Contents

Chapter 1

Introduction

Over years all sorts of applications are developed, used and rooted in an enterprise. Each application manages its own data, processes data on its own initiative. Workflow management system (WFMS) was introduced to organise applications involved in the same logical process so that a single function call at the beginning of a process will trigger a chain of application execution, which returns the wished data at the end of this process (see Figure 1.1).

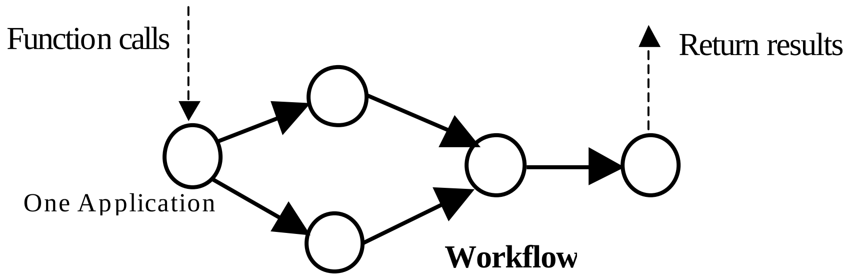

Figure 1.1: Workflow to chain different applications in one logical process

Often the results returned from a workflow is desired to be further processed or stored in a database. Thus the federated database concept is introduced to integrate workflow into a database federation even though workflow output is not aware of any SQL.

Multiple approaches can be applied to realise integration of WFMS to federated database, the wrapper approach and user defined function being two of them [HH02]. The aim of both approaches is to achieve universal access to heterogeneous data sources. The scenarios of both approaches are shown side by side in Figure 1.2.

While user defined function (UDF) approach offers a compact solution of integrating workflow by invoking a single function registered in the Federated Database System (FDBS) to execute workflow, it has its structural limitations.

In this thesis, the integration is tried with help of the wrapper technology standardized by SQL Part 9 Management of External Data (SQL/MED).

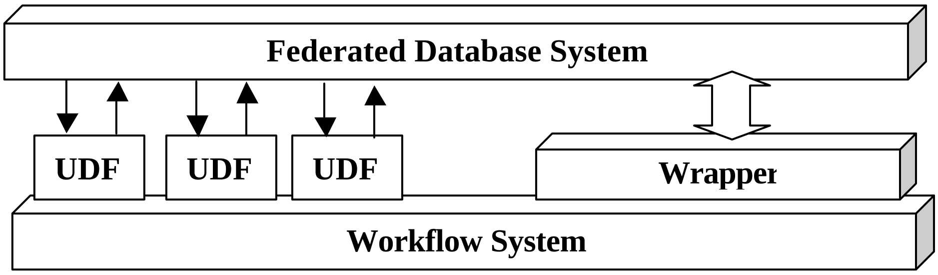

Figure 1.2: Two approaches of integrating workflow

Since integration is done in an object oriented programming environment, SQL/MED will be projected in an object oriented perspective. Wrapper can in some way be seen as the data adaptor of a foreign data source to the federated database system. How data is channeled into workflow process as parameter by wrapper and how the results are collected and converted by wrapper before delivering to federated database system will be one of the most important part of wrapper mechanism.

After the workflow wrapper is implemented, the performance issue arises as it is asked how the structural and conceptual differences between UDF and wrapper will affect them in efficiency as integration tools.

A test based on a simple workflow process is made and its test results will be analysed. An opinion based on the analysis will try to be made.

At the end, suggestions on a better performance of integrating workflow system to federated database management system will be made based on observations on time distribution of Workflow Table Abstraction (WTA) component within wrapper.

Chapter 2

Concepts of Federated Database System

Within an organisation a great varieties of software are used to shore up the information flow and processing, among them databases and their management systems.

A database, according to C.J. Date, is a collection of persistent data used by the application systems of some given enterprise. A database management system (DBMS) is the software that handles all access to the database. A database systems (DBS) is a computerized record keeping system that comprises data, hardware, (usually one) DBMS and users [DA00].

Thus data are managed or accessed exclusively through their DBS by applications along their management system boundaries, though much more powerful applications can be created when these historically segregated data can be integrated.

Over the years various commercial software companies or institutions developed different database systems that base on different standards, employ dialects of query languages and run on numerous platforms, not to mention versions of a same database over years of development. (see Figure 2.1)

Factors, under which an organisation is subject to, like competition, evolving technology, mergers, acquisitions, geographic distribution, and the inevitable decentralization of growth all contribute to this diversity.

However it is only by combining the information from these systems that the enterprise can realize the full value of the data they contain [HL02].

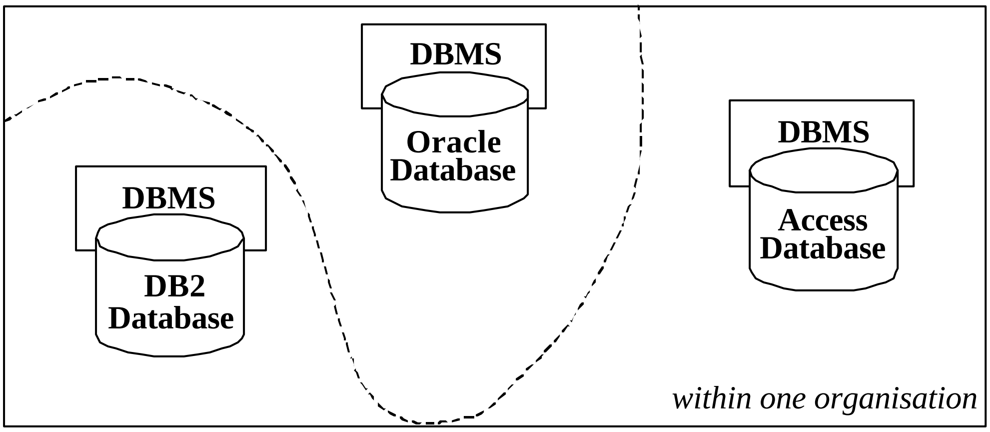

Figure 2.1: A possible engulfment of data along the various management systems

One possibility of integrating two different database, e.g., an Oracle and a DB2, is through so-called **gateway**. A gateway is a 'special' program that simulates access from one database to another by coding protocols of interactions, by mapping query dialects and data types, by maintaining catalog of the target and so on. For each database there has to be a tailor made gateway, e.g., DRDA-Gateway in forms of a DLL-file for accessing DB2 family databases.

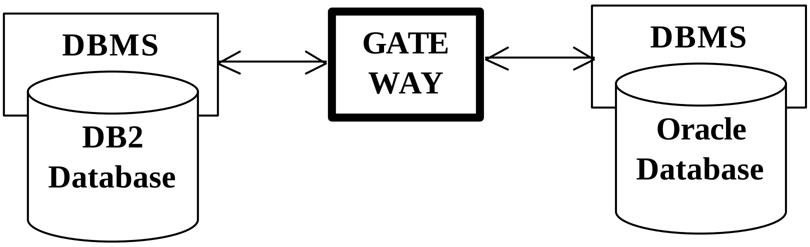

Figure 2.2: Gateway for integration of two different databases

However such a point-to-point Gateway paradigm has its limitations, for one each sort of database a specific gateway is needed, which implies a huge amount of time both for design of gateways and data accessing through gateways. A distinctive application needs to call gateways of each involved database. Further data transparency is difficult to achieve since the user has to have reasonable knowledge over the database at the other end of the gateway before its data can be accessed. An alternative approach[1] would be "**middleware**" architecture[2].

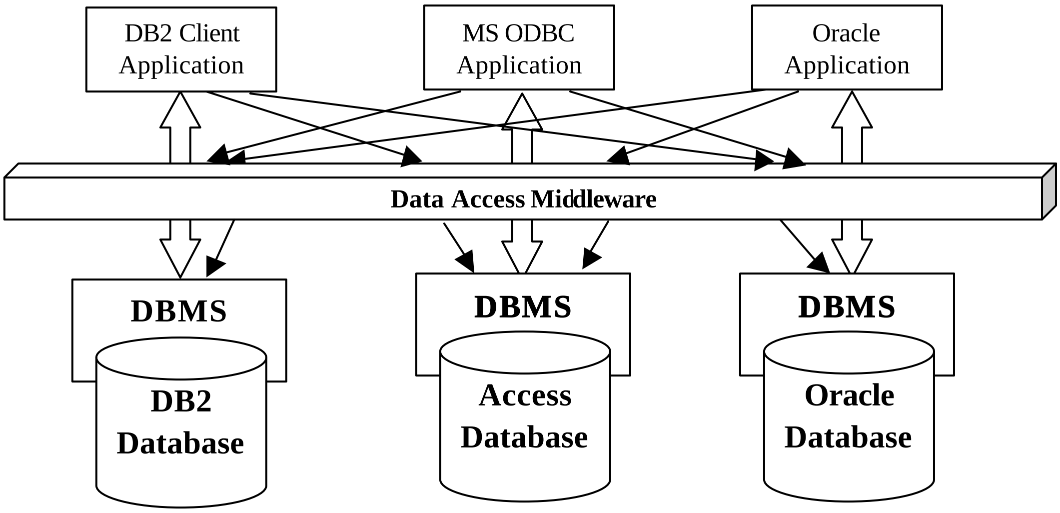

Figure 2.3: Data access middleware architecture

In a middleware-involved architecture, while the database applications access their native database systems directly (indicated by broad line arrows in Figure 2.3), they rely on a uniform access mechanism provided by the middleware layer to perform querying on foreign databases (indicated by line arrows in Figure 2.3).

The aim of the database middleware is to provide a transparent and uniform view of multiple data sources and maintain this interface for database applications in case a new data source becomes available.

An integration of multiple database through the middleware approach brings about the concept of **Federated Database System** (FDBS).

---

[1] Gateway is classified by some authors as a simple point-to-point middleware solution with only two dissimilar database systems involved. Here middleware architecture is meant to involve multiple applications and databases.

[2] Using terminology in [DA00] here the middleware is referred as *data access middleware*, which deals only with database issues. However middleware as a concept is widely used in other fields like networking.

2.1 Characteristics of a federated databases system

The term FDBS emerged at the beginning of 1990's as integration of access to multiple dissimilar DBSs gained its significance.

A set of multiple DBSs can be characterized in three orthogonal dimensions: distribution, heterogeneity and autonomy [SL90] (see Figure 2.4).

Distribution refers to how a single DBS physically store its data. Network technologies nowadays allow DBS to partition and/or replicate its data and maintain them on different nodes and/or in different databases without compromising their accessibility.

Heterogeneity of DBS often occurs due to the fact that independently developed database systems always yields different solutions.

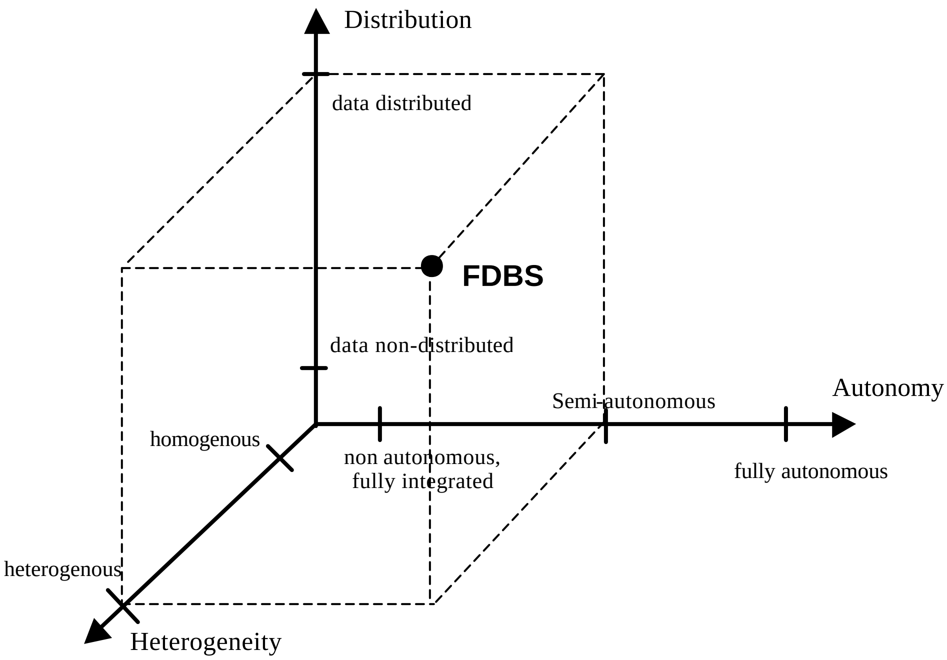

Figure 2.4: FDBS characterized in dimensions of distribution, heterogeneity and autonomy

The heterogeneity can be classified in many different ways. One of them is to distinguish syntactical, data model and logical heterogeneity [BU99]. Different hardware platforms, operating systems, communication methods and access languages constitutes the syntactical aspects of heterogeneity. Data models, e.g. relational vs. object-oriented, can be another source of heterogeneity. Logical heterogeneity comprises of schema- and semantic-specifics of DBS. One example of the logical heterogeneity is various representations and interpretations of data, e.g., an attribute may have the same format but indicates values of different units (US$ vs. Euro).

Autonomy character of a DBS reflects how much independent control a concerning DBS has over its own data in terms of design, communication and execution. Autonomy of a DBS X can be defined as such that X is capable of carrying out all its operations without dependencies on any other DBS. A fully autonomous DBS runs on its own and is not aware of fellow DBSs, at the other end of the spectrum it is a fully integrated DBS, which renders its control to a higher hierarchy, say the management of a DBS federation. An autonomous DBS yielding partial control of its data is said to be a semi-autonomous DBS.

A federated database system is a collection of distributed, heterogeneous and (semi-)autonomous DBSs integrated through a federation layer. In Figure 2.4 FDBS is put into a perspective of three orthogonal characters [ÖV99]. The degree, to which DBSs are autonomous or integrated into a FDBS may vary, yet DBSs participating in a FDBS are always scaled as heterogeneous and distributed within this FDBS. These participating DBSs are called **Component DBSs** (CDBS).

2.2 Classification of federated database systems

Based on each of the three dimensions mentioned above, a set of multiple DBSs (MDBS) can be classified into different types.

Focusing on the distribution side of taxonomy there are types of *centralised* **multi-database system** (MDBS) and *distributed* MDBS.

Applying the criteria of heterogeneity there are also two types of MDBS. An MDBS is called a *homogeneous* MDBS if all its member DBSs are the same type. Otherwise it is a *heterogeneous* MDBS.

With the autonomy of CDBSs in mind and thus data sharing in the foreground, an MDBS can be classified as either *federated* or *non-federated* MDBS. (see Figure 2.5) A non-federated MDBS has only non-autonomous DBS and is sometimes referred as a unified MDBS.

A federated database system (FDBS) consists of component DBSs that are autonomous yet participate in a federation to allow partial and controlled sharing of data [SL90], i.e., CDBSs are semi-autonomous. Thus a FDBS is a distributed, heterogeneous and semi-autonomous MDBS.

The FDBSs can be further divided into two classes, *loosely coupled* and *tightly coupled*, depending on who bears the responsibility of maintaining the federation.

In the case of tightly coupled FDBS the administrator of such a FDBS creates and maintains the federation and actively control the access to CDBSs [SL90]. Users of a tight FDBS are offered an universal access interface, not aware of the individual access schema of CDBSs. The higher transparency of data access is achieved at the cost of CDBS autonomy.

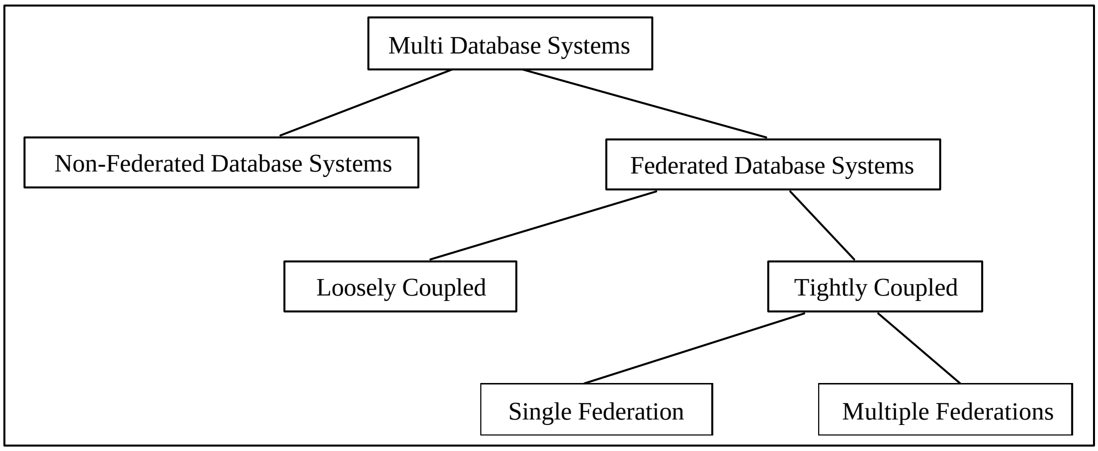

Figure 2.5: Classification of Multi-Database Systems

Query statements issued to a tight FDBS will go through a query planning phase, in which FDBS analyses the segments of the query statement against the actual capacities of individual CDBSs, and determines whether FDBS need to compensate for the missing capability of a certain CDBS by processing the query results from this CDBS.

In absence of a uniform (or federated) access schema a FDBS is called a loose federation. The federation holds up on a uniform query language (*multi-database query language*) which abstracts from CDBSs and hide technical and dialect heterogeneity [BU99]. Users define integrated views on CDBS data and make available to other users as if such integrated views were global relations.

A loosely coupled FDBS may always support multiple federated schema, upon which query operations are performed. Tightly coupled FDBSs may have one or more federated schema, those with one federated schema are said to have *single federation* while those FDBSs allowing creation and management of multiple federated schemas are said to have multiple federations.

2.3 Three layer architecture of FDBSs

Figure 2.6 shows a general three-tier architecture of an FDBS.

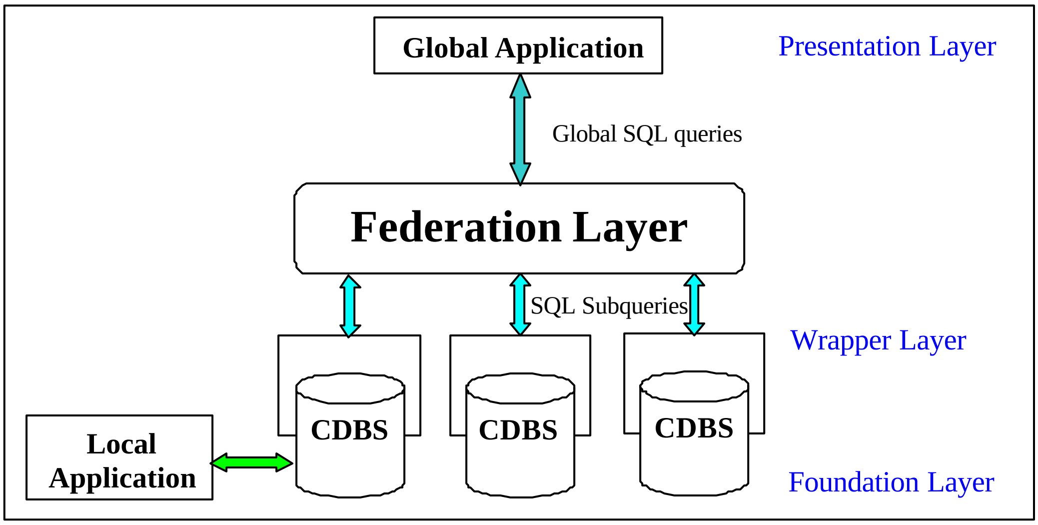

Figure 2.6: Three layer architecture of a FDBS

The foundation layer consists of (semi-)autonomous database systems (CDBSs). The CDBSs manage their data in their own rights, processing requests of local applications (issued by local users) and carrying out due operations. Global applications, which are accessible to FDBS-clients, send data requests in form of global SQL queries to the federation layer. The federation layer, also called **federated database management system** (FDBMS), acts as data access middleware. As a result the global SQL query will be broken down into sub-queries (segments of a query statement) and sent to the corresponding CDBS. The CDBSs respond by sending back query results to the FDBMS. The FDBMS assembles all the results into an integrated schema and returns it to the global application. Between the federation layer and the foundation layer, i.e., between the FDBMS and CDBSs, there is a thin layer for so called wrappers. They are adapters of individual DBS to the federation schema. The thesis will come to the concept of wrappers later.

2.4 DB2 UDB as FDBS

IBM defines a FDBS as a database management system (DBMS) that supports *distributed requests*, which means applications and users may submit SQL statements referencing two or more DBMSs or databases in a single statement [IBM00].

DB2 UDB (Version 7) as a FDBS not only supports distributed requests, but goes beyond in several aspects. DB2 UDB supports Oracle, Sybase, Microsoft SQL Server along IBM DB2 family DBMS as standard CDBSs. DB2 UDB provides clients access to its CDBSs with *location transparency*, i.e., queries are independent of data locations, UDB makes sure of accessibility of the data queried once they are moved physically. UDB can provide *compensation* to the capability of its CDBS by doing “refinery” work on the query results returned by a CDBS to meet demands of UDB’s own client.

