Verteilte Datenbanken - Federated Database System

Marius Hackenberg [1] Alexander Krawczyk[2] Patrick Hasenauer [3] Nicolai Fröhlig [4]
Markus Betz [5] Patrick Bug[6]

Versionshistorie

Version	Datum	Autor(en)	Änderungen
0.1	10.01.18	Alle	Erste Version: Parser, Integritätsbedingungen, Metadaten-Management
0.2	01.02.18	Alle	3.3.4 FDBS-Statement-Resolver, 3.3.6 FDBS-Statement Executer, 3.3.7 FDBS-ExecuteUpdate Unifier, 3.3.8 FDBS-ResultSet Unifier, 3.4 Anpassungen im Kapitel der Metadaten

Abstract: Kurze Zusammenfassung folgt.

[1] University of Applied Science Fulda, Applied Computer Science, Leipziger Str. 123, 36037 Fulda, Germany marius.hackenberg@informatik.hs-fulda.de
[2] University of Applied Science Fulda, Applied Computer Science, Leipziger Str. 123, 36037 Fulda, Germany alexander.krawczyk@informatik.hs-fulda.de
[3] University of Applied Science Fulda, Applied Computer Science, Leipziger Str. 123, 36037 Fulda, Germany patrick.hasenauer@informatik.hs-fulda.de
[4] University of Applied Science Fulda, Applied Computer Science, Leipziger Str. 123, 36037 Fulda, Germany nicolai.froehlig@informatik.hs-fulda.de
[5] University of Applied Science Fulda, Applied Computer Science, Leipziger Str. 123, 36037 Fulda, Germany markus.betz@informatik.hs-fulda.de
[6] University of Applied Science Fulda, Applied Computer Science, Leipziger Str. 123, 36037 Fulda, Germany patrick.bug@informatik.hs-fulda.de

Inhaltsverzeichnis

1 Einleitung

Verteilte Datenbanken und -systeme sind heute in jedem größeren Unternehmen vertreten. Sie werden in modernen IT-Anwendungen benötigt und müssen dabei sehr leistungsfähig sein. Dieser Leistungsbedarf ist durch eine ausgeklügelte Verteilung der Datenbanken mit entsprechenden Datenintegritätsprüfungen umsetzbar. Hierbei stellt die sehr gute Skalierbarkeit von verteilten Datenbanksystemen einen weiteren Vorteil dar. Nötige Ressourcen können zu Spitzenzeiten hoch und wieder herunter skaliert werden. Dabei spielt die Organisation und die Verteilung der einzelnen Tabellen in verteilten relationalen Datenbanksystemen eine große Rolle. Verteilte Datenbanksysteme wirken für den Endanwender im Idealfall wie eine einzige große lokale Datenbank. Der Anwender übermittelt die entsprechenden SQL-Befehle an eine organisierende Middleware, welche diese anhand des angewendeten SQL-Dialekts und der jeweiligen Partitionierungsinformationen interpretiert und die Tabellen über mehrere lokale Datenbanksysteme aufteilt. Die Aufteilung der Tabellen wird dabei intern gespeichert und ist für weitere SQL-Befehle ausschlaggebend. Je nach Verteilung müssen die Befehle auf ihre semantische Korrektheit geprüft werden und auf die zuständigen Datenbanken aufgeteilt werden. Diese Aufteilung erfolgt auf Grundlage der internen Partitionierungsinformationen, welche von der Middleware stets aktuell gehalten werden müssen. Die anhand dieser Logik aufgeteilten SQL-Befehle werden danach an die jeweiligen Datenbanksysteme übermittelt. Diese liefern wiederum mehrere Ergebnisse, welche unter Berücksichtigung der Datenintegrität zusammengefügt werden und schlussendlich an den Anwender übermittelt werden. Zusätzlich sind Schnittstellen zur Steuerung dieser Middleware nötig. Durch diese umfassende Aufgabenstellung an die Middleware wird eine hohe Reaktions- und Leistungsfähigkeit vorausgesetzt, da hier sonst ein Flaschenhals entstehen würde. Das Umsetzen und Implementieren einer solchen Middleware ist durch die hohen Anforderungen eine nicht triviale Aufgabe. Sie wird im Master der Hochschule Fulda Fachbereich Angewandte Informatik angewendet. Die Aufgabe wird im folgenden Abschnitt detailliert erläutert und dargestellt werden.

2 Aufgabenstellung

Die Aufgabenstellung wird in Gruppenarbeit an der Hochschule Fulda Fachbereich Angewandte Informatik, Master im Wintersemester 2017/18 Modul Verteilte Datenbanken durchgeführt. Die Aufgabe stellt umfangreiche Anforderungen. Ziel ist es ein Federal Database Management System (FDBS) zu entwerfen und mit Java zu implementieren. Die Grundfunktionalität wurde bereits im vorherigen Kapitel im Grundsatz erläutert und wird hier folgend detailliert beschrieben. Im Allgemeinen soll das FDBS so implementiert werden, dass minimal eine Datenbank und maximal drei Datenbanken die Daten beinhalten. Es kommen ausschließlich Oracle-Datenbanken zum Einsatz, was zu einem homogenen FDBS führt. Die Datenbanken werden hierbei zur Verfügung gestellt und bieten für jedem Projektteilnehmer eine eigene Umgebung. Die Datenbanken sind wie folgt definiert:

1. pinatubo.informatik.hs-fulda.de:1521:ORALV8A
2. mtsthelens.informatik.hs-fulda.de:1521:ORALV9A
3. krakatau.informatik.hs-fulda.de:1521:ORALV10A

Zur Verteilung der Daten soll Standard-SQL mit einer zusätzlichen Definition für horizontale und vertikale Verteilung eingesetzt werden. Diese "horizontal/vertical-clause" kann vertikal oder horizontal definiert sein. Vertikal bedeutet hierbei, dass die einzelnen Spalten einer Tabelle auf die drei Datenbanken verteilt werden kann, wobei jede Spalte nur einmal pro Datenbank vorhanden sein soll. Horizontal hingegen zerlegt nicht die einzelnen Tabellenattribute, sondern definiert einen Wertebereich pro Datenbank für die angegebene Spalte, sodass alle Datenbanken zwar die gleiche Tabellenstruktur besitzen, allerdings die Zeilen unter Betrachtung eines Wertebereichs einer bestimmten Spalte auf zwei oder drei Datenbanken verteilt werden können.

Diese Aufteilung der Tabellen muss in einer Instanz über eine gesonderte Meta-Daten Tabelle konsistent aufbewahrt und aktuell gehalten werden. Hierzu ist somit der Entwurf und die Umsetzung einer eigenen Meta-Daten-Datenbank nötig, welche auf entsprechende DDL-Statements reagiert und die Struktur der Inhalte beschreibt. Für den Anwender ist diese Meta-Daten-Tabelle nicht einsehbar, sondern wird vom FDBS intern verwaltet und stellt somit interne Methoden zur Abfrage der Datenbankstruktur bereit. Die Meta-Daten sind unmittelbar an das Identifizieren der verschiedenen Statements gekoppelt. Um diese Information zu erhalten, an welche Datenbank welche Teil-Statement gesendet werden soll und die syntaktische Korrektheit zu prüfen, müssen die Statements zuerst geparsed werden. Hierfür ist ein SQL-Statement Parser notwendig, welcher die Befehle ausliest und in einer entsprechenden Datenstruktur - einem abstrakten Syntaxbaum (AST) - speichert.

Je nachdem, wie die Tabellen schlussendlich auf die Datenbanken aufgeteilt werden, wird eine Instanz benötigt, die auf Grundlage der Meta-Daten-Tabelle und des eingehenden Statements dieses in einzelne unabhängige Statements pro benötigter Datenbank zerlegt und vorher die Semantik validiert. Somit wird aus einem Dialekt-SQL-Statement ein ganzes Set aus unabhängigen Standard-SQL-Statements generiert.

Damit die einzelnen Statements ausgeführt werden können ist zudem eine Komponente nötig, die aktiv eine Verbindung zu den drei Datenbanken hält und diesen die Standard-SQL-Statements übermittelt, sowie das Ergebnis zurückerhält. Unmittelbar an diese Komponente wird eine weitere benötigt, welche die möglichen drei Resultsets erhält und zu einem endgültigen Resultset zusammenfasst. Das resultierende Ergebnis wird anschließend an den Anwender lesbar übergeben. Dabei ist eine weitere Anforderung, dass das Ergbnis für den Anwender aussieht, wie aus einer einzelnen Datenbank generiert.

Die gesamten internen Vorgänge sollen hierbei stets geloggt werden und somit nachvollziehbar sein. Ebenfalls sind ist auf eine ausreichende Performance des FDBS zu achten, die gemessen werden soll, sowie eine Dokumentation in welcher die Systemarchitektur und

deren Komponenten detailliert beschrieben werden. Die Umsetzung und die zu nutzenden Werkzeuge sind nicht festgelegt, sondern Teil der Gruppenarbeit. Die Gruppe darf dabei aus maximal 7 Mitgliedern bestehen, welche sich unabhängig zu den anderen Gruppen organisiert. Dazu gehört das Festlegen der gemeinsam verwendeten Werkzeuge, wie z.B. Collaboration-Tools oder das Quellcode-Versionsverwaltungssystem. Die Identifikation und interne Aufteilung der Teilaufgaben ist zusätzlich eine interne Verwaltungsaufgabe der Gruppe.

