Hochschule Fulda
University of Applied Sciences

Prof. Dr. Peter Peinl

Fachbereich Angewandte Informatik

Distributed Database and Transaction Systems

Assignment GP (to be worked on in groups of 3 or 4)

Task: Partial implementation of a Federative Database System (FDBS)

Background information: Federative Database Systems

See also conference article: “Teaching implementational aspects of distributed data management in a practical way” by Peinl/Pape available on ResearchGate.

General description of the task: Partial implementation of a FDBS based on a set of homogenous centralized database systems (CDBS)

The overall task of this assignment is to develop a federative layer as described in the article, which implements an FJDBC interface that enables you to write programs on a FDBS. This assignment addresses the simplest FDBS scenario, an FDBS that integrates a number of homogeneous CDBS, i.e. DBS of the same type. In our case three Oracle database systems (instances) available at the university (oralv8a, oralv9a, oralv10a) will be used. All user ids available on oralv9a are also available on the other two databases. Passwords are the same as those on oralv9a.
The (domain) names of the servers are :

pinatubo.informatik.hs-fulda.de (oralv8a)
mtsthelens.informatik.hs-fulda.de (oralv9a)
krakatau.informatik.hs-fulda.de (oralv10a)

The FDBS at least has to be capable of

1. managing horizontally and vertically partitioned tables and to
2. execute typical SQL statements in a distributed environment, i.e. SQL DDL statements CREATE TABLE und DROP TABLE, DML statements INSERT, UPDATE and DELETE and a very limited subset of the SQL SELECT statement.

As a consequence, your FDBS layer has to be capable to analyze SQL statements invoked/sent by the FJDBC interface, if necessary decompose them into a number of adequate SQL statements. Thereafter, they have to be sent to the appropriate CDBS for processing. If a global query implies queries to several CDBS the results of those queries need to be consolidated into a single result set as the result of the global query. In all cases the result of your distributed statement should be identical to the same statement on a single CDBS storing all data of the federation of databases. This will be checked as part of the evaluation of the assignment.

Essential subtasks of our FDBS, among others, are:

- A Federative DB catalogue (management of the distribution schema)
- Syntax analysis of SQL statements
- Query analysis and query distribution
- Query optimization
- Result set management

General approach: Dynamic mapping from FJDBC to JDBC

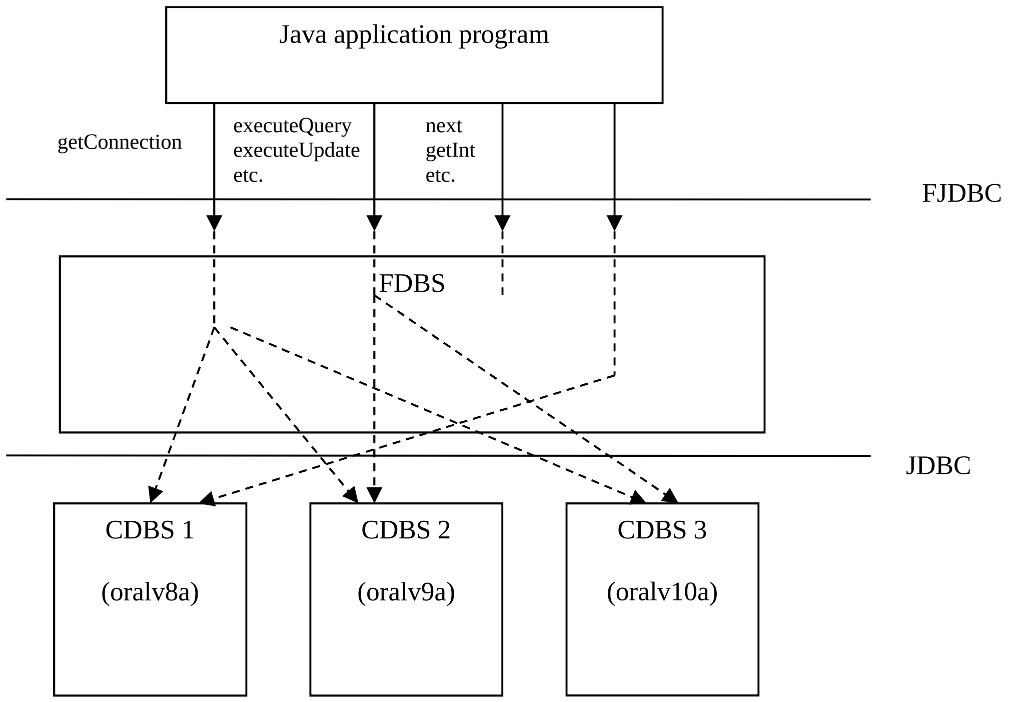
The principal interactions among an application program (a Java program invoking the FJDBC interface), the FDBS layer implementing the FJDBC interface and the JDBC interface to be invoked by your implementation are depicted in the following figure.

