Coupling of FDBS and WfMS for Integrating Database and Application Systems: Architecture, Complexity, Performance

Klaudia Hergula[1] and Theo Härder[2]

[1] DaimlerChrysler AG, ITM, Databases and Data Warehouse Systems (TOS/TDW),
Epplestr. 225, HPC 0516, 70546 Stuttgart, Germany,
klaudia.hergula@daimlerchrysler.com
[2] University of Kaiserslautern, Dept. of Computer Science (AG DBIS),
P. O. Box 3049, 67653 Kaiserslautern, Germany,
haerder@informatik.uni-kl.de

Abstract. With the emergence of so-called 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 novel and extended kind of integration. In particular, so-called federated functions combining functionality of one or more application system calls (local functions) have to be integrated. Starting with the overall architecture, we explain the functionality and cooperation of its core components: a federated database system and, connected via a wrapper, a workflow management system composing and executing the federated functions. Due to missing wrapper support in commercial products, we also explore the use of user-defined table functions. In addition to our workflow solution, we present several alternative architectures where the federated database system directly controls the execution of the requested local functions. These two different approaches are primarily compared w.r.t. their mapping complexity and their performance.

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. Solutions primarily focusing on problems of data heterogeneity exist in the form of federated database systems (FDBSs) and multidatabase systems, even if there are still open questions [1, 2]. 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, so-called application systems. One of the most frequently used application systems is, for example, SAP R/3 [3], 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 anymore, since traditional DBSs have to be accessed using a generic query language (SQL) whereas application systems only provide data access via predefined functions.

We introduce an example in order to illustrate how users work with application systems today. The sample scenario is located in the purchasing department of an enterprise and can be found in similar forms in any other department. Assume the employee must decide whether he should order a new component delivered by a supplier already known. A purchasing system supports the employee by providing a function `DecidePurchase`. This function proposes a decision based on a calculated grade of quality and reliability and the number of the considered component. Unfortunately, the employee only knows the component name as well as the supplier number. As a consequence, he has to query some other systems to get the required input for the function `DecidePurchase`. Fig. 1 illustrates the single steps the employee has to go through, i.e. the functions he must call. He gets the quality as well as the reliability rate for the supplier calling the functions `GetQuality` of the stock-keeping system and `GetReliability` of the purchasing system with the supplier number. He then uses these results as input for the calculation of the component's grade by means of the function `GetGrade` and gets the first required input value for `DecidePurchase`. Moreover, he calls the function `GetCompNo` of the product data management system to query the corresponding number for the component name. With these values – the component's number and grade – he finally can call `DecidePurchase` to make his decision.

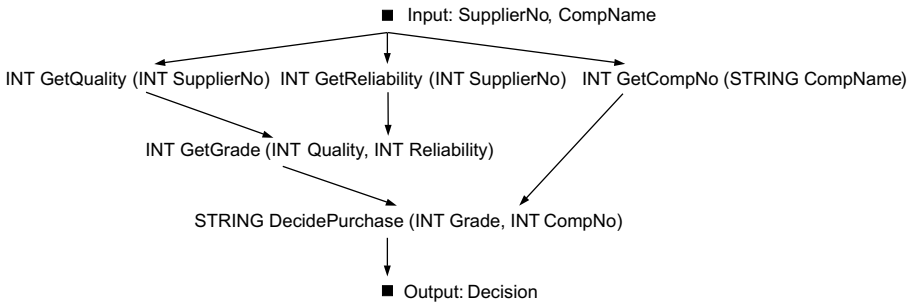

Fig. 1. Workflow process for the federated function `BuySuppComp`.

During the decision process, the user has to cope with three different application systems and three different user interfaces. Technically spoken, he manually achieves a kind of integration by calling the application systems' functions and

copying and pasting result values between them. So the user's interaction represents the glue between the application systems. In addition, we observe that there are steps that are processed in the same order again and again. This fact has led to our idea to support the user by providing so-called federated functions that implement single calls of the local functions and that hide these steps from the user. In our example, the user then has to call one federated function – let's denote it `BuySuppComp` – instead of five local functions.

Thus, an integration of functions or application systems is needed. Since also database systems could be involved in such a user request, 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. Many user applications are SQL-based to take full advantage of its properties. A query involving both databases and application systems includes SQL predicates as well as some kind of foreign function access.

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). Obviously, a WfMS is quite a big engine which seems to be oversized as part of a middleware. Hence, questions crop up, why not directly accessing each of the local functions by a user-defined table function (UDTF for short) instead of using the workflow engine. Therefore, we have implemented both alternatives to be able to examine the differences between them regarding mapping complexity and performance.

In the remainder of this paper, we describe architectures based on UDTFs with and without the workflow system in Sect. 2. In Sect. 3, we point out the mapping complexity these architectures are able to implement. Afterwards, we present the results of our performance tests in Sect. 4, answering the question how much time is consumed by the WfMS. In the remaining sections, we briefly review related work and summarize our ideas.