Um die Komplexität im Rahmen eines Wintersemesters zu halten, wurden bestimmte Einschränkungen für die SQL-Statements getroffen. Die wichtigsten sind folgend aufgelistet:

- Data Definition Language
  - Die Datentypen der einzelnen Spalten dürfen nur vom Typ INTEGER oder VARCHAR sein
  - Einsatz der simpelsten Variante zur korrespondierenden JDBC-Methode implementieren
  - Prepared Statement Klasse muss nicht implementiert werden
  - Es existiert eine feste Hierarchie der Datenbanken (Reihenfolge wie oben aufgelistet)
  - Alle Key Constraints betreffen immer exakt eine Spalte
  - Check Constraints enthalten nur NOT NULL, BETWEEN und einfache boolesche Ausdrücke
  - Es kann immer nur eine oder keine "horizontal/vertical-clause" existieren
  - keine "horizontal/vertical-clause" bedeutet immer die erste DB
- Data Manipulation Language
  - Inserts bestehen immer nur aus einzelnen Tupel
  - Das Update Statement ist optional
- Query Language
  - Select Statement beinhaltet nur voll qualifizierte Attributnamen
  - Aggregierte Funktionen sind limitiert für ein GROUP BY Statement
  - FROM nur auf maximal 2 Tabellen
  - Maximal ein Join in WHERE-Clause oder gar keinen
  - kein kartesisches Produkt oder äußere Verbunde
- Data Control Language
  - Nur vollständige Commits (Alles oder Nichts)

Die gesamten Anforderungen können aus der im Anhang A.2 beigefügten vollständigen Aufgabenstellung "Group assignment - Partial implementation of a Federative Database System" abgeleitet werden.

2.1 Problembeschreibung

Die Problembeschreibung ergibt sich unmittelbar aus der Aufgabenstellung und den Anforderungen an das FDBS. Eines der ersten Probleme ist eine Steuerungsmöglichkeit des FDBS für den Endanwender zur Verfügung zu stellen. Das folgende Problem ist das Auslesen der Partitionierungsinformationen, die einen SQL-Dialekt bilden. Um die Statements in Java abbilden zu können, müssen die Statements eingelesen und geparsed werden sowie in eine syntaktisch korrekte, entsprechend der definierten Sprachen gekapselte Datenstruktur überführt werden. Ein weiteres Problem stellt die interne Partitionierungsverwaltung dar. Der FDBS-Layer muss zu jeder Zeit wissen, welche Tabellen mit welchen Einschränkungen (Constraints) auf welcher Datenbank abgelegt sind. Hierfür wird eine eigene Meta-Table benötigt, welche komplett unabhängig korrekt arbeiten und sich stets aktuell halten muss. Ist diese Eigenverwaltung der FDBS umgesetzt, ist ein weiteres Problem die semantische Analyse der Statements, welche sinnvoll prüfen soll, ob ein Statement nicht nur syntaktisch korrekt ist, sondern zusätzlich semantisch Sinn ergibt. Im Anschluss kann mit weiteren Problemlösungen fortgefahren werden. Diese sind der Statement Resolver, welcher aus einem allgemeinen SQL-Statement mehrere Teil-Statements generiert. Nachdem die Teil-Statements mit Hilfe des Statement Executors an die jeweilige Datenbank übermittelt wurde, müssen die einzelnen Resultsets korrekt miteinander verbunden werden und dem Anwender korrekt und lesbar übergeben werden. Zusätzlich zur programmiertechnischen Problemdarstellung gehört das Erstellen einer sinnvollen Architektur des FDBS-Layer, sowie die Protokollierung der einzelnen Teilschritte, um diese detailliert nachvollziehen zu können. Für die aufgelisteten Problemstellungen müssen entsprechende Konzepte entwickelt werden, die eine Lösung ermöglichen. Die Lösungsansätze werden im folgenden Kapitel kurz erläutert.

2.2 Lösungsansatz

Dem Problem der Architektur wird mit einer strukturierten Klassenarchitektur begegnet, welche in Kapitel 3.1 beschrieben wird. Diese Architektur ermöglicht das Aufteilen der Aufgabenstellung in mehrere Teilbereiche mit entsprechender Schnittstellendefinition. Um ein SQL-Statement in eine geeignte Datenstruktur zur Weiterverarbeitung umwandeln zu können, wird mit Hilfe von JavaCC ein entsprechend den Anforderungen eigener SQL-Dialekt-Parser implementiert. Dieser kapselt das Statement in einen abstrakten Syntaxbaum, welcher von den folgenden Komponenten angesprochen werden kann. Eine detaillierte Einsicht zum Parser kann in Kapitel 3.3.1 eingesehen werden. Die interne autarke Partitionierungsverwaltung wird mit einer Meta-Table-Datenbank gelöst, welche

eine eigene Tabellen-Struktur besitzt. Diese ist in der Lage anhand der Analyse des AST die Partitionierungsinformationen stets aktuell zu halten. Zusätzlich bietet sie Methoden für den Statement Resolver. Detaillierter wird die Funktionsweise in Kapitel 3.3.5 erläutert. Das Aufteilen des SQL-Dialekt-Statements in mehrere syntaktisch sinnvolle wird durch den Statement Resolver und dem Semantic Validator gelöst. Diese... (Informationen zum Resolver und Validator)

Jegliche Vorgänge werden mit Hilfe der Logger-Klasse protokolliert und mögliche Fehler mit der FedException-Klasse beschrieben. Steuerungsmöglichkeiten werden über eine Properties-Datei gelöst, welche Key-Value Paare enthält. Diese können vom Administrator angepasst werden. Die PropertyManager Klasse liest diese Informationen aus und stellt sie anderen Klassen zur Verfügung.

Das Problem des Zusammenfügens von mehreren Resultsets zu einem Fed-Resultset wird... (Informationen Resultset)

2.3 Organisation und Team-Mitglieder

Mitgliedsname	Matrikelnummer	Aufgabenbereich
Alex Krawczyk	730939	Statement Resolver, Statement Executer
Markus Betz	236401	executeUpdateUnifier, MetaDaten
Marius Hackenberg	932006	Connection, Property-Manager
Nicolai Fröhlig	639186	MetaDaten
Patrick Bug	839798	MetaDaten, Statement Executer
Patrick Hasenauer	432475	SQL Parser, Semantik Validator

3 FDBS-Struktur

Beschreibung folgt.

3.1 FDBS-Architektur

Beschreibung folgt.

3.2 FDBS-Paketstruktur

Innerhalb des Projektteams wurden entsprechende Namenskonventionen eingeführt, um Namenskonflikte zu verhindern und die Übersichtlichkeit zu bewahren. Es wurde sich

auf die Lower Camel Case Schreibweise geeinigt. Um weitere Übersichtlichkeit und Namenskonflikte zu verhindern wird neben der Standard-Ordnerstruktur (Source Packages, Test Packages, Libraries,...) mit Packages in Standard-Java Namenskonvention gearbeitet, welche Klassen in Namensbereiche kapseln. Die Namenskonvention entspricht der Punkt-getrennten Angabe einer Paket-Hierarchie, angefangen beim Organisationsnamen (hier: fdbs), über den Funktionsbereich bis zur detaillierten Paketfunktion. Zur Verdeutlichung können hier die Identifier-Klassen des SQL-Dialekt-Parsers hergenommen werden, welche sich im Package *fdbs.sql.parser.ast.identifier* befinden. Somit ergibt sich eine übersichtliche Struktur, die in die Unterpakete *app* (Anwendungsbezogene Klassen und Dateien), *sql* (SQL-bezogene oder abbildende Klassen und Dateien) , *util* (Hilfsklassen und Dateien) aufgeteilt werden. Die weiteren Hierarchie-Ebenen spalten die zuvor benannte in weitere Namensbereiche auf, welche den Funktionsbereich (hier z.B. *fdbs.sql.parser.** *fdbs.sql.meta.**) und die detaillierte Paketfunktion (hier z.B. *fdbs.sql.parser.ast.clause*) definieren. Die gesamte Struktur ist in folgender Abbildung 2 einzusehen:

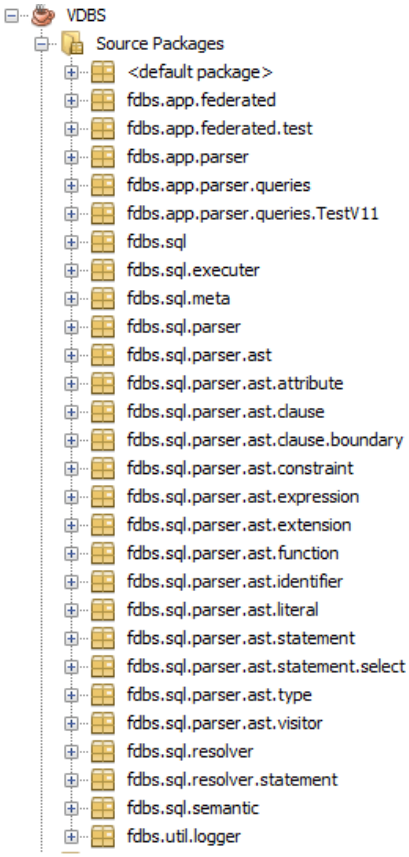


Abb. 1: Die Paketstruktur des Projekts [Stand 10.01.2018]

3.3 FDBS-Komponenten

Beschreibung folgt.

3.3.1 FDBS-SQL Parser

Der FDBS-SQL Parser ist für das Überprüfen der syntaktischen Korrektheit eines SQL Statements zuständig, der über die *executeQuery()* oder *executeUpdate()* Methode eines FedStatement übergeben wird. Als Resultat einer erfolgreichen syntaktischen Überprüfung gibt der SQL Parser einen abstrakten Syntaxbaum (AST) des SQL Statements zurück, der die Weiterverarbeitung des SQL Statements vereinfacht.

