corresponding condition evaluates to true. A condition identifies data dependencies as well as control flow dependencies between activities such that the proposed action keeps the process semantics.

For the *Performance Improvement Property* the following observations are important:

- All rewrite rules reduce the number of activities which in turn reduces the processing overhead. A reduced number of SQL activities in the choreography layer leads to fewer database calls. This reduces the overhead for translating, optimizing and processing SQL statements. Furthermore, this may reduce the size of intermediate result and thus the amount of data transferred between the database level and the choreography layer.
- By merging SQL statements in the choreography layer, we create more sophisticated SQL statements, that provide a higher potential for optimization on the database level than a single SQL statement. This will most likely enable further opportunities at the database level, e.g., detecting common subexpressions.
- The application of a rewrite rule may enable the application of further rewrite rules, possibly enforcing the two positive effects mentioned above.

As our experimental results in Section 7 show, performance improvements of several orders of magnitude are achievable. In the following section, we discuss dependencies between rewrite rules and show that typically the application of a rule enables the application of further rules.

6. CONTROL STRATEGY

In this section, we explain the control strategy for rule-based optimization of business processes. It divides the overall process in several optimization spheres and applies rewrite rules considering their dependencies.

6.1 Enabling Relationships

Our control strategy exploits dependencies among rewrite rules, i.e., the application of one rule may enable the application of another rule. The dependencies depicted in Figure 7 are as follows: The application of any *Activity Merging*

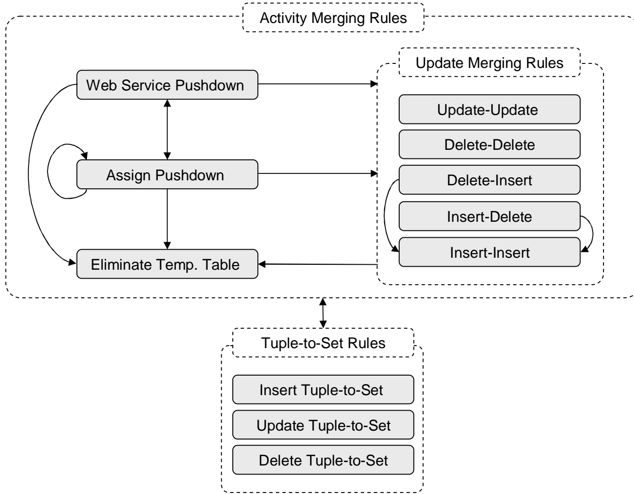


Figure 7: Enabling Relationships

rule to the activities inside a ForEach activity may reduce the number of these activities to one, and, in turn, may enable the application of the *Tuple-to-Set* rule. The application of a *Tuple-to-Set* rule constructs a new SQL activity that might further be merged via application of an *Activity Merging* rule. If an assign activity cuts the data flow between two activities, pushing the assign activity into the succeeding activity may result in a direct data dependency between the two activities and thereby enable the application of *Activity Merging Rules*. The same holds for the *Web Service Pushdown* rule enabling further *Activity Merging Rules*. The application of an *Update Merging* rule may reduce the number of updates on a table to a single one. If such a single update is executed on a temporary table, the *Eliminate Temporary Table* rule might become applicable. Among the *Update Merging Rules*, there is no specific order except for the *Insert-Insert*, *Delete-Insert* and *Insert-Delete* rule. The application of the latter two results in an INSERT statement that uses an EXCEPT to combine an insert as well as a delete operation. This may enable the application of the *Insert-Insert* rule. Among the *Tuple-to-Set Rules*, there is no specific order. Each of these rules addresses a different update statement in the loop body.

Merging activities produces more sophisticated SQL statements. Besides the enabling relationships on the choreography level, this may also enable optimization at the database level, that would not be possible otherwise. The performance gain depends on the optimization potential of the SQL statements as well as on the capabilities of the query optimizer of the database management system that processes these statements.

6.2 Optimization Spheres

Due to the structure of a process, we have to consider boundaries that must not be crossed during the optimization procedure. This is necessary to avoid changes to the original process semantics when applying rewrite rules.

Such a boundary is given in BPEL by scope activities that define the execution context of activities. A BPEL process defines a global processing scope for all its embedded activities. Scopes can be nested to create global and local execution environments for activities.

