How Foreign Function Integration Conquers Heterogeneous Query Processing

Klaudia Hergula
DaimlerChrysler AG
Research and Technology (Dept. FTK/A)
HPC 0516, Epplestr. 225, 70546 Stuttgart, Germany
+49 711 17-96565
klaudia.hergula@daimlerchrysler.com

Theo Härder
University of Kaiserslautern
Dept. Of Computer Science (AG DBIS)
P.O. Box 3049, 67653 Kaiserslautern, Germany
+49 631 2054030
haerder@informatik.uni-kl.de

ABSTRACT
With the emergence of application systems which encapsulate databases and related application components, pure data integration using, for example, a federated database system is not possible anymore. Instead, access via predefined functions is the only way to get data from an application system. As a result, retrieval of such heterogeneous and encapsulated data sources needs the combination of generic query as well as predefined function access. In this paper, we present a middleware approach supporting such a novel and extended kind of integration. Starting with the overall architecture, we explain the functionality and cooperation of its core components: a federated database system and a workflow management system connected via a wrapper. Afterwards, we concentrate on essential aspects of query processing across these heterogeneous components focusing on the impact of the functions included. We discuss the operations the wrapper should provide in order to extend the workflow system's native functionality. In addition to selection and projection, these operations could include aggregation and the support of subqueries. Moreover, we point out modifications to the traditional cost model needed to consider the cost estimates for the function calls as well.

Keywords
Federated database system, workflow management system, function integration, wrapper, heterogeneous query processing, cost model.

1. MOTIVATION
Most enterprises have to cope with heterogeneous system environments where different network and operating systems, database systems (DBSs), as well as applications are used to cover the whole life cycle of a product. Initial approaches primarily focusing on problems of data heterogeneity were federated database systems (FDBSs) and multidatabase systems. So there exist adequate solutions for database integration even if there are still open questions [5][15].

But the database environment is changing now. While many enterprises had selected "their" DBS and designed their tailored DB schema in the past, they are now confronted with databases being delivered within packaged software. In such cases, the database system and the related application are integrated, and an application programming interface, the so-called API, is the only way to access the data. Thus, a (generic) database interface is not supported anymore. In the following, we call systems realizing such an encapsulation concept application systems. One of the most frequently used application systems is, for example, SAP R/3 [13], whose data can be accessed via predefined functions only. The same characteristics can be found in proprietary software solutions implemented by the enterprises.

As a consequence, pure data integration is not possible, since "traditional" DBSs have to be accessed using a generic query language (SQL) whereas application systems only provide data access via predefined functions. Instead, a combined approach of data and function access has to be achieved. Such scenarios can be encountered in many practical and/or legacy applications.

We consider an FDBS as an effective integration platform, since it provides a powerful declarative query language. Furthermore, it offers a large set of numerical processing functions as well as a broad range of scalability. Many applications are SQL-based to take full advantage of these properties. A query involving both databases and application systems includes SQL predicates as well as some kind of foreign function access. According to SQL99 [7] such a reference may occur as a function or as a table. In our view, the most important case is the reference to a function as a table, strictly considered as an abstract table.

To implement such an extended kind of integration, we have developed an integration architecture consisting of two key components: an FDBS and a workflow management system (WfMS). The FDBS is responsible for the integration of data whereas the WfMS is used to implement a kind of function integration. As a result, the WfMS provides so-called federated functions which are made available to the FDBS. Obviously, efficient query processing requires that these two components work together very closely. Such heterogeneous query processing reveals interesting aspects, since two completely different models – a data model and a function model – must be able to communicate and to work together.

In the remainder of this paper, we discuss basic questions about a smooth cooperation of these two components and we examine

CIKM'01, November 5-10, 2001, Atlanta, Georgia, USA.

various implementation aspects of heterogeneous processing. For this purpose, we introduce our integration architecture in Sect. 2 by depicting the structure and the participating systems. Since the connection of FDBS and WfMS based on a wrapper is a fundamental part of our architecture, we describe it in detail in Sect. 3. Moreover, we consider requirements on its functionality and outline its support for heterogeneous query processing in Sect. 4. The impact on the cost model is shown in Sect. 5. Finally, we briefly review related work and summarize our ideas.

2. OVERALL ARCHITECTURE

The goal of our three-tier integration architecture is to enable the applications to transparently access heterogeneous data sources, no matter if they can be accessed by means of SQL or functions (see Figure I). Applications accessing the integrated data sources comprise the upper tier, and the heterogeneous sources represent the bottom tier. Due to space limitations, we focus on the middle tier, the so-called integration server, which consists of two key components: an FDBS achieving the data integration and a WfMS which realizes a kind of function integration by invoking and controlling the access to predefined functions. In our terms, function integration means to provide federated functions combining functionality of one or more local functions [6].

In principle, specialized wrappers could be used to access each of the local functions typically supplied by different applications systems. These functions are frequently called together in a way where the output data of a function call is the input data of a subsequent function call. The execution of the single functions could be directly controlled by the FDBS. However, such an approach would require substantial extensions of the FDBS components in addition to the writing of the specialized wrappers. Furthermore, the FDBS had to cope with the different application systems and their local functions which could be distributed, heterogeneous, and autonomous.