Method calls to the FJDBC interface have to be analyzed by your implementation of the FDBS layer and potentially be decomposed into several calls to the JDBC or completely handled in your FDBS layer.

- Imagine an INSERT that adds a new tuple to a global table, which is partitioned over three tables in the respective CDBS. The FDBS layer In that case has to determine the appropriate partition(s) and potentially decompose the original INSERT into several INSERTs into the appropriate CDBS. This equally applies to UPDATE- and DELETE statements.
- Processing a SELECT statement (query) is considerably more complex than DDL or DML statements, even if it refers to only one global table which is partitioned. Things get really complicated when a query combines several global tables via join operations. In this case all relevant partitions have to be addressed by appropriate queries.

Details have to be worked out by you before the implementation and then implemented. Likewise, meta data, required to describe the (vertical and or horizontal) partitioning of global tables over the CDBSes in the federation, need to be designed and implemented. These meta data should be stored in either one or all databases to later enable your statement parser to identify and locate all the partitions to be queried to generate the global result of a SELECT statement. The design of the format of these meat data (the catalogue of the FDBS) is part of this exercise and to your discretion.

Evidently, it is not expected that you accomplish the above mentioned tasks in a fully general and complete way. Therefore the minimum required functionality will be specified below as complete and as detailed as possible.



<u>The FJDBC interface</u> : Overview of classes and methods

Lacking a better designation I have given the interface between an application program and the FDBS the name FJDBC referring to the well known, open Java interface to a CDBS (JDBC). FJDBC should provide a subset of the JDBC functionality for a federative database environment. Whenever possible the FJDBC methods (calls) are simplified versions of the corresponding JDBC methods. A certain amount of JDBC functionality and corresponding methods have been not been included into FJDBC.

The following list comprises the names of the FJDBC classes and methods:

```
FedPseudoDriver
        getConnection

FedConnection
        setAutoCommit
        getStatement
        commit
        rollback

FedStatement
        executeUpdate
        executeQuery
        close

FedResultSet
        next
        getInt
        getString
        close
        int getColumnCount() throws FedException
        String getColumnName(int index) throws FedException
```

Two additional classes that do not exist in the JDBC framework have been included into the federal JDBC interface definitions. Their methods inform about the structure (schema) of a result set. This is a less complex solution than the JDBC class ResultSetMetaData, which need not to be implemented.

The Federated JDBC classes and their respective methods essentially should work like the corresponding elements of the JDBC interface. However, there is no need to implement a class corresponding to the JDBC class PreparedStatement. Hence, that one is missing from the previous list. A file containing class definitions needed in the assignment will be provided. Also see below.

<u>The FJDBC interface</u>: Details of classes and methods

See provided Java code of class and method definitions.

<u>Functionality of the FDBS layer to be implemented</u>: In detail

It is obvious that the implementation of a full-fledged SQL parser, i.e. one that supports the totality of SQL supported by Oracle would require an incommensurable and unrealistic amount of work. Therefore, the minimal subset of SQL to be handled by your FDBS layer is summarized and described in the following list of SQL (type) statements. You may, of course, implement additional functionality at and mention that in your project (assignment) documentation. Furthermore, you the remarks in the subsequent paragraphs point out some of the problems you may (will) encounter in the implementation of data partitioning. You need to come up with adequate solutions to each of the mentioned topics.

