Hochschule Fulda
University of Applied Sciences

Prof. Dr. Peter Peinl

Fachbereich Angewandte Informatik

Distributed Database and Transaction Systems
Verteilte Datenbanken

Group assignment (groups of 6 or 7 members)

Task:
Partial implementation of a Federative Database System

Background information: Federative Database Systems (FDBS)

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

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

As a consequence, your FDBS layer has to be capable

- to analyze global SQL statements invoked/sent by the FJDBC interface,

- if necessary decompose them into a number of adequate SQL statements and
- invoke the appropriate CDBS for local processing.
- If a global query implies necessitates local 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.

An application program (in our case written in Java) invoking the FedInterface must not see and not be affected by the distribution of the data over the underlying CDBS. Therefore, there is only one global database schema defined by a set of CREATE TABLE statements. The distribution of those global tables over the CDBS is defined by the horizontal and the vertical partitioning rules. The local CDBS schemata have to be managed by you federation layer, i.e. the implementation of the FedInterface.
As a consequence, any statement invoked by an application program via the FedInterface has to produce identical results as the same statement executed by 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 and local schemata)
- Syntax analysis of SQL statements
- Query analysis and query distribution
- Query optimization
- Result set management

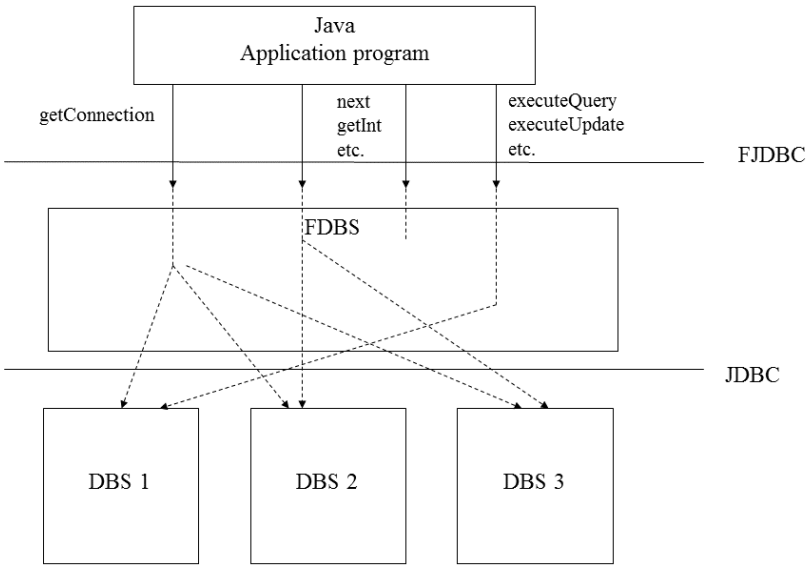
General approach: Dynamic mapping from FJDBC to JDBC

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.

Calls to the FJDBC interface have to be analyzed by your implementation of the FDBS layer. In most cases the need to be decomposed into appropriate JDBC calls to one, several or all CDBS. However, some calls can be handled completely in the FDBS layer without invoking the CDBS.

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. Likewise, meta data, required to describe the (vertical and or horizontal) partitioning of global tables over the CDBS 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.

The FJDBC interface : 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 needs 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 Zip-file containing classes and interface definitions of the FedInterface will be made available. For details, also see below.

The FJDBC interface: Details of classes and methods

See provided Java code of class and method definitions.

Detailed description of the functionality of the FDBS layer

To be implemented as group assignment

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.

Keywords and names of identifiers for tables, columns, etc. will always be written in capital letters. However, your implementation may also support small letters.

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

String constants in SQL are delimited by single quotes. Characters to be taken into account 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 no valid 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.

Rule *fdbs-create-table* defines a simplified version of the SQL CREATE TABLE statement with two optional extensions, the *horizontal-clause* and the *vertical-clause.*

Rule *create-table* further defines the subset of the SQL CREATE TABLE statement to be supported by the implementation of the FedInterface. Only the data types INTEGER and VARCHAR need to be supported. There may also be a DEFAULT specification. Standard SQL also allows to specify NOT NULL for an attribute. In our implementation this will be handled in the constraints definition part via a check constraint, for example CHECK (NAME IS NOT NULL).

Not taking into account the partitioning clauses, the following CREATE TABLE examples are consistent with the above grammar.