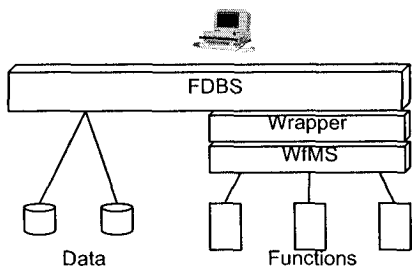

Figure 1. Integration Architecture.

Our key idea is to use a workflow for the execution of a federated function where its activities embody the local function calls and where the WfMS controls the parameter transfer together with the precedence structure among the local function calls. Then, a unified wrapper can be used to isolate the FDBS from the intricacies of the federated function execution and to bridge to the WfMS thereby supplying missing functionality (glue) and making various query optimization options available.

The mapping from federated to local functions is guarded by a precedence graph and it typically consists of a sequence of function calls observing the specific dependencies between the local functions. As a key concept of our approach, we use a WfMS as the engine processing such a graph-based mapping [6].

The workflow to be executed is a production workflow representing a highly automated process [10]. We decided to use a WfMS as a generic vehicle to obtain function access and, at the same time, to make such accesses transparent for the FDBS. Using such an approach, we can fall back on existing technology incorporated in commercially available products that supports complex mapping scenarios and transparent access to heterogeneous platforms. Moreover, it provides interfaces to the application systems to be integrated and can cope with different kinds of error handling, thereby facilitating distributed program execution across heterogeneous applications. Another important point is the fact that the solution is easier to adapt when the environment has to be changed.

FDBS and WfMS are connected by an interface realized by means of a wrapper according to the draft of SQL/MED (Database Languages - SQL - Part 9: Management of External Data, [S]). Actually, this wrapper consists of two parts: one part is located at the FDBS side serving the SQL/MED interface at the FDBS. The other part should be placed at the WfMS side wrapping the WfMS interface. Thus, communication between FDBS and WfMS is basically realized between these two wrapper parts. As a result, the WfMS provides so-called federated functions used by the FDBS to process queries across multiple external data sources.

The applications (users) can access the integration server via an object-relational interface connecting them to the FDBS. The FDBS's query processor evaluates the user queries and those parts requiring foreign function access are handed over to the wrapper which activates the WfMS. The workflow engine performs the function integration by calling the local functions of the referenced application systems as specified in the predefined workflow process. The wrapper returns the result back to the FDBS where it is mapped to an abstract table. The remaining parts of the user query are processed by the FDBS, i.e., the query is divided into the appropriate SQL subqueries for the SQL sources. Eventually, the subquery results are further processed by the query processor, if necessary, and merged to the final result.

All issues of query processing in the FDBS are well explored [2][11]. Therefore, we will focus on the new aspects of our architecture. How can we smoothly integrate calls of predefined functions into the overall query processing of an FDBS? For this purpose, we have a deeper look into the wrapper-based connection between FDBS and WfMS and its realization in the next section.

3. HOW FUNCTIONS STEP INTO THE RELATIONAL WORLD

Before discussing how to handle functions within heterogeneous query processing, we address the key problem of function evaluation and how to map functions to tables. Moreover, we point out the requirements on the wrapper's functionality.

3.1 Mapping Functions to Tables

When mapping functions to tables, we have to distinguish between structural and semantic aspects. The structural mapping defines the way how to represent a function's signature in the form of a table so that it can be referenced in SQL queries. Semantic mapping, on the other hand, considers the semantics of the functionality implemented by a specific function and its mapping to function references in the form of abstract tables

expressed by SQL queries. Unfortunately, this information cannot be derived automatically from a function's signature. Most functions to be integrated are provided by legacy systems which support a hierarchical view on their data. In most cases, the input parameter values are used as input for predicates processing an "equals" operation like "Give me the numbers of all departments with a budget of X dollars": `SELECT deptno FROM departments WHERE budget = X`. Such legacy system functionality is rather fixed and, therefore, inadequate for the support of SQL queries using, for example, range conditions like `SELECT deptno FROM departments WHERE budget < X`. Obviously, the semantics of "<" has to be reproduced by multiple compensating function calls.' Hence, future systems should provide more flexible APIs allowing, for instance, to pass in addition to input values the related (comparison) operator as parameter. In that case, the expressiveness of SQL queries could be more adequately simulated by function calls.

Next, we will describe our way of structurally mapping functions to tables.

Structural Mapping

Realizing a structural mapping by a wrapper, we have to bridge the conceptual differences between FDBS and WfMS. Relational database systems are based on a data model. WfMSs, in turn, are built on a functional model. The wrapper enables access to the WfMS from the relational FDBS through abstract tables. A canonical mapping is to represent each function as a tuple of its input values and output values. Such a structure can be represented as a table.