Shown in the Figure 2.7 DB2 UDB plays the role of DBMS for the FDBS (FDBMS as parallel to DBMS), which supervises various data sources. To process distributed requests coming from various DB2 clients, UDB has to maintain information about catalog entries identifying data sources and their characteristics. The major hierarchies of the catalog are the objects of wrapper, server and nickname.

Wrapper objects represent the module to access a group of data source. Server objects hold the information of data sources for the UDB like server names, types, log-in information etc. Nickname objects are pseudo-tables with references to the physical data in the CDBS.

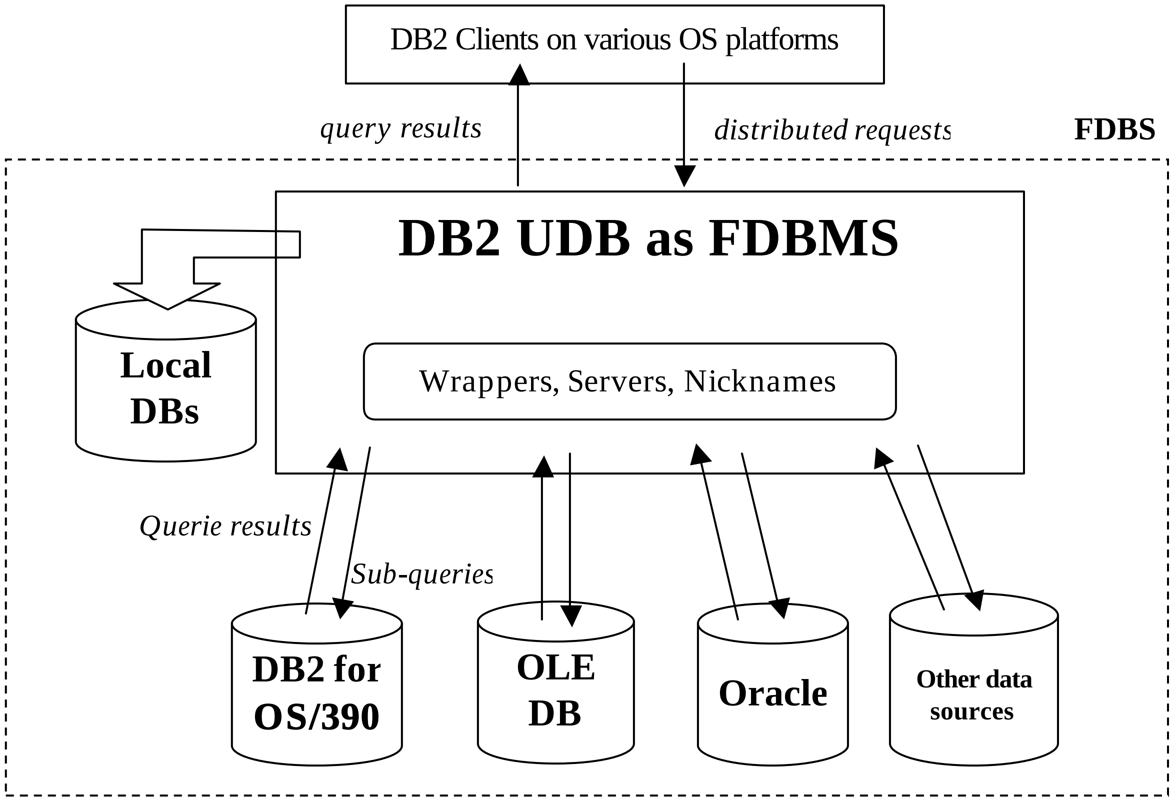

Figure 2.7: DB2 UDB as FDB Management System

DB2 UDB registers information of a specific CDBS (here an OLE DB) in its catalog, the most important catalog entries are e.g. for the OLE-tailor-made wrapper's name, file path, the name, type, version and capacity of the OLE server, the name and column structures of a nickname representing a view or table in the OLE database.

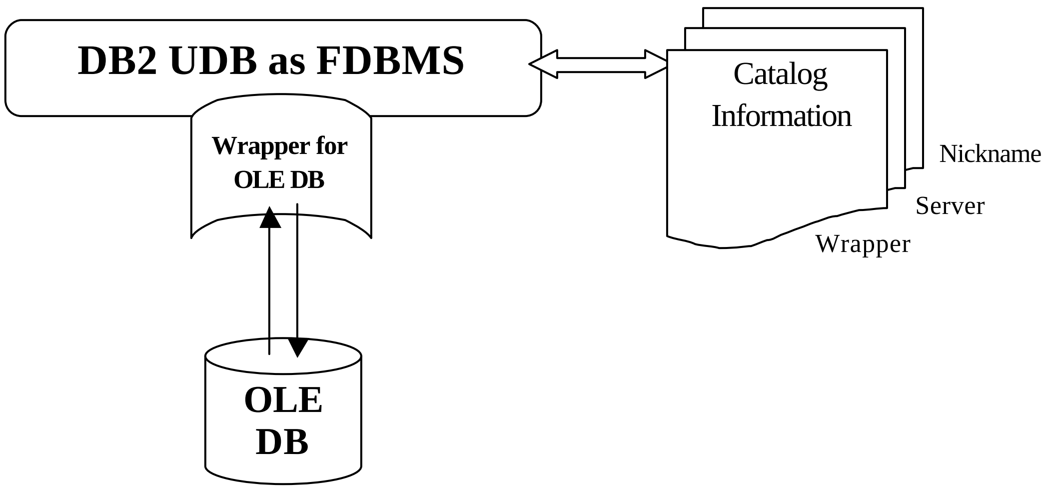

Figure 2.8: Catalog Information of a CDBS maintained in DB2 UDB

As to execute a distributed request (i.e., a query that involves more than one data source) from a DB2 client, UDB check which “nicknames” are set to be queried, then determines through catalog information which CDBS contains the physical data. Supposedly here data in the OLE DB and another CDBS are queried (see Figure 2.8). UDB breaks the request into sub-queries (query segments). The segment that is meant for OLE DB will be analysed by UDB, determining whether OLE DB could handle it and fulfil the requirement on query returns. This is done with help of catalog entries on OLE DB server, which contains the capacity of it.

Since relational queries are non-procedural and there are typically several different implementations of each relational operator and many possible orderings of operators to choose from in executing a query. UDB considers the various possible strategies, modelling the likely cost of each, and choosing the one with the least cost. Typically, cost is measured in terms of system resources consumed. [HL02] This phase is called optimisation of query operations.

Once the planning is finished, UDB reads necessary catalog information on how the OLE DB server can be reached, and how the query segment must be transformed to conform to the dialect of Microsoft SQL. Login data saved in the catalog will be read and used for a connect between UDB and OLE DB server.

Wrapper information came into play as UDB determines with which wrapper module it can access the OLE DB server, establish connection, feed the query segment to OLE DB server, collect query results from it, transform the results into formats acceptable to UDB.

As UDB complete this process, it can integrate these results with those from the other query segment and return in its totality to the user. The user get the query results without having to know details about both CDBSs involved except the nicknames and the data they represent.

A DB2/UDB-based FDBS shall be classified as a tightly coupled single federation FDBS because UDB maintains the federated schema through objects like wrapper, server and nickname.

Worth to mention is that IBM provides support for users that need to write his own wrapper that integrates a data source not included in the standard CDBSs mentioned above. With this wrapper development kit a wrapper is programmed to integrate IBM MQSeries Workflow into the FDBS schema, which is practical basis of this thesis.

With tailor-made wrappers as interface of FDBS, DB2 UDB is in the position of not only integrating foreign database systems, but non-database data sources as long as the source is managed through some mechanism, e.g., a structured file. In this sense DB2 UDB plays a role of what was called as Federated Information System (FIS) in [BU99].

Chapter 3

SQL/MED and Wrappers

Management of External Data, abbreviated as MED, prescribes the way to extend Database Language SQL to support management of external data through the use of foreign-data wrappers and data type *datalink* [ISO02].

SQL/MED addresses two problems of accessing data external to a given database system (thus a given SQL environment) [MM01]. The first problem is how to access through the SQL interface non-native-SQL data. Non-native-SQL data no only includes data existed in databases that use other SQL dialects, but those data accessible through the native SQL-data, yet residing on a different database system as well. SQL/MED does not prescribe implementation details to solve the outstanding problem. Instead the interactions between the concerning SQL-server and foreign-data wrappers are specified.

The second problem concerns with management of data residing in non-database system, namely in file systems. SQL/MED introduced a new data type called DATALINK to meet the need of SQL-server to access those files or data in files.

SQL/MED has set a standard for heterogeneous federated database management [EM00]. The goal of SQL/MED is to specify an open interface, with which integration of foreign data sources to an existing SQL-server can be made by either SQL-server-vendor or users of SQL-server.

This thesis will concentrate on the approach of SQL/MED to address the first problem. IBM's DB2 UDB supports SQL/MED standards in DB2 runtime environment and UDB-compatible wrappers.

3.1 Concepts in SQL/MED in FDBS context

Following important concepts are defined in SQL/MED as the first step.

External Data are those not managed by an SQL-server but accessible through a **Foreign Server**, which manages these external data. Information on foreign server will be maintained by SQL-server in **Foreign Server Descriptor**, which includes foreign server name, name of the wrapper designated for the foreign server, optionally type and version of foreign server.

Within the framework of FDBS (see Figure 3.1) FDBMS plays the role of SQL-server while a set of external data and its server represents a participating (semi-)autonomous database, in other words a CDBS.

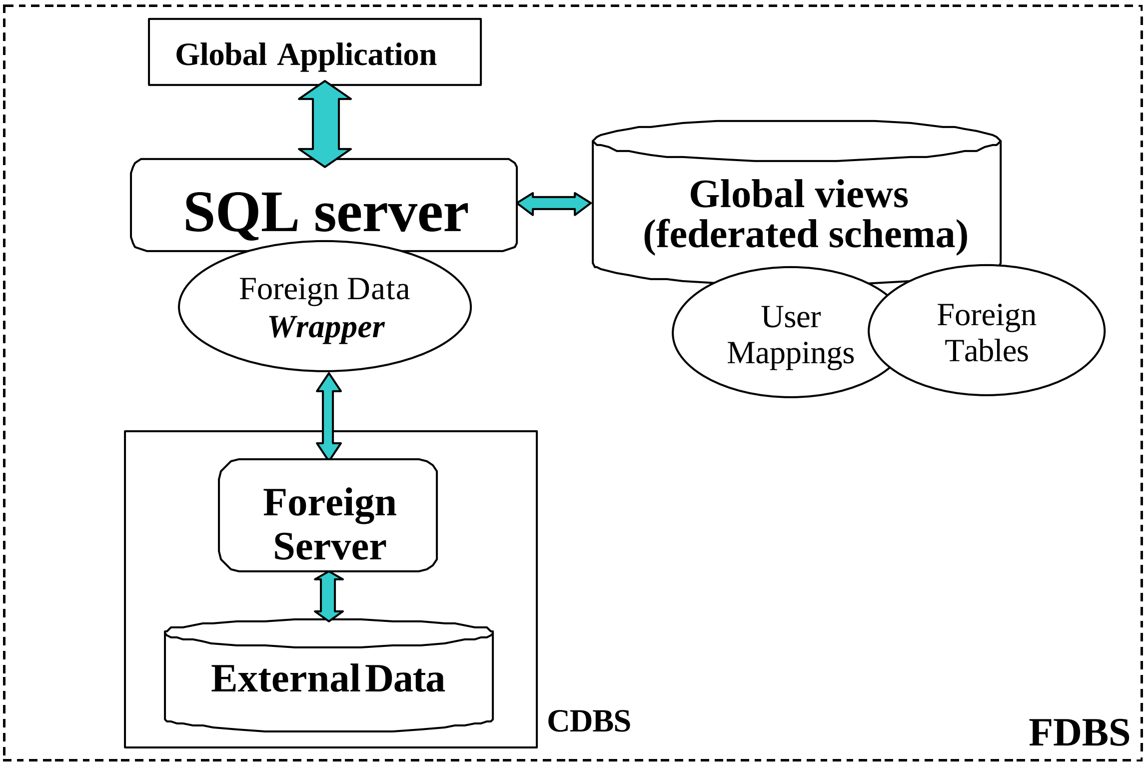

Figure 3.1: SQL/MED architecture in a FDBS context

The external data is represented by **Foreign Tables** and organised in global views, two components managed by SQL-server. SQL-server gains its access to the global views with its SQL-syntax. Foreign tables are virtual tables that carry meta-information of the external data. Analogical structure can be found in FDBS framework too.

To complete the information of external data of various sources there are also **User mappings** needed. They save the authorization data, i.e., user names and passwords, of external data sources. In case that foreign server offers API for execution of its built-functions, SQL-server will maintain information on how to invoke these functions in **Routine mappings** information.

With appropriate information of foreign tables and user mappings FDBMS has sufficient logical information on queries put forward by a global application. It is then the job of FDBMS to translate a query of FDBS-specific syntax into a CDBS-specific query (in certain cases it involves breaking-down of a query into multiple query fragments for multiple CDBSs) and deliver to the relevant CDBS.

The communication between SQL-server and foreign server is delegated to Foreign Data Wrappers[3]. The wrapper mechanism is the most important component of SQL/MED. It is made of a set of routines programmed to access specific foreign servers. SQL server saves information like wrapper name, library file invoked by wrapper to access foreign server, name of the language used to write the library file, etc in Wrapper Descriptor.

Wrappers is the focal point of SQL/MED, for they encapsulate underlying foreign data sources and mediate between foreign data sources and SQL-server (in FDBS structure they are CDBS and FDBMS).

Generic Options for objects like wrapper, foreign server, foreign table and user mapping are kept in Generic Option Descriptors, which are part of the information on the corresponding objects. These options help to customize these objects with minimal efforts. For example foreign server descriptor might keep a generic option “pushdown” to indicate whether to entrust foreign server with evaluation operations like ‘join’ or only assign ‘select’ operations to it.

[3] In this work foreign data wrapper will be referred as wrapper.

3.2 Interactions of MED components

3.2.1 New registration of an external data source (CDBS) to the FDBMS

First of all the administrator of a CDBS teams up with the administrator (FDBA) of the FDBS to complete registration of the newly-coming CDBS into the catalog of FDBMS. The FDBA designs and implements a wrapper (or adapting an existing one) for the CDBS. FDBA then informs FDBMS the existence of such a foreign data wrapper by executing:

```
CREATE FOREIGN DATA WRAPPER <wrapper-name>
[ AUTHORIZATION <authorization identifier> ]
[ LIBRARY <library-name> ]
LANGUAGE <language-name>
[ generic-options ]
```

Figure 3.2a: Create foreign data wrapper in MED

Besides the name of the wrapper concerned, there are some additional information to be included in the SQL-statement. AUTHORIZATION specifies the owner of the wrapper, while LIBRARY identifies the location of the library for wrapper routines.

Once the wrapper is in place, the reference to the foreign server can be created in the FDBMS. MED provides the following SQL-command for the purpose.

```
CREATE SERVER <server-name>
[ TYPE <server type> ]
[ VERSION <server version> ]
[ AUTHORIZATION <authorization identifier> ]
FOREIGN DATA WRAPPER <wrapper name>
[ generic-options ]
```

Figure 3.2b: Create foreign server in MED

The most important part of this command is to name the foreign data wrapper that is associated to the foreign server. Once the association is established, the wrapper will be initialised every time the foreign server is sought after by FDBMS.

Till this point FDBMS has yet to have any logical data of its own. The next step is to integrate external data on the foreign server into FDBMS. Since SQL is based on the relational data model, MED requires any external data to be represented in a relational model in FDBMS too.
In other words external data are to be organised into foreign tables. However foreign tables are virtual ones. The physical data remain under the jurisdiction of their respective foreign server.

Foreign tables describe the columns and their data types in the SQL-statement. Within the statement the foreign server, which hosts the logical data described in a foreign table, must also be named.

```
CREATE FOREIGN TABLE <table-name>
[ {Column definition} ]
SERVER <foreign server name>
[ generic-options ]
```

Figure 3.2c: Create foreign tables in MED

Should CDBS offer more access than conventional interfaces and its corresponding wrapper is capable of taking advantage of it, then FDBMS has the option of importing multiple tables at once. The SQL-command is the following:

```
IMPORT FOREIGN SCHEMA <schema-name>
[ LIMIT TO {foreign tables list}
  | EXCEPT {foreign tables list} ]
FROM SERVER <foreign server name>
INTO <local schema name>
```

Figure 3.2d: Import schema from foreign server in MED

Now the external data is transparent to global applications of FDBMS. The clients of FDBMS see the foreign tables as if they are local tables of a DBS. When the clients query data of foreign tables to FDBMS, the later finds out of its catalogue which foreign server hosts the physical data of a foreign table. Further the foreign data wrapper associated to the involved foreign server is picked up by FDBMS to coordinate interactions between FDBS and CDBS.

Wrappers needs to log in to foreign servers in most of the cases to access their data. If necessary such information shall be provided to wrappers through SQL statement as the following:

```
CREATE USER MAPPING FOR <user name>
SERVER <server name>
[ generic-options ]
```

Figure 3.2e: Create user mapping in MED

Generic options that appear at the end of the SQL-Statements above save information about foreign data wrapper, foreign server, foreign table or user mapping respectively. Each generic option consists of an option name and an option value. The options and the range of their values must be predefined by the wrapper so that as FDBMS prepares SQL-statement, the options can be accepted and used for queries.

Registration of an external data source is somewhat like setting a logical copy of the source in FDBMS. Once the process is completed, FDBMS view the external data through a “mirror” of wrapper as its own (see Figure 3.3).

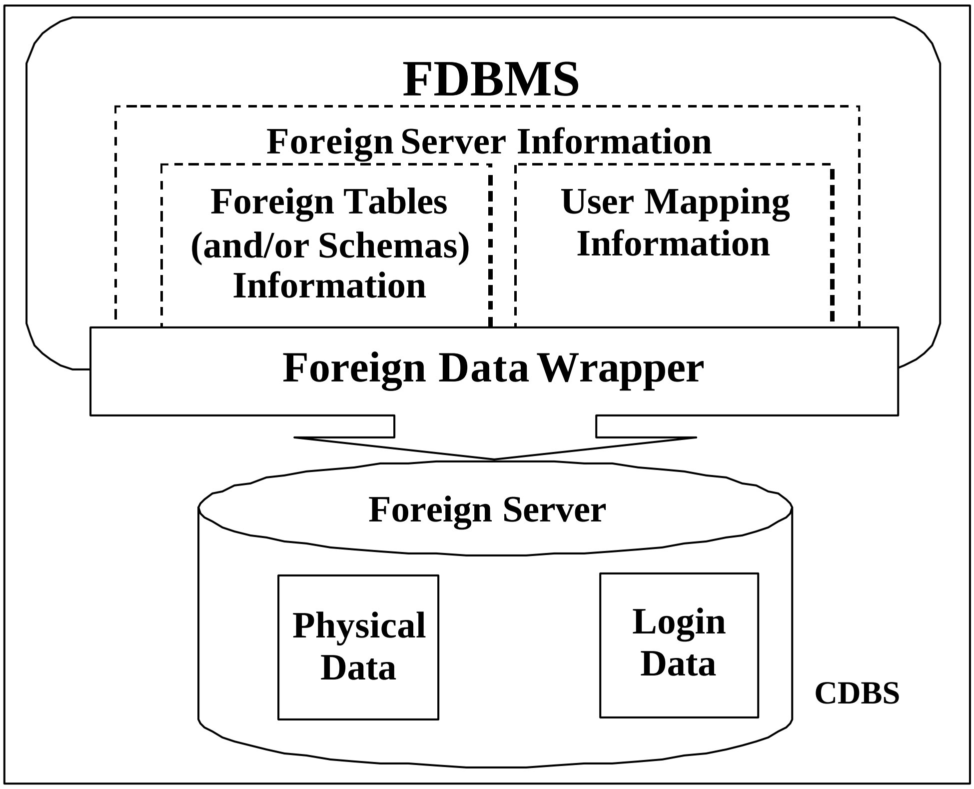

Figure 3.3: Setting an “logical copy” of CDBS data in FDBMS

3.2.2 Querying registered external data source

A global application (FDBS-client) submits an SQL statement in FDBMS-syntax (Arrow "1" of Figure 3.4). The query demands a customized view from foreign tables registered in FDBMS.

FDBMS analyses the SQL-statement and decides in which of the two modes it will issue the query requests to CDBS – **pass-through** or **decomposition**.

If all the columns of the SQL-statement deal with one single CDBS, FDBMS will go for pass-through mode. By choosing pass-through option FDBMS forwards the entire SQL-statement to a relevant wrapper and leave all the decisions and operation to that wrapper and its associated foreign server and wait for the results from wrapper's execution.

More often and complex comes the mode of decomposing a global SQL-statement. FDBMS check up in which foreign tables the columns in SQL-statement reside. Then it looks into the catalog entries of foreign tables to find out the responsible CDBSs, breaks up the SQL-statement into **query segments**, each for one CDBS. (Arrow 2)

While processing each query segment FDBMS initialises the wrapper that is associated with the concerning CDBS (Arrow "3").