The list of SQL to be supported is subdivided into different types of statements.

Data Definition Language (DDL)

The DDL consists of all SQL statements, that deal with the meta data of a DBS. Meta data describe the structure and type of the data of the data seen by the direct or indirect user of the DBS. Therefore, the various CREATE, ALTER and DROP statements for tables and views belong to the DDL. However, views are not part of this assignment. The same holds for ALTER statements.

Among the data types defined by the SQL standard only INTEGER and VARCHAR, written in capital letters, have to be considered.

String constants in SQL are delimited by single quotes. Characters to be considered are from the set defined by the regular expression 0-9_a-zA-Z, table and column names likewise. SQL keywords like SELECT, INSERT, etc. are not allowed as table or column names.

That leaves the CREATE TABLE and the DROP TABLE statements.

DROP TABLE in this context is rather uncritical.

Deviating from and/or extending the SQL standard and its Oracle implementation the CREATE TABLE statement will modified for our assignment by clauses specifying the partitions of a table to be created. Both vertical and horizontal partitioning are to be supported. However, there will be no redundancy in the form of replication.

In the simplest way to partition a table horizontally is by specifying exactly one attribute (a column) and a logical expression on that attribute. Together they control the allocation of data to the members of the federation CDBSes. You will only have to implement this simple version and consequently are not asked to consider an even more flexible allocation controlled by a combination of columns and a boolean expression on the combination. Instead of a more general boolean expression on the attribute partitions will be defined as disjoint intervals of the domain of the attribute. If, for example, the attribute ZIPcode (data type INTEGER) controls horizontal partitioning and there are, as in our case, 3 CDBS, the intervals might be defined as follows (-infinity up to and including 39999, 40000 up to and including 69999, 70000 to +infinity). To reduce complexity horizontal partitioning will only be required on attributes of data type INTEGER.

As there is no standard syntax for partitioning in the SQL standard, we will define our own small extension of the CREATE TABLE statement by appending a horizontal clause to the end of the statement. The horizontal ZIPcode partitioning of the example above is formulated as follows:

```
CREATE TABLE PERS (PNR INTEGER, NAME VARCHAR(30), …, PLZ INTEGER)
HORIZONTAL (PLZ (39999,69999))
```

Syntax rules for horizontal partitioning are as follows :

```
fdbs-create-table ::= create-table-non-partitioned | create-table-partitioned
create-table-non-partitioned ::= create-table
create-table-partitioned ::= create-table HORIZONTAL (attribute(list-of-boundaries))
list-of-boundaries ::= boundary [,boundary]
```

Please note that the syntax diagram allows for a maximum of three intervals, i.e. there will be no need to allocate more than one partition to a server.

Rules create-table and attribute are the rules according to Oracle of a table definition and a columns name. Rule list-of-intervals contains at least one INTEGER constant. As can be seen in the example above each constant of the list determines the upper bound of an interval, where the implicit assumption is minus infinity for the lower bound of the first and plus infinity as the upper bound of the last interval. If the number of intervals defined is less than the number of CDBSes, then the data are to be distributed over the “first” CDBSes. If no partitioning clause is given, the entire table hast to be stored on the “first” of your CDBSes.

You only have to support integrity constraints defined explicitly after the column definitions and if they begin with the keyword constraint.

Example:

```
CREATE TABLE ABC (
A INTEGER, B INTEGER, C INTEGER, D VARCHAR(30),
CONSTRAINT ABC_PS PRIMARY KEY(A),
CONSTRAINT ABC_SK UNIQUE(B),
CONSTRAINT ABC_FS FOREIGN KEY(C) REFERENCES XYZ(W)
)
```

Keys in all constraints are simple, i.e. there is exactly one column.
You have to support constraints of types PRIMARY KEY and UNIQUE.
Support of FOREIGN KEY constraints is optional. If you decide to implement that restrict yourself to the INSERT statement. Referential integrity when deleting or updating existing tuples need not be supported.