Die Implementierung des FDBS-SQL Parsers wird mithilfe des Java Parser Generators *Java Compiler Compiler (JavaCC)* und der im Auftragsdokument vorgegeben Grammatik umgesetzt. Im Anhang A.1 befindet sich ein Klassen-Diagramm, welches die Struktur des FDBS-SQL Parser sowie den Zusammenhang zwischen dem Parser, dem AST und einen AST-Visitor dargestellt.

3.3.2 FDBS-Semantic Validator

Der FDBS-Semantic Validator führt fest definierte semantische Überprüfungen sowie die Überprüfung von Integritätsbedingungen in bestimmten Fällen durch. Bei den jeweiligen Überprüfungen werden Metadaten sowie Abfragen auf einzelnen verteilten Datenbanken benötigt. Abgerufene Informationen werden während der Überprüfung an den AST vermerkt, um das erneute Abrufen dieser Informationen einzusparen und die Performance nachfolgender Komponenten zu steigern, die ebenfalls den AST für ihre Verarbeitung verwenden und die gleichen Informationen benötigen. Folgende Überprüfungen sind geplant:

3.3.2.1 Typprüfungen Die geplanten Typprüfungen beinhalten die Überprüfungen von Typen bezüglich definierten Attributtypen und angegebenen Attributwerten in Statements sowie die Überprüfung, ob Vergleichsoperatoren mit den richtigen Operandentypen verwendet werden.

- In CREATE TABLE Statements soll geprüft werden, ob der Typ bei angegeben Default-Werten mit den Attributtyp übereinstimmt.
- In einer HORIZONTAL Clause soll geprüft werden, ob der Boundary-Typ mit dem Attributtyp des angegeben Attributs in der HORIZONTAL Clause übereinstimmt.
- In INSERT und UPDATE Statements soll geprüft werden, ob die angegebenen Value-Typen mit den jeweiligen Attributtypen übereinstimmen.

- In Bedingungen von Constraints und WHERE Clause soll geprüft werden, ob die Vergleichsoperatoren mit den richtigen Operandentypen verwendet werden.

Bei den Typprüfungen ist zu beachten, dass ein bestimmter Wert von einem bestimmten Typ in einen anderen Typ automatisch von der Datenbank konvertiert werden kann. Beispielsweise kann ein INTEGER 1 in den Typ VARCHAR konvertiert werden sowie ein VARCHAR '123' in den INTEGER 123. Die automatische Konvertierung ist aber nicht in allen Fällen möglich. Beispielsweise ist die Konvertierung in einem WHERE eines SELECT Statements von einen INTEGER in ein VARCHAR nicht möglich.

3.3.2.2 Existenzüberprüfungen Die geplanten Existenzüberprüfungen beinhalten die Überprüfung von Tabellen-, Attribut- und Constraint-Namen, die in einem Statement verwendet werden. Folgende Existenzüberprüfungen sind aktuell geplant:

- In CREATE TABLE Statements soll geprüft werden, ob die Tabelle schon existiert und ob ein Attributname mehrmals definiert wird.
- Bei einem CREATE TABLE Statements mit einem Constraint soll geprüft werden, ob der Constraint-Name schon exisistiert, da er für alle Datenbanken einzigartig sein soll.
- In einer VERTICAL Clause soll geprüft werden, dass ein Attribute nicht auf mehrere Datenbanken verteilt wird und dass jedes Attribute auf eine bestimmte Tabelle aufgeteilt wird.
- In INSERT Statements soll geprüft werden, ob alle notwendigen Werte für die jeweilige Tabelle angegeben sind.
- In einem DROP TABLE Statement soll geprüft werden, ob die zu löschende Tabelle existiert.
- In allen anderen Statements soll geprüft werden, ob angegebene Tabellen- und Attributnamen existieren.

3.3.2.3 Überprüfung von Integritätsbedingungen Die geplanten Überprüfungen von Integritätsbedingungen beinhalten die Prüfung von bestimmten Fällen bezüglich eines PRIMARY KEY Constraints, UNIQUE Constraints, FOREIGN KEY Constraints und CHECK Constraints bei INSERT sowie UPDATE Statements. Bei einer Tabelle mit einer horizontalen oder vertikalen Aufteilung müssen für Überprüfung eines Constraints gegebenenfalls mehrere Datenbankabfragen getätigt werden. Welche Fällen geprüft werden müssen, sind im Kapitel 3.4.2 definiert. Folgende Punkte beschreiben was in einem bestimmten Fall für ein PRIMARY KEY Constraint, UNIQUE Constraints, FOREIGN KEY Constraint und CHECK Constraint überprüft werden muss.

- Bei einem INSERT oder UPDATE auf eine Tabelle mit einem PRIMARY KEY Constraint muss geprüft werden, ob der Primärschlüsselwert in allen Teil-Tabellen einzigartig ist.
- Bei einem INSERT oder UPDATE auf eine Tabelle mit einem UNQIUE Constraint muss geprüft werden, ob der Attributwert des einzigartigen Attributs in allen Teil-Tabellen einzigartig ist.
- Bei einem INSERT oder UPDATE auf eine Tabelle mit einem CHECK Constraint muss geprüft werden, ob die Werte des Statements gegenüber dem CHECK Constraint valide sind.
- Bei einem INSERT oder UPDATE auf eine Tabelle mit einem FOREIGN KEY Constraint muss geprüft werden, ob der angegebene Fremdschlüssel auch in der referenzierten Tabelle existiert.

Alle Überprüfungen werden mit möglichst wenigen Datenbankabfragen und möglichst wenigen Traversierungen durch den abstrakten Syntaxbaum implementiert. Die Liste der Überprüfungen wird während der Entwicklung gegebenenfalls erweitert.

3.3.3 FDBS-Property Manager

Der FDBS-Property Manager ist für das Einlesen und Speichern von Konfigurationsinformationen zuständig, die in einer separaten Datei gespeichert und von den Klassen der FDBS-Middleware verwendet werden. Diese Konfigurationsinformationen bestimmen unter anderen das Logging-Verhalten der FDBS-Middleware sowie Informationen über die einzelnen Datenbanken, auf die die Daten verteilt werden sollen (z.B. Hostnames, IP-Adressen, Ports und Zugangsdaten).

Für das Einlesen und Speichern der Konfigurationsinformationen wird die Standard-Java-Klasse *java.util.Properties* verwendet. Benötigte Methoden für das Abrufen und Setzen eines Properties werden vom FDBS-Property Manager über bereits vorhanden Methoden der Properties-Klasse bereitgestellt. Konfigurationsinformationen werden als Schlüssel-Wert-Paare gespeichert und abgerufen, wie in Ausschnitt 1 dargestellt.

```
LogLevel=INFO
Database1=jdbc:oracle:thin:%username%/%password%@pinatubo ...
Database2=jdbc:oracle:thin:%username%/%password%@mtsthelens ...
Database3=jdbc:oracle:thin:%username%/%password%@krakatau ...
```

List. 1: Ausschnitt der Konfigurationsdatei

Um mögliche Probleme hinsichtlich mehrfachen bzw. parallelen Zugriff auf die Konfigurationsdatei zu vermeiden, wird der FDBS-Property Manager als Singleton implementiert.

3.3.4 FDBS-Statement Resolver

Um die an den Parser übergebenen Statements semantisch zerteilen zu können und somit erst eine Föderation der Datenbanken zu ermöglichen, wird eine Komponente benötigt, die spezifische Objekte erzeugen kann. Die korrespondierenden Klassen enthalten dabei einen Algorithmus zur Aufteilung des globalen Statements und leiten die daraus gewonnenen Daten an eine Komponente weiter die eine Datenstruktur der Sub-Statements hält, die dann ggf. in einem nächsten Schritt eine variable Anzahl der verfügbaren Datenbanken ansprechen kann.

In unserer Architektur bezeichnen wir diese Komponente den Statement Resolver, da dieser ein globales Statement in mehrere Sub-Statements aufteilt, die direkt auf den Datenbanken der Föderation ausgeführt werden können. Dabei verwenden wir bei dem Design der Komponente das sog. Factory-Pattern, welches einen geeigneten Weg darstellt spezifische Objekte zu erzeugen ohne die Erstellungslogik zu entlarven. Ein Client kann bei Bedarf ein spezfisches Objekt über eine Schnittstelle referenzieren, welches in ihrer korrespondierenden Klasse erzeugt wird.