After initialisation FDBMS will engage the wrapper in an exchange of information in order to achieve the optimal execution of the query segment. SQL/MED uses the concept of "handles" to facilitate the communication between FDBMS and wrappers. The idea is that FDBMS passes integer parameters while invoking wrappers routines indicating its requirements on query results. Wrappers are pre-programmed to understand the parameters and strive to meet the requests. These handles will be discussed in greater details in the next chapter when it comes down to standard interfaces between FDBMS, wrappers and foreign servers.

The wrapper analyses the segment and reports to FDBMS on how much its foreign server can handle from the assignment. In case that a foreign server is unable to execute the query segment in its totality, extra efforts must be orchestrated by FDBMS to compensate for the gap. For example, a certain foreign server can't perform a complicated query segment as its wrapper notices, FDBMS has to improvise a tactic of querying the foreign server with multiple simple statements and have the results being re-filtered to meet the original request.

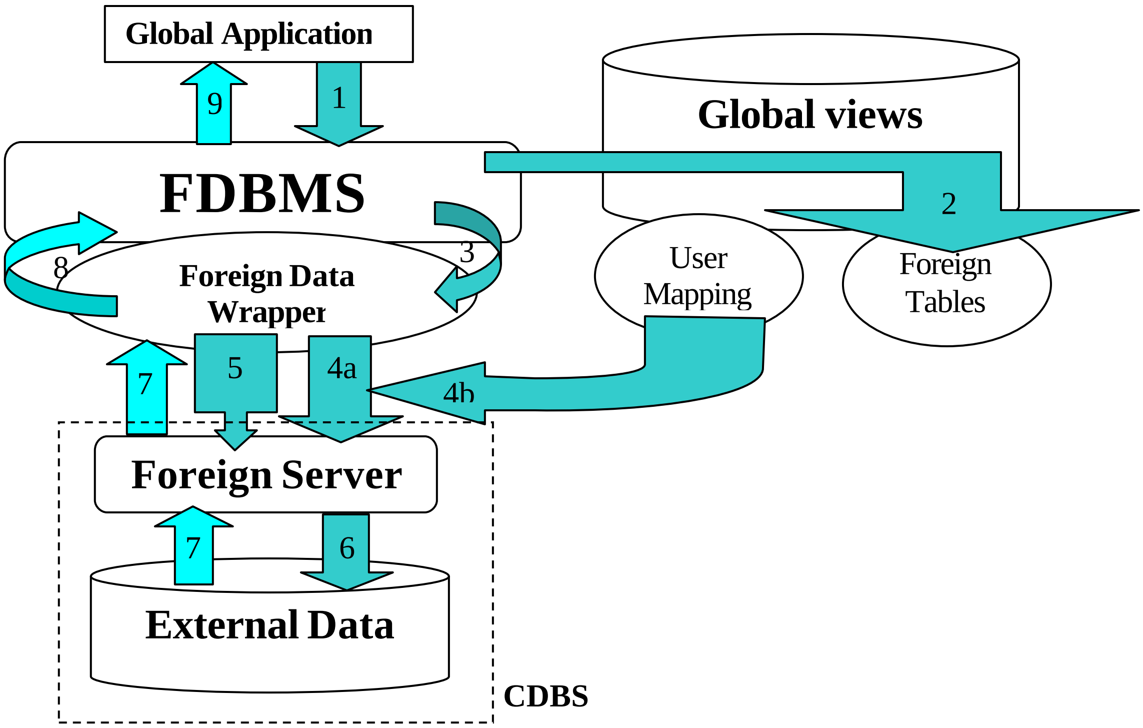

Figure 3.4: Data flow between FDBMS, Wrapper and CDBS

Once FDBMS has pieces of plan allocated to each involving wrapper, FDBMS calls for a connection to be established with the foreign server by the wrapper (Arrow "4a"). The log-in data at the foreign server will be provided to the wrapper too (Arrow "4b").

The wrapper connects to the foreign server and logs in on behalf of FDBMS. Then it confirms to FDBMS that the foreign server is query-ready. FDBMS passes the SQL-segment to the wrapper. The wrapper translates it into query segment into foreign server's dialect and feeds to the foreign server (Arrow "5").

The foreign server processes the wrapper requests and deliver query results back to the wrapper (Arrow 7).

The wrapper converts these return data into table formats set by FDBMS (Arrow "8"). Once FDBMS receives return data for all segments of the global SQL-statement and compensates for foreign server query limitation, the combined data can be returned to the FDBS client (Arrow "9").

Chapter 4
Workflow Management System

4.1 Concept

4.1.1 Business process and workflow

Before defining the concept of workflow, it is helpful to take a look at the concept of business process, to which workflow comes as a support.

An enterprise is divided into different organisational units, each being assigned to a specific area of operation. To fulfil a customer's request or to deliver an output to a market, certain units of the enterprise need to integrate their activities beyond their organisational boundaries to complete the process. A business process can be defined as "a set of logically related tasks performed to achieve a defined business outcome" [DS90] by the participating units.

Often a process repeat themselves along a well defined path, employing same subunits and resources. For example, to produce and deliver a certain amount of a product A, the input of workers, raw materials and transportation means shall be reserved. Further, management of the production and other administration activities like quality control are also required. All these resources are channelled into the production process in a sensible order – first the materials shall be provided by the logistical department, necessary workers are planned for assemble lines, then the production will start while quality control personnel will put up parallel efforts to ensure the finished products conforming prescribed standards.

Such a business process will start over and over again to meet demands on a certain product. The structure of a business process can be represented in a *process model*, in which all activities, paths and resources are detailed. To start a business process means to execute an instance ("*process instance*") of its process model, called "*process template*".

Now that business processes are heavily computer-aided, the amount of needed workers, materials are calculated through specific computer programs, the quality standards of the product are registered in a database etc. Thus (parts of) business processes can be transformed into a structured set of computer-supported activities, which are called workflows. A workflow is defined as the computerised facilitation or automation of a business process, in whole or part [WfM95]. The process model of a business process can be converted into a *workflow model*.

Figure 4.1 depicts a workflow supported by four applications linked by procedures (process execution) for moving data and managing the process.

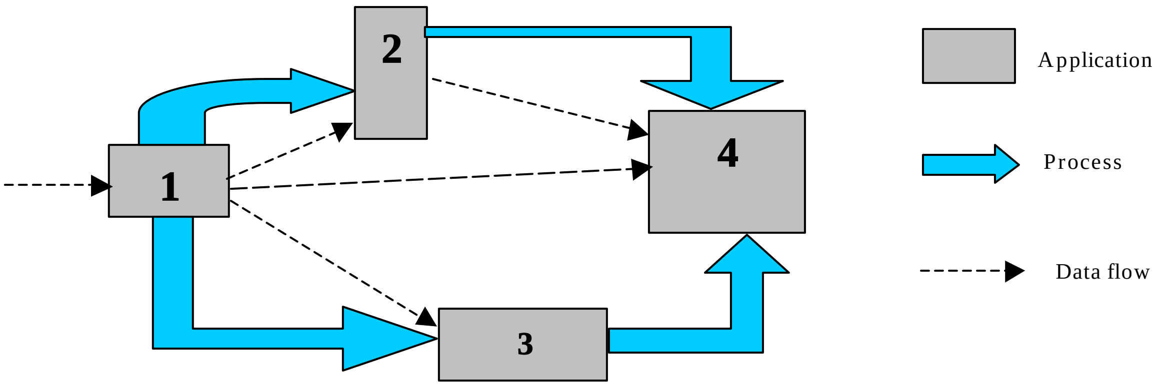

Figure 4.1: A sample workflow

4.1.2 Dimensions and categories of workflow

According to [LR00] there are three dimensions to a workflow, the first of which is *process logic*. The process logic answers the "what"-question on the workflow – the goal of the workflow, the topology of the workflow, the tasks of each single activity, etc.

The second dimension is organisation of the workflow. It defines *organisation* of the human factor in the process execution – who are needed for each piece of work and how they are grouped. Each job function is defined as role and assigned to someone to perform.

The last dimension is the IT-infrastructure, which deals with *which* materials are required for the workflow. That may include a spectrum of hardware and software needed for the execution of the process.

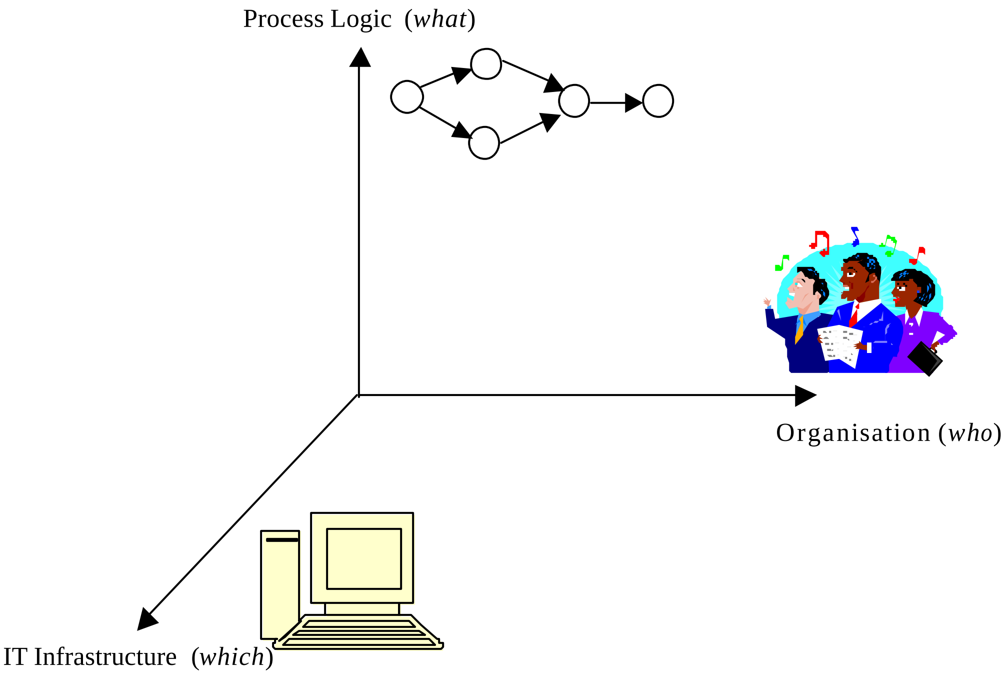

Figure 4.2: Dimensions of a workflow

Once the workflow model is instantiated and put into run, it is carried and driven by these three factors to completion.

One of the goals of a workflow is to separate process logic from individual software applications [MO01]. Separating the two brings about benefits in software reuse and integration of heterogeneous applications which are developed in isolation into different process logics.

Workflows, depending on their business values and recurrence, can be classified into four groups [LR00].

Business value represents the importance of a workflow to an enterprise. Workflows of a high business value are next to the core competence of the company, in other words these workflows are what the company is all about, like logistic company transporting goods of a customer upon order, an automobile manufacturer making a vehicle.

Repetition is the frequency of a workflow being executed. Since defining a workflow is not effortless, repetition is a criteria of judgement whether it is worth the time and work spent on building a workflow model. Workflows of high recurrence are more suitable of being automated and monitored with help of computers.

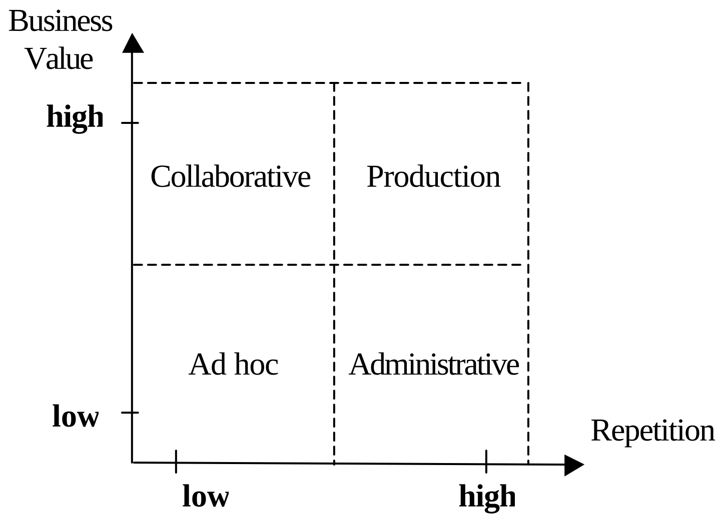

Figure 4.3: Classification of workflows

4.2 Workflow Management System

For precise execution of a business process there is always an administrator defining, scheduling, monitoring and coordinating its execution. The administrator for workflows is a "*Workflow Management System*".

A **Workflow Management System** (WFMS) is a system that completely defines, manages and executes workflows through execution of a computer representation of the workflow logic. [WfM95]

WFMS involves the different dimensions of workflow in the following paradigm (Figure 4.4) [WfM95].

According to Workflow Management Coalition (WfMC) classification there are three functional areas of a WFMS.

First it is the *buildtime*. Before a workflow is executed, the business process it represents shall be analysed and transformed from a real world issue into a formal computer-processable model. The outcome of the analysis is the process definition, or *process model*, *process template*. A standard way of defining the process is through *Workflow Process Definition Language* (WPDL).

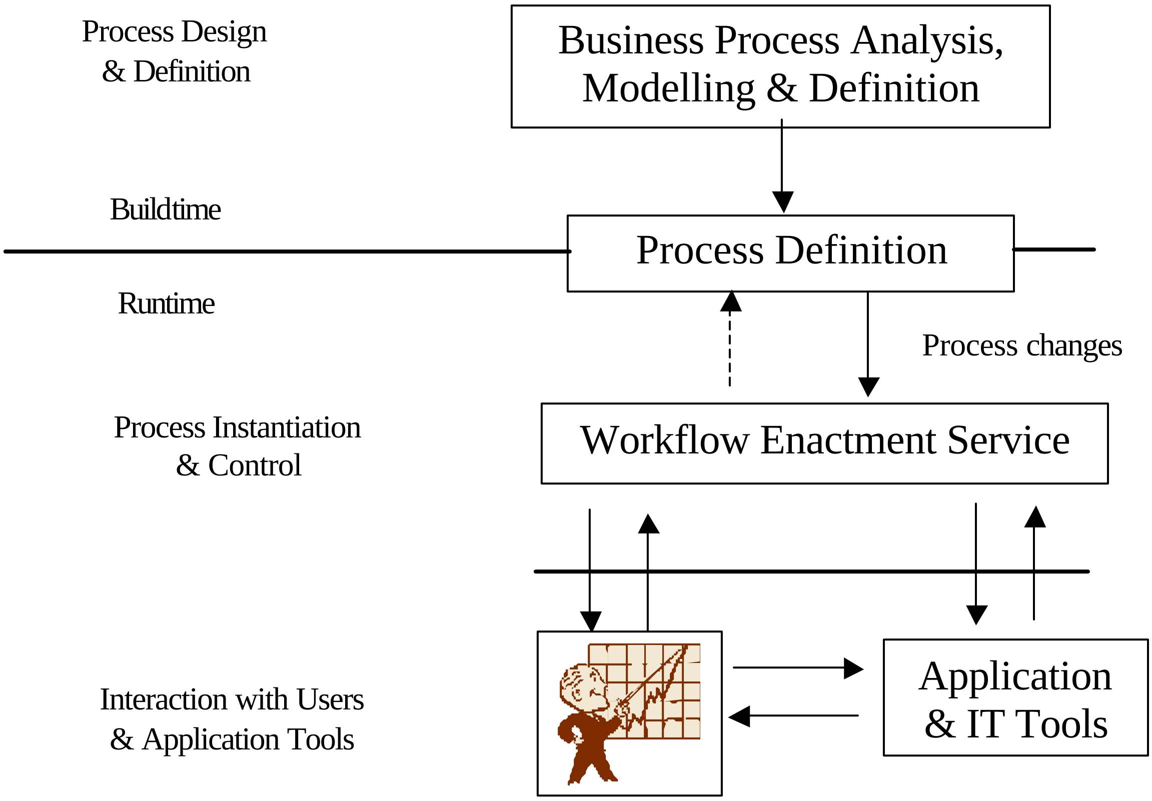

Figure 4.4: WFMS characteristics

Then this process model (often in WPDL) will be translated into *runtime* operations that control and coordinate activities of a workflow. The operation command centre is called *Workflow Enactment Service (the engine for workflow execution)*, whose responsibilities compass creation and deletion of a workflow process, assignment of activities during runtime of a process, interaction with the workflow users and applications.

The successful completing of a workflow process will come down to completions of each single tasks assigned by the Workflow Enactment Service. These tasks are delegated to the human resources, who use IT tools to have the assignment done, or fully automated applications. The interactions between the workflow engine, the human factor and the IT tools consist the another character of a WFMS.

4.3 MQSeries Workflow

For the integration with FDBS (DB2 UDB in this work) IBM MQSeries Workflow is chosen as the Workflow Management System.

4.3.1 Architecture of MQSeries Workflow

MQSeries Workflow (MQWF) is the process engine of IBM's business process management software. MQWF bases its client/server structure on IBM MQSeries and WebSphere. Through its client and server components MQWF supports features like web clients of an e-business implementation, open Web standards, event handling etc.

The client and server components of MQWF can be put into a two-tier architecture[4] [IBM01]

MQWF clients in the tier one interact with users and provide API-access to MQWF management system, which enables users to control the work items and processes. The communication channel between the MQWF clients and the servers in the second tier (MQWF management system) is realised after the client/server paradigm.

The MQWF management system (all components in Tier two of Figure 4.5) comprises usually of MQWF server components, the database system supporting the server and components for two major phases of a workflow – buildtime and runtime.

The execution of the processes in a workflow are supervised cooperatively by MQWF server components, in order to channel processed data through the workflow for the final results. The cooperation between MQWF server components are achieved by message queues underlying IBM MQSeries .

[4] A three-layer architecture view is introduced in the same IBM manual too, which puts database components of runtime and buildtime into a separate base layer. In that case, FDL Import/Export is seen as between both databases instead of between Buildtime components and MQWF server components.

Data over the workflow composition, node activities and process transitions will be grouped and maintained in either runtime or buildtime databases. Buildtime data are descriptions of workflow models and definition of system resources. MQWF encapsulates the buildtime data through MQ Workflow Flow Definition Language (FDL), which enables MQWF server components to read or write buildtime data in a graphical editor.

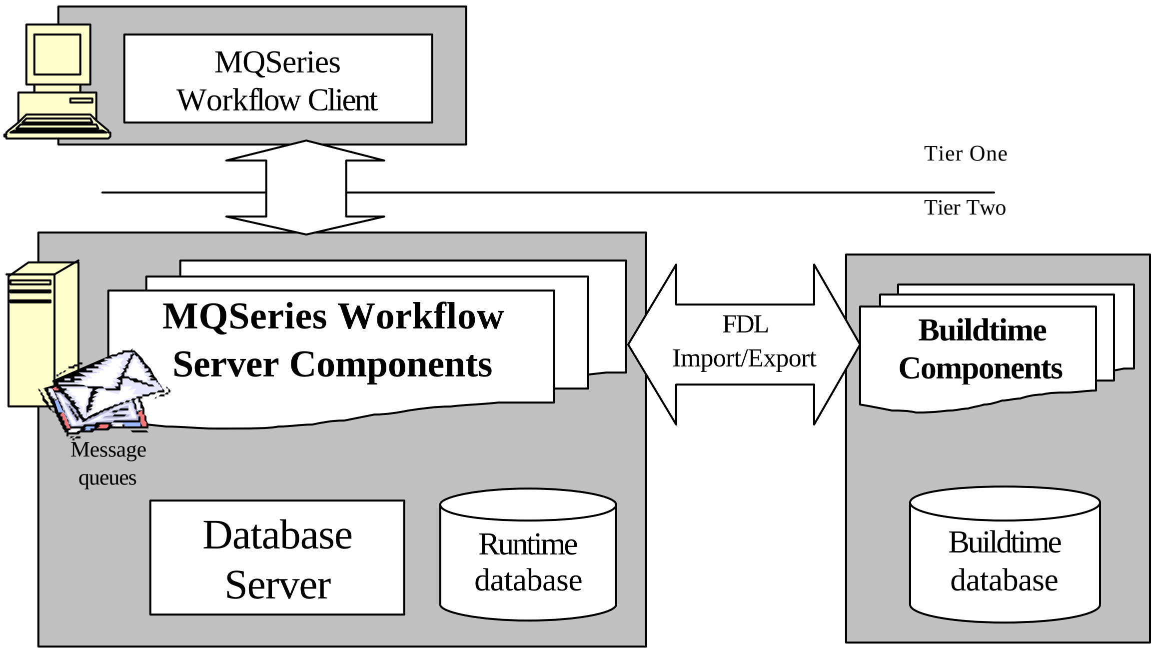

Figure 4.5: MQSeries Workflow architecture (2 tier structure)

The FDL-encapsulated workflow model will be imported into server components to translate into a *process template*. Runtime process template will be instantiated into a executable *process instance*, which oversees assignments on each node of the process to be completed.

In Figure 4.6 [IBM01] are the components in both client and server of MQWF architecture shown. On the MQWF server side runs the *Administration Server*, which is the centrepiece of a WFMS. It supervises, coordinates and communicates with all other server components. *Workflow Execution Server* issues execution commands of workflow processes and monitors them while *Application Execution Server* calls applications associated with process activities. *Scheduling Server* controls and manages notification for activities that have time out limits. *Cleanup Server* deletes process instances once they are completed. Underlying server components are the Runtime DBS client.