Data Manipulation Language (DML)

The DML consists of the SQL statements INSERT, DELETE and UPDATE. Compared to the full functionality of Oracle-SQL you only need to implement the special cases outlined below.

1. Only single tuple INSERT operations are to be supported. Set oriented INSERT operations are not to be supported. Attributes will be constants, not expressions, as in the following examples!

   ```
   INSERT INTO PERS VALUES (1, 230, ’Meier’, ’Max’,43.6)
   ```

INSERT INTO PERS VALUES (2, 40, ’Kunz’, ’Max’, null)

Recall the consequences of partitioning for primary keys, candidate keys (unique) and foreign keys. Try at least to implement consistency for primary key.

2. To simplify parsing of DELETE statements only two very special cases will have to be implemented. Firstly, enable the deletion of whole tables, for example DELETE FROM TAB. Secondly, enable pinpointed deletions defined by a very simple logical expression in the WHERE clause. The expression consists of exactly one attribute, one comparison operator and one constant, as in the following examples.

   DELETE FROM PERS WHERE PNR = 17
   DELETE FROM PERS WHERE NAME = ‘Müller’
   DELETE FROM PERS WHERE BONUS > 0

3. Implementation of the UPDATE functionality in the assignment is optional. If you decide to implement UPDATE the scope will be restricted as in the case of the DELETE statement. Though SQL allows for changing multiple attributes in a single UPDATE statement (for example, UPDATE PERS SET BONUS = 50, GEHALT = 80 WHERE PNR = 23), we will restrict ourselves to exactly one column. This significantly facilitates parsing. Similar to the limitations applied to the DELETE statement above, either the whole table may be updated or the scope of the update is defined by a simple WHERE clause (one attribute, comparison operator and constant) in the UPDATE statement. Do not forget to consider changes of the value of the partitioning attribute and take appropriate actions!

Syntax rules for SQL DML statements :

fdbs-DML-statement ::= {fdbs-insert-statement | fdbs-delete-statement | fdbs-update-statement}
fdbs-delete-statement ::= DELETE FROM table [WHERE fdbs-dml-where-clause]
fdbs-update-statement ::= UPDATE table SET column = constant [WHERE fdbs-dml-where-clause]
fdbs-dml-where-clause ::= column comparison constant
comparison :: = {= | != | > | >= | < | <=}
fdbs-insert-statement ::= INSERT INTO table VALUES (constant [,constant]…)

Query Language (QL)

In general, a QL enables dynamic combination of data in the database and their retrieval by programs and applications. The QL of SQL consists of only one, yet extremely powerful, statement named SELECT. The dynamic mapping of all subclauses of the SQL select (SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY) to a federation of CDBSes would take too much effort in an assignment.

Furthermore, the complexity of an SQL SELECT essentially depends on the number of tables (FROM) and the type of combination of those tables (Cartesian product, Join, Natural Join, Outer Join, Union, Intersect, …). Imagine, for instance, a Natural Join between two tables EMP and DEPT (SELECT * FROM ABT NATURAL JOIN PERS), where both tables are each distributed over 3 CDBS. There exist several strategies to implement such a join, and you are to implement some of those at your choice.

A completely general solution as assignment is impossible because the effort and time needed would be too much Therefore, your only have to handle the following special cases.