2 Integration Architectures

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 Fig. 2). Applications referring to a (homogenized) view to the data comprise the upper tier, and the heterogeneous data 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 [4] as introduced in our sample scenario. Considering the federated function `BuySuppComp`, one can see that the mapping from federated to local functions is guarded by a prece-

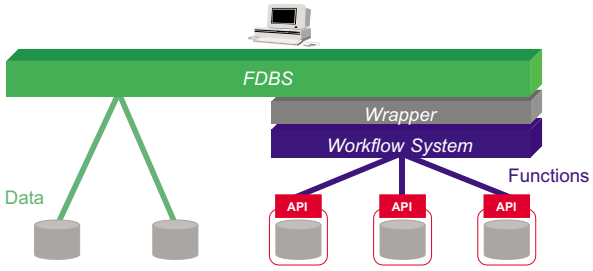

Fig. 2. Integration architecture.

dence 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 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 [4]. The workflow to be executed is a production workflow representing a highly automated process [5]. 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. In order to be independent of vendor-specific solutions, a standardized wrapper interface according to the draft of SQL/MED (Database Languages – SQL – Part 9: Management of External Data, [6]) is used. As a result, the WfMS provides so-called federated functions used by the FDBS to process queries across multiple external data sources.

We have decided to use the WfMS because we want to use existing technology instead of implementing the engine by ourselves. Moreover, the basic concept of a workflow engine matches our mapping graph and supports very complex mapping scenarios. In addition, the workflow engine enables transparent access to different platforms, hides the interfaces to the application systems to be integrated, and copes with different kinds of error handling. As a result, the WfMS implements distributed programming over heterogeneous applications and, thereby, abstracts the function integration towards the FDBS which then has to deal with only one interface, that is, that of the workflow engine resp. that of the wrapper. Finally, the implemented mapping is much easier to maintain when realized by means of a workflow product.

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.

At the moment, there is no database vendor supporting the SQL/MED wrappers. As an alternative, we decided to replace the wrapper by user-defined table functions (UDTFs, see Fig. 3). These UDTFs can be referenced in the FROM clause of an SQL query and return their result as a table to the FDBS. They can be implemented in different programming languages accessing any kind of data source. In such an architecture, each federated function is represented by a UDTF. In our example, a UDTF `BuySuppComp` can be referenced in a select statement starting the appropriate workflow process. Unfortunately, UDTFs only support read access, i. e., we are not able to propagate inserts, deletes, and updates. But since we want to get a first impression of the workflow performance, read access is a sufficient first step.

Of course, the use of a workflow engine seems to be oversized to some reader, since, in principle, specialized wrappers or UDTFs could be used to access each of the local functions which are often 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 or UDTFs. Furthermore, the FDBS had to cope with the different application systems and their local functions which could be distributed, heterogeneous, and autonomous.

Despite these potential drawbacks, we discuss possible solutions without a WfMS. In such cases, the integration logic has to be implemented by means of UDTFs only, where each single local function is connected by a UDTF to the FDBS. In the following, we present a spectrum of architectures based on UDTFs.

Simple UDTF Architecture

While providing only a simple connectivity for the local functions, the first approach burdens the application with the integration problem. Each local function is separately accessed by means of a UDTF, which then can be used in SQL queries. Since these UDTFs allow only for a single function access, we will call them *Access UDTFs* or *A-UDTFs* for short (see Fig. 3). The actual integration is not supported by this architecture. Instead, it is achieved by the application[1] by issuing several SQL statements referencing the A-UDTFs and perhaps composing the related result sets even 'manually'. For our sample scenario, we have to implement five A-UDTFs for the corresponding local functions. These A-UDTFs can then be referenced in SQL statements that are embedded in the application programming code. The logic of the federated function `BuySuppComp` is represented by the following select statement. Please note, that the syntax shown in the examples is based on the implementation of IBMs DB2 UDB v7.1 that introduces UDTFs with the key word TABLE and and a mandatory correlation

[1] Or rather by the application programmer.

name. UDTFs can be only referenced in the FROM clause which is processed in left-to-right order. This means that there is a precedence structure among the UDTF calls that is determined by the availability of the input parameter values. Since, for instance, UDTF `GetGrade` is dependent on the output values of UDTF `GetQuality` and `GetReliability`, it cannot be executed until the other two UDTFs have been finished. During the processing of the SQL statement, the table functions are called returning the output values. In the SELECT clause, the user can specify which output values to project. Although the statement seems to specify a cross join, we get only single values in the following example:

```
SELECT DP.Answer
FROM TABLE (GetQuality(SupplierNo)) AS GQ,
TABLE (GetReliability(SupplierNo)) AS GR,
TABLE (GetGrade(GQ.Qual, GR.Relia)) AS GG,
TABLE (GetCompNo(CompName)) AS GCN,
TABLE (DecidePurchase(GG.Grade, GCN.No)) AS DP
```

Obviously, this approach is not satisfactory at all, since the integration logic is hidden within the application code. If the developers have to change the integration scenario by adding or removing application systems and their local functions, they have to understand the current implementation possibly done by developers not present anymore. Usually, its documentation is incomplete or even missing, so they will need much more time to understand and to adjust it. Therefore, we will not further consider this approach.