The server components can be accessed by client components through server APIs. On the client side, *Administration Utility* and *Application Execution Agent* invoke corresponding server components. *MQWF Client* is the handle for users to start a process, *Custom Client* and *Applications for Custom Client* are components to customize interfaces for specific users.

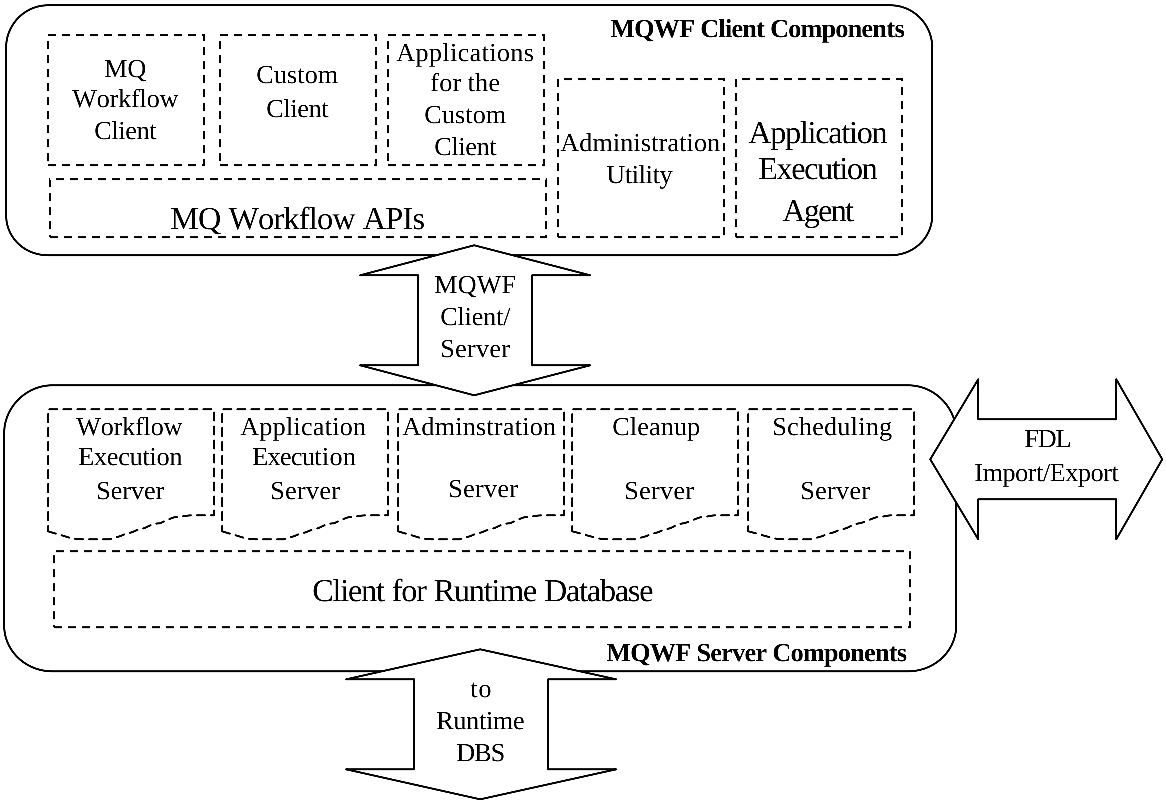

Figure 4.6: Components of MQWF Client and Server

4.3.2 Modelling the process

To build a workflow the first job is to build a solid process logic, with whose framework activities and data flows can be added.

A model for a business process has a starting point and a terminating pointing, called *data source* and *data sink* respectively. Between these two nodes there are nodes of *activities* as relatively independent assignments, which are connected by control and data *connectors*. A control connector leads to a sequential execution of multiple activities, triggered by a transition condition. A data connector specifies flow of data from one activity to another. In the following simple example of credit approval the basic concepts of activities and connectors are shown.

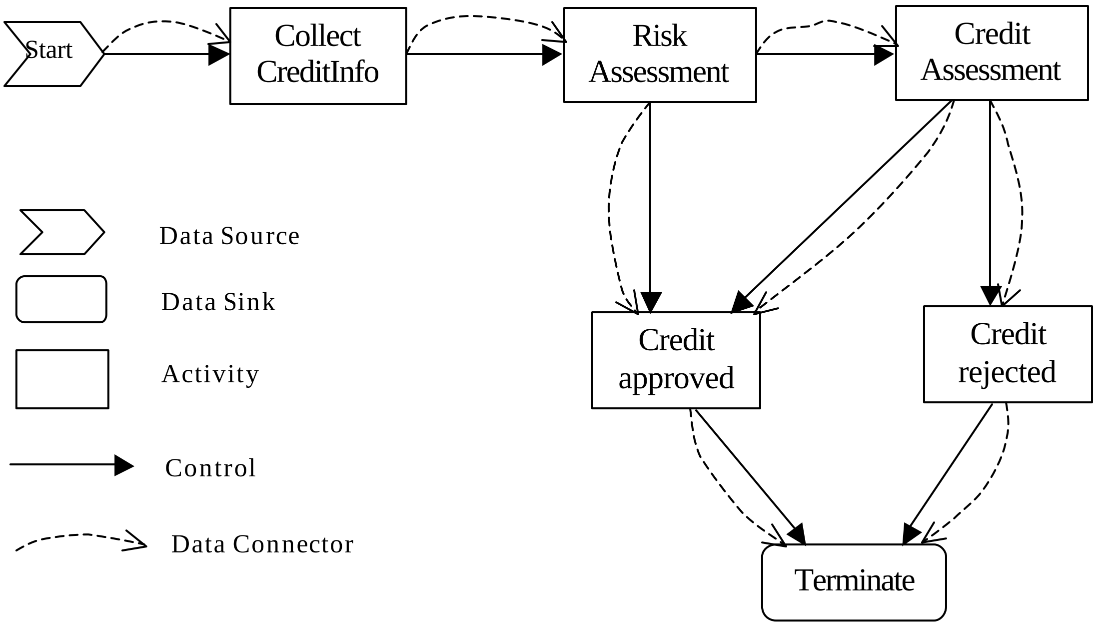

Figure 4.7: Simplified process model of credit approval in IBM MQSeries

In the Figure 4.7, the process was started from the node of *data source*, Control Connector passes the process focus to the activity of “Collect CreditInfo”, along with data passed by Data Connector. “Collect CreditInfo” executes its program with the data (e.g., name and credit history of the concerned customer) from Data Connector.

Once the program is successfully completed, the next activity “Risk Assessment” is to be brought to life by Control Connector, with the relevant data from the previous activity passed along.

“Risk Assessment” makes a decision whether the credit requirement is bigger than the automatic granted credit line to the customer (say, for a EC-card holder, 600 Euros of credit can be drawn by the customer without extra approval of his bank). For requests under the pre-granted amount, a positive answer will be made right away and the process will move to the point of “Credit approved”.

For credits beyond the amount premeditated, the data and process control will be turned over to the activity of “Credit Assessment”. It will be a more complicated decision with a lot of discretions involved. Once the decision is made whether the credit will be granted or not, the process will move further, with one step away to conclude the entire process and reach the node of *data sink*.

4.3.3 Buildtime

Buildtime is a phase in which all three dimensions of a workflow are specified: a business process with well defined activities will be combined with the necessary personnel and IT resources.

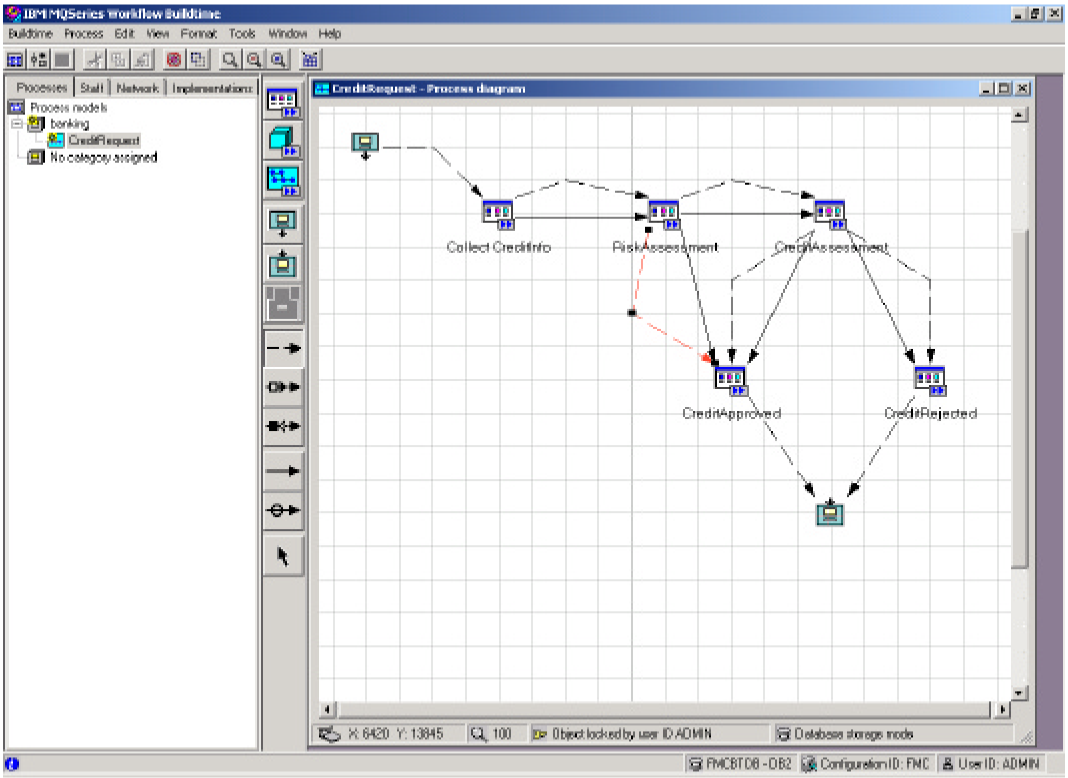

Figure 4.8: IBM MQSeries Buildtime graphical editor and a simple model

With help of a graphic editor of MQWF the process model can be put into a workflow diagram. *Activities*, *data connectors*, *control connectors*, among other elements, complete the workflow topology (see Figure 4.8).

Once the process logic is drawn, the human factor shall be specified for each activity. Three concepts of human factors are involved in MQWF. First it is a *Persons* tree, which is a list of personnel in an enterprise. A *Staff* Tree divides personnel into different levels according to their capacities, for example department chefs, secretaries, interns. *Roles* tree defines roles in an enterprise. A *Role* is a function one person or a organisation unit has, in other words a person may have many roles while a group of persons have the same role. In MQWF organisational units can also be defined.

Further, how the staffs for the activity are associated with the executable programs are also laid out through a notification mechanism. Once an activity is entered by the process control, persons associated to this activity will be put on notice.

The programs to be executed by activities will be also named in the buildtime phase.

4.3.4 Runtime

The workflow model defined through the buildtime graphic editor will be translated into process template using FDL, which can be imported by the server components of MQWF for runtime execution.

Upon request a *process template* will be instantiated into a copy called *process instance* by process execution server. The process instance comes into the status of "*Created*" (see Figure 4.9).

MQWF set a process instance to run by entering the very first activity of the process. In the life cycle of a process instance in Figure 4.8 [LR00] starts a process instance from the status of "Created" to focal point of a life cycle "*Running*". After successful execution the process instance migrates to "*Finished*" and will eventually exit after "*Deleted*". The process instance could be driven to "*InError*" status or be "*Suspended*" by the user. Users with required authorization may terminate a process while it is still running, the instance then go through the "*Terminating*", "*Terminated*" status till it takes the exit at "*Deleted*" status.

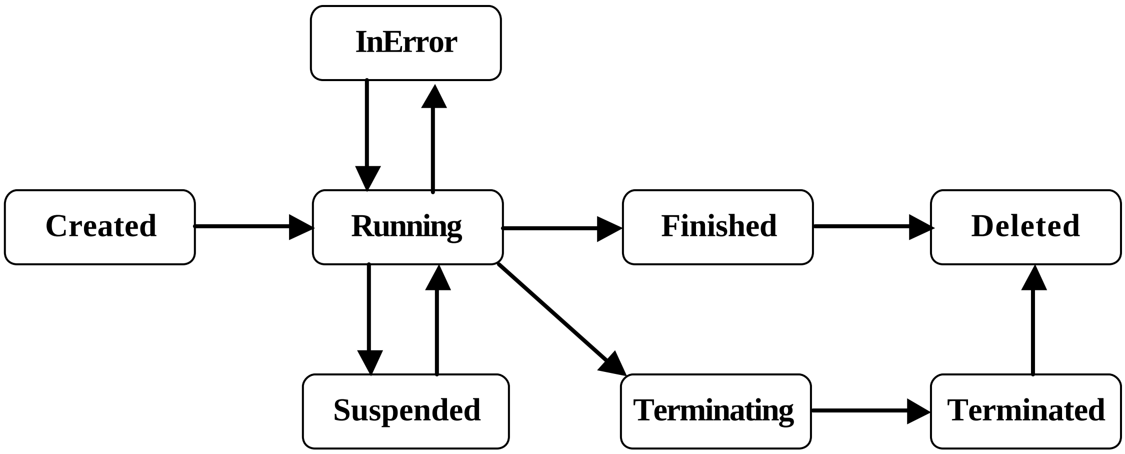

Figure 4.9: Life cycle of a process instance

During the runtime, working data is stored in "*containers*" and delivered between activities for processing.

At the starting node a workflow initialises its *process input container*, and passes it through the activities, which have their own input and output container. Upon the successful execution of a workflow the processed data package reaches the terminating node and is puts into a *process output container* as a result.

Along the flow of process execution the data flow between two activities and their containers are shown in Figure 4.10. As the control flow reaches activity A, the data flow brings the data from the last activity also to A's input container, which will be processed within A. Then A put the results into its output container. The control flow's next stop will be activity B, so shall the data be transferred to the input container of B. However, the differences in data structure (names, types) between these two containers require usually a container data mapping to ensure B getting the right data in a right way.

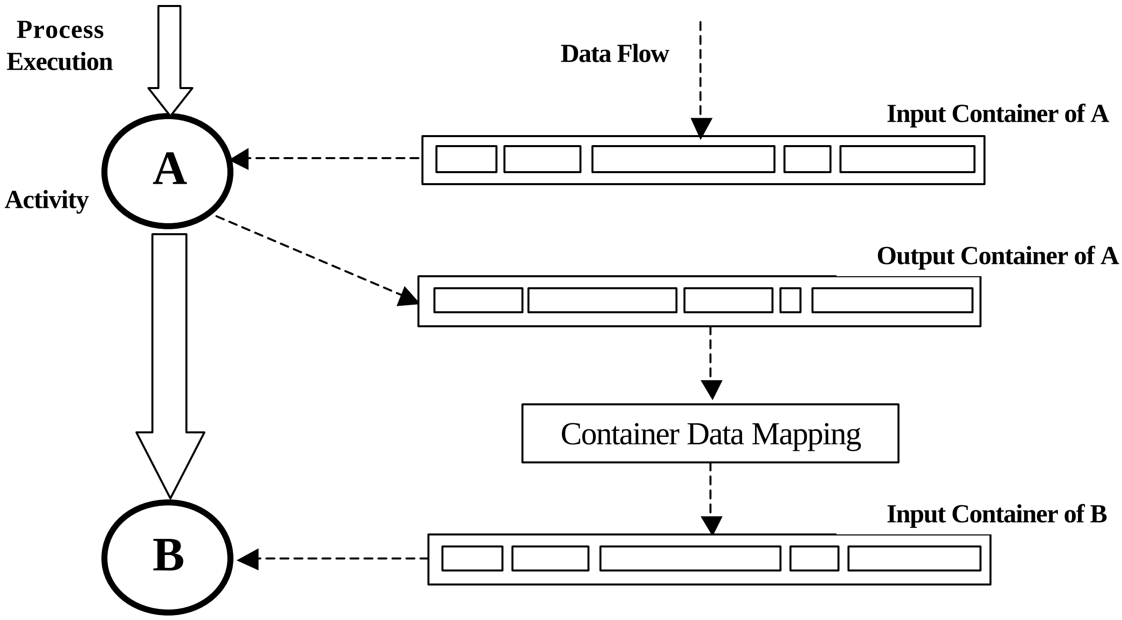

Figure 4.10: Data flow between Containers and activities

Chapter 5

Workflow Wrapper Structure

Workflows can be integrated to FDBS by different means like wrappers within the framework of SQL/MED [RS97] [HH01], user defined function (UDFs) [HH00] and stored procedures.

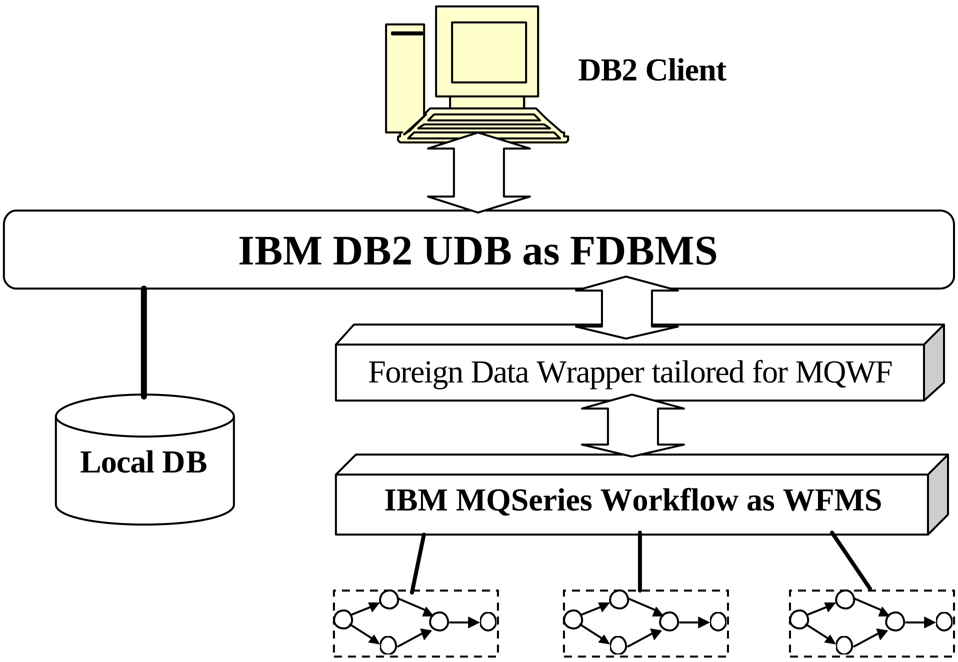

Figure 5.1: The integration architecture of wrapper approach

In this work, the wrapper approach is chosen. A three tier structure is built in the integration scheme, with IBM DB2 UDB as FDBS and IBM MQSeries Workflow as underlying workflow management system (see Figure 5.1). The wrapper customized for MQWF is called *workflow wrapper* in this work and is abbreviated just as *wrapper* in this chapter.

Thus interactions will be taking place in two levels in this integration scheme, one between the interfaces of UDB[5] and workflow wrapper and the other between the interfaces of workflow wrapper and MQWF. The making of the workflow wrapper in this work is based on IBM wrapper development support, which conforms to SQL/MED. Therefore discussions involving workflow wrapper will be presented within the SQL/MED framework.

5.1 Interfaces between UDB and wrapper

5.1.1 Interface described in SQL/MED standards

A foreign data wrapper provides FDBMS a mechanism to access multiple foreign data sources by its routines, which can be invocated through the wrapper's application programming interfaces (APIs).

Most commonly a foreign data source is managed by a SQL-aware server known to FDBMS but independent to it. A wrapper can therefore be viewed as a register and a data "adaptor" of the foreign server. To identify the data structure both on the FDBMS and foreign server sides, *descriptors* are used. Foreign server's descriptor saves information of server name, type, version, authorization, generic options and the name of the corresponding wrapper. Wrappers keep descriptors of their own too, which comprise information of wrapper names, authorization, generic options, library names and interface routines. Descriptors represent general data structures maintained in the foreign data wrapper.

Routines are the interactions between FDBMS and foreign servers, which are catalogued in foreign data wrappers. There are four categories of routines: resource allocation and deallocation, connection control and monitoring of foreign servers, negotiating SQL-capacities between FDBMS and foreign servers and control over the execution of SQL-statement by the foreign server.

Executing and controlling routines is implemented together with the use of *handles*. By supplying handles as parameters of routine invocation, state

[5] UDB or FDBMS in this chapter are all referenced to IBM DB2 UDB.

of foreign servers and foreign server wrappers can be obtained or altered. Handles are represented in INTEGER values.

Routines

According to SQL/MED standards routines can be divided into the following categories, each reflecting an area of work that routines are designed to do.

Handle Routines provide access to objects involved in the UDB, foreign server and wrapper interactions. A typical handle routine is a 'get' method associated with a handle object, which returns the information of the handle. A routine to find out the name of a foreign server associated with the server handle in form of GetServerName (…ServerHandle one_handle…) is for example a typical handle routine.

