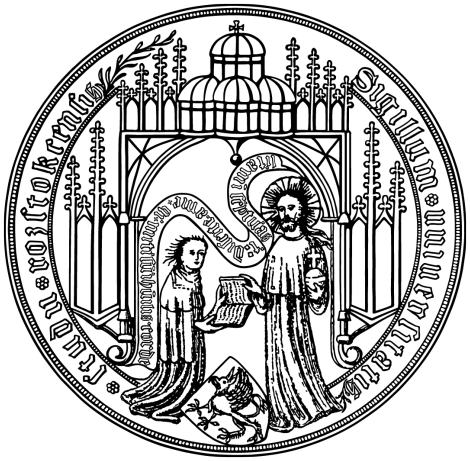

Bachelorarbeit

Aufbau einer Forschungsdatenverwaltung für chemische und physikalische In-Situ-Daten aus der Ostsee

vorgelegt von Mark Lukas Möller
Matrikelnummer 212204377
am 18. Februar 2016

am Lehrstuhl für Datenbanken- und Informationssysteme
der Universität Rostock

Erstgutachter
Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter
Dr. rer. nat. Ralf Prien

Betreuer
Dipl.-Inf. Ilvio Bruder
Dr. rer. nat. Susanne Feistel
Dipl.-Inf. Susanne Jürgensmann

Abstract

Zur Speicherung von Datenbeständen und Analysefunktionen in einem Datenbankmanagementsystem, das den Ansprüchen an Provenance genügen soll, sind Konzepte nötig, die über das reine Speichern der Daten hinausgehen. Diese Arbeit beschäftigt sich mit der Speicherung von Daten, die im Rahmen des GODESS Projektes des Leibniz Institutes für Ostseeforschung Warnemünde entstanden sind. Auf Basis dieser Daten wird ein Prototyp für ein Framework zur Speicherung dieser Daten entworfen, das die Anforderung an Provenance berücksichtigt. Es werden dabei insbesondere temporale Konzepte in Kombination mit Metadaten verwendet, die die Rekonstruktion von Daten sicherstellen sollen. Das Umsetzung erfolgt unter Verwendung des Datenbankmanagementsystems MySQL.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CLOB	Character Large Object
CTD	Conductivity, Temperature, Depth
DBMS	Datenbanken-Managementsystem
DDL	Data Definition Language
DML	Data Manipulation Language
EAV	Entity-Attribute-Value
ER	Entity-Relationship
GODESS	Gotland Deep Environmental Sampling Station
IOW	Leibniz-Institut für Ostseeforschung Warnemünde
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
PIP	Profiling Instrumentation Platform
SP	Stored Procedure
UDF	User Defined Function
XML	Extensible Markup Language

1 Einführung

Ein Konzept für eine konsistente Forschungsdatenverwaltung ist für die langfristige Speicherung heterogener wissenschaftlicher Daten unabdinglich. Kommen zusätzlich Ansprüche an die Nachverfolgbarkeit und Rekonstruktion von Daten hinzu, so treten Fragen und Probleme auf, die über die eigentliche Datenverwaltung hinausgehen. Im Rahmen dieser Bachelorarbeit sollen geeignete Methoden und Konzepte untersucht und diskutiert werden, um diese Probleme exemplarisch für GODESS-Forschungsdaten des *Leibniz-Instituts für Ostseeforschung Warnemünde (IOW)* zu lösen.
In diesem Einführungskapitel soll das GODESS-Projekt mitsamt seiner aktuellen Forschungsdatenverwaltung vorgestellt werden. Es soll aufgezeigt werden, warum eine andere Art der Forschungsdatenverwaltung als bisher nötig ist und erklärt werden, worin die Motivation für Provenance liegt. Es wird eine Zielstellung festgesetzt und die Gliederung der Arbeit beschrieben.

1.1 Das GODESS-Projekt

Die GODESS ist Teil des Forschungsgegenstandes der Arbeitsgruppe für chemische In-Situ-Sensoren des Leibniz-Instituts für Ostseeforschung Warnemünde. GODESS ist ein Akronym für „*Gotland Deep Environmental Sampling Station*“ [PFPSB15]. Hierbei handelt es sich um eine Messstation in Form einer profilierende Verankerung, die mit einer Vielzahl von Messinstrumenten und Sensoren zur Messung von heterogenen Daten von physikalischen und chemischen Umweltbedingungen wie beispielsweise dem ph-Wert, dem Sauerstoffgehalt und der elektrischen Leitfähigkeit der Hochsee ausgestattet ist [PSB15]. Durch ein Verankern der GODESS im Gotlandbecken wird damit ein Messbild der Ostsee über einen Zeitraum von mehreren Monaten erzeugt [IOW01], um Veränderungen physikalischer und chemischer Parameter in Abhängigkeit von der Zeit am Ort des Messens nachvollziehen zu können. Die GODESS verfügt zusätzlich über Einrichtungen wie eine Seilwinde und Auftriebsplatten, die dafür sorgen, dass Messdaten in unterschiedlichen Tiefenbereichen von etwa 30 m bis 180 m unter der Wasseroberfläche erfasst werden können [PFPSB15].