Examples:

```
CREATE TABLE ABC (
A INTEGER, B INTEGER, C INTEGER, D VARCHAR(30),
CONSTRAINT ABC_PK PRIMARY KEY(A)
)

CREATE TABLE ABC (
A INTEGER, B INTEGER DEFAULT(0),
C INTEGER, D VARCHAR(50) DEFAULT('To be or not to be this is the question'),
CONSTRAINT ABC_PK PRIMARY KEY(A),
CONSTRAINT ABC_B_NN CHECK(B IS NOT NULL)
)
```

Syntax rules for CREATE TABLE including horizontal and vertical partitioning:

fdbs-create-table ::= create-table {horizontal-clause | vertical-clause}
horizontal-clause ::= HORIZONTAL (attribute-name(list-of-boundaries))
vertical-clause ::= VERTICAL ((list-of-vertical-attributes),(list-of-vertical-attributes) [,(list-of-vertical-attributes)])
create-table ::= CREATE TABLE table-name (list-of-attributes , list-of-constraints)
list-of-attributes ::= attribute [,attribute]…
list-of-vertical-attributes ::= attribute-name [,attribute-name]…
attribute ::= attribute-name attribute-type [DEFAULT({integer-constant | string-constant})]
attribute-type ::= {INTEGER | VARCHAR(length)}
list-of-boundaries ::= boundary [,boundary]
boundary ::= {integer-constant | string-constant}
length ::= integer-constant
list-of-constraints ::= primary-key-constraint [,unique-constraint]… [,foreign-key-constraint]… [,check-constraint]…
primary-key-constraint ::= CONSTRAINT constraint-name PRIMARY KEY(attribute-name)
unique-constraint ::= CONSTRAINT constraint-name UNIQUE(attribute-name)
foreign-key-constraint ::= CONSTRAINT constraint-name FOREIGN KEY(attribute-name) REFERENCES table-name(attribute-name)
check-constraint ::= CONSTRAINT constraint-name CHECK ({not-null | between | comparison})
not-null ::= attribute-name IS NOT NULL
between ::= attribute-name BETWEEN constant AND constant
comparison ::= {attribute-attribute-comparison | attribute-constant-comparison }
attribute-attribute-comparison ::= attribute-name comparison-operator attribute-name
attribute-constant-comparison ::= attribute-name comparison-operator constant
comparison-operator ::= { < | <= | > | >= | = | != }
fdbs-drop-table ::= DROP TABLE table-name [CASCADE CONSTRAINTS]

Kommentiert [PDPP1]: Metaklammern {} aus Konsistenzgründen einmgefügt

Kommentiert [PDPP2]: Der Vollständigkeit halber die Syntax für ein DROP TABLE eingefügt. Kein Unterschied zum SQL-Standard

Rule *horizontal-clause* defines the horizontal partitioning of the table. 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.

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.

Horizontal partitioning means cutting the table into pieces of entire tuples (rows). Those partitions are then allocated to different CDBS. The simplest way to specify horizontal partitioning is to choose exactly one attribute (a column) and specify a logical expression. Together they control the allocation of data to the members of the federation CDBS. You will only have to implement this simplest version. Consequently are not asked to consider even more flexible allocation procedures controlled by a combination of columns and a boolean expression on the combination. Instead of general boolean expression on the attribute, partitions will be defined as disjoint intervals of the domain (i.e. INTEGER or VARCHAR) of the attribute.

If, for example, the attribute ZIPcode 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). The horizontal ZIPcode partitioning of the example above is formulated as follows:

Example:

```
CREATE TABLE PERS (
        PNR INTEGER, NAME VARCHAR(30),
        STATE VARCHAR(2), PLZ INTEGER,
        CONSTRAINT PERS_PK PRIMARY KEY(PNR)
)
HORIZONTAL (PLZ (39999,69999))
```

Assuming that states are identified by two letter codes, three partitions might be defined as follows. First partition states AA to CT, second partition CU to TX, third partition TY to ZZ.

Example:

```
CREATE TABLE PERS (
        PNR INTEGER,   NAME VARCHAR(30),
        STATE VARCHAR(2), PLZ INTEGER,
        CONSTRAINT PERS_PK PRIMARY KEY(PNR)
)
HORIZONTAL (STATE ('CT','TX'))
```

Rule *list-of-boundaries* contains at least one 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 CDBS, then the data are to be distributed over the "first" CDBS. If no partitioning clause is given, the entire table hast to be stored on the "first" of your CDBS.

Rule *vertical-clause* defines the vertical partitioning of the table. 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 vertical clause to the end of the statement.

Vertical partitioning means cutting the table into pieces of entire attributes (columns). Those partitions are then allocated to different CDBS. Thus vertical partitions are sets of columns that are

allocated to one of the CDBS in the federation. Those sets of columns will be define in the vertical clause in parentheses. Naturally, there will be at least two partitions, otherwise the table is entirely stored on exactly one CDBS.