Initialisation Routines, as the name suggests, provide initialisation of descriptors by AllocDescriptor(…), global data structures for wrappers by AllocWrapperEnv(…), connections to foreign servers by ConnectServer(…), request and reply handles of a SQL-request by InitRequest(…), SQL-Statement preparation by AdvanceInitRequest(…) etc.

Access Routines are involved during the transmission and execution of a particular query segment. In each query session, (i.e., wrapper receives a fragment of a SQL-statement from FDBMS and process it in cooperation with foreign server), FDBMS will determine whether to feed SQL-statement to wrappers in *pass-through* mode or *decomposition* mode. In the pass-through mode FDBMS invokes TransmitRequest(…) routine to let wrapper pass SQL-statement in the native SQL of the foreign server. In decomposition mode FDBMS prepares resources for query segment execution through open(…) routine and retrieves results in rows through Iterate(…) routine until all data is read. Close(…) routine frees resources allocated for the query session as ReOpen(…) routine reinitialise resources for processing another query segment of the same foreign data source in the same query session.

Termination routines free resources allocated by initialisation routines and then terminate a querying process through wrappers.

FreeWrapperEnv(…), FreeFSConnection(…), FreeExecutionHandle(…) and so on may find their counterparts in the initialisation routines.

Handles

Handles are of type INTEGER. Each is created to identify a resource in the query session. They are passed around as arguments while invoking routines to obtain or set information. The followings handles are introduced in the order of their creation in the life cycle of a query session.

Wrapper Handle is created by FDBMS to reference a foreign data wrapper. While creating an instance of a wrapper, **WrapperEnv Handle** and **Descriptor Handle** are also created. The former is a reference to the wrapper environment in which FDBMS's requests will be processed and the later returns a reference to the wrapper descriptor.

Server Handle references to the involved foreign server and can provide information on the server's name, type, version among others. **FSConnection Handle** is in turn the reference to the connection built between the wrapper and the foreign server. The lifetime of FSConnection Handle is called also a **foreign server session**, during which a chain of operations will be performed by the wrapper on this handle. To establish such a connection user mapping in the foreign server shall be maintained and made available through **User Handle**.

After preparing for a query session a **Request Handle** will be created by FDBMS to reference a SQL-statement and the foreign tables, columns that are involved in the SQL-statement. Wrappers determine which part(s) of the query the designated foreign server can process and create a **Reply Handle** to reference to the query segment. Along side the reply handle an **Execution Handle** is created by the wrapper to maintain information of query execution and resulting data sets.

For every foreign table and each of its columns FDBMS creates a handle. A **Table Reference Handle** is associated to a foreign table object to keep information of table names, column names, etc. The detailed column information is kept up in another handle called **Value Expression Handle**. **QueryContext Handle** is a special indicator of identical value expression handles in multiple query requests, which gives wrapper room for optimisation.

FDBMS additionally maintains a **Routine Mapping Handle** to reference routine mapping descriptions.

Descriptors

Descriptor areas (descriptors) provide interfaces to description of values between FDBMS and foreign server. In other words the input values for query executions and results of query executions.

Descriptors are made of header and item fields. Routines **GetDescriptor(...)** and **SetDescriptor(...)** retrieve and modify values in descriptors through individual handles.

There are generic option descriptors for each kind of objects using generic options. Those objects are foreign data wrappers, foreign servers, foreign tables, columns in foreign tables and user mappings. In these descriptors pairs of option name and value are listed.

Following variants of Descriptors are employed in the wrapper structure:

Table Reference Descriptor describes the foreign table queries in a SQL-statement and is associated with TableReferenceHandle.

Each **Wrapper Row Descriptor** (WRD) contains a row of results from the pass-through query. It is associated with an execution handle.

Server Row Descriptor (SRD) describes types and locations of query results and is also associated with an execution handle.

Wrapper Parameter Descriptor is used to describe input values of pass-through query execution.

Server Parameter Descriptor (SPD) specifies types and locations of query input values.

5.1.2 User interface of wrapper in UDB

From the view of client users there is a set of commands available to exploit FDBS capability of querying external data sources. The standard SQL commands in SQL/MED were introduced in Chapter 3. Users of UDB have such commands with small variations.

First a wrapper module shall be named to take charge of querying external data. UDB will search the module and register in its catalog a mapping between the wrapper and the module. SQL/MED includes authorization information in the "CREATE FOREIGN DATA WRAPPER …", but in UDB wrapper commands can only be issued by users with DBS administrator authority. UDB itself stores no standard generic options, however wrapper programmer may create and maintain his own.

```
CREATE WRAPPER <wrapper name>
        [<library name and path>]
        [<generic options>]
```

To delete or alter a certain wrapper entry from UDB catalog the command "DROP WRAPPER …" or "ALTER WRAPPER …" may be issued.

Each foreign server has to be associated to an existing wrapper (A wrapper may associate with multiple servers). The UDB format of server definition is:

```
CREATE SERVER <foreign server name>
        [TYPE <server type> VERSION <server version>]
        [AUTHORIZATION <authorization information>]
        WRAPPER <wrapper name>
        [<generic options>]
```

Server's information on type, version and generic options can be deleted or altered through "DELETE SERVER …" or "ALTER SERVER …".

Through definition of foreign server the foreign data source is now integrated into UDB. To explore external data UDB must register them into *nicknames*, UDB-specific naming of what are foreign tables in SQL/MED.

```
CREATE NICKNAME <table name>
    <column list and definition>
    SERVER [foreign server name]
    [<table generic options>]
```

Once the nickname is created, the description of its columns and generic options will be saved into UDB catalog. When the foreign table (nickname) or its columns are referenced in a SQL-statement, a chain of commands is set off to invoke the responsible wrapper module, to locate and connect the appropriate server, to retrieve the data queried and to convert the results into the format clients request. Nickname information and options can be modified through "ALTER NICKNAME …".

UDB registers user mapping information by naming first a user name of UDB, and then provides log_in information of an existing foreign server as such like user name, domain name, password. UDB uses automatically these data to log on to a foreign server for data retrieval.

```
CREATE USER MAPPING FOR < udb_user_name >
    SERVER [foreign server name]
    OPTIONS [<foreign server log_in information>]
```

UDB clients have now a transparent view of the bound foreign data source residing in a foreign server. Nicknames can appear in Select-statements just as any native tables of UDB. One limitation is for the current version of UDB only read operation can be performed by UDB on foreign servers.

5.1.3 Interactions between UDB and Wrapper

When a client issues a SQL-statement to UDB in which a registered nickname is in the "FROM …"-clause, UDB generates a query segment for this foreign table (nickname).

UDB manipulates objects of wrapper, server, nickname by invoking routines and initiate a series of interactions.

In Figure 5.2 one simplified interaction paradigm between UDB and wrapper is drawn. The actions from UDB/wrapper are marked with arrows and reactions of the other side are marked in dotted arrow lines.

Since wrapper may choose between two modes, *pass-through* or *decomposition*, depending on the type and capacity of relating CDBSs, the interaction paradigms are slightly different for two mode. Here the interactions of *decomposition* mode is shown because wrapper enters this mode to access WFMS.

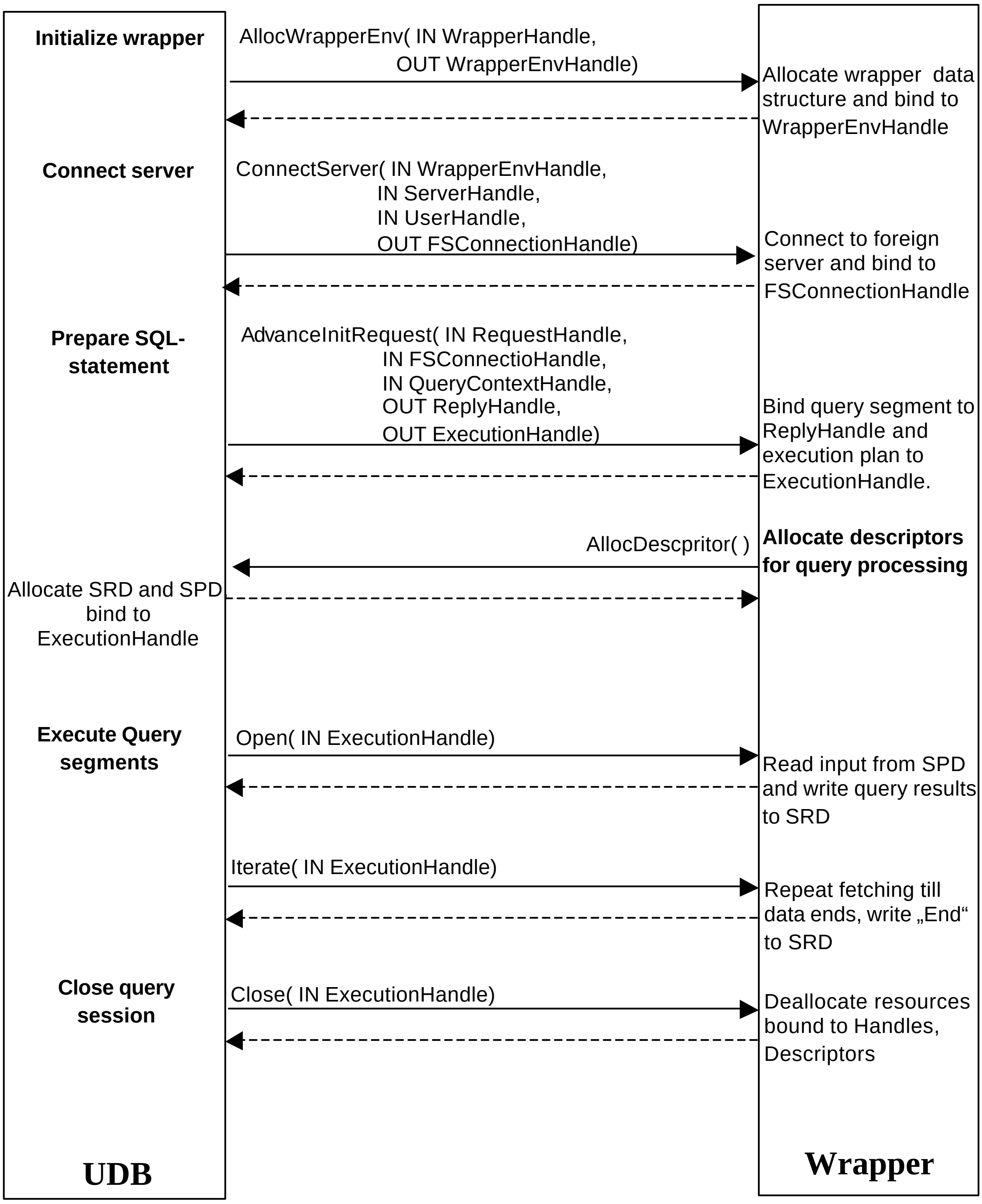

Figure 5.2: Interaction of UDB and Wrapper in SQL/MED paradigm

UDB invokes rountine AllocWrapperEnv(…) to initialise the wrapper; wrapper initialises its data struture, associates it with WrapperEnvHandle and returns the handle to UDB.

UDB generates information about registered foreign server and user mapping according to its catalog (stored by "CREATE SERVER …", "CREATE USER MAPPING …"), then UDB initialises ServerHandle and UserHandle and associates them to the data created above.

With the handles for wrapper environment, foreign server and user mapping complete, UDB requires wrapper to establish a connection to the foreign server and associates the server session to FSConnectionHandle. Wrapper retrieves information on foreign server and user mapping with repective handles, which is needed for building a connection. Wrapper also does customized initialization of foreign server before returning the connection handle to UDB.

UDB initialises RequestHandle and associates it with the query segment(s) intended for the wrapper, initialises ReplyHandle and ExecutionHandle, invokes the routine AdvanceInitRequest (…). Wrapper executes this routine to analyze how much the foreign server can handle the query segment(s) pointed by RequestHandle, then answers to UDB in ReplyHandle as return.

Additionally UDB consults with wrapper about its query execution plan pointed by ReplyHandle, wrapper returns the actual execution plan of the query segment(s) through ExecutionHandle to UDB.

Wrapper takes the initiative to ask UDB to allocate resources for SRD and SPD as preparation for query execution. Both descriptor will be associated with ExecutionHandle. UDB obliges and write input parameter in SPD and invokes routine Open(…).

Wrapper uses ExecutionHandle to access SPD, reads parameter and open the foreign server for query execution. UDB repeats routined of Iterate(…) to retrieve data that wrapper gets from the foreign server and puts in SRD. When the queried data are all retrieved, wrapper signals an "End" to UDB.

UDB invokes Close(…) routine to end this foreign server session, wrapper deallocates resources bound to its descriptors and handles.

5.2 Workflow Table Abstraction

To communicate with MQWF and to integrate it into UDB through a Workflow-Wrapper, a *workflow table abstraction* (WTA) was introduced [RW01]. The purpose of WTA is to encapsulate interaction details between workflow client (here the wrapper) and WFMS (see Figure 5.3).

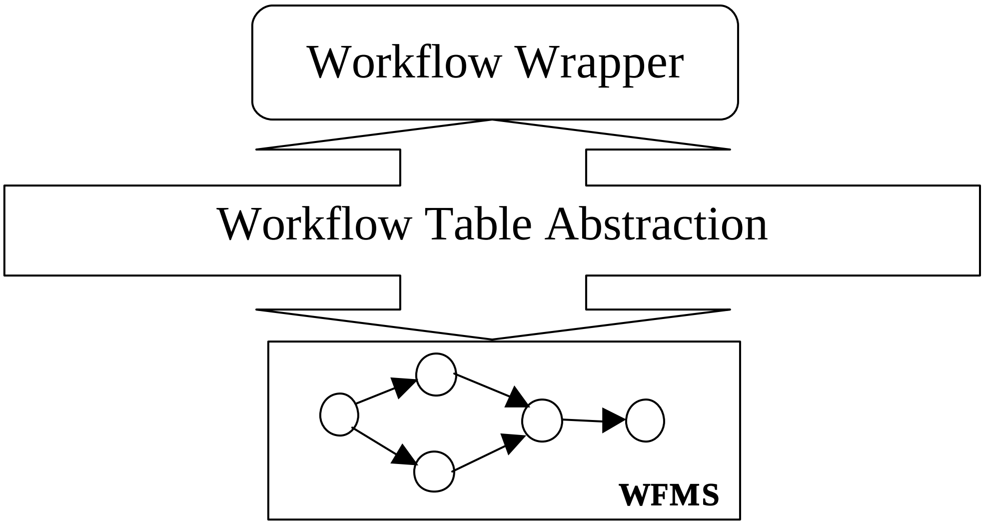

Figure 5.3: WTA as “middle man” between wrapper and WFMS

5.2.1 WTA Functions

There are two major functions abstracted through WTA. For one WTA encapsulates routines that access APIs of WFMS (WAPI).

A class called WFMS_API with no implementation details is created as a general interface to access MQ Series Workflow API (WAPI) (see Figure 5.4). A complete implementation of interactions with WAPI is realised in the class DSSC_MQWF [RW01].

In the class DSSC_MQWF following WAPI routines are integrated into its methods:

Connection Function is integrated into the member function DSSC_MQWF::ConnectToWFMS(…) to initialise WAPI and connect to workflow server.

Process Control Function is integrated into the member function DSSC_MQWF::createProcessInstance (…) to create a process instance.

Activity Control Function is integrated into the member function DSSC_MQWF::executeProcessInstance() to execute activities of the process instance.

The input needed for execution is encapsulated in a data container by generic methods of DSSC_MQWF:: writeParameterIntoWF(…). The results of process instance execution is saved in a second data container and returned through methods of DSSC_MQWF:: readResultFromWF(…).

Finally the connection to workflow server is cut by invoking the method DSSC_MQWF::disconnectFromWFMS(…).

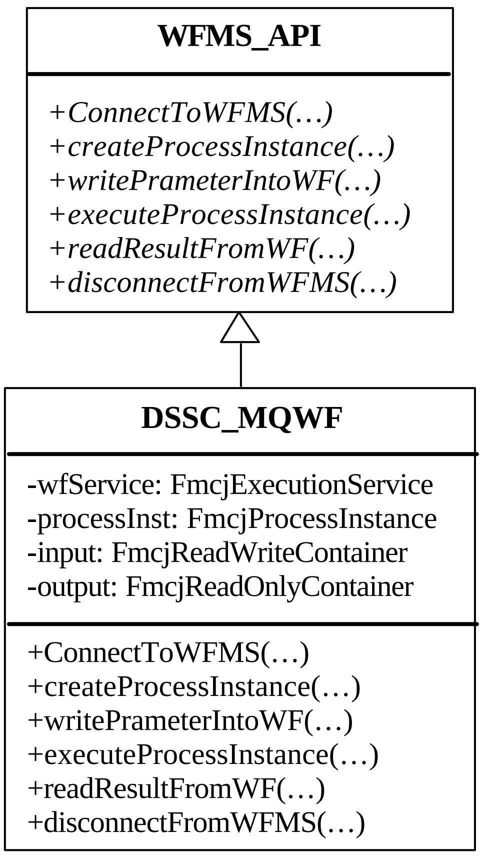

Figure 5.4: Class diagram in UML of classes accessing workflow API

The second function of WTA is to save and to organise the results of workflow activity execution in a manner that would enable wrapper to read and convert data format with little effort.

A class hierarchy is constructed to represent the data (see Figure 5.5). It begins with the class ExternalTable, which is the handle to access the whole data. It stores an array of objects from class ExternalRow, which represents a record of data.

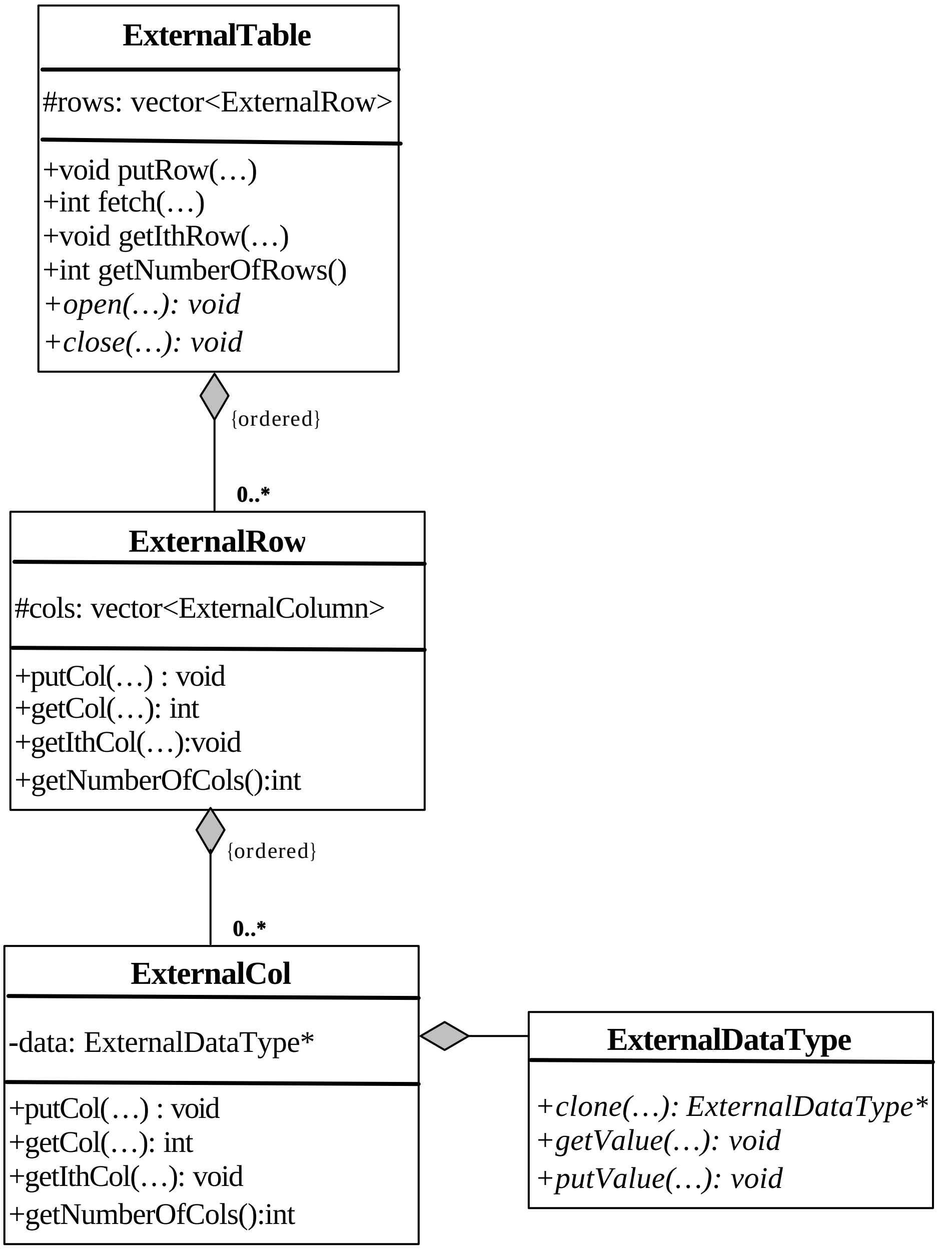

Figure 5.5: Class diagram in UML of WTA table classes

ExternalRow in turn contains an array of objects from class ExternalCol, which is an equivalent of column data. Once instantiated, the whole ExteralTable object can be accessed by browsering through its structure (see Figure 5.6).