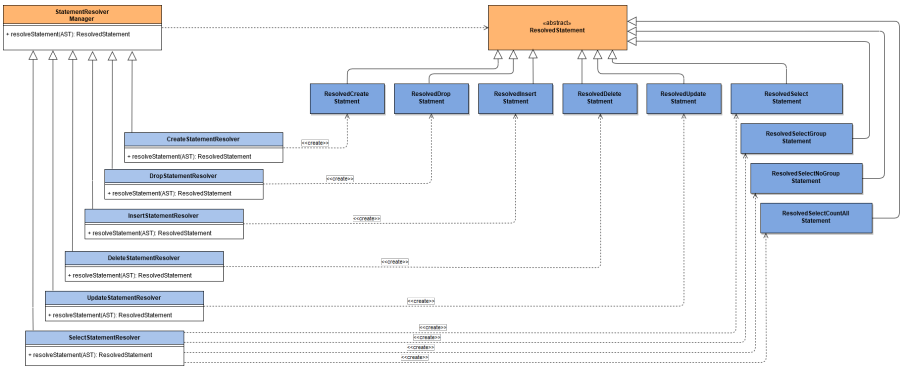


Abb. 2: Folgendes Diagramm verdeutlicht die verwendete Architektur

Die Schnittstelle dieser Komponente ist der StatementResolverManager. Ruft der Client über das FedStatementInterface die Methode executeUpdate() oder executeQuery() auf, wird der geparste AST sowie die Datenstruktur der FedConnection an den Manager übergeben. Anhand des AST entscheidet der Manager anhand seiner Factory-Logik, welche Statement Resolver Klasse das konkrete Statement behandeln soll und leitet die oben genannten Objekte dann an eine ausgewählte Statement Resolver Klasse weiter. Diese spezifische Klasse instanziiert bzw. nutzt den Metadata Manager, um auf die benötigten Metadaten zuzugreifen. Die spezifischen Resolver Klassen kümmern sich dementsprechend um die Zerlegung der globalen Statements, implementieren die Logik zur Erstellung der Sub-Statements für jede angesprochene Datenbank und sorgen für eine Weitergabe der entsprechenden ausführbaren SQL-Statements an den Statement Executer. Es gibt für jeden Typ eines Statements eine eigene Resolver Klasse und ein korrespondierendes

ResolvedStatement Objekt, dass von einer abstrakten Oberklasse ResolvedStatement erbt. In den folgenden Abschnitten werden die verschiedenen Typen detaillierter erläuter.

3.3.4.1 Create-Statement Resolver Der Create-Statement Resolver behandelt sämtliche globalen Create Statements, die an den Parser übergeben werden. Dabei instanziiert er den Metadata Manager und übergibt ihm den AST sowie die FedConnection, damit dieser die benötigen Metadaten-Tabellen erstellen kann, die im weiteren Verlauf für die Abfragen oder Manipulation der Daten innerhalb der föderativen Tabelle verwendet werden müssen. Dabei verwendet der Create-Statement Resolver verschiedene Methoden die sich mit den Einzelheiten bezüglich der Partitionierung der zu erzeugenden föderativen Tabelle und deren Integritätsbedingungen auseinandersetzen. Er sorgt dafür, dass die Tabellenstruktur abhängig von den Vorgaben zur Partitionierung eingehalten wird und die Teiltabellen die Bedingungen implementieren, die ohne ein Eingreifen des Semantic Validators möglich sind. Am Ende seiner Routine überträgt er sämtliche Sub-Statements an den Statement Executer, der diese dann auf den vorhergesehenen Datenbanken ausführt.

3.3.4.2 Drop-Statement Resolver Bei dem Drop-Statement Resolver handelt es sich um die zweite Komponente, die das Thema der Data Definition Language abdeckt. Der Resolver zerlegt dabei ein globales Drop Table Statement im Falle einer vertikalen oder horizontalen partitionierten föderativen Tabelle in geeignete Sub-Statements, die ebenfalls eine Anweisung zum Löschen sämtlicher Bedingungen beinhalten, die die referenzierte Tabelle betreffen. Diese Daten werden ebenfalls an den Statement Executer weitergegeben.

3.3.4.3 Insert-Statement Resolver Der Insert-Statement Resolver hat die Rolle ein globales Insert Statement im Falle einer vertikalen oder horizontalen Partitionierung zu zerlegen, dabei ist es wichtig dass er eine Instanz des Metadata Managers aufruft und sich somit die benötigten Informationen zum Partitionierungstyp sowie den Integritätsbedingungen einholen kann, um das globale Statement in geeignete Sub-Statements aufzuteilen und an den Statement Executer weiterzugeben.

3.3.4.4 Delete-Statement Resolver Bei dem Delete-Statement Resolver erfolgt der Ablauf der Verarbeitung ähnlich zu dem des Insert-Statement Resolvers, mit dem Unterschied, dass dieser Resolver nur im Falle einer WHERE-Klausel eine Prüfung der Integritätsbedingungen vornehmen muss. Tritt dieser Fall ein muss eine SELECT-Abfrage alle Tupel über den Primärschlüssel selektieren und das resultierende ResultSet prüfen, um eine fehlerfreie Löschung der gewünschten Datensätze zu ermöglichen, ansonsten ist einzig und alleine die Partitionierung der föderativen Tabelle entscheidend in wieviele Sub-Statements das globale Statement aufgelöst wird und anschließend an den Statement Executer übergeben werden.

3.3.4.5 Update-Statement Resolver Der Update-Statement Resolver stellt analog zum Delete-Statement einen identischen Ablauf der Verarbeitungslogik zur Verfügung und muss ebenfalls eine Prüfung der WHERE-Klausel vornehmen, falls diese angegeben wurde.

3.3.4.6 Select-Statement Resolver

Wenn ein Statement als Select Statement identifiziert wird, wird dieses von dem FDBS-Select-Statement Resolver verarbeitet. Dieser hat die Aufgabe, dass Select-Statement in einzelne Statements für die jeweiligen Datenbanken zu zerlegen. Die zerlegten Statements werden dann an die nächste Komponente für die weitere Verarbeitung übergeben. Select-Statements sind in insgesamt vier verschiedene Unterarten aufgeteilt (CountAllTable, SelectNoGroup, SelectGroup, SelectHaving). Die jeweiligen Select-Statements werden zuerst eingeordnet und danach verarbeitet. Je nach Art des Select-Statements und der Partitionierung der einzelnen Tabellen, welche von dem Select-Statement betroffen sind, verhält sich die Zerlegung und Weiterverarbeitung verschieden. Die erste Select-Art welche betrachtet wird ist CountAllTable: SELECT COUNT (*) FROM TAB. Falls die Tabelle TAB, auf die das Select-Statement ausgeführt wird, weder horizontal noch vertikal partitioniert ist, wird das Statement direkt zusammengebaut und für die Weiterverarbeitung an DB1 übermittelt. Dies gilt generell für alle Arten der hier behandelten Select-Statements. Wenn die Aufteilung von der Tabelle TAB horizontal ist, muss auf allen Datenbanken das Select-Statement ausgeführt werden. In Folge müssen die Werte der einzelnen Result-Sets später addiert werden. Ist der Typ der Partitionierung vertikal, wird das Select-Statement auf der ersten von der Partitionierung betroffenen Tabelle ausgeführt. Da die Tabelle vertikal partitioniert ist, ist die Anzahl der Reihen auf jeder Tabelle gleich und somit wird das Select-Statement nur auf einer Tabelle ausgeführt und liefert das Ergebnis. Der nächste Fall von möglichen Select-Statements ist SelectNoGroup. Der erste Teil von SelectNoGroup beinhaltet Statements wie: SELECT * FROM TAB. Betrachtet man den Fall das TAB horizontal partitioniert ist, wird das Select-Statement auf allen partitionierten Tabellen ausgeführt. Im weiteren Verlauf werden die Result-Sets nach einander in geordneter Reihenfolge bei einem FedResult.next() abgearbeitet. Bei einer vertikalen Partitionierung, wird das Select-Statement auf allen Tabellen, welche von der Partitionierung betroffen sind ausgeführt. Alle implizite geordnete Result-Sets dieser Abfragen kombiniert ergeben das Fed-Result-Set. Anstatt des All-Attribute-Identifiers können auch einzelne Attribute gesucht werden, wie SELECT Att1 , Att2 , Att3 FROM TAB. In diesem Fall ist das Verhalten des Resolvers ähnlich, wie mit All-Attribute-Identifier. Bei horizontaler Aufteilung, wird die Abfrage auf den einzelnen Partitionen durchgeführt. Diese Result-Sets werden wiederum nach einander bei einem FedResult.next() abgearbeitet. Bei vertikaler Partitionierung werden solang Abfragen getätigt bis alle Attribute von TAB abgefragt wurden. Hierbei kann es sein, dass weniger Result-Sets entstehen als mit All-Attribute-Identifier. Dies liegt daran, dass die Attribute z.B. auf nur einer Datenbank liegen können.

Es kann sein, dass die SelectNoGroup- und die SelectGroup-Statements um eine WHERE-Clause erweitert werden. In der WHERE-Clause werden die verschiedenen Join- und NonJoin-Conditions verarbeitet. Der erste WHERE-Clause Fall der betrachtet wird, ist die Erweiterung des Selects, um eine NonJoin-Condition. Bei dieser Condition wird ein Attribut gegen eine Konstante geprüft. SELECT $*$ FROM TAB WHERE ((ATT1 Operator Constant)). Bei horizontaler Partitionierung wird das komplette Select-Statement auf den einzelnen Partitionen ausgeführt und die einzelnen Result-Sets wiederum nach einander bei einem FedResult.next() abgearbeitet. Bei vertikaler Partitionierung wird das Select-Statement zuerst auf die Tabelle in der das Attribute ATT1 liegt ausgeführt. Je nachdem welche weiteren Attribute gesucht werden, wird ein Select-Statement für jede Reihe auf den anderen Tabellen bei einem FedResult.next() ausgeführt. Nun ist es möglich, dass diese NonJoin-Condition, um eine weitere NonJoin-Condition disjunktiv erweitert wird, wie SELECT $*$ FROM TAB WHERE ((ATT1 OP C) OR (ATT2 OP C)). Sollte die Tabelle TAB horizontal aufgeteilt sein, wird das komplette Statement auf den einzelnen Tabellen ausgeführt und die einzelnen Result-Sets wiederum nach einander bei einem FedResult.next() abgearbeitet. Bei einer vertikalen Partitionierung wird nun zudem auf die Verteilung der Attribute geachtet. Wenn sich ATT1 und ATT2 die gleichen Datenbank teilen, verhält sich die vertikale Partitionierung gleich wie bei einem Select-Statement mit nur einer NonJoin-Condition. Sobald sich die Datenbanken der einzelnen Attribute unterscheiden, wird das Select-Statement ohne die zweite NonJoin-Bedingung auf der Datenbank, in der ATT1 liegt, ausgeführt. Als nächstes werden alle anderen gewählten Attribute aus den vertikalen Partitionierungen bei einem FedResult.next() mittels des PK der aktuellen Reihe ermittelt und zusammengesetzt. Sobald das erste Result-Set mit FedResult.next() vollständig durchlaufen wurde, wird das gleiche mit ATT2 durchgeführt.