Functions can be classified in two categories. Scalar functions return a single result value or result tuple for each set of valid input values whereas relation-valued functions return a set or list of result values or result tuples for each set of valid input parameters. Our canonical mapping of relation-valued functions creates a conventional abstract table consisting of a set of one or more rows containing the given input values and the corresponding output values. In addition, projection as well as selection operators can be applied to such a table in order to minimize data shipped between wrapper and FDBS. This enables a wrapper to take over parts of the query evaluation by performing such functions on behalf of the FDBS. A scalar function returns for each tuple of input values only a single output tuple. Therefore, the simple canonical mapping of scalar functions always creates a table with a single row.

Proceeding this way, the mapping is absolutely transparent to (external) users or applications. Neither they have to pay attention to restrictions nor they have to learn a particular use of SQL when accessing abstract tables.

3.2 Needed Wrapper Functionality

First, we take a look at the tasks the wrapper must fulfil in order to provide for a relational FDBS access to a WfMS. In order to achieve this functionality, the wrapper must mediate between both systems and map the functional model to the data model, as well as provide support for abstract table queues returning the result to the FDBS.

From the FDBS side, it must be able to accept a partial query[2] regarding data access to the abstract tables. This query may include operators such as projection and selection using comparison predicates.

Based on its knowledge about the abstract table representations, the wrapper prepares a local access plan. In the first place, such a local access plan consists of calls to WfMS process instances. Additionally, it may include wrapper-internal operations for requested functionality not natively supported by the WfMS. As usual, the optimization target is to push down as many predicates as possible to the WfMS.

During the execution of a local access plan, the wrapper invokes functions embodied by WfMS process instances. For each call, the wrapper derives input values from the query and receives result values from the WfMS. The input/output value pairs of each WfMS function can be processed by further operations. Finally, the query result set is returned to the FDBS query execution engine as an abstract table queue.

In addition to these basic tasks, the wrapper may be extended supporting further operations. We group these functions in core, base, and extended wrapper services. As our rationale, we follow the idea of realizing as much optimization effect with as little implementation effort as possible. Afterwards, we explain why particular operations should not be addressed by the wrapper, but left to the FDBS instead.

Core functionality consists of all functions needed to map between abstract tables and WfMS function calls. If no additional functionality is available, each reference to an abstract table in an SQL query has to be realized by materializing and delivering the complete foreign table. For this reason, wrappers to be integrated into DBMS or FDBS processing should allow for basic optimization options, that is, they should enable push-down of selection and projection operations in the first place.

Hence, *base functionality* on abstract tables should include projection to a subset of columns and selection of a subset of rows using any boolean combination of comparison predicates of columns with constants or list of constants. Such comparison predicates contain an operator $\Theta \in \{=, \neq, <, \leq, >, \geq\}$.

Core and base functionality could be *extended* by more advanced query execution operations such as grouping and aggregation, subqueries regarding abstract tables, and set comparison. Furthermore, the join of two abstract tables might be implemented within the wrapper in order to minimize communication caused by wrapper calls of the FDBS. On the other hand, join operators have been implemented in FDBSs in very efficient ways. Therefore, it should be critically evaluated whether a reimplementation of join functionality is justified.

Because wrappers receive only queries regarding abstract tables, there is no need to implement any functions beyond a subselect (such as union, except, intersect). Separate subselects are usually represented by independent queries to be combined by the FDBS.

[1] In this paper, we will not discuss synchronization issues to be performed by the application systems, if repeatable results are required.

[2] In the following, we denote a partial query delegated to the wrapper as query for short.

Subqueries regarding FDBS-managed tables should not be executed by the wrapper, since such functions can be handled better by the FDBS.

Transfer of the result set is performed via an abstract table queue which may be achieved in a pipeline mode depending on the availability of result tuples or in a block mode at the end of the function call. Furthermore, a result set returned by a function invocation may have some useful characteristics important for the overall query optimization. For example, a result contains an interesting sort order. Such information should be passed as a hint to the FDBS query processor to avoid further sort operations.

4. QUERY PROCESSING ACROSS FDBS AND WfMS

So far, we have described how FDBS and WfMS are connected. In the following, we focus on heterogeneous query processing performed by these systems. First, we introduce some terms and definitions describing the native functionality supported by the workflow engine. Afterwards, we focus on the realization of the operations defined as base and extended functionality within the wrapper. Since optimization aspects like pushing down operations and reducing the amount of data shipped between FDBS and WfMS have been sufficiently analyzed in the past [16], we will not elaborate on them.

4.1 Terms and Definitions

In the following, we describe our forma1 representation of the query processing. It is based on the relational algebra extended by two additional operators. Taking the signature of a function $f(i_1, \ldots. i_m, o_1, \ldots. o_n)$ as a starting point, we define $I:=\{i_k \mid k=1, \ldots. m\}$ and $O:=\{o_l \mid l=1, \ldots. n\}$ as the sets containing the input and output parameters. Hence, the function signature can also be written as $f(I, O)$.