1. The full SQL SELECT clause allow for attributes, constants and expressions of those to be output. There also can by single valued functions and aggregate functions, for example “SELECT A, B, 5, MOD(B,10), MAX(C) FROM T”. In the assignment, the SELECT clause will only contain fully qualified attribute names, For example “SELECT R.A, S.B, S.C”. The use pf aggregate functions is limited to statements with a GROUP BY clause.
2. The full FROM clause supports an arbitrary number of tables to be combined by Joins and Cartesian Products. In the assignment there will be either one or two tables in the FROM clause.
3. Furthermore, in the assignment a SELECT statement will maximally have one Join or none at all. Further, there will be no Cartesian Products or Outer Joins. Join conditions will be specified in the WHERE clause, i.e. according to the original (first) specification by the creators of the SQL language. For example, instead of the formulation “FROM EMP JOIN DEPT ON (EMP.DNO = DEPT.DNO)” the formulation will always be “FROM EMP, DEPT WHERE EMP.DNO = DEPT.DNO”.

4. Furthermore, apart from a Join condition the WHERE clause will be limited to simple attribute value comparisons which are connected either by one AND or one OR operator. You need not consider the logical NOT operator. Nested selects, set operations, etc. also need not be dealt with. Just focus on a clever implementation of the Join operation on two partitioned tables
5. The WHERE clause of a SELECT statement, by which two tables are joined will therefore be limited to the following cases. The keyword WHERE will be followed by the JOIN condition in parentheses. Sometimes the SELECT statement will have one additional non-join conditions on one or both tables. In this case the logical operator AND will always be used and the comparisons will be parenthesized.

   The following examples outline the types of statements that your federation layer should be able to execute.

   SELECT without Join
   - WHERE (R.A = R.C)
   - WHERE (R.A = 10) AND (R.D = 'Kunz')
   - WHERE (S.D = 'Kunz' OR S.E = 100)

   SELECT with Join only
   - WHERE (R.A >= S.C)
   - WHERE (R.D = S.E)

   SELECT with Join and Non-Join conditions
   - WHERE (R.A >= S.C) AND (S.D = 'Kunz')
   - WHERE (R.A >= S.C) AND (S.D = 'Kunz') AND (R.E > 100)
   - WHERE (R.A >= S.C) AND (S.D = 'Kunz') OR (R.E > 100)
   - WHERE (R.A >= S.C) AND (S.D = 'Kunz') AND (S.E > 100)

   SELECT with Group
   - SELECT R.A, COUNT(*) FROM R GROUP BY R.A

6. GROUP BY only has to be supported if there is only one table in the FROM clause. In this case there will be exactly be one grouping attribute. The only aggregate function you need to supported is COUNT(*).
7. The support of the HAVING clause is optional.
8. The support of the ORDER BY clause is optional.
9. All statements given to your federative layer will be syntactically correct.

Following are some examples of SELECT statements that your federative layer should be able to execute.

SELECT * FROM PERS, ABT WHERE (PERS.PNR = ABT.PNR) AND (PERS.NAME = 'Meier')
SELECT ANR, COUNT(*) FROM PERS GROUP BY ANR
SELECT ANR, SUM(SALARY) FROM PERS GROUP BY ANR
SELECT PERS.PNR, PERS.PNAME, ABT.ANAME FROM PERS, ABT WHERE (PERS.PNR = ABT.PNR) AND (ABT.ORT != 'Mainz')

Syntax rules for select are as follows :

```
fdbs-select ::= { fdbs-select-no-group | fdbs-select-group}
fdbs-select-no-group ::= SELECT {* | list-of-attributes | COUNT(*)} FROM list-of-tables
                         [WHERE fdbs-where-clause]
fdbs-select-group ::= SELECT table.attribute, {COUNT(*) | SUM(attribute)} FROM table
GROUP BY table.attribute
list-of- attributes ::= table.attribute [,table.attribute]…
list-of- tables ::= {table | table1, table2}
fdbs-where-clause ::= {fdbs-where-join-only| fdbs-where-join-and-non-join |
                       fdbs-where-non-joins}
fdbs-where-join-only ::= (single-join-condition)
single-join-condition ::= table1.attribute1 comparison table2.attribute2
fdbs-where-join-and-non-joins ::= (single-join-condition) AND (non-join-conditions)
fdbs-where-non-joins ::= (non-join-conditions)
comparison ::= = {= | != | > | >= | < | <=}
non-join-conditions ::= (non-join-condition) [{AND | OR } (non-join-condition)]
non-join-condition ::= table1.attribute1 comparison constant
constant ::= {integer-constant | string-constant}
```

Though the above statements look simple in the case of a centralized DBS, the task to build a united federated result set (class FedResultSet of the FedInterface) for all the partitions in the FDBS is not easy, especially if there is also a Join. The calculation of the by the federative layer should be achieved in a reasonable amount of time, even if the tables contain several thousand tuples.