Sollten die Non-Join Bedingungen konjunktiv verbunden sein, ändert sich das Verhalten bei vertikaler Partitionierung. Sollten die Attribute sich auf der gleichen Datenbank befinden, wird das Statement mit der kompletten WHERE-Clause auf der gemeinsamen DB ausgeführt und die restlichen Attribute wie bei der disjunktiven Verbindung mittels des PK der jeweiligen aktiven Reihe abgerufen. Wenn ATT1 auf einer anderen Partition wie ATT2 liegt, dann wird zuerst ein Select mit der ersten Condition auf die Tabelle des ersten Attributes ausgeführt. Als nächstes wird ein Select mit der zweiten Condition auf der Tabelle des zweiten Attributes ausgeführt. Insgesamt werden somit zwei Result-Sets erzeugt, die jeweils in jeder Reihe den PK enthalten. Mit Hilfe dieses PK wird eine Überprüfung der beiden Result-Sets durchgeführt. Die konjunktive Verknüpfung der beiden Conditions ist nur erfüllt, wenn der jeweilige PK in beiden Result-Sets auftaucht. Sollten danach noch Spalten in der Tabelle fehlen, welche nicht über den Zugriff der ersten beiden Datenbanken erhalten wurden, können diese mit einem weiteren Select mittels des PKs aus den fehlenden Datenbanken geholt werden.

Neben den NonJoin-Conditions können auch Join-Conditions in der WHERE-Clause auftreten. Betrachtet man Selects wie

`Select * FROM TAB1, TAB2 WHERE ((T1.ATT1 Join-OP T2.ATT2))`, müssen die Partitionierungen von allen genutzten Tabellen betrachtet werden. Hierbei wird je nach Partitionierung anders vorgegangen. Betrachtet man den Fall das TAB1 und TAB2 horizontal partitioniert sind, wird zuerst ein SELECT * FROM TAB1 auf allen Partitionierungen ausgeführt. Durch diesen Select, über alle Partitionen von TAB1, erhält man einzelne Reihen mit dem Wert des Attributes TAB1.ATT1. Nun wird der Wert für das Attribut TAB1.ATT1 von der aktuellen Reihe mit einem weiteren Select gegen TAB2 und ihre Partitionen geprüft. Mit SELECT * FROM T2 WHERE Row.AT JOP TAB2.ATT2 werden alle passende Reihen für den Wert des Attributes TAB1.ATT1 abgerufen, die für die aktuelle Reihe des Attributes ATT1 die möglichen Joins repräsentieren und solange bei einem FedResult.next() verarbeitet werden, bis kein Eintrag in dem zweiten Result-Set von TAB2 mehr vorhanden ist. Danach wird dieser Schritt für jede Reihe im ersten Result-Set wiederholt, bis alle Reihen TAB1 abgehandelt wurden. Sollte eine Tabelle von keiner Partitionierung betroffen sein verringert sich die Anzahl der einzelnen Result-Sets die durchlaufen werden müssen. Wenn TAB1 vertikal und TAB2 horizontal partitioniert sind, ähnelt sich das Verhalten. Der erste Select auf TAB1 wird genau auf die partitionierte Tabelle ausgeführt in der das Attribut TAB1.ATT1 liegt. Zusätzlich werden nach dem Join die restlichen Spalten, welche durch die vertikale Partitionierung fehlen, für jede Reihe abgefragt und ergänzt. Bei einer vertikalen Partitionierung beider Tabellen hingegen wird genau auf die partitionierten Tabellen in der ATT1 und ATT2 liegen, die einzelnen Selects ausgeführt. Die Überprüfung erfolgt weiterhin für jede Reihe der einzelnen Result-Sets. Wenn die Bedingung in einer Reihe erfüllt ist, werden die restlichen benötigten Spalten geholt. Im Falle, dass anstatt allen Attribute nur spezielle Attribute per Select abgefragt werden, werden in beiden intern ausgeführten Selects nur die bestimmten Attribute ermittelt. Wenn einer Join-Condition eine weiteren NonJoin-Condition folgt, muss je nach Partitionierung die intern ausgeführten Selects um eine WHERE-Clause, welche die zweite Condition enthält, erweitert werden.

3.3.5 FDBS-Metadata Manager

Die Komponente FDBS-Metadaten Manager stellt grundlegenden Funktionen zu Verfügung, die für das Abrufen und Erstellen von Metadaten benötigt werden. Für die Bereitstellung der Funktionen implementiert die FDBS-Metadata Manager Klasse (MetaDataManager) folgendes Interface:

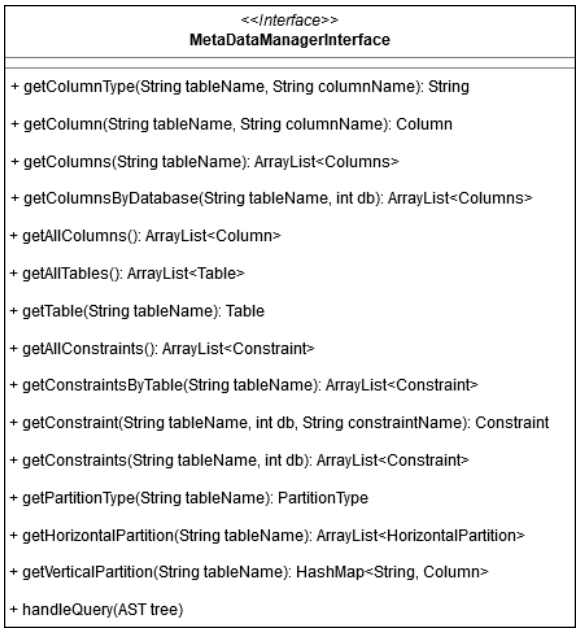


Abb. 3: Das Interface des MetaDataManager

Alle verfügbaren Methoden bis auf handleQuery werden benutzt, um Metadaten abzurufen. Dadurch können alle Metadaten der Tabellen, verschiedenen Datenbanken, Spalten und Einschränkungen ermittelt werden. Dabei werden die Informationen, die für das Interface benutzt werden, einmal beim Instanziieren ermittelt, um nur eine Datenbankabfrage auszuführen. Dadurch wird die Performance des MetaDataManagers verbessert. Die Methode handleQuery(AST tree) wird benutzt, um die Metadaten aktuell zu halten. Dieser Methode wird der Parser-Baum des ausgeführten Statements übergeben. Dann wird überprüft, welches Statement vorliegt. Bei einem CREATE-Statement werden die neuen Tabellen mit Partition, Spalten und Einschränkungen in die Metadaten eingeführt. Bei einen DROP-Statement werden die Einträge der zu löschenden Tabelle gelöscht. Dadurch können die Metadaten zu jedem ausgeführten Statement aktuell gehalten werden.

3.3.6 FDBS-Statement Executer

Der FDBS-Statement Executer führt die mit Hilfe des Resolvers generierten SQL-Statements auf den Zieldatenbanken aus. Er stellt das Bindeglied zwischen dem Resolver- und Unifier-Modul dar und kapselt die Datenbankanfragen in einer “Task”-Datenstruktur. Jede Instanz eines Tasks repräsentiert die benötigten Abfragen, die für die Umsetzung eines FedStatements benötigt werden. Da eine Abfrage sich aus mehreren Unterabfragen der zugrunde liegenden

Datenbanken zusammensetzen kann, müssen diese ebenfalls separat in der Datenstruktur abgelegt werden. Für diesen Zweck wird ein “SQLExecuterSubTask”-Objekt genutzt.
Dieses Objekt speichert die genutzte SQL-Abfrage und das resultierende Ergebnis und erlaubt die Abfrage, ob es sich um eine Update-Operation auf der Datenbank gehandelt hat. Um die Abfragen den einzelnen Datenbanken zuordnen zu können, werden die Subtasks innerhalb eines Tasks in Listen für die jeweilige Zieldatenbank gespeichert. Der Executer führt jeden SubTask eines Tasks aus, speichert die Ergebnisse in den ResultSets der Subtasks und wird im Anschluss an die Unifier-Komponente weitergereicht, die aus den erzeugten SQL-Resultsets der Unterdatenbanken das Ergebnis für das FDBS-Statement konstruiert. Hierbei können durch Nutzung von Vererbung auch weitere Unterklassen von Tasks gebildet werden, die falls nötig zusätzliche Eigenschaften besitzen, die das Auswerten der Ergebnisse vereinfachen können.
Vor allem diese Komponente wird im Laufe der Implementierung mehrere Iterationen erfahren, um die Anforderungen der vorherigen und nachfolgenden Komponenten erfüllen zu können.