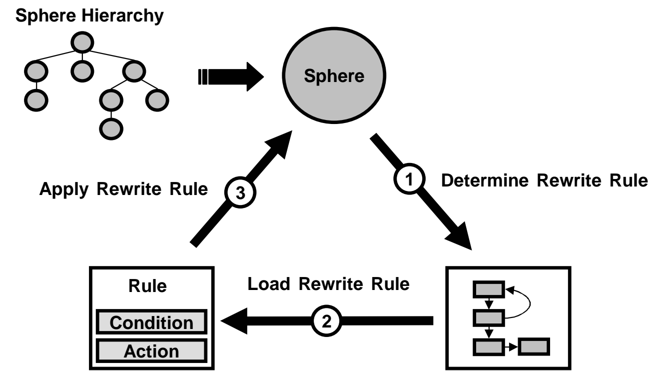


Figure 8: Optimization Procedure

A scope can declare fault- and compensation handlers defining undo steps of arbitrary complexity that reverse the effects of partially completed processing steps. Removing or inserting an activity from or into a scope causes an adaptation of these handlers in order to preserve the original process semantics. But determining and adapting affected handlers is a sophisticated task. Thus, in order to preserve the original semantics and to reduce complexity, we defined transformations that do not cross scope boundaries. We consider a scope as a closed optimization space, that we call Scope Optimization Sphere (SOS). The effects of a rewrite rule are always local to a SOS.

We consider Loop Optimization Spheres (LOS) as another type of optimization sphere. They comprise a ForEach activity with its nested activities and all surrounding activities that are necessary for applying a *Tuple-to-Set* rule. In analogy to SOSs, all effects of a rule's action are local to a LOS.

6.3 Overall Control Strategy

As shown in Figure 3, our optimizer engine gets the PGM representation of a process as input. In a first step, the optimizer identifies all optimization spheres within this representation. The nesting character of a process defines a tree that represents a hierarchical ordering on all optimization spheres. The global processing scope defines the root and the most nested optimization spheres define the leaves of this tree. We traverse this tree in a depth-first manner (in post-order) to guarantee that all nested spheres are processed prior to an enclosing sphere. When optimizing an enclosing sphere, we treat all nested SOSs as black boxes. However, we consider their data dependencies to activities within the enclosing optimization sphere, in order to prove a rule's condition. Unlike nested SOSs, we do not treat nested LOSs as black boxes, since we want to exploit the enabling relationships provided by the LOS's optimization taking place before.

For each optimization sphere, we apply our ruleset according to the control strategy that we discuss below. For each sphere type, we use a different control strategy that comprises the associated ruleset and the order in which the rules are applied. This way, a sphere is optimized in an iterative way, until we have processed all rules.

Figure 8 illustrates the optimization procedure. In a first step, the next rule to be considered is determined by the appropriate control strategy. Within the sphere, we search for a group of activities that fulfills a rule's condition. If found,

we apply the rule's action to these activities. When this rule is not applicable any more, we proceed with the next rule according to the control strategy. This proceeds until we have applied the full ruleset. This way, we optimize all spheres within the hierarchy, until we reach the root sphere that is optimized in a final step.

Algorithm 1 optimizeSphereHierarchy

Require: sphere-hierarchy sh
Ensure: optimized sphere-hierarchy sh
  while sh is not fully traversed **do**
    $s \leftarrow getNextSphere(sh)$
    $optimizeSphere(s)$
  end while

Algorithm 2 optimizeSphere

Require: sphere s
Ensure: optimized sphere s
  $cs \leftarrow getControlStrategy(s)$
  while cs is not finished **do**
    $r \leftarrow getNextRule(cs)$
    while s is not fully traversed **do**
      $a \leftarrow getNextActivity(s)$
      $m \leftarrow findMatch(a, s, r)$
      if $m \neq \emptyset$ **then**
        $applyRule(m, r)$
      end if
    end while
  end while

Algorithms 1 and 2 illustrate this procedure. Method *optimizeSphereHierarchy* (Algorithm 1) traverses a given sphere hierarchy *sh*, and calls method *optimizeSphere* (Algorithm 2) for each optimization sphere in *sh*. This method is responsible for the rule-based optimization of a sphere. Function *getNextSphere* implements the depth-first traversal strategy. Method *optimizeSphere* first calls function *getControlStrategy* that either returns a LOS or SOS control strategy *cs* depending on the given sphere type of *s* (see Section 6.4). *getNextRule* delivers the next rewrite rule *r*, until we have processed *cs*. In a next step, we try to find all matches for *r* in sphere *s*. Therefore, function *getNextActivity* implements a depth-first traversal strategy in this sphere returning all activities in *s* exactly once and ignoring activities that are part of a nested LOS or SOS. For each delivered activity *a*, function *findMatch* checks, whether *r*'s condition matches to a subprocess in *s* starting with *a*. It returns match *m*. The rule conditions guarantee that there is at most one match for each activity *a*. Match *m* serves as input for method *applyRule* that transforms *s* by applying *r*'s action. Otherwise, we skip this method and consider the next rewrite rule in the following iteration. The separation of the two algorithms allows to adapt the control strategy on both levels towards given constraints.