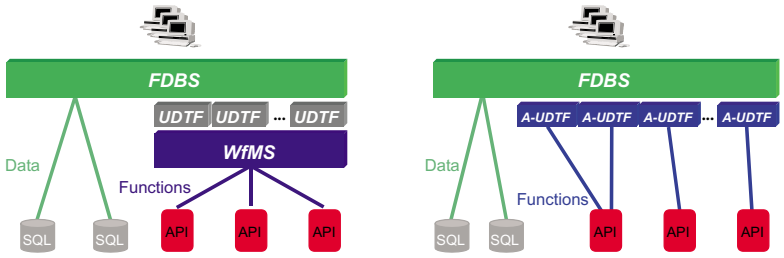

Fig. 3. WfMS approach (on the left) and simple UDTF approach (on the right).

Enhanced SQL UDTF Architecture

Next, we enhance the simple UDTF approach by pushing down the integration logic from the user code into the FDBS. In order to flexibly compose a federated function using multiple local functions, we introduce so-called *Integration UDTFs* or *I-UDTFs*. These I-UDTFs consist of an SQL statement which includes references to A-UDTFs, thereby implementing the integration logic. Hence, they incorporate our federated functions and lead to our enhanced SQL UDTF architecture (see Fig. 4). Unfortunately, in the product we used, the function body

may contain only one single SQL statement, i.e., the logic has to be expressed by one SQL statement. This restriction obviously results in further restrictions regarding the mapping complexity to be implemented. Since we are able to express the mapping logic of our example with one statement, the definition of the I-UDTF `BuySuppComp` looks as follows:

```
CREATE FUNCTION BuySuppComp (SupplierNo INT, CompName VARCHAR)
RETURNS TABLE (Decision VARCHAR) LANGUAGE SQL RETURN
SELECT DP.Answer
FROM TABLE (GetQuality(BuySuppComp.SupplierNo)) AS GQ,
TABLE (GetReliability(BuySuppComp.SupplierNo)) AS GR,
TABLE (GetGrade(GQ.Qual, GR.Relia)) AS GG,
TABLE (GetCompNo(BuySuppComp.CompName)) AS GCN,
TABLE (DecidePurchase(GG.Grade, GCN.No)) AS DP
```

In contrast to the simple UDTF architecture, the application code contains a rather simple select statement now:

```
SELECT BSC.Answer
FROM TABLE (BuySuppComp(SupplierNo, CompName)) AS BSC
```

Assessing the enhanced SQL UDTF architecture we can state that it is able to provide the applications with federated functions which can be referenced within SQL statements and, therefore, be combined with references to other federated functions or local and remote tables. Since the federated functions are implemented by means of SQL at the FDBS side, the maintenance of them is much more convenient than for the simple UDTF architecture.

Enhanced Java UDTF Architecture
This architecture is based on the same idea like the enhanced SQL UDTF architecture. The difference is found in the specific implementation of the integration logic to be realized by means of Java (see Fig. 4). So if we consider this architecture from bottom to top, each local function is made accessible to the FDBS via an A-UDTF written in Java. These A-UDTFs can now be used in the FROM clause of an SQL statement. The federated functions that are mapped to the local ones are realized based on Java I-UDTFs which include JDBC calls invoking the A-UDTFs. Proceeding this way, we can avoid the 'one SQL statement' restriction. Instead, the Java I-UDTF can issue as many SQL statements as needed in order to implement federated functions of much more complexity. Moreover, we can make use of all the features a programming language provides like, for instance, control structures. Transferring our sample to this architecture, the Java I-UDTF would contain the same SQL statement like the SQL I-UDTF since the logic can be expressed by one select statement.

The enhanced Java UDTF architecture seems to be the most powerful solution to implement federated functions within the FDBS. However, the maintenance becomes more difficult again, since the integration logic is partially hidden within the programming code. In our view, logic implemented by means of SQL

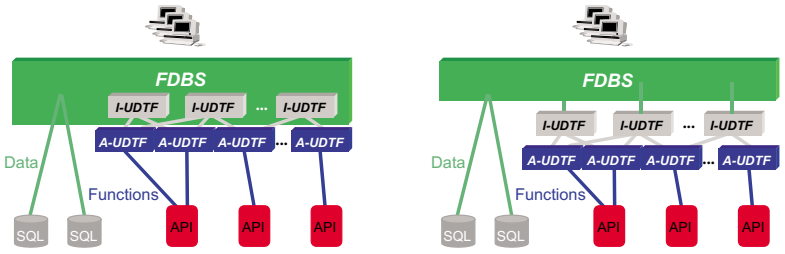

Fig. 4. Enhanced SQL UDTF (on the left) and Java UDTF approach (on the right).

only is easier to understand. Of course, we could use PSM (persistent stored module) stored procedures which support SQL as well as procedural extensions[2], so that we can obtain the same mapping complexity by means of SQL only. However, stored procedures have to be handled as 'procedures', that is, they can only be invoked by a CALL statement. This restriction, in turn, means that a user is not able to reference a stored procedure (which represents a federated function) in a select statement. Hence, such a mechanism cannot be combined with references to other federated functions or tables.