3.3.7 FDBS-ExecuteUpdate Unifier

Der FDBS-ExecuteUpdate Unifier wird verwendet um die einzelnen im FDBS-Statement Resolver (Kapitel 3.3.4) aufgeteilten und im FDBS-Statement Executer (Kapitel 3.3.6) ausgeführten Ergebnisse zu einem Integer-Wert zu vereinen. Dabei wird der ExecuteUpdate Unifier immer dann verwendet, wenn ein DML- oder DDL-Statement ausgeführt wurde. Dies ist notwendig, da bei einer horizontalen oder vertikalen Datenaufteilung der Tabelle stets eine unterschiedliche Anzahl von Reihen betroffen sein können, und somit pro Datenbank und pro Befehl unterschiedliche Ergebnisse erhalten werden. Der FDBS-ExecuteUpdate Unifier unterscheidet dabei zwischen DDL und DML. Während im Falle eines DDL die beiden Statements "Create"und "Drop" betroffen sind, geht es bei DML, um "Insert" , "Delete" und "Update" . Bei einem DDL Statement wird eine 0 zurückgeliefert, wenn bei allen ausgeführten DDL-Statements kein Fehler aufgetreten ist, ansonsten 1. Bei einem DML Statement wird die höchste Anzahl der betroffenen Reihen aller Tabellen zurückgeliefert. Die Steuerung und Unterscheidung der Fälle erfolgt dabei über den "ExecuteUpdateUnifierManager" welcher über die Methode "unifyUpdateResult" angesprochen wird. Die Gesamte Struktur des FDBS-ExecuteUpdate Unifier ist in folgender Abbildung 4 einzusehen:

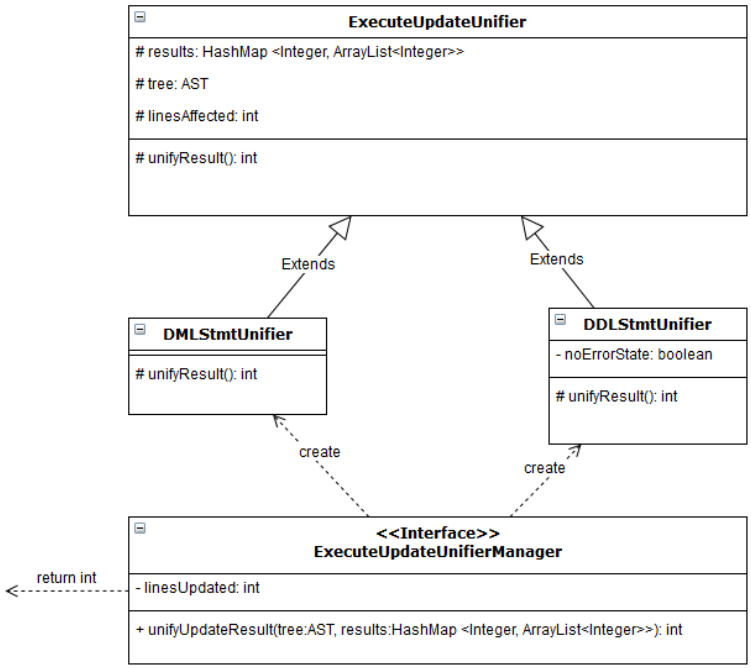


Abb. 4: Struktur des FDBS-ExecuteUpdate Unifier

3.3.8 FDBS-Resultset Unifier

Anders als der FDBS-ExecuteUpdate Unifier, wird der FDBS-ExecuteQuery Unifier verwendet, um Select-Queries, welche vom FDBS-Statement Resolver (Kapitel 3.3.4) aufgeteilten und im FDBS-Statement Executer (Kapitel 3.3.6) ausgeführten ResultSets zu einem FedResultset zusammenzufassen. Hierzu werden die einzelnen ResultSets des FDBS-Statement Executers entgegengenommen und anhand des mit übergebenen AST's analysiert. Dabei werden zunächst die drei grundlegenden Select-Queries (SelectGroup-Statement, SelectNoGroupStatement, SelectCountAllStatement) betrachtet, welche vom ExecuteQueryUnifierManager aufgerufen werden. Der Fall des SelectHavingStatements wird im SelectGroupStatement als Option mit abgebildet. Eine der Herausforderungen dabei ist es die Datenstruktur so zu gestalten, dass möglichst effizient die Spalten aus den bis zu drei Datenbanken zusammen in Relation gesetzt und untereinander anhand der eventuell auftretenden WHERE und HAVING Anweisungen abgebildet werden. Dabei ist eine "Fetch-Size" zu beachten, sodass bei großen Datensätzen nicht unnötig viele Handshake-Transaktionen durchgeführt werden müssen und somit die Performance leidet oder ein OutOfMemory-Fehler entsteht. Innerhalb der drei verschiedenen Select-Statements werden zudem einschränkende Funktionen ausgeführt die zuvor im FDBS-Statement Resolver

nicht möglich sind, wie z.B. MAX, SUM, COUNT(*) oder Joins. Die somit gefilterten ResultSets werden in einem FedResultSet gespeichert und können in diesem anhand eines Zeigers zeilenweise durchlaufen werden. Das FedResultSet stellt somit den Endpunkt einer erfolgreichen Transaktion dar.

3.3.9 FDBS-Helper

Beschreibung folgt.

3.4 Metadaten

Dieser Abschnitt erläutert die Implementierung des Metadaten-Managers. Die verwalteten Metadaten beinhalten Informationen über die vertikale und horizontale Aufteilung der Tabellen und deren Inhalt. Mit ihrer Hilfe wird das Federated Database System entscheiden, wie Einfügeoperationen und Abfrageoperationen verteilt werden, um ein korrektes Ergebnis für den Endanwender zu konstruieren. Deshalb werden die hier verwendeten Datenstrukturen im weiteren Verlauf vom Statement Resolver, Semantic Validator und Statement Unifier verwendet. Die grundlegenden Informationen, die durch die Metadaten gespeichert werden müssen, teilen sich wie folgt auf:

- Die verfügbaren Tabellen mit deren Name
- Der Partitionstyp der Tabelle, falls dieser vorhanden ist
- Horizontale Partitionierung mit den verschiedenen Wertebereichen der Datenbanken
- Vertikale Partitionierung mit Informationen, auf welcher Datenbank welche Spalte vorliegt
- Keine Partitionierung, wenn alles auf der ersten Datenbank gespeichert wurde
- Die Spalten der Tabelle mit deren Namen und Typ
- Die Einschränkungen der verschiedenen Datenbanktabellen, um Statements validieren zu können

3.4.1 Verwaltung der Metadaten

Die oben genannten Informationen werden auf dem ersten Datenbanksystem gespeichert. Diese Datenstruktur wird nicht auf allen Datenbanken abgespeichert, da der Ausfall einer Datenbank zum Versagen des ganzen Datenbanksystems führen würde. Sie ist wie folgt definiert:

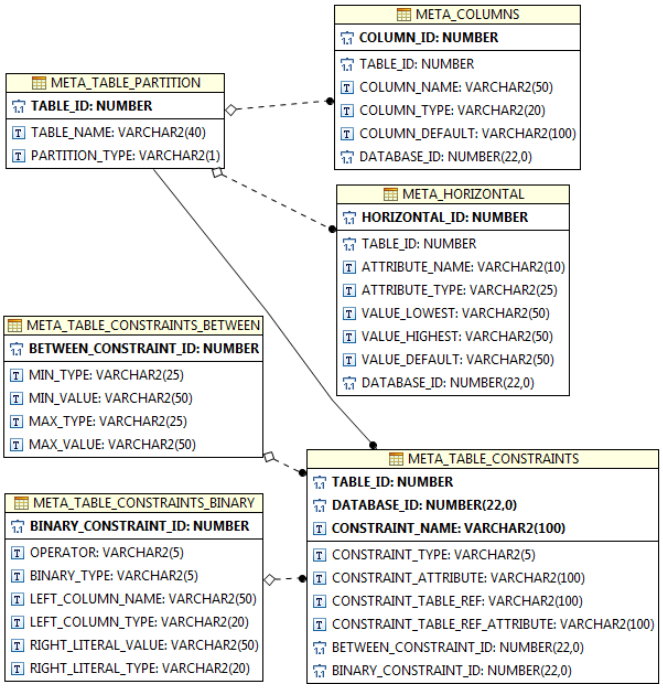


Abb. 5: Die Datenstruktur der Metadaten

Eine Tabelle kann entweder vertikal oder horizontal partioniert werden. Der Partitionierungstyp wird als VARCHAR gespeichert, der die Länge 1 besitzt und drei Werte annehmen kann: 'H' für horizontale, 'V' für vertikale Partitionierung oder 'N' für keine Partitionierung. Diese Informationen werden in der Tabelle META_TABLE_PARTITION gespeichert, die zusätzlich noch jeden Eintrag mit einem ganzzahligen Primärschlüssel identifiziert. Die Nutzung eines Integer-basierten Primärschlüssels dient vor allem Performancegründen, da der Abgleich von Strings bei ggf. langen Tabellenbezeichnungen ineffizient wäre und eine unserer Anforderungen eine möglichst gute Performance ist.

Die Spalte einer Tabelle wird durch ihren Typen, ihren Standardwert und ihren Namen definiert. Unabhängig von der gewählten Partitionierung werden alle Spalten in den Metadaten abgelegt und mit Hilfe des TABLE_ID-Fremdschlüssels mit der entsprechenden Tabelle eindeutig verknüpft. Falls eine vertikale Partitionierung vorliegt, wird zusätzlich noch die Datenbank gespeichert, auf der die Daten dieser Spalte abgelegt werden. Liegt keine vertikale Partionierung vor, so wird für die Spalte DATABASE_ID null eingetragen.