6.4 Control Strategy for Optimization Spheres

As shown in Figure 9, there are different control strategies for the sphere types, a LOS control strategy and a SOS control strategy. Both are based on the enabling relationships discussed in Section 6.1 and shown in Figure 7.

When optimizing a LOS, the objective is to remove the whole loop by applying a *Tuple-to-Set* rule. Due to their enabling relationships, we first apply the class of *Activity Merging Rules* to the activities that are part of the LOS. Since they may have enabling relationships on other rules, we start with the two Pushdown rules. Due to their mutual enabling relationships we alternate between both rules as long as they are applicable. Afterwards, we apply all *Update Merging Rules* before the *Eliminate Temporary Table* rule. The order among the *Update Merging Rules* is as follows: we start with the *Update-Update* and *Delete-Delete* rule. Due to their enabling relationships, we execute the *Delete-Insert* and *Insert-Delete* rule before executing the *Insert-Insert* rule. After having executed all *Activity Merging Rules*, we finally apply the remaining *Tuple-to-Set Rules* according to the ordering shown in Figure 9. In the best case, we have succeeded in replacing the loop by a single SQL activity, otherwise the loop will be retained.

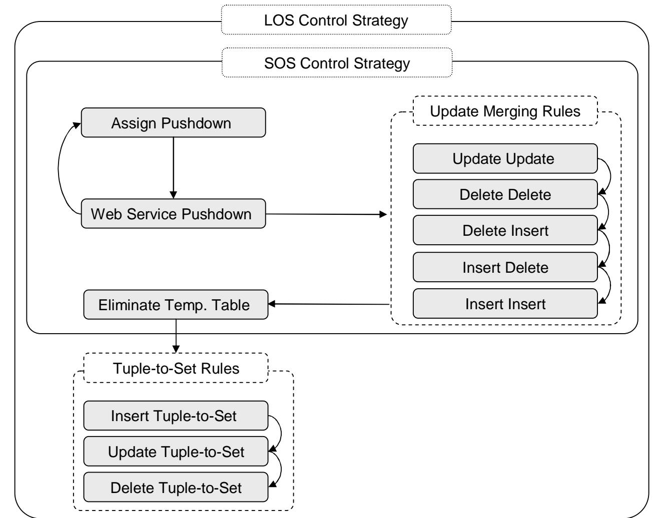


Figure 9: Control Strategy for Optimization Spheres

Since we consider nested loops within a SOS as a single optimization sphere, all rewrite rules except the *Tuple-to-Set Rules* take also part in the SOS control strategy.

6.5 Termination

For the set of rewrite rules and the presented control strategy, we can guarantee, that the optimization process terminates after a finite number of steps for the following reasons.

The sphere hierarchy is traversed from bottom to top, i.e., when the optimization process leaves an optimization sphere and ascends to its enclosing parent sphere, the optimization process will never re-descend to it. Hence, the traversal finally ends in the optimization sphere at top-level.

The traversal strategy within an optimization sphere guarantees, that each activity of an optimization sphere is considered exactly once, when searching for a rule match. This ensures, that the process of match finding will terminate within an optimization sphere in a finite number of steps.

With the current set of rules, each rule application reduces the number of activities. So, oscillating rule applications are impossible, i.e., there can not be a cycle among rules, where some rules undo the transformations of other rules.

Our ruleset consists of a finite number of rules and there is a finite number of activities in each optimization sphere. Furthermore, if for a certain source activity a merging rule

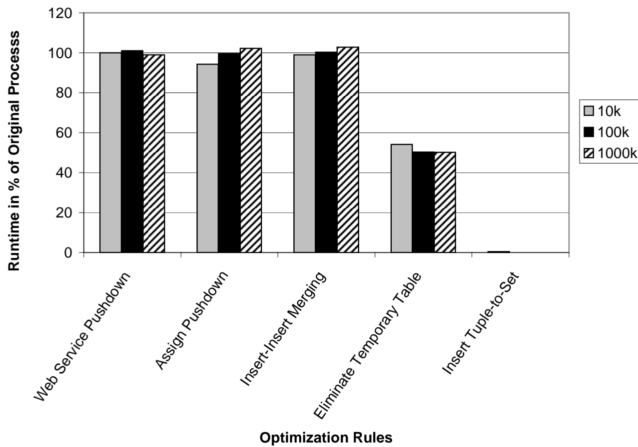


Figure 10: Effectiveness of Rewrite Rules

has resolved all its data dependencies to other activities that enable a rule's application, then the rule can not be applied to this activity any more. Each *Tuple-to-Set rule* can only be applied once to the activities within a LOS because it always eliminates some of the pattern matching activities. Therefore, we conclude, that each rule can only be applied a finite number of times to the same activity.