Each attribute of the federated table as seen by the application program will be assigned to exactly one partition. However, the implementation needs to be able to recompose the individual segments of each tuple to the complete tuple as seen by the application program

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

The following example defines a vertical partitioning of table PERS. PERS has 7 attributes, which are allocated to three partitions, i.e. the first partition consists of three attributes (PNR, NAME, FIRSTNAME), the second partition consists of two attributes (AGE, CITY), and the third partition consists of two attributes (STATE, PLZ).

Example:

```
CREATE TABLE PERS (
        PNR INTEGER, NAME VARCHAR(30), FIRSTNAME VARCHAR(30),
        AGE INTEGER, CITY VARCHAR(20),
        STATE VARCHAR(2), PLZ INTEGER,
        CONSTRAINT PERS_PK PRIMARY KEY(PNR)
)
VERTICAL ((PNR, NAME, FIRSTNAME), (AGE, CITY), (STATE, PLZ))
```

You only have to support integrity constraints defined explicitly after the column definitions. They will always begin with the keyword CONSTRAINT. In our case, there will be at least one constraint, i.e. one defining the primary key.

In our implementation you need to handle 4 kinds of constraints that exist in the SQL standard, which are defined by the rules *primary-key-constraint*, *unique-constraint*, *foreign-key-constraint* and *check-constraint*.

Careful inspection of the grammar will validate the following observations.

- For all key-related constraints, keys are simple, i.e. there is exactly one column. The syntax is standard.
- The first constraint defines the primary key.
- There is at most one secondary key (unique), and it is defined after the primary key
- If there are one or more foreign keys, they are specified following the candidate key.
- Eventual check constraints are defined last. The syntax is standard, but only three cases have to be supported, i.e. NOT NULL, BETWEEN and a simple Boolean expression that either compares two attributes or one attribute to a constant.

The following CREATE TABLE statements exemplify the various aspects of the rules pertaining to the constraints part of the grammar.

Examples:

```
/* Primary key only */
CREATE TABLE ABC (
        A INTEGER, B INTEGER, C INTEGER, D VARCHAR(30),
        CONSTRAINT ABC_PS PRIMARY KEY(A)
)

/* Primary key and one foreign key */
CREATE TABLE ABC (
```

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

/* Primary key and several foreign keys */

CREATE TABLE ABC (
        A INTEGER, B INTEGER, C INTEGER, D VARCHAR(30),
        CONSTRAINT ABC_PS PRIMARY KEY(A),
        CONSTRAINT ABC_FS1 FOREIGN KEY(C) REFERENCES XYZ(W),
        CONSTRAINT ABC_FS2 FOREIGN KEY(D) REFERENCES UVW(W)
)

/* Primary key and several foreign keys */

CREATE TABLE ABC (
        A INTEGER, B INTEGER, C INTEGER, D VARCHAR(30),
        CONSTRAINT ABC_PS PRIMARY KEY(A),
        CONSTRAINT ABC_CHK1 CHECK(B IS NOT NULL),
        CONSTRAINT ABC_CHK2 CHECK(B BETWEEN 0 AND 100),
        CONSTRAINT ABC_CHK3 CHECK(B > C)
)
```

Meta data

As mentioned above, 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. Typical examples of meta data of relational DBS are the names of tables and columns, the data types of the columns, the names and types of the constraints, etc.

As is the case with the commercial relational database system used in this assignment, Oracle, the collection of the meta data is often called the catalogue or the data dictionary. Most DBS organize meta data in the same way as the organize the user data, i.e. in the form of tables. Most of those meta data are visible to the user, but they certainly cannot be changed directly by the user program. However, every DDL statement changes the meta data in the catalogue. For example, a statement that creates a new table XYZ will insert the name of this new table into “the table of tables” in the catalogue. In Oracle this “table of tables” is called USER_TABLES.

Your implementation of the FedInterface will also need a certain amount of meta data, for instance to permanently store information about the partitioning schema. Those meta data will have to be maintained by your FedInterface implementation. You should organize this meta data in the same way as the DBS, i.e. as tables. The schemata and number of those Federal catalog tables is part of the assignment. Those tables have to be created as part of your federative layer and have to be stored in the Oracle database as well.

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.

INSERT statements

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

2. Recall the consequences of partitioning for primary keys, candidate keys (unique) and foreign keys.

DELETE statements

3. 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
```

UPDATE statements

4. 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-name comparison-operator constant
comparison-operator :: = {= | != | > | >= | < | <=}
fdbs-insert-statement ::= INSERT INTO table-name 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 sub-clauses of the SQL select (SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY) to a federation of CDBS 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 allows for attributes, constants and expressions to be output. There are also 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 of 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.
6. Furthermore, a SELECT statement counting the number of rows in a table needs to be supported.