Alternative Architectures

Besides the presented architectures, there is also the possibility to implement an integration based on the WfMS only. In this case, the workflow system represents the top layer of an integration architecture accessing functions as well as data (via an FDBS, for instance). But since we focus on the data we get by means of function calls and its further processing, we believe that a database system provides an engine that is more suitable.

Moreover, there are further solutions possible without using an FDBS and a WfMS at all. For instance, a J2EE compliant application server could represent the integration engine implementing access to database and application systems by means of appropriate J2EE connectors. Enterprise Java Beans have to contain the mapping and integration logic. Another solution could be based on a message broker which is also able to execute a kind of precedence graph to some degree. Or the integration engine is completely implemented by ourselves supporting exactly the functionality needed without introducing a possibly oversized engine like the WfMS.

These hand-made solutions do not seem suitable for several reasons. First, we have to process data, no matter if accessed via SQL or functions, and we believe that a DBMS represents the best solution for processing data in a fast, reliable, and secure way. Second, we want to integrate data and functions providing a flexible and generic interface supporting references to data and functions. Third, it is not desirable at all to implement the integration logic by ourselves, since maintenance as well as further development is quite difficult. Instead, we want

[2] PSM stored procedures are defined by SQL99 and are also supported by DB2.

to use existing technologies and products that can be adapted or extended if necessary.

In the following sections, we concentrate on the WfMS approach (Fig. 3) and the enhanced SQL UDTF architecture (Fig. 4), since they best meet our requirements described above. Moreover, the workflow architecture is quite a new solution which has to be compared with more 'traditional' solutions regarding mapping complexity and performance.

3 Supported Mapping Complexity

In this section, we compare the enhanced SQL UDTF and the WfMS approach w.r.t. their mapping complexity. This complexity is mainly caused by the heterogeneity gap to be overcome when mapping federated functions to local functions. In the following, we classify the different forms of heterogeneity, listed by increasing complexity:

- **Trivial case**: One federated function is mapped to exactly one local function and their signatures are identical. Only the names of the functions and the parameters may differ.
- **Simple case**: In contrast to the trivial case, the signatures may be different, i. e., the number and data type of parameters do not match.
- **Independent case**: A federated function is mapped to more than one local function. Since the local functions are independent of each other, they can be processed in parallel.
- **Dependent case**: The next step allows dependencies between the local functions including linear, (1:n), (n:1), and cyclic dependencies.
- **General case**: Different forms of dependencies may occur and have to be handled together when more than one federated function has to be mapped to a set of local functions.

In the following, we examine to what extent these cases can be implemented by means of the UDTF and WfMS approach. For this purpose, we introduce suitable examples to illustrate the separate cases. In our sample scenario, three application systems are used. A stock-keeping system provides information about the components in stock, the corresponding supplier as well as their quality. A product management system stores the bill of material, whereas a purchasing system keeps information about the suppliers and their reliability. The data of these systems can be accessed by local functions.

Trivial Case
In the trivial case, a federated function `GibKompNr` represents a German version of an English local function `GetCompNo`. In this case, different function and parameter names have to be resolved by the mapping. Using the UDTF approach, this mapping is achieved by hiding the local function's signature behind that of the federated function.

The same concept is implemented by the WfMS approach where the signature of the connecting UDTF hides the names of the functions and parameters handled by the workflow process.

Simple Case
For the simple case, we demonstrate a type cast by changing the output parameter's data type. In addition, we have to cope with differing numbers of parameters in the function signatures. Assume a federated function `GetNumberSupp1234` that returns the stock-keeping number of a given component number for supplier `1234`. It is mapped to the local function `GetNumber` which asks for two input parameter values. Since the federated function provides only one (`CompNo`), we have to specify a constant value for the second input parameter. This constant is defined by the federated function which returns information about supplier `1234`. In addition, the resulting data type has to be converted from `INT` to `LONG`. As a result, the following federated function may be composed:

```
CREATE FUNCTION GetNumberSupp1234 (CompNo INT)
RETURNS TABLE (Number INT)
LANGUAGE SQL RETURN
SELECT BIGINT(GN.Number)
FROM TABLE (GetNumber(1234, GetNumberSupp1234.CompNo)) AS GN
```

The WfMS approach introduces so-called helper functions which are defined as additional activities in the workflow process and which implement the required type conversions. Comparable to the UDTF approach, the workflow solution can supply a constant value when calling the local function.