7. EXPERIMENTS

As part of a proof of concept, we analyzed the effectiveness of rewrite rules in two scenarios. The first one comprises small business processes to which only a single rewrite rule is applicable. This allows to evaluate rewrite rules in isolation. The second scenario is more complex and allows the application of several rewrite rules according to the control strategy explained in the previous section.

7.1 Experimental Setup

The experimental setup consists of three components: the runtime environment for business processes and for Web services as well as the database management system. All components run on a Windows Server 2003 system with two 3.2 GHz processors and 8 GB main memory. We used WebSphere Process Server version 6.0.1 [7] as the runtime environment for business processes. The business processes access data managed by DB2 version 8.1. The table that is accessed by data management activities in the processes was provided in varying size, ranging from 10,000 up to 1 Million rows. For the experiments presented in Figure 11, we additionally used Oracle 10g Release 2.

7.2 Results

In a first set of experiments, we focused on individual rewrite rules. We designed various processes that (i) allow the application of exactly one rewrite rule, and (ii) consist of a minimum set of activities. In Figure 10 we show results for processes that allow the application of the *Web Service Pushdown*, the *Assign Pushdown*, the *Insert-Insert Merging*, the *Eliminate Temporary Table*, and the *Insert Tupel-to-Set* rule, respectively. These results allow to evaluate the isolated effects of each rewrite rule. Any business process and the corresponding optimized process were executed more than 100 times. The average runtime of the original pro-

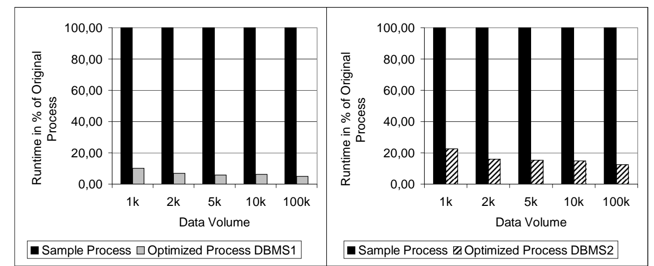


Figure 11: Performance Improvement for Sample Scenario (see Figure 1) on different DBMSs

Table 1: Effectiveness of Insert Tuple-to-Set rule: Factor by which runtime is reduced when applying Insert Tuple-to-Set rule

Data Volume	10k	100k	1000k
Factor	245	2090	≈15000

cess is taken as 100%. The corresponding average runtime of the optimized version is reported in Figure 10. In order to identify scalability effects, we repeated all experiments with varying data volume (10k, 100k, 1000k).

Figure 10 shows two groups of rewrite rules. The *Pushdown* and *Update Merging Rules* belong to a first group of rules that do not substantially influence process performance. Performance improvement and performance degradation are both within a 5% range to the 100% margin. These rules are nevertheless indispensable due to their enabling property as discussed in Section 6.1.

A more detailed analysis reveals why these rules did not improve process performance. The *Web Service Pushdown* rule had no effect because calling a Web service in a SQL statement causes approximately the same overhead than calling it on the process level. The *Insert-Insert Merging* rule combines two INSERT statements into a single one. As the optimizer of the database system was not able to identify a more efficient query plan for the combined SQL statement, the process performance was not affected. The same argument also applies to the remaining *Update Merging Rules*.

The other group consists of the *Eliminate Temporary Table* rule and the *Tuple-to-Set Rules*. They lead to significant performance improvements. Applying the *Eliminate Temporary Table* rule always cuts the process runtime by at least a factor of two. A detailed analysis reveals the main influencing factors: (i) There is no overhead for the lifecycle management concerning the temporary table. (ii) The rule replaces a set reference that refers to a temporary table directly by the SQL statement providing the content of this temporary table. Hence, two SQL statements are merged into a single one, which, in this case, allows the underlying database system to identify a more efficient query plan.

The *Tuple-to-Set Rule* cuts the runtime of a process by several orders of magnitude. Table 1 shows this effect in more detail. *Tuple-to-Set Rules* combine the iterated execution of SQL statements into one data management activity. In the 10k case for example, 10,000 tuple-oriented INSERT statements are merged into one set-oriented INSERT statement that is processed much more efficiently and that causes less processing overhead.