In Sect. 3.1, we have explained how to map functions to relations. To enable a simple mapping, the function $f(i_1, \ldots. i_m, o_1, \ldots. o_n)$ is represented as an abstract relation $R(i_1, \ldots. i_m, o_1, \ldots. o_n)$. Since the input and output parameters are mapped to the attributes of the relation, it also can be described as $R(I, O)$.

A function call then represents the following algebraic expression:

$$f(I,O) \leftrightarrow \pi_O(\sigma_I(R))$$

Hence, a function call is equivalent to a selection on relation R specified by predicates based on the values of the input parameters. In addition, there is a projection on the output parameters. Since this term exactly describes the functionality of our WfMS, we introduce a new operator φ representing the workflow functionality:

$$\varphi(R) := \pi_O(\sigma_I(R))$$

In order to be able to map a SELECT * functionality, i.e. the projection on all attributes $\pi_{I,O}$, the input parameters have to be concatenated to the corresponding tuples of the result set returned by the WfMS. This operation is described by the operator κ:

$$\kappa_I(\varphi(R)) := I \| \varphi(R) = I \| \pi_O(\sigma_I(R)) = \pi_{I,O}(\sigma_I(R)) = \sigma_I(R)$$

Based on these definitions, we will now discuss the functionality which should be made available by the wrapper in order to extend the native query support provided by the WfMS.

4.2 Base Functionality

In our view, the combined functionality of wrapper and WfMS should at least contain selection and projection in order to reduce data shipping between FDBS and WfMS. Therefore, the wrapper has to extend the WfMS core functionality by additional functions. In the following, we discuss the realization of such a functionality and its specific characteristics.

4.2. I Selection

Before describing the selection operation in detail, we present some prerequisites which are essential for our considerations. Afterwards, we differentiate between selections on input parameters and those on output parameters.

Prerequisites

As described above, the values for the input parameters have to be derived from the predicates in the WHERE clause of the SQL query to be evaluated. Only if the WHERE clause specifies a comparison predicate (containing an equals operator) for each single input parameter, i.e. q(R), the result set can be captured by a single function invocation. In any other case, additional function calls have to be initiated to compensate the missing input parameter values; in the extreme case, all possible values of a missing input parameter have to be supplied. In order to perform such substitutions, that is, to simulate the original query by additional function calls, some restrictions on the value count as well as the value set of an input parameter have to be applied:

1. The value count $V_c(R_a, i)$ of each input parameter of an abstract table R_a has to be finite.
2. It must be possible to enumerate the elements of its value set $V_s(R_a, i)$.

Otherwise, the number of compensating function calls cannot be determined and, consequently, the query must be rejected.

Selection Based on Input Parameters

Assume the comparison operator specified in the predicate matches the operator implemented by the function and all required input values are given by constant predicates. Then, the selection is defined as follows causing exactly one function call:

$$\sigma_I(R) = \kappa_I(\varphi(R))$$

As described above, the operation $o,(R)$ is processed by calling exactly one function. If one input parameter value v_i is missing, $V_c(R, i)$ function calls must be executed to go through all possible combinations of the given values and the missing input parameter value. This number of calls is multiplied with the value count of each unspecified input parameter value. Consequently, the operation $\sigma_{i1}(R)$ with only a single input parameter value causes the following number of function calls:

$$1 + V_c(R,i_2) \times V_c(R,i_3) \times \ldots \times V_c(R,i_m) = 1 + \prod_{k=i_2}^{i_m} V_c(R,k)$$

Thus, if $I=\{i_1, \ldots. i,\}$ and we want to process $\sigma_{(i=1)}(R)$, we get the union of the result sets of the required function calls:

$$\sigma_{(i=1)}(R) = \left(\kappa_I\left(\varphi^1(R)\right) \cup \kappa_I\left(\varphi^2(R)\right) \cup \ldots \cup \kappa_I\left(\varphi^n(R)\right)\right)$$

In this equation, n describes the number of function calls needed to compute the entire result set and corresponds to the number derived above based on the value counts.

Similar aspects have to be considered, if the comparison operator of a predicate does not match the operator implemented by the function. For instance, a function *f(i,* o) computes the predicate *(i = constant)* whereas the SQL statement contains the predicate *(i ≤ constant).* This case also results in multiple function calls to derive the anticipated result. Again, the restrictions described above have to be applied to enable the derivation of the required function calls. In our example, a function with *(i = constant)* and further functions for every single element of the value set of *i* less than the specified constant value must be processed. Consequently, the union of the result sets of several selections has to be derived as shown in the following example (provided that $V_s(R, i) = \{1, 2, 3, \ldots, 10\}$):

$$\sigma_{(i\leq 5)}(R) = \sigma_{(i=5)}(R) \cup \sigma_{(i=4)}(R) \cup \sigma_{(i=3)}(R) \cup \sigma_{(i=2)}(R) \cup \sigma_{(i=1)}(R)$$

Selection Based on Output Parameters

Considering predicates including output parameters, we again identify two different cases. First, we suppose that all required input values are specified by predicates in the SQL statement. Then, the selection has to be applied on the result set returned by the function call:

$$\sigma_{I,O}(R) = \sigma_O(\sigma_I(R)) = \sigma_O(\kappa_I(\varphi(R)))$$

If, on the other hand, the predicates are based on output parameters only, all combinations of input parameter values have to be determined for the corresponding functions. Based on the union of the returned result sets, the selection on the output parameter can be processed. Obviously, this case can lead to an enormous number of function calls.