Independent Case
A federated function `GetSubCompDiscounts` returns the sub-components and the related supplier for a given component number which can be purchased with a given discount by calling the local functions `GetSubCompNo` and `GetCompSupp4Discount`. This operation requires the composition of the single result sets of the local functions to a common abstract table:

```
CREATE FUNCTION GetSubCompDiscounts (CompNo INT, Discount INT)
RETURNS TABLE (SubCompNo INT, SupplierNo INT)
LANGUAGE SQL RETURN
SELECT GSCD.SubCompNo, GCS4D.SupplierNo
FROM TABLE (GetSupCompNo(GetSubCompDiscounts.CompNo)) AS GSCD,
TABLE (GetCompSupp4Discount(GetSubCompDiscounts.Discount)) AS GCS4D
WHERE GSCD.SubCompNo=GCS4D.CompNo
```

The local functions return separate result tables for which the join predicate is used to select the tuples relevant for our query. For instance, result tuples of `GetCompSupp4Discount` representing component numbers that are not subcomponents of the given component are removed.

Using the WfMS approach, the independent case is still a rather simple task to be accomplished. The independent, i. e. parallel execution of functions is implemented by defining parallel activities whose results are combined by a helper function.

Dependent Case
The local functions are dependent on each other resulting in a precedence structure among the function calls, i.e., the output value of one local function is used as the input value of a subsequent local function. We have identified four different cases of dependency: linear, (1:n), (n:1), and cyclic dependency.

Linear dependency: Two local functions have to be composed for a federated function `GetSuppQual` which returns the quality of a supplier for a given supplier name. Since the local function `GetQuality` returns the quality for a given supplier number, the local function `GetSupplierNo` has to be called first to get the corresponding number. Its result is then used as input for `GetQuality`. This is the point where the UDTF approach encounters limiting factors the first time. Since SQL is a declarative language, there is no way to specify a particular order of function calls within a query. One possible workaround would be to nest the function calls like, for instance, `GetQuality(GetSupplierNo(SupplierName))`. Unfortunately, nesting of functions is not supported. Nevertheless, the DBMS we used for our prototype supports another implementation. In our case, we are able to reference two types of parameters as input of a local function: input parameters of the federated function as well as output parameters of other local function. In this way, we can model a kind of dependency between two functions. In our example, the input parameter of `GetQuality` references the output parameter of `GetSupplierNo` and, therefore, is dependent of `GetSupplierNo` and its result. This, in turn, implies that `GetQuality` cannot be processed before `GetSupplierNo`. We implement this solution by performing a cross product between the result values of our local functions. `GetSupplierNo` gets the input value of the federated function and, therefore, can be executed immediately. In contrast, the input of `GetQuality` is specified as the output value of `GetSupplierNo`. But this value is only available after `GetSupplierNo` has been executed.

```
CREATE FUNCTION GetSuppQual (SupplierName VARCHAR)
RETURNS TABLE (Qual INT)
LANGUAGE SQL RETURN
SELECT GQ.Qual
FROM TABLE (GetSupplierNo(GetSuppQual.SupplierName)) AS GSN,
TABLE (GetQuality(GSN.SupplierNo)) AS GQ
```

Please note that this solution is supported by the product we used. There is no guarantee that other products also enable such a proceeding. Moreover, one should keep in mind that SQL actually is not intended for such a procedural use[3]. There is no further selection required if the local functions return single values. However, if the result consists of tables, a selection has to be specified in the WHERE clause.

Considering the WFMS approach, a simple sequential order has to be defined by the control flow.

[3] Except for the procedural extensions for SQL stored procedures.

(1:n) dependency: Assume a federated function is mapped to three local functions where one local function is dependent on the other two local functions. This case is a combination of the independent case and the linear dependent case. Consequently, the implementation is comparable, because the dependency of the one local function is realized by using the output values of the other local functions which, in turn, get the input values of the federated function.

With a WfMS, a workflow process is defined in which two function calls are specified as parallel activities. The control flow specifies that the third function has to be processed after the first two functions have finished.

(n:1) dependency: In contrast to the (1:n) dependency, several local functions are dependent on a single local function. This mapping is solved in the same way as in the dependent cases above.

The WfMS approach can handle this case by the appropriate forks in the control flow.

Cyclic dependency: This kind of dependency cannot be implemented by the UDTF approach, since there are no control structures like a loop which are needed to iterate the cycle. At the moment, such control structures are only supported in PSM stored procedures. But when we use stored procedures representing federated functions, we are not able to combine them with other function or table references. On the other hand, the WfMS approach provides such control structures. The cyclic case is implemented by defining sub-workflows containing activities to be invoked several times. Such a sub-workflow is then activated in a do-until-loop which realizes the cycle.