Die horizontale Partitionierung ist charakterisiert durch die Aufteilung von Zeilen unter Betrachtung des Wertes einer bestimmten Spalte. Es muss möglich sein, einen Wertebereich bestehend aus Minimal- und Maximalwert und eine Zieldatenbank zu definieren, in der

Zeilen, die Teil dieses Wertebereichs sind, gespeichert werden sollen. Deshalb muss eine weitere Metadaten-Tabelle definiert werden, die die horizontale Verteilung eines Attributes zu einer Tabelle zuordnet. Diese Tabelle ist als META_HORIZONTAL definiert und speichert die genannten Informationen über den Wertebereich (vgl. VALUE_LOWEST, VALUE_HIGHEST) sowie den Namen der Spalte, die für die Verteilung berücksichtigt werden soll. Weiterhin wird der Typ der Spalte unter ATTRIBUTE_TYPE gespeichert. Alle Einträge außer HORIZONTAL_ID und TABLE_ID in dieser Metadaten-Tabelle sind vom Typ VARCHAR und werden vom Metadaten-Manager innerhalb der Anwendung dynamisch als Integer oder Strings interpretiert, um für die anderen Komponenten des FDBS die Vergleichsoperationen zu vereinfachen und das Speichern von Wertebereichen flexibel gestalten zu können. Aus Performancegründen wird auch für diese Tabelle eine zusätzliche Spalte definiert, die einen ganzzahligen Primärschlüssel enthält und jede Zeile eindeutig identifiziert. In der Tabelle META_HORIZONTAL werden nur Informationen gespeichert, falls eine horizontale Partitionierung für eine Tabelle explizit gewünscht ist. Damit unterscheidet sie sich von der Tabelle META_COLUMNS, die die Informationen über jede Spalte auch bei nicht gewünschter vertikaler Partitionierung abspeichert und damit die Gesamtstruktur des Datenbanksystems vorhalten kann.

Um später die Semantik der Statements überprüfen zu können, werden weiterhin die Einschränkungen der verschiedenen Tabellen der Datenbanksysteme abgespeichert. Diese Einschränkungen enthalten einen Typ, der beschreibt, was für eine Einschränkung vorliegt (Primary Key, Foreign Key, Unique, CheckNull, CHeckBinary, CheckBetween). Diese Einschränkung wird dann über die Spalten CONSTRAINT_ATTRIBUTE, CONSTRAINT_TABLE_REF, CONSTRAINT_TABLE_REF_ATTRIBUTE spezifiziert. Eine Einschränkung liegt entweder für eine bestimmte Datenbank einer Tabelle vor (Wert 1 - 3) oder für alle Datenbanken (Wert 0) vor. Wenn die Einschränkung für alle Datenbanken vorliegt, dann muss diese vom Semantic Validator überprüft werden. Dadurch wird der Wert der DATABASE_ID durch die Integritätsbedingungen bestimmt. Um auch die verschiedenen BINARY-Constraints und BETWEEN-Constraints abzubilden, werden die Constraints durch zwei weitere Tabellen abgebildet. Im Falle des BETWEEN-Constraints werden hier die beiden zu vergleichenden Werte und Typen gespeichert, um diese später benutzen zu können. Bei dem BINARY-Constraint wird abgespeichert welcher Operator genutzt wurde, ob eine Attribute-Attribute-Vergleichsbedingung oder eine Attribute-Konstanten-Vergleichsbedingung verwendet wird und welche zu vergleichenden Werte und Typen benutzt werden. Dadurch können die spezifischen Einschränkungen einer Tabelle und auch einer Datenbank abgefragt und überprüft werden.

Diese Datenstruktur für die Metadaten wird über den MetaDataManager angesprochen und aktualisiert. Das heißt, dass bei jedem CREATE-Statement oder DROP-Statement diese Daten für die erstellte oder gelöschte Tabelle aktualisiert werden. Der MetaDataManager kann weiterhin angesprochen werden, um alle benötigten Informationen zu den verschiedenen Tabellen abfragen zu können. Die Datenstruktur wird dabei einmal abgerufen, sobald der Manager durch die Connection instanziiert wird. Dadurch wird die Performance verbessert,

da nur eine Datenbankabfrage ausgeführt wird. Dieser Manager hat dabei eine eigene Verbindung zu der ersten Datenbank, um alle benötigten Abfragen ausführen zu können.

Durch diese Managerinstanz und der Metadatenstruktur kann auf alle benötigten Funktionen zugegriffen werden, um die richtigen Statements im StatementResolver zu erstellen, die Statements im Semantic Validator zu überprüfen und die Ergebnisse im ResultSet Unifier wieder zusammenzusetzen.

3.4.2 Integritätsbedingungen

Die Maßnahmen für die Einhaltung von Integritätsbedingungen unterscheiden sich je nach Constraint-Typ. Die allgemeine Idee ist, dass Constraints soweit wie möglich von einer Tabelle bzw. Teiltabelle überprüft werden und nur datenbankübergreifende Contraints von der Middleware, um die Middleware möglichst einfach und fehlerfrei zu halten.

3.4.2.1 PRIMARY KEY Um die Integritätsbedingungen für die Fälle der Primärschlüssel nicht zu brechen und die Konsistenz der Datenbank zu wahren, wurde für alle 3 Partitionierungstypen ein Regelwerk implementiert welches bei der Zusammensetzung der Datensätze gewährleistet, dass eine Reihe zu jeder Zeit über einen Primärschlüssel selektiert werden kann. Unabhängig davon ob eine Teiltabelle einen Constraint selbst umsetzen kann oder nicht, wird in jedem Fall auf jeder beteiligten Teiltabelle eine *"CONSTRAINT constraintname PRIMARY KEY (attribute)"* Bedingung angelegt.

Eine föderative Tabelle die per Default partitioniert wird, liegt nur auf einer Datenbank. Dementsprechend ist eine Prüfung der Primärschlüssel Bedingung einfach zu implementieren.

Wird eine föderative Tabelle vertikal über mehrere Datenbanken aufgeteilt, stellen wir sicher dass jede dieser Teiltabellen ein Duplikat des Primärschlüsselattributs enthält (außer der Tabelle auf der das Primärschlüsselattribut ursprünglich liegen soll). Somit wird gewährleistet dass zu jeder Zeit eine Selektion einer ausgewählten Spalte auf jeder Teiltabelle durchführbar ist. Dementsprechend kann auf jeder vertikalen Teiltabelle eine Primärschlüssel Bedingung eingesetzt werden, die die Teiltabelle selbst prüfen kann. Wird eine förderative Tabelle horizontal über mehrere Datenbanken aufgeteilt, kann eine direkte Umsetzung auf jeder Teiltabelle nur dann von den Teiltabellen selbst erfolgen, wenn das Primärschlüsselattribut auch jenes Attribut ist, welches für die horizontale Partitionierung der föderativen Tabelle zuständig ist. Aus diesem Szenario ergibt sich dass eine Wertegrenze für jede dieser Teiltabellen existiert, die dafür sorgt dass die Primärschlüsselattribute einzigartig bleiben. Wird jedoch die horizontale Aufteilung nicht durch das Primärschlüsselattribut bestimmt, ist eine direkte Integritätsprüfung durch die Teiltabellen nicht möglich. In diesen Szenarien muss zusätzlich zum Setzen der Bedingungen auf den Datenbanktabellen, die Middleware bei jeder Manipulation der Daten innerhalb einer Teiltabelle die Integritätsprüfung vornehmen.

3.4.2.2 UNIQUE Um die Integritätsbedingung für ein einzigartiges Attribut (UNIQUE) umsetzen zu können, ist es wieder notwendig die verschiedenen Partitionierungsmethoden zu betrachten. Ist eine föderative Datenbank per Default gesichert, ist eine simple Übergabe einer UNIQUE Bedingung an die Teiltabelle ausreichend um die Konsistenz zu gewährleisten.

Bei einer vertikalen Tabellenaufteilung, wird die UNIQUE Bedingung in der Teiltabelle implementiert, in der sich das betroffen Attribut befindet somit besteht keine weitere Notwendigkeit eine Prüfung in der Middleware miteinzubeziehen.

Ist die föderative Tabelle horizontal aufgeteilt wird eine Integritätsprüfung analog zu dem Vorgehen bei einer Primärschlüssel Bedingung durchgeführt. Handelt es sich bei dem einzigarten Attribut um das Attribut, welches die Tabelle horizontal aufteilt, muss die Middleware nicht eingreifen. Ist dies nicht der Fall, wird einzig und allein die Middleware entscheiden, ob ein Insert oder Update gültig ist, um die Konsistenz der Daten zu bewahren.