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

SELECT count all table
- SELECT COUNT(*) FROM R

SELECT all table
- SELECT R.A, R.C FROM R
- SELECT * FROM R

SELECT without Join (WHERE clause)
- 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 clause)
- WHERE (R.A >= S.C)
- WHERE (R.D = S.E)

SELECT with Join and Non-Join conditions (WHERE clause)

- 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 By (WHERE clause)

- SELECT R.A, COUNT(*) FROM R GROUP BY R.A
- SELECT R.A, COUNT(*) FROM R WHERE ((R.B = 10) AND (R.D = 'Kunz')) GROUP BY R.A
- SELECT R.A, COUNT(*) FROM R WHERE (R.B = S.C) AND ((R.D = 'Kunz')) GROUP BY R.A
- SELECT R.A, COUNT(*) FROM R WHERE (R.B = S.C) AND ((R.D = 'Kunz') OR (R.D = 'Kunz')) GROUP BY R.A

SELECT with Group By Having (WHERE clause)

- SELECT R.A, MAX(R.B) FROM R GROUP BY R.A HAVING (COUNT(*) > 2)
- SELECT R.A, SUM(R.B) FROM R WHERE ((R.B = 10) AND (R.D = 'Kunz')) GROUP BY R.A HAVING (COUNT(*) > 5)
- SELECT R.A, COUNT(*) FROM R WHERE (R.B = S.C) AND ((R.D = 'Kunz')) GROUP BY R.A
- SELECT R.A, COUNT(*) FROM R WHERE (R.B = S.C) AND ((R.D = 'Kunz') OR (R.D = 'Kunz')) GROUP BY R.A

Reminder: Capital versus small letters in are handled in the following way. Keywords and identifiers (names of tables and their attributes) may be written in capital or small letters or a mix of those. For instance SELECT, select and SeLEcT are all treated as SELECT. In other words, they are all translated into capital letters before further syntax checking. The only exceptions are string literals in single quotes, i.e. 'Smith' is different from 'SMITH'. To simplify syntax processing, you might first translate all words in the statement to capital letters except those in single quotes.

Asssurance: All statements given to your federative layer will be syntactically correct.

Syntax rules for SELECT :

fdbs-select ::= { fdbs-select-count-all-table | fdbs-select-no-group | fdbs-select-group | fdbs-select-having }
fdbs-select-count-all-table ::= SELECT COUNT(*) FROM table-name
fdbs-select-no-group ::= SELECT {* | list-of-attributes} FROM list-of-tables [fdbs-where-clause]
list-of- tables ::= {table-name | table-name, table-name}
list-of- attributes ::= table-name.attribute-name [,table-name.attribute-name]…
fdbs-where-clause ::= WHERE {join-condition | non-join-conditions | join-condition AND non-join-conditions}
join-condition ::= (table-name.join-attribute-name { = | <= } table-name.join-attribute-name)
non-join-conditions ::= ((non-join-condition) [{AND | OR } (non-join-condition)])
non-join-condition ::= table-name.attribute-name comparison-operator constant
comparison-operator ::= {= | != | > | >= | < | <=}
constant ::= {integer-constant | string-constant | NULL}
fdbs-select-group ::= SELECT table-name.attribute-name, {COUNT(*) | {SUM | MAX}(table-name.attribute-name)} FROM list-of-tables [fdbs-where-clause] GROUP BY table-name.attribute-name
fdbs-select-having::= fdbs-select-group HAVING (COUNT(*) comparison-operator integer-constant)

Though the above statements look simple in the case of a centralized DBS, the task to build a combined 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, or a Join with a grouping or Join with a restricted grouping. 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, you will be given a short program (App.java) and a directory of text files that allows you to run a test suite on your implementation of the FedInterface. The test program sequentially executes SQL statements that are organized into several text files (.SQL) based on specific tasks. For example, one file (DROP.SQL) drops all existing tables of the test database. Another file (CRETAB.SQL) creates the test database schema from scratch. Furthermore, there a several files that contain SQL SELECT statements of similar complexity.

To validate your implementation we will run several other benchmarks of SQL statements on your federative layer, check the correctness of the results and measure the response times. Your code will be linked to an extended version of the program described above. SQL statements, especially the SELECT ones, to be tested will be published some time before the evaluation.

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 on the console optionally). The file should be named fedprot.txt and created at the program's start. In this file all statements invoked by the test program and all resulting JDBC queries to the CDBS should be logged. Output format roughly resemble 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, etc. 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.