4.2.2 Projection

Regarding projection we have to consider three different cases: projection on a) input parameters only, b) output parameters only, and c) both types of parameters.

When applied on input parameters, projection can be realized without calling any function in some cases. For that, the wrapper has to check if the specified input values are included in the corresponding value set. If not, the result set is 0 and no function call is needed. Moreover, if the function's signature contains only one input parameter, the result of $\pi_i(R)$ is *i,* if the value specified is part of $V_s(R, i)$, derived without any function call. In any other case, the wrapper must initiate the required functions and add the input values to the result set. If there is a projection on a subset of the input parameters, the wrapper has to cut off the parameter values not needed.

Looking at projection on output values, i.e. $\pi_O(R)$, it is obvious that it matches exactly the native functionality $\varphi(R)$ of the WfMS. Consequently, the result set of the WfMS can be passed on to the FDBS. If there is a projection on the subset of the output parameters, the wrapper has to cut off the other ones by applying the operation $\pi_O(\varphi(R))$.

Finally, input as well as output parameters are projected. The general case of projecting on all parameters (SELECT *) is described in Sect. 4.1. In any other case, the projection on subsets of input and output parameters is the concatenation of the results processed for each parameter type as described above. For instance, the SQL statement

```
SELECT  i1,  o2,
FROM  R
WHERE . . .
```

is then processed as follows:

$$\pi_{i_1,o_2}(\varphi(R)) = \kappa_{i_1}(\pi_{o_2}(\varphi(R)))$$

Summarizing our ideas, we get the following results (see Table 1):

Table 1. Projection

Case a)	$\pi_I(R) \subseteq V_s(R, I)$	$\pi_{i_1}(R) \subseteq V_s(R, I)$
Case b)	$\pi_O(R) = \varphi(R)$	$\pi_{o_1}(R) = \pi_{o_1}(\varphi(R))$
Case c)	$\pi_{I,O}(R) = \kappa_I(\varphi(R))$	$\pi_{i_1,o_2}(R) = \kappa_{i_1}(\pi_{o_2}(\varphi(R)))$

4.3 Extended Functionality

Next, we discuss functionality which, in our opinion, is not strictly necessary, but could improve query optimization by reducing the amount of data shipped between FDBS and WfMS.

*4.3.1 Grouping **and** Aggregation*

Typically, the combination of grouping and aggregation significantly reduces the number of result tuples and, therefore, should be supported within the wrapper. Like in the cases before, we have to distinguish between aggregations or groupings based on input or output parameters. Since the considerations on aggregation operations are similar to those on projection, we will now focus on the grouping aspects.

Let γ_L represent the effect of grouping and aggregation. L is a list of elements, each of which is either a grouping or an aggregated attribute. In the following, we will not consider cases including a WHERE clause, since selection already is discussed in Sect. 4.2.1. In addition, we assume that the set of grouping attributes and that of aggregated attributes are disjointed.

We start with grouping based on input parameters with any kind of aggregation on any parameter x $\gamma_{1,aggr(x)}(R)$. No matter if the grouping is based on all input parameters or only a part of them, the number of function calls n_{fc} is always the same in order to get all values:

$$n_{fc} = \prod_{k=i_1}^{i_m} V_c(R, k)$$

The difference is found in the number of result tuples which is dependent on the value count $V_c(R, i)$ for a single input parameter or the combination of several of them. In order to illustrate this result, we consider the following example. Assume a function *costs* which returns as output values a department's *target* costs, *actual* costs, as well as a classification based on the actual costs. The input parameters include a department number *deptno* and a *year*. This function is mapped to an abstract table *Costs* containing attributes based on the function's parameters. Now we want to process the operation $\gamma_{deptno,sum(actual)}(Costs)$ on the abstract table Costs, adding up the actual costs over all years for each department:

```
SELECT  SUM(actual)
FROM  Costs
GROUP  BY  deptno
```

For each single department, i.e. each element x of V_s(Costs, deptno), the actual costs for every year have to be summed up by processing the following operations:

$$\gamma_{deptno=x,sum(actual)}(Costs) = \kappa_{deptno}\left(sum\left(\pi_{actual}\left(\sigma_{deptno=x}(Costs)\right)\right)\right)$$

As described above, $\sigma_{(deptno=x)}$(Costs) requires a function call for each combination of the deptno values x and the values for the stored years. This is necessary, because both deptno and year have to be provided as input values for the function costs (cf section 4.2. I). The overall result of this grouping/aggregation operation is the union of the result tuples of each group resp. department:

$$\gamma_{deptno,sum(actual)}(Costs) = \bigcup_{x \in V_s(Costs,deptno)} \gamma_{deptno=x,sum(actual)}(Costs)$$