Data Control Language (DCL)

DCL contains all SQL statements that either enforce the right of access to data (GRANT, REVOKE) or transaction control (COMMIT, ROLLBACK, SAVEPOINT) and in a wider sense triggers. Those are not to be implemented in the assignment. However, there are methods Commit and Rollback of the FedConnection class.

Test and acceptance of the assignment: In detail

In order to test your own implementation adequately, I recommend that you write the following short programs:

1. Write a simple program that reads a string of characters (containing the SQL statement to be tested) from the console, invokes your FJDBC-implementation and prints the results on the console.
2. Write a simple program similar to the previous assignment that creates and loads a large global distributed table with several thousand tuples.
3. Write one or more simple programs that execute queries on big tables and measure the response time

To validate your implementation we will run a benchmark of SQL statements on your federative layer, check the correctness of the results and measure the response times. Your code will be linked to programs equivalent to programs (see above) provided by us. SQL statements, especially the SELECT ones, to be tested will be published some time before the tests.

Protocol (documentation) of operations within the FDBC layer: In detail

In order to validate your implementation please log the processed statements in a file (and via console optionally). The file should be named fedprot.txt and created on the program's start. In this file all received statements and all resulting JDBC queries to the CDBSes should be logged. Output should be formatted roughly according (similar) to the following example. The exact time (up to a granularity of milliseconds) has to be printed at the beginning of each line.

```
<12:10:23.100> Start FDBS
<12:10:23.101> Connect 1 oralv9a, vdbs24
```

```
<12:10:23.103> Connect 2 oralv8a, vdbs24
<12:10:23.104> Connect 3 oralv10a, vdbs24
<12:10:23.105> Received FJDBC: Insert into pers values (12, ‘Meier’,63001)
<12:10:23.105> Sent oralv8a: Insert into pers values (12, ‘Meier’,63001)
<12:10:23.106> Received FJDBC: Insert into pers values (45, ‘Mehler’,29556)
<12:10:23.106> Sent oralv9a: Insert into pers values (45, ‘Mehler’,29556)
<12:10:23.107> Received FJDBC: Insert into pers values (99, ‘Zehner’,81324)
<12:10:23.107> Sent oralv10a: Insert into pers values (99, ‘Zehner’,81324)
```

Feel free to log also additional information and details but keep the output human-readable.

Documentation of your software

Design and implementation shall be documented, especially the design decisions.

The documentation at least has address the following topics:

1. Name and matriculation number of each group member.
2. Contributions to the common solution listed by each group member; every member has to do some part of the coding.
3. Systems architecture, i.e. components, verbal description of the tasks/functionality of the components, structure of the system, interaction among the components. Some well-designed diagrams may be helpful.
4. The functionality of each component ought to be explained verbally and in diagrams. This includes first and foremost the discussion of problems that arose during the transformation of federal statements into equivalent statements to the centralized DBSes. Examples describing the handling of federative constraints, the decomposition of queries, the implementation of distributed joins are very welcome. Where it helps to improve intelligibility you may use pseudo code to explain essential aspects. Complete code of classes or unimportant details are not wanted here. If you have tried to optimize the execution of distributed SQL statements, you are welcome to document your approaches, ideas and result of this effort. A prominent example is the acceleration of joins in a distributed environment.
5. If you have made your own tests on big tables with the software and measured response times you may describe the tests, discuss the response times and try to explain them.

The documentation need not comprise:

1. The complete code or part of it, except as an annex (Jar).
2. Detailed descriptions of the code, i.e. UML-diagrams class diagrams etc.