In Figure 11, we demonstrate the effect of applying rewrite

rules to a more complex scenario, i.e., the sample process shown in Figure 1. The number of executions of the SQL activity inside the ForEach activity is proportional to the table cardinality. Hence, the execution time of the original process, denoted as *Sample Process*, grows linear to the table cardinality. In Figure 11, we show the average runtime of the original process as 100%. As already shown in Section 2.2, the *Optimized Process* is the result of applying three rewrite rules to the *Sample Process*: The *Web Service Pushdown* rule, a *Tuple-to-Set Rule* and finally, the *Eliminate Temporary Table* rule. The only difference between the two optimized processes in Figure 11 is that the SQL statements of the process are executed by different database systems from different vendors. The figure shows that (i) performance improvements of an order of magnitude are also achievable in such complex scenarios, (ii) that they are largely independent of the table cardinality, and (iii) that they are independent of the underlying database system. A more detailed analysis reveals that the performance improvement is mainly caused by the *Tuple-to-Set* rule and the *Eliminate Temporary Table* rule. This finding is consistent with the results presented in Figure 10. Nevertheless, the application of additional rewrite rules was necessary to exploit enabling effects, i.e., *Web Service Pushdown* enables the *Tuple-to-Set Rule*, which in turn enables the *Eliminate Temporary Table* rule (as described in Section 6).

8. CONCLUSION

In this paper, we have shown a promising approach to extend the optimization of business processes to data processing affairs. Our approach adds another level of optimization in between well-known process-level optimization and database-level optimization. As a result, and for the first time, business process specifications can be optimized over the whole spectrum from the process level to the data level.

We introduced an optimization framework that is built on a rule-based rewrite approach combined with a sophisticated multi-stage control strategy to guide the optimization process. The distinctive property of our approach is that the rewrite rules are based on a process graph model that externalizes data dependencies as well as control flow dependencies of the business process. We have shown that these two kinds of dependencies are indispensable in order to guarantee for sound rewrite rules that keep the semantics of the original business process. Based on a prototype and a case study, we have shown that independent of the underlying DBMS, there is a huge optimization potential that induces significant performance improvements.

In future work, we will extend the scope of applicability with respect to the supported subset of BPEL, additional workflow languages as well as support for non-SQL data management, e.g., based on XQuery.

9. REFERENCES

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *Proc. SIGMOD Conference*, 2004.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.

[3] N. Dalvi, S. Sanghai, P. Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In *Proc. PODS Conference*, May 2001.

[4] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[5] K. Hergula and T. Härder. Coupling of FDBS and WfMS for Integrating Database and Application Systems: Architecture, Complexity, Performance. In *Proc. EDBT Conference*, 2002.

[6] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proc. of 3rd International Conference on Business Process Management*, 2005.

[7] IBM. *Information Integration for BPEL on WebSphere Process Server*. http://www.alphaworks.ibm.com/tech/ii4bpel.

[8] IBM. WebSphere Business Integration Server Foundation v5.1. http://www-306.ibm.com/software/integration/wbisf.

[9] V. Josifovski, P. Schwarz, L. M. Haas, and E. T. Lin. Garlic: a new flavor of federated query processing for DB2. In *Proc. SIGMOD Conference*, 2002.

[10] D. Kossmann. The State of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[11] T. Kraft, H. Schwarz, R. Rantzau, and B. Mitschang. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *Proc. VLDB Conference*, 2003.

[12] F. Leymann, D. Roller, and M.-T. Schmidt. *Web services and business process management*. IBM Systems Journal, 41(2), 2002.

[13] Microsoft. Windows Workflow Foundation. http://msdn.microsoft.com/windowsvista/building/workflow.

[14] OASIS. Web Services Business Process Execution Language Version 2.0. Committee Draft, Sept. 2005. http://www.oasis-open.org/committees/download.php/14616/wsbpel-specification-draft.htm.

[15] Oracle. Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html.

[16] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible Rule-Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD Conference*, 1992.

[17] A. Rosenthal and U. S. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. In *Proc. VLDB Conference*, 1988.

[18] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proc. SIGMOD Conference*, 2000.

[19] T. Sellis. Multiple-Query Optimization. *TODS*, 13(1):23–52, 1988.

[20] T. K. Sellis and L. D. Shapiro. Query Optimization for Nontraditional Database Applications. *IEEE Trans. Software Eng.*, 17(1):77–86, 1991.

[21] S. Shankar, A. Kini, D. DeWitt, and J. Naughton. Integrating databases and workflow systems. *SIGMOD Record*, 34(3):5–11, 2005.

[22] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query Optimization over Web Services. In *Proc. VLDB Conference*, 2006.