Finally, we want to return to our sample introduced in Sect. 1 to demonstrate the difference between the UDTF and workflow approaches by a more complex example including several of the introduced heterogeneity cases. The resulting I-UDTF for the enhanced SQL UDTF architecture solution is illustrated once again below:

```
CREATE FUNCTION BuySuppComp (SupplierNo INT, CompName VARCHAR)
RETURNS TABLE (Decision VARCHAR) LANGUAGE SQL RETURN
SELECT DP.Answer
FROM TABLE (GetQuality(BuySuppComp.SupplierNo)) AS GQ,
TABLE (GetReliability(BuySuppComp.SupplierNo)) AS GR,
TABLE (GetGrade(GQ.Qual, GR.Relia)) AS GG,
TABLE (GetCompNo(BuySuppComp.CompName)) AS GCN,
TABLE (DecidePurchase(GG.Grade, GCN.No)) AS DP
```

Comparing this function definition to the mapping graph resp. workflow process shown in Fig. 1, it is obvious that it is quite difficult to identify the relations and dependencies among the local functions in the `CREATE FUNCTION` statement. In contrast, the workflow solution in Fig. 1 is much clearer. Moreover, we would like to point out again that this solution is supported by the product we used, but cannot be taken for granted in general.

The following table summarizes our results:

Case	UDTF approach	WfMS approach
trivial	hidden behind the federated function's signature	hidden behind the federated function's signature
simple	cast functions, supply of constant parameters	helper functions
independent	join with selection	parallel execution of activities
dependent: linear	join with selection; execution order defined by input parameters[a]	sequential execution of activities
dependent: (1:n) and (n:1)	join with selection; execution order defined by input parameters[a]	parallel and sequential execution of activities
dependent: cyclic	not supported	loop construct with sub-workflow

[a] Not supported in general.

Obviously, the UDTF approach is able to support many of the cases we examined, going a long way towards a restricted but lightweight workflow technology. However, the WfMS supports still more functionality like conditions, that cannot be expressed by SQL. The examples show that it is easier to specify and implement the mappings by means of workflows. Moreover, with an increasing number of local functions involved, the SQL statements become more and more complex and confusing. Nevertheless, the UDTF approach can be beneficial for simple applications that do not need a full-flegded WfMS.

4 Performance

In the following, we will explore the performance of our reference architectures. For this reason, we have implemented the workflow as well as the enhanced SQL UDTF architecture and have built a test and measurement environment for several examples representing the different cases introduced in Sect. 3. The processing time, i. e. the elapsed time, for these examples has been measured for each solution and compared to each other. The implementation is based on IBM's DB2 UDB v7.1 and MQ Series Workflow v3.2.2.

Because of security restrictions in DB2 UDB, we had to modify our architecture slightly by introducing a so-called controller. This controller is needed to connect a UDTF to a database on the same server (which is the case in our test implementation). It ensures that the UDTF process and the connection to the database are two different processes. The same process isolation is implemented in the workflow architecture in order to separate the UDTF process from the process invoking the workflow. In addition, the controller is started only once at the beginning when the whole environment is booted. It calls the WfMS providing the connect information and keeps the WfMS active. If these tasks would not be performed by the controller, each single integration UDTF would have to repeat it each time it calls the WfMS. Hence, we optimize the WfMS access, since the execution time for a federated function is reduced by the time needed for

connecting the workflow engine. Since these modifications have the same impact on both solutions, we can likewise compare the alternative implementations.

First of all, we consider the processing time for function calls in three different situations: right after the entire system has been booted, after some other function has been invoked, and after the same function has been processed. Of course, the initial function calls are the slowest, since all underlying processes have to be started and memory as well as caches are empty. As expected, the repeated function call is the fastest. Please note, that the cyclic dependent case is not implemented and, therefore, not measured for the UDTF approach. Moreover, we note that the function `GetSuppQualRelia` based on parallel activities is processed faster than the function `GetSuppQual` with a sequential processing order in the workflow architecture. In contrast, the UDTF approach achieves processing times which show a contrary result. Obviously, the workflow approach can process parallel function calls in a more efficient way.

The following measurement results are based on repeated function calls. First, we directly compare the measured processing times to each other. Fig. 5 shows that the WfMS approach is up to three times slower than the UDTF solution. Furthermore, we can observe that the processing times do not rise as intensely for the UDTF approach as for the workflow approach when the number of functions increases. Taking the facts into account that the WfMS approach implies the use of a second, very big engine and that the time scale remains in the expected range, the gap between the processing times measured is acceptable.