In the next example, we consider grouping based on output parameters $\gamma_{O,aggr(x)}$(R). Thus, we examine the following statement returning the number of identified cases for each classification:

```
SELECT  COUNT(deptno)
FROM    Costs
GROUP   BY  class
```

Obviously, the number of function calls is n_{fc} again, since we have to receive all output parameters based on the combination of all input parameter values. In order to identify the different groups, this intermediate result has to be sorted, denoted by the operator τ_{class}(Costs), where class represents the attribute of Costs the sort operator is based upon. Thus, an intermediate result R_{int} for $\gamma_{class,count(deptno)}$(Costs) is built up as follows:

$$R_{\text{int}} = \tau_{class}\left(\kappa_{deptno}\left(\pi_{class}(Costs)\right)\right)$$

Since the count aggregation must be applied to each group of deptno, the overall result set is obtained by:

$$\gamma_{class,count(deptno)}(Costs) = \bigcup_{x \in V_s(Costs,class)} \gamma_{x,count(deptno)}(R_{\text{int}})$$

If a sort operator is not supported by the wrapper, grouping based on output parameters cannot be processed within the wrapper. Instead, the unsorted R_{int} has to be passed on to the FDBS for further processing. When grouping is based on input as well as output parameters, there are no new aspects to consider.

The huge number of function calls can be reduced if the evaluation of the HAVING clause is added to the wrapper's functionality. Since it represents a selection on the grouping attributes, it can be used to restrict the required function calls. Unfortunately, this is only true for grouping on input parameters. In the case of output parameters, we still get n_{fc} function calls, since a selection on output parameters has to be processed.

4.3.2 Subqueries and Set Comparison

Evaluation of subqueries is limited to abstract tables. In addition, the subquery must not include operations that are not supported by the wrapper or WfMS. These requirements have to be met in order to guarantee that the subquery can be processed completely by the wrapper and the WfMS. Moreover, we have to distinguish between correlated and uncorrelated subqueries. In the uncorrelated case, the subquery has to be evaluated only once and does not reveal any new aspects. Therefore, we will focus on the correlated case, where the subquery has to be evaluated for each row of the outer relation. Since most subqueries are introduced by set comparison predicates like IN, EXISTS, or SOME, set comparison operations should be supported by the wrapper in order to minimize the derivation steps as well as the amount of data shipped between the systems. Assume the following SQL statement has to be processed on a relational table $R_r(a_1, a_2)$ and an abstract table R_a(i, o):

```
SELECT  a1
FROM    Rr
WHERE   a2  IN  (SELECT  o
                 FROM    Ra
                 WHERE   i  =  a2)
```

In order to call the wrapper only once, all values of a_2 are passed to the wrapper at a time. These values are used to prepare the function calls needed to process the result of the correlated subquery. For each element of $V_s(R_r, a_2)$ a function is called with $i=a_2$. Assume the wrapper supports set comparison; it can compare the function's output with a_2 and can build a result tuple containing a_2 as well as "true" or "false" representing the result of the set comparison. The union of the results of all the functions is then returned to the FDBS as the basis for the selection of a_1. So the work to be done in the wrapper is the following, where the operation *IN(x,* Y) represents the test whether x is an element of the set Y:

$$\bigcup_{x \in V_s(R,a_2)} \kappa_x\left(IN\left(x, \varphi_{(i=x)}(R_a)\right)\right)$$

5. NEW ASPECTS OF THE COST MODEL

In this section, we outline new aspects which have to be taken into account when elaborating a cost model including function calls. In general, the costs of a query include the costs for communication time, processor time, disk I/O's, and memory. In our case, the costs for communication time and those for disk I/O's are the lion share of the costs for a query execution plan (QEP). Regarding communication time, the existing cost models for heterogeneous query processing in FDBSs are also valid for our approach. In order to minimize communication time between FDBS and WfMS, the amount of data shipped should be reduced. This is basically realized by pushing down as many operations as possible to the WfMS or its wrapper.

However, the traditional cost calculation for accessing a relation R has to be modified when including function calls. Selinger et al. [14] introduced the following cost estimate based on disk I/O's and processor time:

$$\text{Cost(R)} = pagefetches(R) + W \times systemcalls(R)$$

In this formula, the function *pagefetches* describes the number of physical accesses to the external storage (mainly disks). The function *systemcalls* represents the number of calls to the access system which is derived from the number of tuples needed for the query processing. Consequently, this function is a good indicator for the expected processor load. Factor W is used to include the computer configuration. If the system is CPU-bound, W should be a rather big value in order to make QEPs with low CPU demand preferable. If the system is I/O-bound, W should be small to avoid the selection of I/O-intensive QEPs.