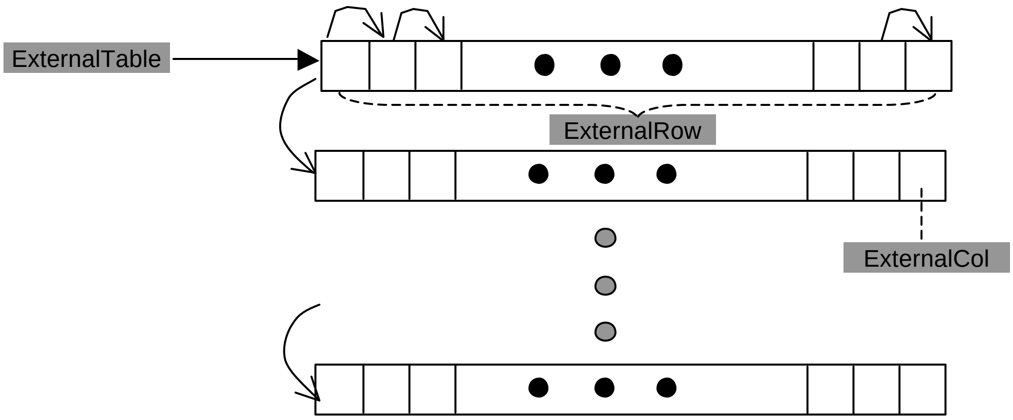

Figure 5.6: Simplified structure of an instantiated class ExternalTable

From the entry point of object from class ExternalTable, the structure stretches to a list of ExternalRow objects, each of them containing a list of ExternalCol objects. Not shown in Figure 5.5 is how different objects of ExternalDataType may be stored in each cell of ExternalRow.

Each of these classes have built-in public *get*-methods to enable access to their non-public member data, a common practice of object-oriented programming. For the getIthCol method two parameters are needed, the first one using a pointer to pointer for an efficient function output, the second parameter as input to indicate which of the ExternalColumns shall be fetched (see following).

```
void getIthCol (ExternalColumn** col, int index)
```

Since there are several types of data as return from WAPI, WTA build a class hierarchy to represent each of these types.

It begins with the abstract class of ExternalDataType. With these types formatting data before putting them into containers or reading them out of containers will be easier to perform, which saves the wrapper of similar task.

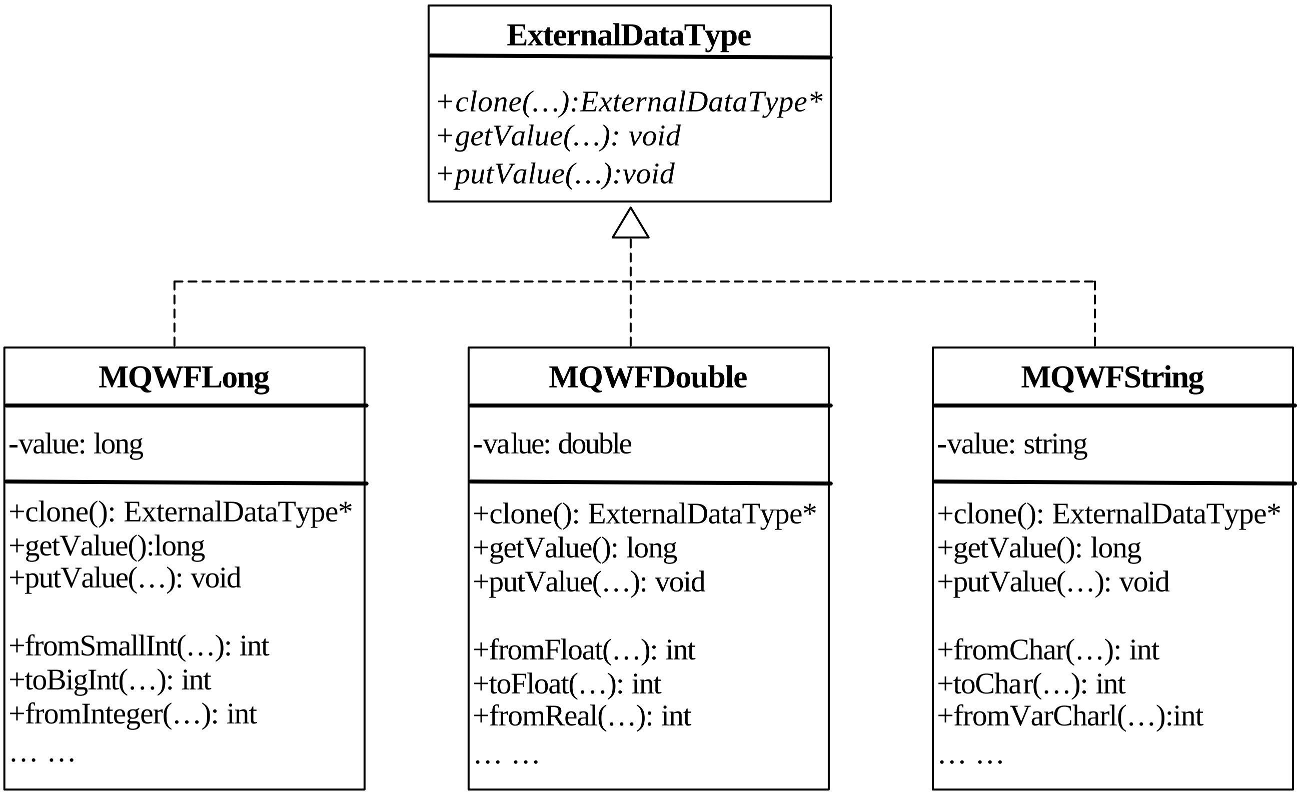

Figure 5.7: Class diagramm in UML of ExternalDataType

In the case of WTA-wrapper, value saved in the class MQWFLong can be converted to/from SQL_INTEGER, that of MQWFDouble to/from SQL_DOUBLE, that of MQWFString to/from SQL_CHAR or SQL_VARCHAR, all of which are built-in DB2 data types (Figure 5.7).

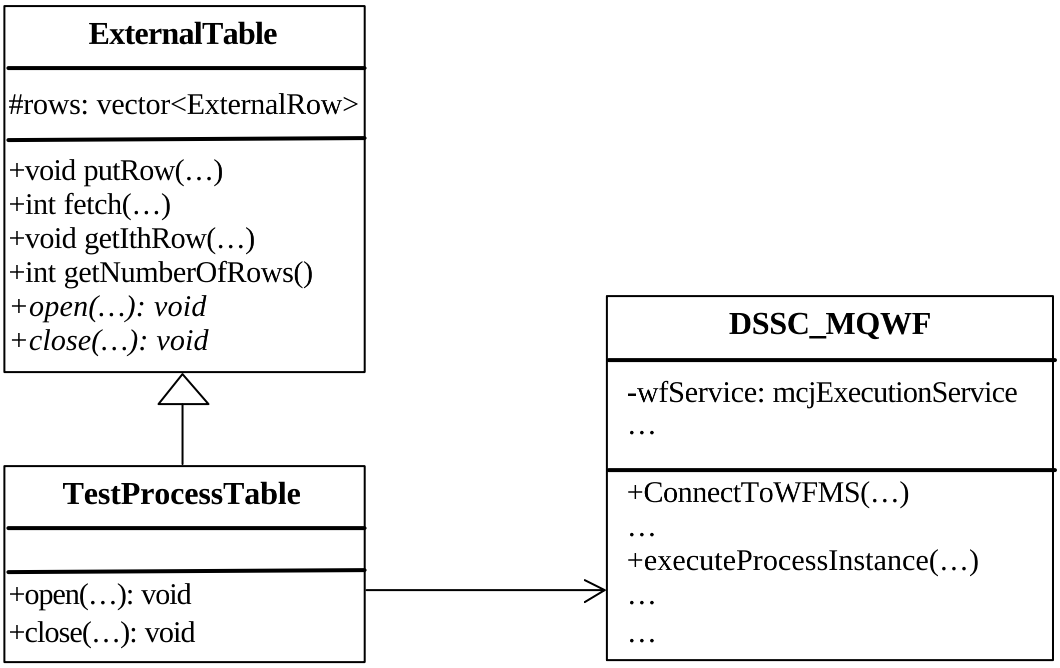

Figure 5.8: concrete class TestProcessTable acting as WFMS client

In the class ExternalTable there are two abstract methods Open(…) and Close(…) defined (Figure 5.8). For each workflow, there shall be a subclass that is derived from ExternalTable and the two abstract methods have to be implemented in the subclass, in which details of interactions between WFMS and WTA are set.

As an example, a concrete class TestProcessTable is derived from ExternalTable as execution handle for a certain workflow process instance called TestProcess. For the routines to access WAPI, TestProcessTable calls functions of class DSSC_MQWF and saves results in the data structure in itself, which is inherited from ExternalTable.

5.2.2 Interaction between Wrapper, WTA and WAPI

To take an object of the class TestProcessTable as WTA instance, put it into the context of WTA wrapper and WAPI, interactions between wrapper[6], WTA and WAPI follows the paradigm in Figure 5.9.

First wrapper initialize an instance of WTA through "TestProcessTable test_wta", WTA allocates resources of the newly created object. Then wrapper instructs test_wta to establish a connection with WAPI by invoking its open method.

test_wta executes the routines class DSSC_WFMS offers, which connect to WAPI, create a process instance, write the execution parameters into the container, then read the data out of container and save them in its own data structure, then disconnect from WAPI.

After finish opening WAPI, test_wta has the data in its ExternalTable hierarchie in an orderly manner. Wrapper calls its own routine of fetching data, which goes with the data structure of test_wta row after row, grasps data saved in objects ExternalCol, converts them into DB2-compatible format by calling routines built in ExternalDataTypes.

[6] Here and in the proceeding paragraphs wrapper means specifically workflow wrapper with integrated WTA functions.

As fetch routine reaches the last row, thus completes its work, wrapper calls for destruction of test_wta and thereby releases the resources.

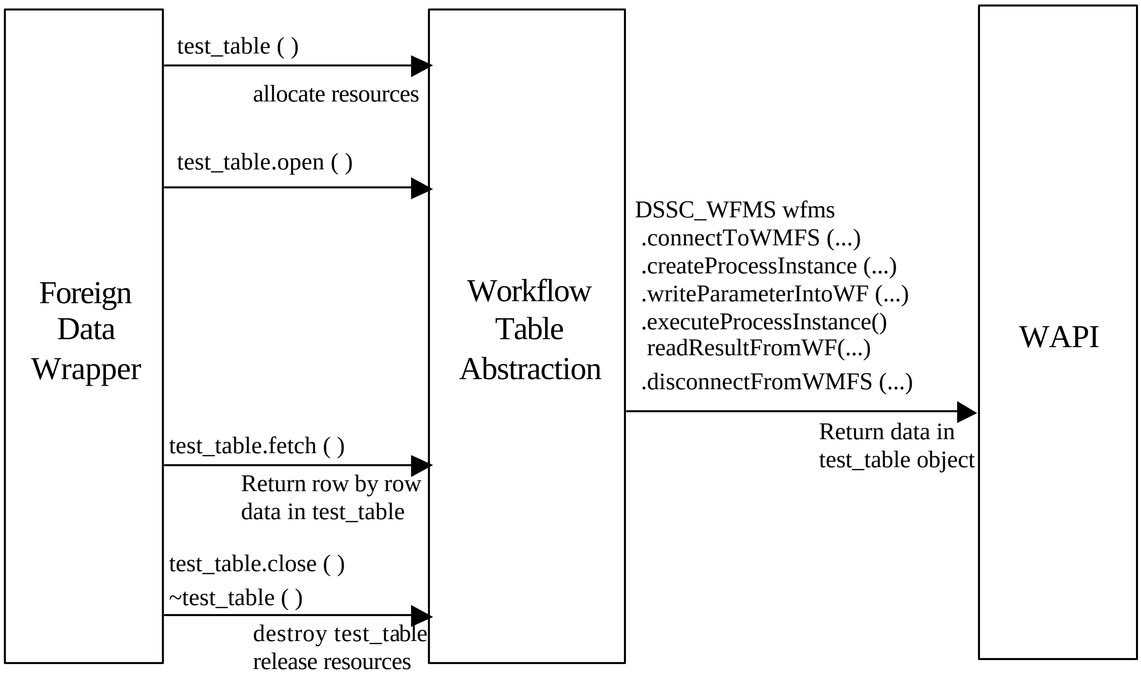

Figure 5.9: Interactions between wrapper, WTA and WAPI

5.3 Implementing workflow wrapper

5.3.1 Organisation of classes for workflow wrapper

SQL/MED describes default behaviours of wrapper while interacting with its FDBMS, without outlining a structual framework of it. That leaves to FDBMS providers considerable discretions implementing their own wrapper class hierarchie.

However the backbone of any such object-oriented wrapper implementation may not vary too much from each other given the premeditated wrapper behaviours. In the following a pseudo-design of basic wrapper classes will be presented. Further this wrapper design will be integrated with Workflow Table Abstraction.

Recalling the set of SQL statements involving wrappers (see Chapter 3.2) available to FDBS clients and the order of which the statements are to be issues[7], the starting point of a wrapper hierarchie has to be a foreign data wrapper (FDW) class. FDW class is instantiated through "CREATE WRAPPER …" command. FDW object shall be associated with **WrapperEnv Handle** and **WrapperHandle**, which are by FDBMS.

A FDW class shall contain interfaces to control connections to foreign server, initiate and terminate SQL-statement execution at the foreign server [ISO02]. Further FDW class shall be designed as such that it may control and monitor a group of foreign servers with minimal adaption. Therefore a foreign server (FS) class is introduced. FS class will be instantiated through "CREATE FOREIGN SERVER …" command. FS object shall be associated with a **ServerHandle**, which is created by FDBMS and available to FDW.

For the same logic a user mapping (UM) class, a foreing table (FT) class are introduced. They are respectively instantiated by "CREATE USER MAPPING …" and "CREATE FOREIGN TABLE …" commands, which are associated to **UserHandle** and **TableRefenceHandle**.

These four classes uphold the framework of a wrapper hierarchie. Indeed they reflect the physical existence of a foreign data souce that is to be integrated through wrapper (see Figure 3.3). The FS class stores information of the foreign server, UM class the login data to that foreign server and FT class represents a logical snapshot of a relation existed on the foreign server. Thus if a wrapper has its way by accessing and using these information, FDBMS shall be able to perform operation on foreign data just as if they are in a local database of FDBMS.

Figure 5.10 shows how FDBMS may instantiate the major wrapper classes mentioned above to gain access to a foreign database. The entry point of FDBMS to wrapper structure is the WrapperHandle. Through handles FDBMS can access User Mapping and Foreign Server object. Combining the information provided by both objects FDBMS will be able to log on the foreign DBS. Then FDBMS refers to Foreign Table object via its handle to find out data structures of a certain relation existing on the Foreign DB. Now with both the permission of foreign DB and the knowledge on data structure of a relation residing in this foreign DB a FDBMS may request foreign DB to apply SQL operations on this relation.

[7] SQL/MED SQL-commands will be used in this section. UDB variation of commands has the same structure and order, which can be adapted easily into the discussion here.

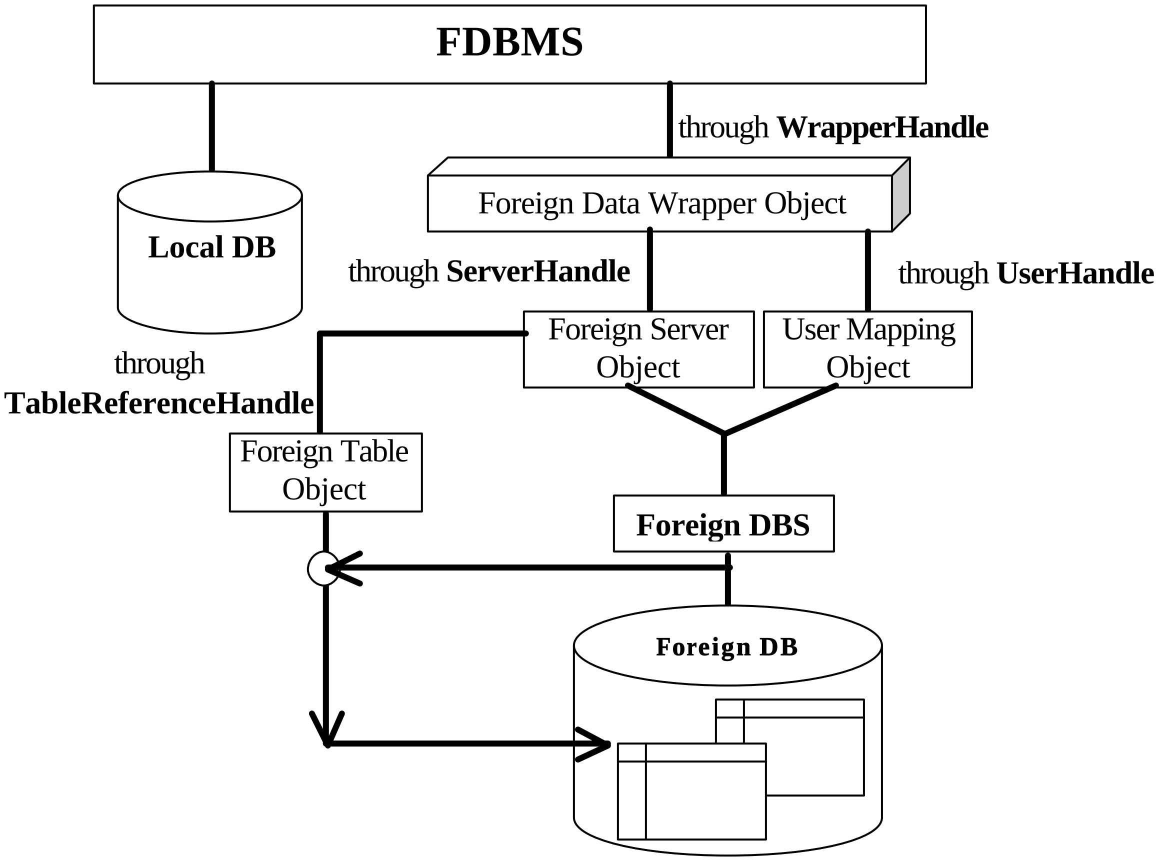

Figure 5.10: FDBMS accesses foreign DB through wrapper classes and handles

Within the framework of SQL/MED there are other important routines to be implemented, like AdvanceInitRequest(…), ConnectServer(…), Open(…), Iterate(…), AllocDescriptor(…), in order to have a completed paradigm of interaction between FDBMS and wrapper (see Figure 5.2).

More classes could be brought into wrapper class design to encapsulate implementation of these routines. For example the concept of Descriptor can be converted into a generic class that represents the general structure of a descriptor. Since objects like FDW, FS, FT, UM etc all desire their own specific information storage in descriptors, it can be extended into subclasses tailor-made for their special needs. In the coming discussion it is assumed that all interactions between FDBMS and wrapper are achieved by classes carrying out routines, not limited to the four major classes just mentioned.

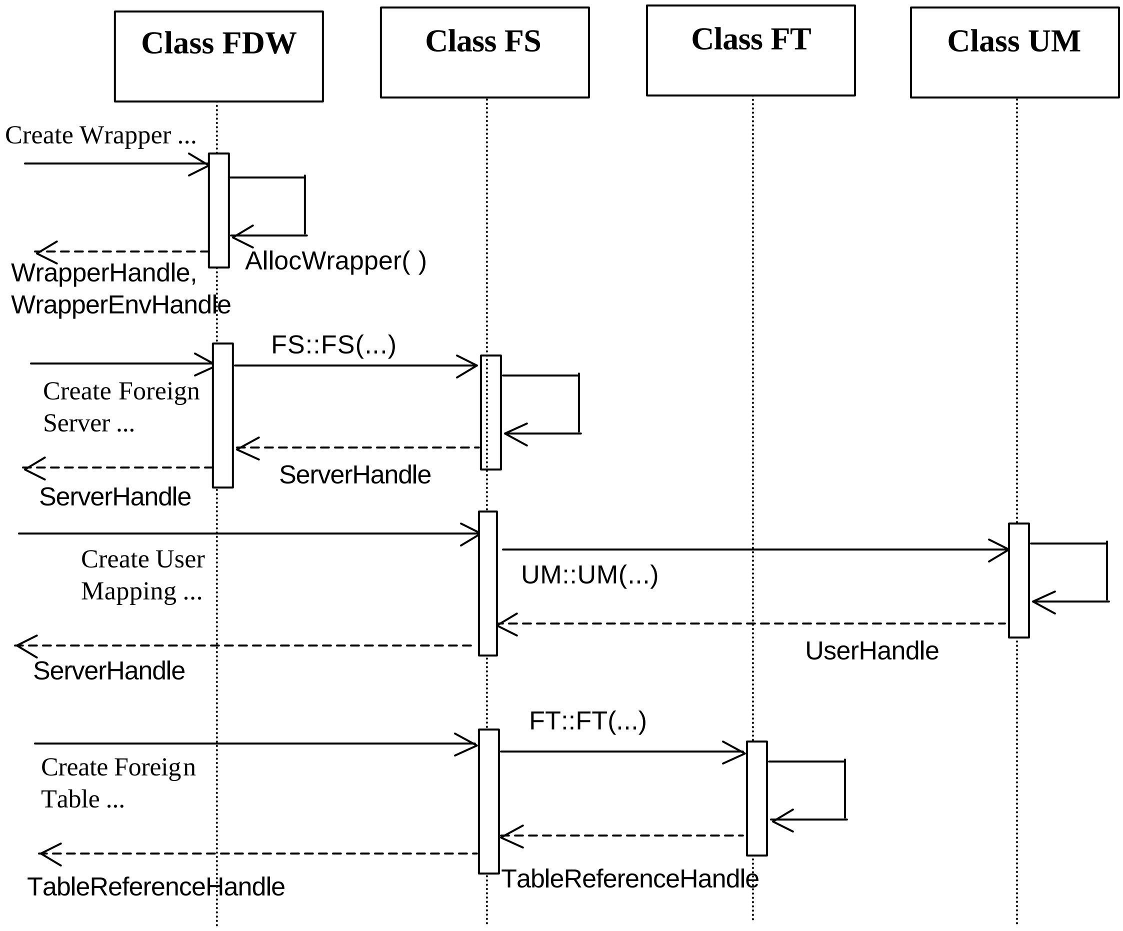

Figure 5.11: FDBMS bootstraps wrapper class hierarchie

The partial program flow for an object oriented wrapper class hierarchie in Figure 5.11 starts with "Create Wrapper …", the result of which is the constructor of FDW is called to fulfill what routine AllocWrapper(…) is expected to do. Class FDW returns WrapperHandle and WrapperEnvHandle to FDBMS.