3.4.2.3 FOREIGN KEY Wie die Überprüfung einer Fremdschlüssel Bedingung durchgeführt werden kann, ist von der Partitionierung der Fremdschlüssel-Tabelle sowie der referenzierten Tabelle abhängig. Unter folgenden Bedingungen wird die Überprüfung einer Fremdschlüssel Bedingung direkt von einer Fremdschlüssel-Tabelle bzw. Fremdschlüssel-Teiltabelle durchgeführt:

- Fremdschlüssel-Tabelle und die referenzierte Tabelle sind per Default partitioniert, dies bedeutet, dass beide Tabellen auf der selben Datenbank liegen **(default to default)**
- Wenn die Fremdschlüssel-Tabelle per Default partitioniert ist sowie die referenzierte Tabelle vertikal aufgeteilt wurde, wird gewährleistet dass eine Teiltabelle der Referenztabelle auf der selben Datenbank liegt und sämtliche Primärschlüssel enthält **(default to vertical)**
- Wenn die Fremdschlüssel-Tabelle vertikal aufgeteilt sowie die Referenztabelle per Default partitioniert ist und die Bedingung erfüllt ist, dass das Fremdschlüsselattribut auf der gleichen Datenbank wie die Referenztabelle liegt **(vertical to default)**
- Falls die Fremdschlüssel-Tabelle vertikal aufgeteilt ist und sich das Fremdschlüsselattribut auf einer Datenbank befindet, auf der die entsprechende referenzierte vertikale Teiltabelle liegt **(vertical to vertical)**
- Ist die Fremdschlüssel-Tabelle horizontal aufgeteilt und die referenzierte Tabelle vertikal aufgeteilt, muss die Bedingung gelten, dass für jede Partition der Fremdschlüssel-Tabelle auch eine Partition der Referenztabelle existiert auf die verwiesen werden kann **(horizontal to vertical)**

Unter folgenden Bedingungen wird die Überprüfung einer Fremdschlüsselbedingung direkt in der Middleware durchgeführt:

- Die Fremdschlüsseltabelle und Referenztabelle sind per Default partitioniert und liegen nicht auf der selben Datenbank **(default to default)**
- Wenn die Fremdschlüsseltabelle per Default partitioniert wurde und die Referenztabelle eine vertikale Aufteilung aufweist bei der keine dieser Teiltabellen auf der Datenbank der Fremdschlüsseltabelle liegt **(default to vertical)**
- Wenn die Fremdschlüssel-Tabelle vertikal aufgeteilt ist und die Teiltabelle mit dem Fremdschlüssel nicht auf der gleichen Datenbank liegt, wie die per Default partitionierte Referenztabelle **(vertical to default)**
- Ist die Fremdschlüssel-Tabelle vertikal aufgeteilt und die Teiltabelle mit dem Fremdschlüssel liegt auf einer Datenbank, auf der sich keine Teiltabelle der referenzierten vertikal aufgeteilten Tabelle befindet (vertical to vertical)
- Wenn die Fremdschlüssel-Tabelle horizontal aufgeteilt ist und die referenzierte Tabelle per Default partitioniert wurde (horizontal to default)
- Die Fremdschlüssel- sowie Referenztabelle sind beide horizontal partitioniert (horizontal to horizontal)
- Falls die Fremdschlüssel-Tabelle horizontal und die referenzierte Tabelle vertikal aufgeteilt wurde sowie keine entsprechende Referenz-Teiltabelle für eine Fremdschlüssel-Teiltabelle vorhanden ist **(horizontal to vertical)**

3.4.2.4 CHECK Ob eine CHECK Bedingung von einer Tabelle bzw. Teiltabelle überprüft werden kann, hängt von dem Typ der CHECK Bedingung sowie von der Aufteilung der föderativen Tabelle ab.

- CHECK Bedingungen die eine IS NOT NULL Bedingung, BETWEEN Bedingung oder einen Vergleich eines Attribut mit einer Konstante beinhalten werden immer von einer Teiltabelle überprüft, da sich diese Fälle immer nur auf ein einziges Attribut beziehen und dieses Attribut dementsprechend in einer Teiltabelle liegt.
- CHECK Bedingungen die einen Vergleich zweier Attribute enthalten werden nur in dem Fall von einer Teiltabelle geprüft, wenn diese Teiltabelle beiden Attribute beinhaltet. Dies ist bei bspw. einer horizontale Aufteilung sowie einer Default Partitionierung der Fall, da beide Attribute in der selben Teiltabelle liegen.

Während der Entwicklungen können noch weitere Überprüfungen in der Middleware implementiert werden, die zwar auch in einer Teiltabelle umgesetzt werden, aber die Performance verbessern. Überprüfungen von Integritätsbedingungen, die von der Middleware durchgeführt werden müssen, werden im FDBS-Semantic Validator umgesetzt. Weitere Details zum FDBS-Semantic Validator im Kapitel 3.3.2.

3.5 Testfälle

Beschreibung folgt.

4 Fazit

Fazit folgt.

A Anhang

A.1 FDBS-SQL Parser Klassendiagramm

Die Abbildung 6 zeigt den Zusammenhang zwischen FDBS-SQL Parser, abstrakten Syntaxbaum (AST) und einem AST-Visitor, der für das Traversieren durch den AST genutzt werden kann. Das dargestellte Diagramm ist bezüglich des AST aus Übersichtsgründen nur bis zur Ebene der Statements aufgebaut.

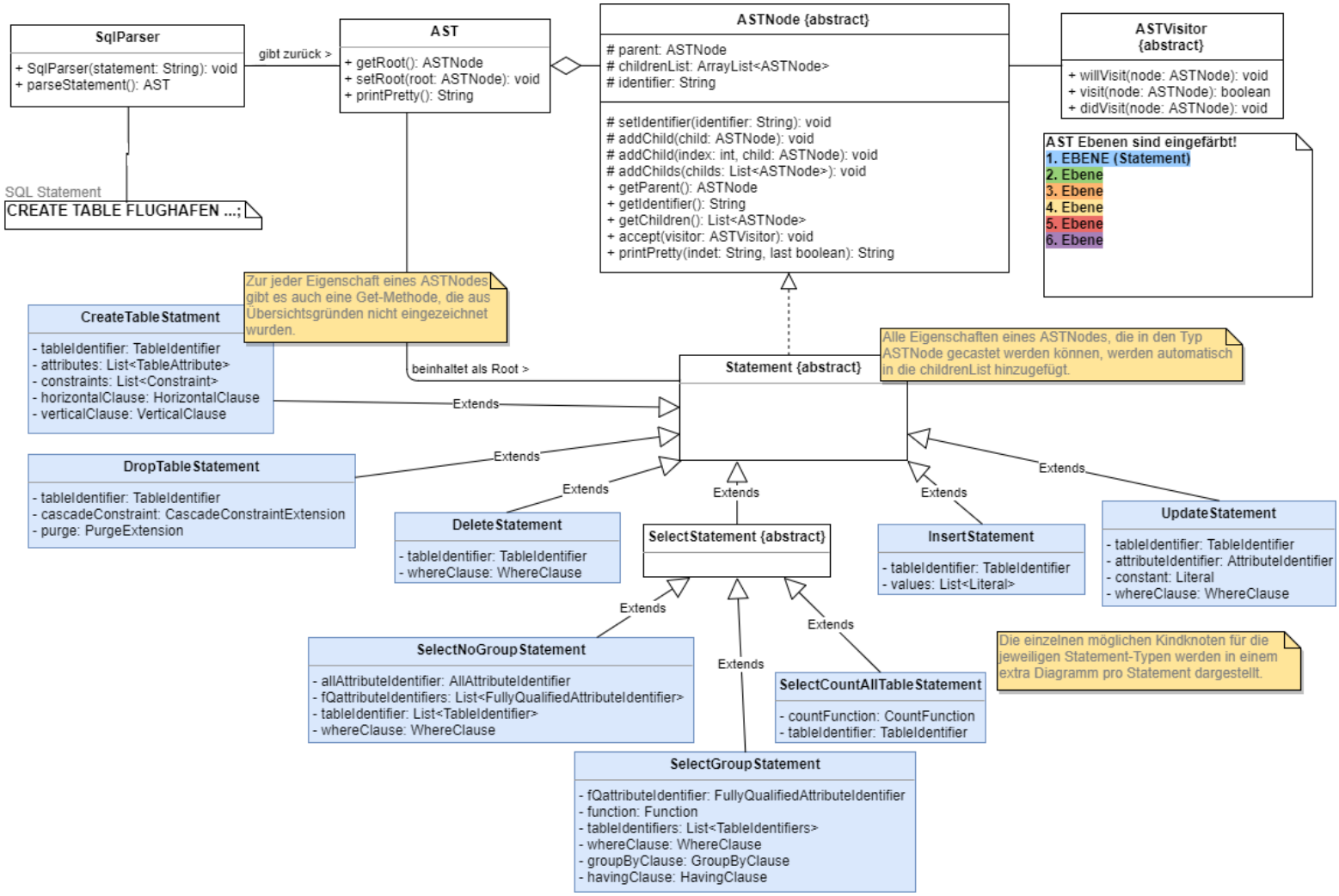


Abb. 6: FDBS-SQL Parser Klassendiagramm bis zur Statement-Ebene

A.2 Group assignment - Partial implementation of a Federative Database Syste

Hochschule Fulda
University of Applied Sciences

Prof. Dr. Peter Peinl

Distributed Database and Transaction Systems
Verteilte Datenbanken

Fachbereich
Angewandte
Informatik

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

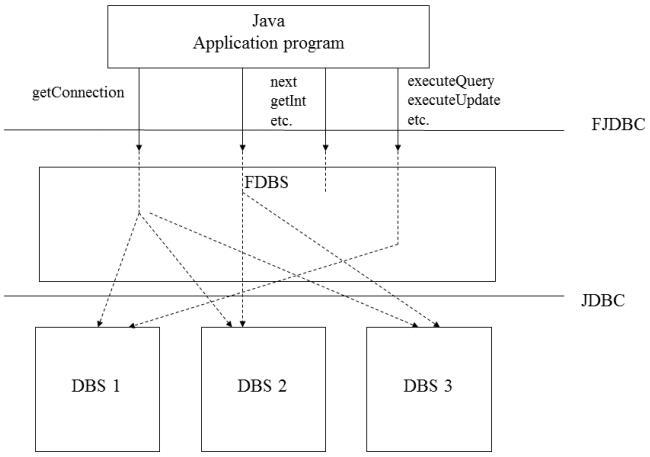
General approach: Dynamic mapping from FJDBC to JDBC

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.