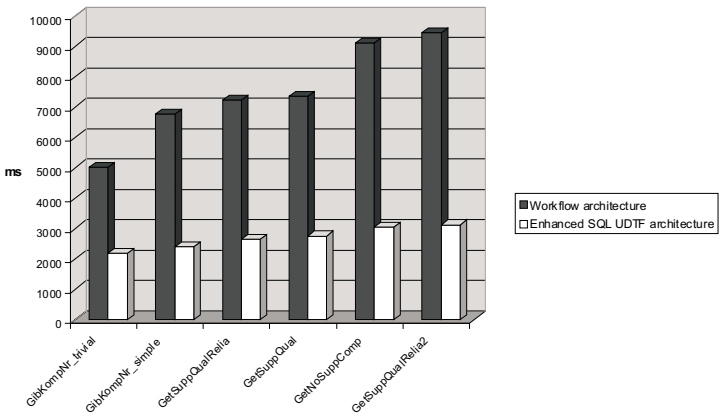

Fig. 5. Comparison of the results for the workflow and the enhanced UDTF approach.

Another interesting aspect is the question about the single actions the overall processing time consists of. Fig. 6 shows how much time the single steps consume when running the federated function `GetNoSuppComp`.

Workflow approach	
Step	Time
Start UDTF	9%
Process UDTF	11%
RMI call	3%
Start workflows and Java environment	10%
Process activities	51%
Workflow	9%
Controller	5%
RMI return	0%
Finish UDTF	2%

UDTF approach	
Step	Time
Start I-UDTF	11%
Prepare 3 A-UDTFs	28%
3 RMI calls	24%
3 controller runs	0%
Process activities	6%
Finish 3 A-UDTFs	21%
3 RMI returns	1%
Finish I-UDTF	9%

Fig. 6. Time portions of the overall function calls in the WfMS and UDTF approach.

Considering the workflow architecture we observe that the processing of activities takes the lion share of the time with 51% of the overall processing time. 23% of the time is needed for preparing actions like calling and processing the UDTF and the controller until the workflow system is involved. Another 19% is consumed by the workflow environment and the remaining time is needed for the controller (5%), the result return, and finishing the UDTF (2%). Starting the workflow process instances and the Java environment for the Java API of the WfMS seem to take up a lot of time with 10%. But this share will be smaller when the number of activities is increasing since it will always take the same constant time, irrespective of how many activities have to be executed. The introduction of the controller makes a difference with a total of 8%.

Looking at the UDTF architecture, the time is divided in other proportions. Starting and finishing the integration UDTF requires 20% of the overall time. This relative time portion is comparable to that of the workflow approach[4] with 22% for starting (9%), processing (11%), and finishing (2%) the integration UDTF. The overall processing time of the three access UDTFs consumes about 49% of the time. The processing of the local functions requires only 6%. Hence, the time portion corresponding to the activities in the workflow approach accounts for 55% (49% + 6%) and, thus, represents a bigger part of the overall elapsed time than the activities in the workflow. The controller consumes a rather big percentage of the time with a total of 25%.

Comparing the two time portions, we observe the extreme difference regarding the various process activities. This is mainly due to the start of the Java programs. In the UDTF solution, the activities are processed within the controller which is already running, whereas the workflow architecture requires the start of a new Java program for each single activity including the booting of the Java virtual machine. Moreover, the workflow activities have the additional task of handling the input and output containers.

[4] However, the ratio for the absolute elapsed time is still 1:3.

Assume, we can implement our prototypes without the controller. Then, the total time of the WfMS solution would decrease by 8%, whereas the UDTF solution would decrease by even 25%. As a result, the overall processing time ratio between workflow and UDTF approach would increase from 3 to 3.7.

Moreover, we have used the cyclic dependent case in the workflow solution to get an impression how the number of functions influences the overall processing time. The federated function `AllCompNames` is realized by means of a do-until loop. We have used this loop structure to measure the impact of an increasing number of calls of the same local function within a single federated function. This scenario shows that the overall processing time rises linearly to the number of function calls. Of course, this is only true, if we always call the identical function. Several measurements have shown that it is pretty difficult to define a formula for the overall time based on the number of functions integrated, since the elapsed time to execute a local function can differ immensely. So we can state that, currently, the parallel execution of functions is the only action we can make an assumption about: activities that are defined as parallel activities in a control flow actually are processed in parallel and, therefore, reduce the time consumed. Our examples `GetSuppQualRelia` and `GetSuppQual` show that the UDTF approach is unable to exploit the advantage of parallelism to such an extent since the execution of the parallel case takes more time than that of the sequential case.

In summary, our evaluated scenario indicates that a workflow system in the middleware does not produce such an overhead that the resulting processing times are not acceptable anymore.

5 Related Work

Most approaches dealing with the integration of heterogeneous data sources focus on the capability to combine different data models and heterogeneous systems providing an interface which is not as powerful as SQL. Approaches like Garlic [7], Information Manifold [8], or TSIMMIS [9] 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 accessing any kind of data source. In our case, all non-SQL sources are integrated by the WfMS (or the use of UDTFs) and the SQL sources are managed by the FDBS. As a consequence, the FDBS has to communicate with a single non-SQL source only: the WfMS. Hence, we can concentrate on a specific solution supporting the interoperability between the FDBS and the workflow system (on the assumption that the access to heterogeneous SQL sources is already provided by the FDBS).