With WrapperHandle FDBMS approaches object of class FDW to create an instance of class FS. FDW object will invoke constructor of FS and gets a ServerHandle in return, which it passes to FDBMS. The same instantiation process goes on with the same pattern when FDBMS is about to execute commands of "Create User Mapping …" and "Create Foreign Table …", with FDBMS's entry directly to FS object through a psuedo-routine getForeignServer(ServerHandle).

At the end of this bootstrapping process, FDBMS manages to build a tree of objects, with FDW at the top, FS at the second level hanging on to FDW, FT and UM at one level further down.

Classes incoorperates related routines as their member function and handles as member data. Take Class FS as an example (see Figure 5.11). It will contain its own descriptor as data member, as well as UserHandle, TableReferenceHandle, all these are set as private. Routines are enbodied in the FS class to provide read access to these private member data or to instantiate UM and FT objects.

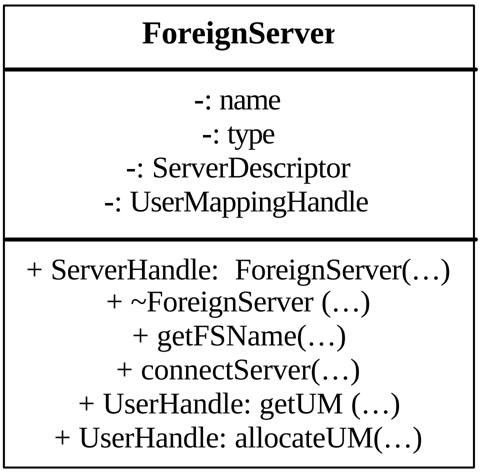

Figure 5.11: pseudo-UML presentation of class ForeignServer

5.3.2 Data mapping between wrapper and WTA

The data mapping between wrapper and WTA happens in two separate processes. First wrapper need to extract from Server Parameter Descriptor (SPD) the parameter values needed by WTA to retrieve data WFMS. SPD is filled by FDBMS with its items fields containing the query parameter(s).

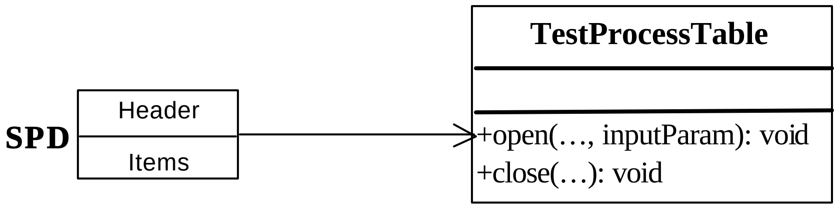

Figure 5.12: Mapping SPD from wrapper to WTA as input parameter

Each SPD has two parts, header and items descriptor fields. Header fields contains the meta data of items field, the latter saves information like type, location of an item. If there is multiple items in the item fields, header fields will indicate the items number in its “COUNT” field.

Wrapper extracts the value and type of the parameter(s) from SPD, invokes WTA to open WAPI with the parameter(s) (see Figure 5.12).

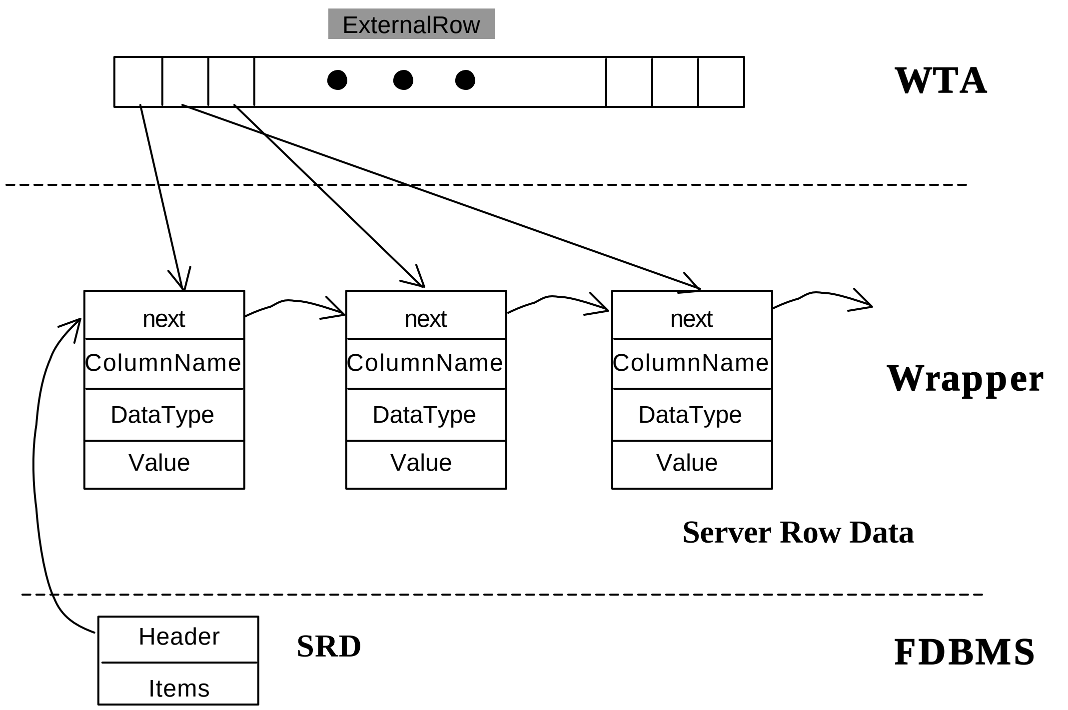

Figure 5.13: Data mapping from wrapper through SRD to FDBMS

The second data mapping takes place from the retrieved data in WTA to wrapper. The retrieved data from workflow container is saved in the object of ExternalTable. Wrapper will have to extract them from ExternalTable and write to Server Row Descriptor (SRD), which is per SQL/MED the designated carrier of data from wrapper to FDBMS.

During the open routine of WTA, the object ExternalTable is filled in with ExternalRow objects, which contains a list of ExternalCol objects. When the retrieval between WTA and MQWF ends, WAPI will be closed and WTA notices Wrapper that opening MQWF is completed successufully. Wrapper fetches one ExternalRow from WTA, going through the list of ExternalCol. Wrapper picks up those ExternalCol objects that are chosen in the SQL-statement. If a SQL-statement goes as “Select Col1, Col3 FROM …”, wrapper will skip the second ExternalCol. The chosen ExternalCol objects will be manipulated to extract the data type and value that is saved in ExternalDataTypes structure within ExternalCol object.

The data type and value extracted will be saved as a structured list of *Server Row Data* in wrapper. The list of Server Row Data is dynamically allocated by wrapper. It has the size of the number of the columns selected, each Server Row Data for one column. Server Row Data has at least four members, one for type, one for value that are extracted from ExternalCol, a pointer to the next Server Row Data, and one storing the identification of the selected column.

When a list of Server Row Data is filled out, wrapper set the data-pointer field[8] of SRD item fields to the top of this list, and runs down the list so that each Server Row Data is passed to FDBMS through SRD.

Then wrapper clears the content in the Server Row Data list, takes the next ExternalRow in ExternalTable and restarts the process again.

As all ExternalRow objects are processed and delivered, wrapper writes a "end_of_data" symbol to SRD, which triggers FDBMS invoking close routine on wrapper.

5.3.3 WTA as a component of wrapper

WTA is integrated to the general wrapper structure as a component with two major functions mentioned above, data mapping and accessing WFMS. The complete wrapper structure will include three wrapper components [RW01] (see Figure 5.14).

In the Error handling component a standard exception catching and throwing norm through return code of routines is described after SQL/MED standards.

Each of the following routine codes indicate one possible scenario of routine execution [SQL00]:

- 0 (zero) indicates **Success** of routine execution,
- 1 indicates **Success** of routine execution but a **warning** is raised
- 100 indicates **Success** of routine excution but **no data** is found

[8] This implementation is only specified for wrappers written in pointer-supporting languages, like C, C++, Ada.

- -1 indicates Error, which results from unsuccessful routine execution (error condition: other than *"wrapper-specific conditoin – invalid handle"*, e.g., data conversion overflow)
- -2 indicates Error, which results from unsuccessful routine execution (error condition: *"wrapper-specific conditoin – invalid handle"*, e.g., handle not intialised before invokation)

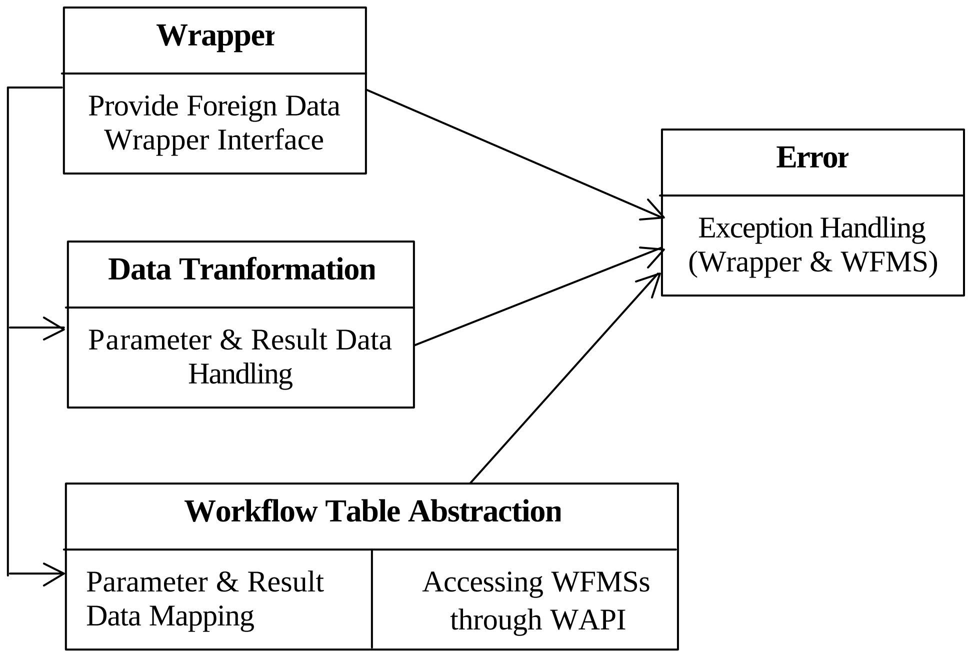

Figure 5.14: Component structure of WTA-Wrapper

Wrapper component is the core of workflow Wrapper, incorporating functionality routines and interfaces laid out by SQL/MED.

Data Transformation Component deals with data exchanges (writing input parameters and retrieving results) between FDBMS and wrapper, while the *Parameter & Result Data Mapping* part of WTA deals with the data exchanges between WTA and WFMS.

Error handling routines accompany the whole process to indicate execution status of each routine.

Chapter 6

Performance Issues of Workflow Wrapper

6.1 Introducing UDTF approach to WFMS integration

The integration of MQWF management system can be achieved through different techniques. As the alternative to the wrapper approach discussed in the work, *Table Function* approach is often used to access SQL-foreign data sources too [HH00]. This approach is called user-defined function (UDF) in SQL3.

It is also implemented in IBM DB2 UDB and is called user-defined table function (UDTF). An example of creating a UDTF in UDB is as the following:

```
CREATE FUNCTION findPerson ( inName
VARCHAR(20) )
        RETURNS TABLE (
                IdNr INTEGER,
                LastName CHAR(20),
                FirstName CHAR(20) )
        EXTERNAL NAME 'findPersonUDTF.dll'
```

In this example, a function findPerson is defined by the user to return a table to UDB that has three columns: IdNr of data type INTEGER, LastName and FirstName of data type CHAR(20). The function takes a parameter called inName. When the function is invoked, the executable module findPersonUDTF.dll written in C shall be invoked and the value of inName passing as the parameter to the module.

A registered UDTF can be used in UDB SQL-Operation, as the table objects it returns can be referred just as any relation in UDB. Using the example above a SELECT-statement may look like this:

```
SELECT F.IdNr, F.FirstName, F.LastName
FROM Table( findPerson ('Smith') ) as F
```

In this statement, as UDB encounters the registered UDTF findPerson it invokes the executable associated to it, findPersonUDTF.dll, which returns a table object. The table object is stored under the name “F”. Finally SQL-operations can be performed on “F”.

```
void SQL_API_FN oneUDTF (
SQL_PARAMETER,
SQL_RETURN_VALUES,
…
SQL_INFO)
{
        switch( SQLUDF_CALLT)
        {
                case SQLUDF_TF_OPEN:
                        …; //implementation
break;
                case SQLUDF_TF_FETCH:
                        …; //implementation
break;
                case SQLUDF_TF_CLOSE:
                        …;break;
        }
}
```

Figure 6.1: Excerpts of a UDTF module

UDTF integrates WFMS to FDBMS by encapsulating details of accessing WFMS in its executable module. In a UDTF executable module, the typical main function has the pattern in Figure 6.1. SQL_API_FN is the macro defined by UDB that specifies the calling convention for a UDTF or stored procedure. Thus the parameters of the macro and inner structure of the function are preset by UDB. For example, the switch condition

SQLUDF_CALLT searches automatically in the parameter SQL_INFO on what types of function call is: open, fetch or close.

By invoking open or fetch methods between the case statements the same WTA classes that are used by wrapper can be employed by UDTF to integrate WFMS to UDB too. However between these two approaches, there are considerable qualitative differences [RW01].

Among the interesting differences are:

Standardization Wrapper interface is standardized in SQL/MED, while UDTF has yet a standardized interface for different database system providers. Thus the cross-vendor wrapper compatibility is high, making the cost for adaptation of a certain wrapper from one FDBMS to another low.

Extensibility Just as shown in Figure 6.1 UDB defines a macro as template for all UDTFs. The advantage of such a simple structure is less effort needed on programming a UDTF. The cost of integrating a workflow process is to extend a new WTA subclass by implementing its open and fetch methods suitable for the workflow container data structure, then bind it into the UDTF macro. The other side of the advantage is less extensibility of existing UDTF.

On the other hand, though wrapper has far more complicated interface and structure, as Chapter 5 shows, SQL/MED concept can be transformed into an object-oriented wrapper class hierarchie, making development based on existing wrapper and generic wrapper solution possible. With a generic wrapper structure and generic options defined in SQL/MED, a single wrapper can be made for multiple external data sources (say various workflow models). Actually in DB2 UDB a wrapper module called drda.dll is developed to integrate all database systems of DB2 family to UDB.

Query planning According to SQL/MED query execution is based on the negotiation between FDBMS and foreign data sources, in which wrapper plays as a middleman. The Reply/RequestHandle are presented by wrapper to achieve the optimal query planning. For SQL-aware foreign data sources (foreign DBS), wrapper may choose to shift part of SQL-operation workloads to foreign DBS or even the total workload by choosing "*pass-through*" mode. However since WFMS is a non-SQL data source, i.e., it can perform no SQL-operation on its data, this gap does not affect wrapper and UDTF approach of integrating MQWF into UDB.

Authentication SQL/MED supports user mapping, thus login information can be given and altered at run time, while for UDTF login must be coded in the executable module.

Wrapper has intrinsically quantitative advantages in many aspects over UDTF, much due to its complexity. In the next section a performance test is carried out to see integration potential of wrapper comes at the price of query operation speed.

6.2 A qualitative comparison of wrapper and UDTF approach

In this performance test the same workflow process is integrated separately by wrapper and UDF both using WTA.

For the UDTF approach, a table function is first defined in UDB, then a select-all statement will be issued to measure the time UDB needs to process it. For the wrapper approach Wrapper, Server and Nickname will first be created before a select-all on the nickname will be issued.

The test is carried out on the platform of an IBM notebook with Pentium III 850 MHz and 320 MB main memory. The benchmark tool provided by DB2 called *db2batch* will be utilized. db2batch reads the SQL-statements and testing variables from a script, then runs DB2 in a shell to monitor the execution process of SQL-commands in the test script. The SQL-statement in the test would be in “select * from one_table” format.

Additionally it will be interesting to find out also how much time WTA classes will need to retrieve data from MQWF, thus a clock mechanism is coded into WTA classes to both approaches (see Figure 6.2).

```
ExternalTable TestTable;
long start, finish;
int start_ms, finish_ms;

struct _timeb timebuffer;
 _ftime( &timebuffer );
 start = (long) timebuffer.time;
 start_ms = (int) timebuffer.millitm;
...
TestTable.open( );
...
_ftime (&timebuffer);
 finish = (long) timebuffer.time;
 finish_ms = (int) timebuffer.millitm;

log << "  clock elapsed in Seconds: " << (finish-start) << “,” <<(finish_ms - start_ms) << endl;
```

Figure 6.2: Pseudo-code for measuring time needed to retrieve data from MQWF

During the test it was observed that for both UDTF and wrapper, it took significantly longer to retrieve data from MQWF with newly started operating system than to repeat retrieval process without rebooting. Therefore for both approaches two test groups are set up, one under the condition that computer is newly booted before query execution while the other group containing results from WTA retrieval without rebooting. They are referred to as wrapper/UDTF “*startups*” and “*non-startups*” in the analysis.

While creating UDTF two modes are available in UDB, “UDTF *fenced*” or “UDTF *not fenced*”. For functions that are considered “safe” to run, it can be created marking as “not fenced”. For those that might be considered as unsafe, user may choose creating the UDTF in a “fenced” mode, which will result in UDB insulating some DB2 internal resources like data buffers from the UDTF. UDTF will be created in both modes and run as non-startups. For UDTF startups only “not fenced” mode is tested. Thus there are five test groups “wrapper non-startups”, “wrapper startups”, “UDTF fenced”, “UDTF non fenced” and “UDTF startups”.

Finally the test results will be statistically taken median instead of average for each test group to avoid single extreme result having an unfair influence of the whole test results.

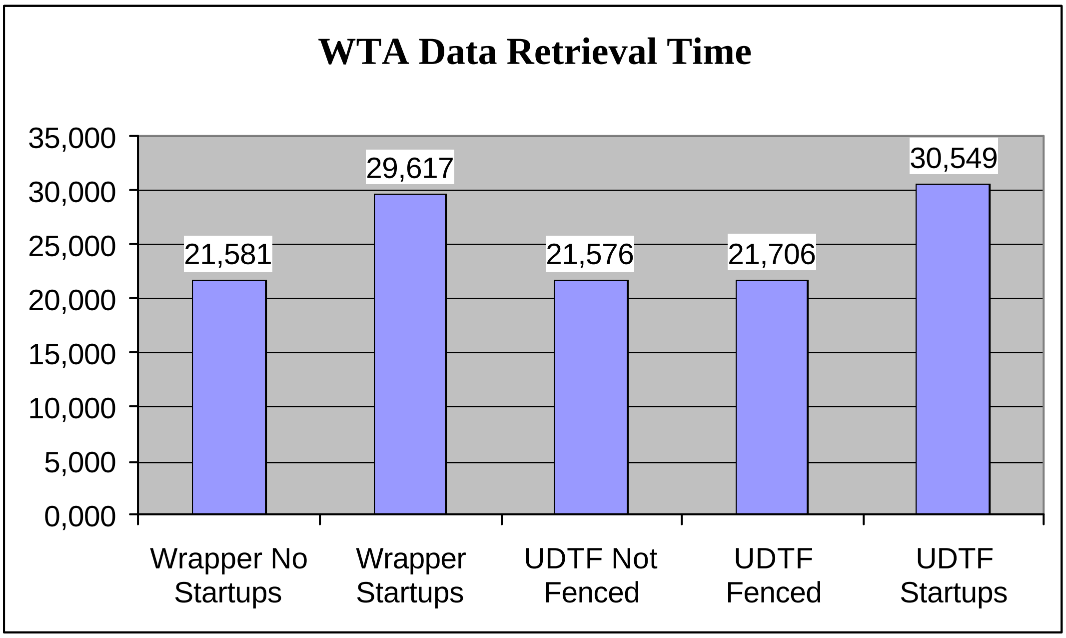

Figure 6.3: Time needed by different test groups to retrieve data from MQWF

The first test results are on the time elapsed for WTA data retrieval from MQWF. Figure 6.3 shows a clear gap between startups and non-startups for both approaches, with startups needing about 30 seconds to finish retrieving data from MQWF while non-startups needing only 21.5 seconds for doing the same work. The reason for that has very likely to do with the loading MQWF into the operating system. IBM MQSeries takes a lot of time and memory to be loaded and have its APIs ready for WTA access. Startups need to load MQWF into blank memory, a very time consuming event, during which time WTA waits for the WAPI initialization to be completed. Actually even no-startups takes a big chunk of time needed for both approaches, thus time saving techniques on workflow data retrieval will be focus of future improvement.