Unfortunately, this formula does not fit to our approach, since we do not access base relations R but abstract relations R_a which are built up by means of one or more function calls. Hence, the function *systemcalls* is replaced by the function *wrapcalls* and

represents the number of calls to the wrapper resp. the WfMS needed to receive the specified contents of the abstract relation R_a. Moreover, we introduce the function *datacomm* which depicts the communication costs for shipping result tuples from the WfMS to the FDBS. Furthermore, physical access to secondary storage is replaced by the number of workflow process instances executed which is described by the function *procexec*. Since the execution of workflows is dependent on the number of their activities invoked, we introduce a factor A indicating the number of activities defined for the workflow. Our performance tests have shown that the execution time of a workflow process increases more or less linearly with the number of activities within that process. Factor W can still be used to adjust the QEP optimization. For instance, if the WfMS process is much slower than that of the FDBS, i.e. the system is WfMS-bound, W should be rather small in order to avoid the selection of WfMS-intensive solutions. Consequently, we get the following cost estimate for accessing an abstract relation R_a:

$$Cost(R_a) = A \times procexec(R_a) + W \times wrapcalls(R_a) + datacomm(R_a)$$

To determine the number of function calls, the query processor must know at least two things: the value count $V_c(R_a, i)$ of each attribute of R_a representing an input parameter as well as each single value of its value set $V_s(R_a, i)$. Many other parameters which are usually contained in the optimizer statistics cannot be determined at all or only for those attributes denoting the input parameters of the function. Therefore, it is often difficult to estimate selectivities for predicates in order to get an idea of the size of intermediate results. As a consequence, standard selectivities as described in [14] have to be used instead.

In the following, we consider the impact on the overall costs if operations like selection, projection, aggregation, and subqueries are pushed down to the wrapper. We will focus on the number of wrapper calls as well as workflow process executions and the amount of tuples shipped between FDBS and WfMS.

When estimating the costs for selection or projection processed by the FDBS and WfMS respectively, the communication costs represent the only varying factor. This is based on the fact that σ_1 is always processed at the WfMS side, since the predicates are needed to determine the function calls. σ_0 is then applied to the result of the function calls, thus reducing the number of result tuples. In the case of projection, the number of function calls is the same, no matter where the operation actually is processed. Since in both cases the data has to be retrieved by several function calls, the communication costs only vary by the amount of data returned to the FDBS. Since these observations are well known from traditional query optimization, we will not further follow up these considerations.

Instead, we discuss the costs for grouping and aggregation as well as subqueries in greater detail. Starting with grouping and aggregation, we get the results for wrapper calls, process executions, and tuples shipped as described in Table 2.

Including a HAVING clause which is based on I, the processing cost in the FDBS remains the same. For the WfMS, we get the following result (with H := {having attributes}):

$$A \times \prod_{i \notin G} V_c(R,i) + W \times 1 + \prod_{h \in H} \frac{1}{V_c(R_a,h)} \times \min\left(\frac{1}{2}T(R_a), \prod_{a \in G} V_s(R_a,a)\right)$$

Obviously, the amount of data shipped between FDBS and WFMS is smaller when grouping is pushed down. If the HAVING clause is also processed by the wrapper, the result set is further reduced and, in turn, the number of process executions is decreased.

Table 2. Cost comparison for grouping and aggregation processed in FDBS and pushed down to WfMS

	Overall cost for grouping and aggregation
FDBS	With $T(R_a)$:= # tuples of R_a $A \times \prod_{k=i_1}^{i_m} V_c(R_a,k) + W \times 1 + T(R_a)$
WfMS	With G := {grouping attributes} $A \times \prod_{k=i_1}^{i_m} V_c(R_a,k) + W \times 1 + \min\left(\frac{1}{2}T(R_a), \prod_{a \in G} V_s(R_a,a)\right)$

Finally, we compare the costs for subqueries and set comparisons. Obviously, the correlated case of subqueries causes a lot of interaction between FDBS and WfMS, since the subquery cannot be processed completely in a single step by the wrapper. Consequently, the cost estimates for the two cases should at least differ in the number of wrapper calls. Our cost estimates are based on the example introduced in Sect. 4.3.2 to illustrate them in a comprehensive way (see table 3).

Table 3. Cost comparison for correlated subqueries

	Correlated subqueries
FDBS	$A \times V_c(R_r,a_2) + W \times V_c(R_r,a_2) + V_c(R_r,a_2)$
WfMS	$A \times V_c(R_r,a_2) + W \times 1 + V_c(R_r,a_2)$

In summary, we can state that the functionality extension of the wrapper can reduce costs not only by minimizing the amount of data shipped, but also by reducing the number of wrapper calls and process executions. However, the support of all comparison operators by means of compensating function calls is inappropriate, since the number of function calls may explode. Assume a function has three input parameters with a value count of only 10 for each of them. If, for instance, a selection with a predicate for only one of these parameters is specified, we already get 10x10=100 compensating function calls. Even worse, if no predicate at all is specified, there are 10x10x10=1000 function calls. Consequently, we must consider alternatives reducing this huge number of function calls. One solution could be an enhanced parser rejecting queries that do not specify predicates for each single input parameter. Another way is to shift the specification of input values from predicates to user-defined functions (UDFs) in the WHERE clause. These UDFs would force the user to provide values for each input parameter.