Furthermore, we focus on the integration of a functional interface on the FDBS side. Chaudhuri at al. [10] 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 spe-

cific selection criterias in the WHERE clause of an SQL statement. Approaches like [11] and [12] propose solutions for this limitation by binding attributes in order to support queries on such data sources.

The approaches mentioned above mainly obtain their solutions by fully-fledged implementations whereas our intention is to use existing technologies and products. Reinwald et al. [13] present a solution based on user-defined table functions to access non-SQL data sources. But these table functions are limited to Windows sources, since they implement them by means of OLE DB.

6 Summary

In this paper, we have introduced an approach for the integration of heterogeneous data sources accessible via a generic query language or predefined functions. We have described the components of our integration architecture introducing the FDBS and the WfMS connected via standardized SQL/MED wrappers. Since there is no product available yet supporting such wrappers, we have decided to implement a first prototype with so-called user-defined table functions. These table functions support read access to any kind of data source returning the result as tables and, therefore, can be referenced in the FROM clause of a select statement. Since the workflow engine may be considered as a component too big for the middleware, we have described several alternative architectures which realize the function integration via direct access to the local functions by means of table functions.

In order to assess the function mapping capabilities, we have compared the workflow architecture and a table function architecture regarding their mapping complexity. The comparison has shown that the workflow approach is able to realize all possible scenarios whereas the table function approach has some limitations. Moreover, we have implemented two prototypes and have measured the performance for each solution. With a workload of increasing mapping complexity, the measurement results revealed a factor of 3 in favor of the UDTF solution. However, when the overall effort for implementing a foreign function integration is considered, the workflow approach represents a solution, easy to implement and to use, which is not as slow as many readers may expect.

Not all questions could be answered concerning the suitability of an architecture for data and function integration where the data sources are autonomous, distributed, and heterogeneous. To the best of our knowledge, empirical studies on their related issues are missing so far. Therefore, an initial study cannot be exhaustive. Further questions to be considered include, for instance, the mapping between the data model and the functional model and how it can be realized absolutely transparently to the application. Moreover, the functionality provided by the wrapper is interesting including the discussion of wrapper-internal operations for requested functionality not natively supported by the WfMS [14]. Further research has to clarify issues of query optimization, scalability, access control, ease of administration and evolution, and so on.

References

1. Härder, T., Sauter, G., Thomas, J.: The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution. VLDB Journal 8:1 (1999) 25–43
2. Sheth, A.P., Larson, J.A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. ACM Computing Surveys 22:3 (1990) 183–236
3. SAP AG: SAP R/3. (2001). `www.sap.com/solutions/r3/`
4. Hergula, K., Härder, T.: A Middleware Approach for Combining Heterogeneous Data Sources – Integration of Generic Queries and Predefined Function Access. Proc. 1st Int. Conf. on Web Information Systems Engineering, Hongkong (2000) 22–29
5. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques, Prentice Hall (2000)
6. ISO & ANSI: Database Languages – SQL – Part 9: Management of External Data, Working Draft (2000)
7. M.Tork Roth, P. Schwarz: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. Proc. 23rd Int. Conf. on Very Large Data Bases, Athens (1997) 266–275
8. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying Heterogeneous Information Sources Using Source Descriptions. Proc. 22nd Int. Conf. on Very Large Data Bases, Bombay (1996) 251–262
9. Papakonstantinou, Y., Garcia-Molina, H., Widom, J.: Object Exchange Across Heterogeneous Information Sources. Proc. 11th Int. Conf. on Data Engineering, Taipei (1995) 251–260
10. Chaudhuri, S., Shim, K.: Query Optimization in the Presence of Foreign Functions. Proc. 19th Int. Conf. on Very Large Data Bases, Dublin (1993) 529–542
11. Florescu, D., Levy, A., Manolescu, I., Suciu, D.: Query Optimization in the Presence of Limited Access Patterns. Proc. ACM SIGMOD Int. Conf. on Management of Data, Philadelphia (1999) 311–322
12. Garcia-Molina, H., Labio, W., Yerneni, R.: Capability-Sensitive Query Processing on Internet Sources. Proc. 15th Int. Conf. on Data Engineering, Sidney (1999) 50–59
13. Reinwald, B., Pirahesh, H., Krishnamoorthy, G., Lapis, G., Tran, B., Vora, S.: Heterogeneous Query Processing through SQL Table Functions. Proc. 15th Int. Conf. on Data Engineering, Sidney (1999) 366–373
14. Hergula, K., Härder, T.: How Foreign Function Integration Conquers Heterogeneous Query Processing. Proc. 10th Int. Conf. on Information and Knowledge Management, Atlanta (2001) 215–222