In Figure 6.4 total elapsed time for the select-all query statement for all five test groups are listed.

Figure 6.4 continues the trend of Figure 6.3, as the time difference between two startups get even bigger. The time spent by wrapper startups other than data retrieval, i.e. 29.803 seconds for wrapper startups in Figure 6.4 minus 29.617 seconds for wrapper startups in Figure 6.3, is 0.186 seconds. UDTF counterpart has a 2.404 seconds difference. It might be inferred that UDTF command requires more system resources, but at startup conditions those resources are occupied by MQWF loading. On the other hand three no startup groups has relatively smooth increase in execution time, which can be taken as that for no startup conditions system resources have been freed for UDTF.

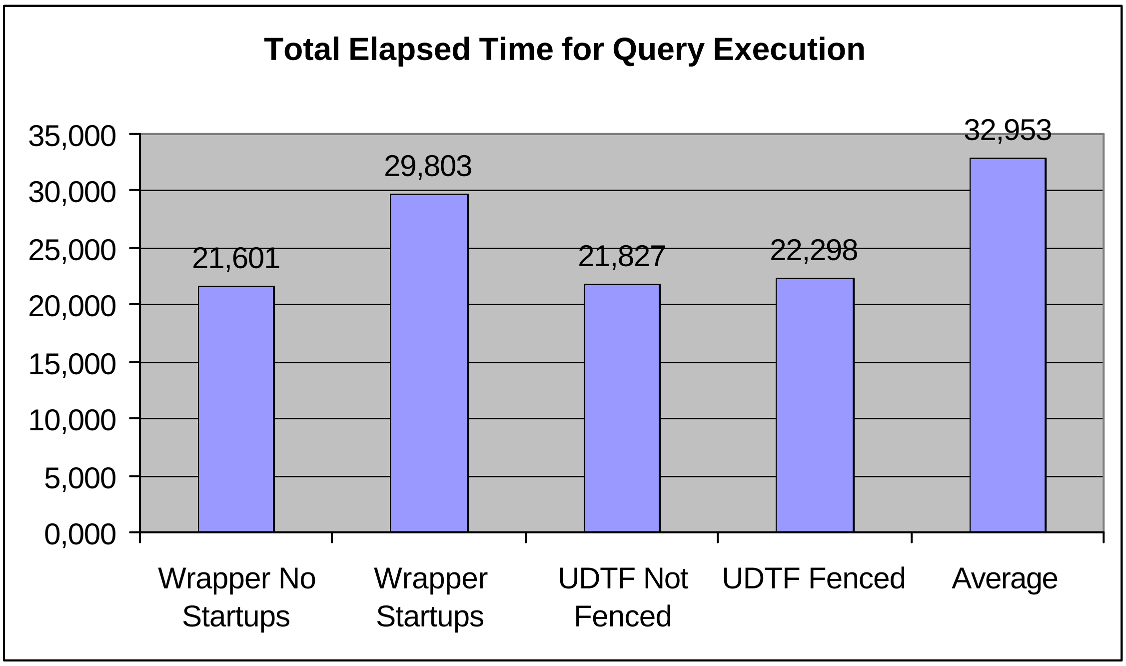

Figure 6.4: Total time elapsed for Query Execution

Time spent in Seconds	Wrapper No_Startups	Wrapper Startups	UDTF not_fenced	UDTF fenced	UDTF startups
Preparation	0,010	0,106	0,010	0,010	0,030
Execution	21,591	29,713	21,821	22,287	32,923
Fetch	0	0	0	0	0

Table 6.1: Distribution of elapsed time on query preparation and execution

The total time elapsed is made of time consumption in three phases in DB2: prepare time, execution time and fetch time. Prepare time refers to the amount of time UDB needs to prepare for the SQL-Statement, its end is marked by completion of query planning. Execution time amounts to what wrapper or UDTF needs starting from the end of query planning till the foreign data source is successfully "opened", during which period e.g. wrapper establishes a connection with the foreign server. Fetch time counts from the first set of data being fetched into UDB system till wrapper or UDTF signals to UDB that all data has been fetched and query execution has reaches its end. The workflow process on test is a rather smaller one, thus UDB spent almost nothing on "fetch", which showed zeros on all five test groups. Elapsed time on query preparation and execution in the table below will be studied.

The preparation time for three no startup groups stays at the same level of 0.010 seconds (see Table 6.1). UDTF startups needs three times for the same job vs. UDTF no startups, yet wrapper startup group need more than ten times of time that its no startups take. One explanation for this dramatic increase might be that wrapper's query preparation is more complex (involving intensive negotiation between UDB and wrapper's foreign server object), therefore more sensitive to system resources available. Startup test conditions, as argued above, reserved more system resources for MQWF initialisation.

A notice can be paid to the execution time needed by UDTF not fenced group and fenced group. With the identical copy of UDTF being created by UDB in two modes, it is evident that fenced UDTF needs more time for query execution (because of system resource insulation) than not fenced group. For query planning both groups take the same amount of time. Therefore system resources provided by UDB plays a very important role in query execution speed.

One of the most important benchmark in evaluating query execution is the *CPU time used by DB manager agent process* (Agent CPU time). It represents total CPU resources needed for a specific query statement, which excludes factors like parallel running processes or out-of-UDB calls like executable calls. It is in other words, the *net weight* of a SQL-statement that UDB has to carry.

In Figure 6.5 a clear tendency can be observed that wrappers of both groups require less CPU time to process the select-all statement than their counterparts of UDTF groups. Further even startup wrappers (worst case group for wrapper) require less CPU time than not fenced UDTF (best case group for UDTF). This is a clear evidence that wrapper has a better overall performance than UDTF while retrieving data from the sample workflow instance.

Agent CPU time can be divided into two parts, System CPU Time and User CPU Time used by Agent. System CPU represents the time spent in system calls, which is the system call overhead for execution of a SQL-statement (or a set of SQL-statement in question). User CPU represents time spent executing database manager code [IBM00].

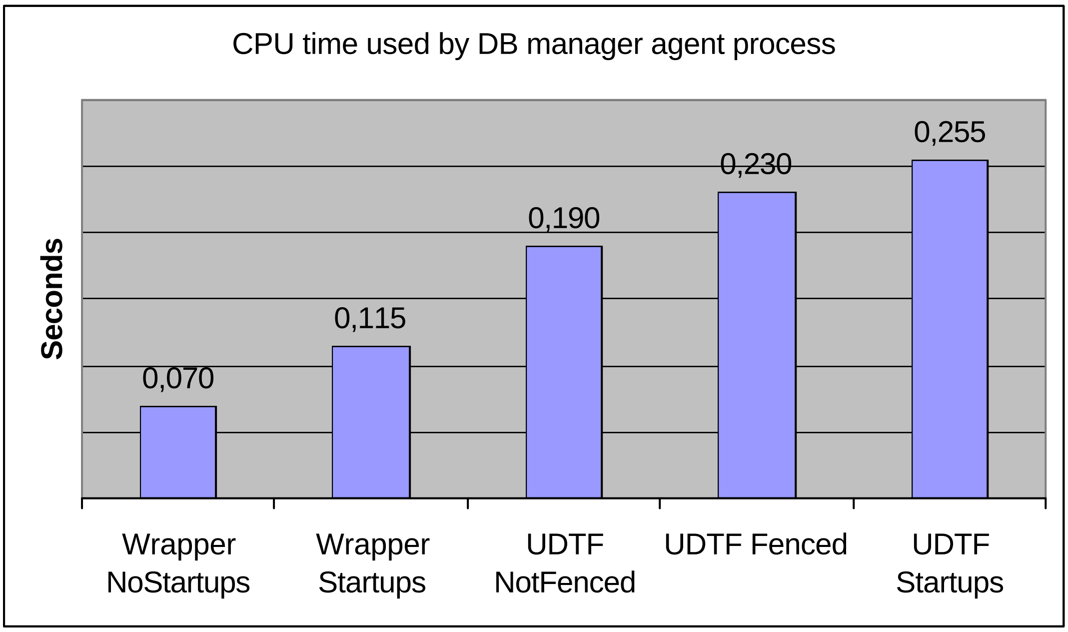

Figure 6.5: CPU time used by DB manager agent process

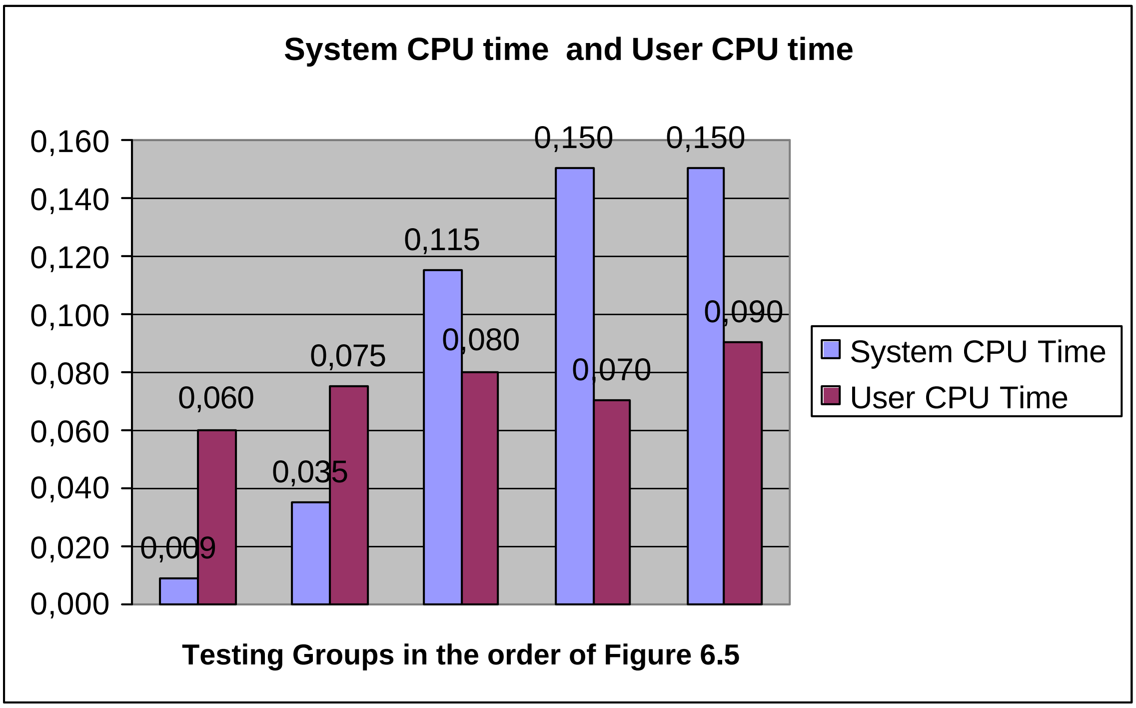

Figure 6.6: Composition of Agent CPU time in System and User CPU time

To break Agent CPU time usage and further analysis on test results, one can find another fact: CPU time devoted to UDB code varies among test groups with slight advantage to wrapper groups, but CPU time used on system calls is significantly higher for UDTF than that needed by wrappers (see Figure 6.6). Obviously UDB needs to reserve more system resources, executing more complex routines to run a SQL-statement on UDTF.

Upon the observations made above, two opinions are made about the performance comparison between the wrapper and UDTF approach.

Wrapper holds the edge over UDTF overall on query performance based on Figure 6.5 and 6.6. However due to time consuming data retrieval from MQWF, such an advantage is almost insignificant to users as evident in Figure 6.4.

Since the test is based on a light weighted workflow process, which returns only four rows of data, reservation must be made that test results may vary for data retrieval on much bigger MQWF containers. More data retrieval means more user CPU time, the bulk of which will be repeated fetch routines. Therefore the overhead cost in terms of system CPU time will get less meaningful. However, judging user CPU time needed by no startup wrappers vs. no startup UDTFs as well as startup wrappers vs. startup UDTFs, it is still safe to draw the conclusion that wrappers will do no worse than UDTFs retrieving large amount of data.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

In this thesis, work has been done to integrate workflow management to federated database system. The practical work is based on IBM DB2 UDB as the Federated Database Management System as one side, and IBM MQSeries Workflow as the workflow management system on the other side of spectrum.

Different integration plans have been done before. The benefits of such an integration is to combine the benefits of using a federated database system for access heterogeneous data and the benefits of employing a workflow system to achieve a federated function of multiple applications.

Two kinds of approach are available for such an integration. One is called user-defined table function (UDTF). It defines a function within the FDBMS that returns a table as execution results. The returned table can be referenced in the SQL-statements to FDBMS as if it were a native relation. The function is bound to a certain executable, which performs operations on foreign data sources and returns the results. UDTF has no standards, its implementation is very much dependent on its database system. At the same time, UDTF has very simple structure, involving mainly a SWITCH-CASE block from a macro template. UDTF's interaction with FDBMS is straight out of the SWITCH-CASE block to be seen, which involves only opening returned table object, fetching data out of it and closing it.

The other kind of approach is wrapper. Wrapper has a far more complex interface and complicated structure. Its interactions with FDBMS are based on numerous routines, descriptors; it views the foreign data source in a refined architecture involving servers, tables, user mappings as if it were a database (which is NOT a necessary precondition to be an accessible data source). SQL/MED has standardized the interactions between wrapper and FDBMS.

MQSeries Workflow has its standard APIs. To integrate MQWF to FDBMS the indispensable part is to access its APIs, through which workflow process will be instantiated and executed, the results can be retrieved from its output containers. A Workflow Table Abstraction was introduced in [RW01] to abstract this process and was built into an UDTF to integrate MQWF to UDB. WTA is utilized by this thesis as a component in the workflow wrapper for UDB too.

The UDTF and wrapper based on WTA are analogically the same type of tires for two different automobiles. A performance test between these two approaches is done to find out how the differences in structure and concept affect them using DB2 UDB powerful query capabilities.

The test showed that although UDTF is simple and compact, it does not turn into advantages in efficiency. Quite the opposite, wrapper based workflow integration seems to be more effectively in cooperation with UDB. This might be the result of the “light-weighted” structure wrapper was built to be run on UDB.

This opinion is far from being a solid conclusion due to limitation of test examples, nevertheless it could be deducted from the test results that UDTF shall not perform any better than wrapper if bigger workflow process involves.

Test also shows that running the test program on an ordinary PC/Notebook with just over 300 MB main memory, assessing workflow systems’ APIs takes between 20 to 30 seconds, while UDB related activities amount to a fraction of a second usually. It is attributed to the massive amount of time MQWF needs to initialise its APIs before it is to take any work from clients.

7.2 Outlook

Based on this observation[9], it is easy to believe, to improve performance of workflow wrapper, thereby improve FDBMS performance involving workflow integration, the primary target shall be to reduce time needed retrieving data from WAPI.

Since WAPI is independent from FDBMS, the efforts shall be concentrated on how to avoid accessing WAPI every time a SQL-statement involves data from workflow.

A no-brainer solution would be to save WTA-structured workflow data in the wrapper hierarchies, so that when repeated SQL-statements arrive to wrapper, the same data can be used as query return. However such an almost effortless solution is hardly a solution according to database system concepts. It uses (very likely) outdated workflow data. Besides when the workflow data retrieved is considerable sized, forcing wrapper to maintain a very large data structure, it will slow down the overall wrapper performance, causing the exact problem this “solution” tries to solve.

Another solution would be to use a proxy server in front of workflow system. Proxy server takes over the responsibility of keeping data of last access. At the same time proxy server can engage workflow system in an preset interval. Thus improve the data retrieval while keeping limit on chances of getting out-of-date workflow data.

WTA based workflow wrapper takes little use of concepts of foreign server or foreign table. Workflow data is read in during the open routine and is processed in the fetch routine. With the introduction of a proxy server, WTA routines of opening WAPI (in reality the interface of the proxy server) can be transferred from wrapper’s open routine to its routines based on the foreign server, decreasing the workload of opening foreign data source. Other WAPI-related activities are to be shed over to proxy server too. The result of such changes is the removal of virtual methods “open” and “close” meant for WAPI from the ExternalTable class.

[9] Running MQWF on more powerful machines with more memory capacities may significantly reduce the initialisation time, thereby making such a bottleneck less severe. At the same time, if a large amount of data is to be handled by workflow wrapper, improvement on data processing by wrapper will gain its significance as a performance issue too.

Until now UDTF and wrapper approaches of workflow integration requires for each single workflow model one WTA-inherited class to be written and bound to UDTF or wrapper executable. The reason is diversity of data structures that might exist in workflow data containers. For each workflow WTA has a part of it devoted to adapt workflow container data structure. An ideal solution would be to have meta data of workflow container data structure passed to wrapper and adaptation will be done at wrapper run time. It would be a generic wrapper for all workflow models.

Wrapper has build-in capability of taking generic options. It keeps them in descriptors. A generic wrapper in this case takes in nickname options, each describing data structure of one column in workflow containers. While running "open" routine wrapper converts these options into a parameter list for WTA. WTA propagates input parameter into workflow and read output data from containers according to this list. Using generic options, changes in workflow container may be reflected by calling "ALTER FOREIGN TABLE …OPTIONS (…)" SQL-command. However it might looks very clumsy if dozens of options must be given as options while creating a foreign table.

Wrapper may also read in meta data from a repository and determines workflow container data structure accordingly.

References

[BU99] Susanne Busse *et al*, *Federated Information Systems: Concepts, Terminology and Architectures*, TU Berlin 1999

[DA00] C.J. Date, *An Introduction to Database Systems*, 7th edition, Addison-Wesley 2000

[DS90] Davenport, T.H. & Short, J.E. (1990 Summer). *The New Industrial Engineering: Information Technology and Business Process Redesign*, Sloan Management Review, pp. 11-27.

[EM00] A. Eisenberg, J. Melton, *SQL standardization: the Next Steps*, 2000

[HH00] K. Hergula and T. Härder, *A Middleware Approach for Combining Heterogeneous Data Sources – Integration of Generic Query and Predefined Function Access*, Proceedings 1th International Conference on Web Information Systems Engineering (WISE) 2000.

[HH02] K.Hergula and T. Härder, *Coupling of FDBS and WfMS for Integrating Database and Application Systems: Architecture, Complexity, Performance*, Proceedings 8th International Conference on Extending Database Technology (EDBT 2002).

[HL02] L. Haas and E. Lin, *IBM Federated Database Technology*, March 2002

[IBM00] IBM DB2 Universal Database Version 7, Online Reference 2000.

[IBM01a] IBM MQSeries Workflow, *Concepts and Architecture Version 3.3.2*, 2001.

[IBM01b] IBM MQSeries Workflow, *Getting started with Buildtime Version 3.3.2*, 2001.

[IBM99] IBM MQSeries Workflow, *Getting started with Runtime Version 3.2.1*, 1999.

[LR00] F. Leymann & D. Roller, *Production workflows – concepts and techniques*, Prentice Hall 2000.

[MM01] J. Melton, J. Michels et al, *SQL and Management of External Data*, 2001

[MO01] Mohan, C. "Workflow management in the internet age", see *www.almaden.ibm.com/cs/people/mohan/*

[ÖV99] M.T. Özsu, P. Valduriez, *Principles of distributed database systems*, 2nd edition, Prentice Hall

[RH98] F. Rezende and K. Hergula. *The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways,* Int. Conf. On Very Large Data Bases (VLDB'98)

[RS97] M Roth, P. Schwarz, *Don't Scrap it, Wrap it! A Wrapper Architecture for Legacy Data Sources*, VLDB Conference Athens, Greece, 1997

[RW01] R. Wagner, *Integration of Workflows into a Federated Database System with SQL/MED*, 2001

[SL90] A. P. Sheth and J. A. Larson, *Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases*, ACM Computing Surveys, September 1990

[SQL99] ISO/IEC, *Database Language SQL, Part 9*, November 1999

[ISO02] ISO/IEC SQL Part 9: Management of External Data (SQL/MED) Jan. 2002

[WfM95] WfMC, *The Workflow Reference Model*, The Workflow Management Coalition Specification TC-1003 Issue 1.1, Jan 1995.