6. RELATED WORK

Most approaches dealing with the integration of heterogeneous data sources focus on the capability to integrate different data models and heterogeneous systems providing an interface which is not as powerful as SQL. Approaches like Garlic [16], Information Manifold [9], or TSIMMIS [12] embody mediator- or wrapper-based solutions where missing functionality of the data sources is compensated by the integration server. In contrast to our work, these approaches provide general solutions and algorithms for integrating any kind of data source. In our case, all non-SQL sources are integrated by the WfMS and the SQL sources are managed by the FDBS. As a consequence, the FDBS has to integrate only a single non-SQL source: the WfMS. Hence, the FDBS just has to integrate SQL sources and the functionality provided through the single WfMS wrapper interface. This means that we can concentrate on a specific solution integrating the workflow system.

Furthermore, we focus on the integration of a functional interface on the FDBS side. Chauduri et al. [1] have discussed this topic very early, thereby demonstrating how references to foreign functions can be expressed in a query language. But they did not address the problem of limited access patterns. In such cases, for example, a particular function input must be stated similar to specific selection criterias in the WHERE clause of an SQL statement. Approaches like [3] and [4] propose solutions for this limitation by binding attributes in order to support queries on such data sources.

7. SUMMARY

In this paper, we have introduced an approach for the integration of heterogeneous data sources accessible via generic queries or predefined functions. The consideration of predefined functions has been motivated by current system environments where databases and applications are encapsulated providing an API with functions instead of a database interface. We have described the components of our integration architecture introducing the FDBS, the WfMS and its wrapper-based connection. Since FDBS and WfMS must cooperate when processing a user query, we have focused on heterogeneous query processing in our approach. First, we have pointed out the functional requirements of such a heterogeneous query processing system. We have described the operations which should be provided by the WfMS and the wrapper in order to support efficient heterogeneous query processing. Based on these requirements, we have shown how much of the required operations are supported natively by the WfMS functionality. Moreover, we have identified aspects which have to be taken into account when extending its native functionality by implementing additional operations like selection, projection, grouping, and aggregation within the wrapper. In addition, we have considered the cost model and modifications to it to estimate the costs for function calls.

8. REFERENCES

[1] Chaudhuri, S., and Shim, K. Query Optimization in the Presence of Foreign Functions. In Proc. 1 9th Int. Conf. on Very Large Data Bases, Dublin, 1993, 529-542.

[2] Czejdo, B., Rusinkiewicz, M., and Embley, D.W. An Approach to Schema Integration and Query Formulation in Federated Database Systems. In Proc. 3rd IEEE Int. Conf. on Data Engineering, Los Angeles, 1987, 477-484.

[3] Florescu, D., Levy, A., Manolescu, I., and Suciu, D. Query Optimization in the Presence of Limited Access Patterns. In Proc. ACM SIGMOD Conf. on Management of Data, Philadelphia, 1999, 3 1 l-322.

[4] Garcia-Molina, H., Labio, W., and Yerneni, R. Capability-Sensitive Query Processing on Internet Sources. In Proc. 1 5th IEEE Int. Conf. on Data Engineering, Sidney, 1999, 50-59.

[5] Harder, T., Sauter, G., and Thomas, J. The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution. VLDB Journal 8: 1, 1999, 22-29.

[6] Hergula, K., and Harder, T. A Middleware Approach for Combining Heterogeneous Data Sources Integration of Generic Queries and Predefined Function Access. In Proc. 1"' Int. Conf. on Web Information Systems Engineering, Hongkong, 2000, 22-29.

[7] IS0 & ANSI. Database Languages - SQL - Part 2: Foundation, International Standard, 1999.

[8] ISO & ANSI. Database Languages - SQL -Part 9: Management of External Data, Working Draft, Sept. 2000.

[9] Levy, A.Y., Rajamaran, A., and Ordille, J.J. Querying Heterogeneous Information Sources Using Source Descriptions. In Proc. 22nd Int. Conf. on Very Large Data Bases, Bombay, 1996, 25 l-262.

[10] Leymann, F., and Roller, D. Production Workflow: Concepts and Techniques, Prentice Hall, 2000.

[1 1] Meng, W., and Yu, C. Query Processing in Multidatabase Systems. In W. Kim (ed.) Modem Database Systems: The Object Model, Interoperability, and Beyond, ACM Press and Addison-Wesley, 1995, 251-572.

[12] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. Object Exchange Across Heterogeneous Information Sources. In Proc. 1 1th IEEE Int. Conf. on Data Engineering, Taipei, 1995, 251-260.

[13] SAP AG. SAP R/3,2001. http://www.sap.com/solutions/r3/.

[14] Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. Access Path Selection in a Relational Database Management System. In Proc. ACM SIGMOD Conf. on Management of Data, Boston, 1979, 25 l-260.

[15] Sheth, A.P., and Larson, J.A. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys 22:3, 1990, 183-236.

[16] Tork Roth, M., and Schwarz, P. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, 1997, 266-275.