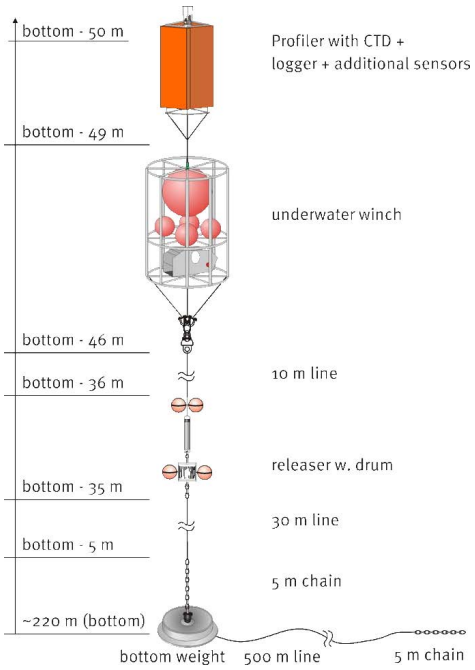

Abb. 1: Entwurfszeichnung der GODESS
[PSB11]

Die GODESS erhebt alle Sensordaten *in situ*. Das bedeutet, dass die Informationen direkt vor Ort gewonnen werden und durch äußere Parameter nicht verändert werden. Der Logger der GODESS speichert diese Sensordaten in Textdateien, die von den Forschern des IOW analysiert, verwaltet und ausgewertet werden können, sobald die GODESS aus dem Gotlandbecken geborgen wird. Zusätzlich zu den eigentlichen Messdaten erhebt die GODESS auch Metadaten wie Datum und Uhrzeit der aktuellen Messung, geografische Koordinaten oder die Maßeinheiten der jeweiligen gemessenen Parameter und speichert diese gemeinsam mit den Nutzdaten.

1.2 Problemstellung und Motivation

1.2.1 Aktuelles Forschungsdatenmanagement

Am IOW werden aktuell MySQL-Datenbanken eingesetzt, um Forschungsdaten vom Ostseemonitoring, von Projekten und von Arbeitsgruppen, wie der AG „Ökologie benthischer Organismen“ zu speichern (vgl. [Mey15]). Allerdings werden die Daten des GODESS-Projektes derzeit nicht in einer solchen Datenbank verwaltet. Ein Forschungsdatenmanagement, welches die Vorteile einer Datenbanktechnologie ausnutzt, kommt hier nicht zum Tragen. Für die Analyse und Weiterverarbeitung wird zum jetzigen Zeitpunkt lediglich auf Dateiebene

gearbeitet. Ein Matlab-Script liest die Daten aus den Dateien ein, transformiert die Daten je nach Anforderung und schreibt die Ergebnisse in eine vorher definierte Datensenke. Anhand dieser Daten lassen sich dann Ergebnisse ableiten und zugehörige Ergebnisse visualisieren.

Das Arbeiten mit Forschungsdaten in Dateien auf dem lokalen Dateisystem hat einige Nachteile. Zum einen liegen die Dateien lokal und nicht zentralisiert vor. Das bedeutet, dass jeder, der mit diesen Daten arbeiten möchte, sich diese auf sein Endgerät kopieren muss und dadurch Redundanz erzeugt. Außerdem können etwaige Änderungen nicht nachvollzogen werden, ohne dass jeder Benutzer die Änderungen in seiner Datenkopie nachholen muss [Mey15]. Alternativen wären an dieser Stelle Dateifreigaben oder Netzlaufwerke, durch die man über das Netzwerk auf diese Daten zugreifen kann. Allerdings stellt auch diese Möglichkeit die Anwender vor Hürden, da in konventionellen Betriebssystemen nur ein Anwender ein Schreibrecht auf diese Dateien bekommt. Alle anderen Nutzer, die diese Dateien gleichzeitig geöffnet haben, sind gezwungen, diese Dateien im Lesemodus zu öffnen [Mey15]. Da die Rohdaten normalerweise nicht verändert werden, ist ein Schreibrecht auf die Daten nicht immer notwendig, allerdings existieren im GODESS-Projekt auch Dateien, in denen die Informationen manuell erfasst werden. Beispiel dafür wären Kommentare und Fehlerbeschreibungen eines Deployments[1]. Durch das Arbeiten auf dem lokalen Dateisystem kann also kein konkurrierender[2] Mehrbenutzerbetrieb sichergestellt werden.

Aspekte zur Nachverfolgbarkeit und Rekonstruktion von Daten, die für die Anforderung an Provenance wichtig werden, werden in Dateien auf einem lokalen Dateisystem kaum beachtet. Informationen zur Bearbeitung oder Löschung von Daten innerhalb einer Datei werden nicht automatisiert mitgeführt, sofern sie nicht manuell ergänzt werden. Es ist nicht möglich zu sagen, ob Daten innerhalb einer Datei hinzugefügt, bearbeitet oder gelöscht worden sind. Lediglich die durch das Betriebssystem verwalteten Metadaten wie der Änderungszeitpunkt der gesamten Datei lassen darauf schließen, *ob* etwas verändert wurde, jedoch nicht *was*. Es findet also keine Historisierung von Änderungen statt, die für ein Provenance Management unabdinglich ist.

Privacy-Aspekte werden im gegenwärtigen GODESS-Forschungsdatenmanagement nur sehr grobgranular realisiert. Die kleinstmögliche Granularität ist hier die Textdatei, auf die der Zugriff für einen Benutzer eingeschränkt werden kann. Aufgrund dieser Tatsache und der, dass die Sensordaten in einem kommasepariert-ähnlichem Format innerhalb weniger Dateien gespeichert werden, ist es an dieser Stelle nicht möglich, dem Benutzer Zugriff nur auf bestimmte Sensorwerte innerhalb einer Datei einzuräumen oder zu verweigern, ohne

[1] „Deployment“ hier: Zeitraum zwischen dem Absenken der GODESS in die Ostsee (dem eigentlichen Deployment) und dem Einholen der GODESS aus der Ostsee (Recovery); also ein Datenmesszyklus

[2] konkurrierender Zugriff: gleichzeitiger Zugriff durch mehrere Benutzer auf den gleichen Datenbestand

die Dateien vorher dafür bearbeiten zu müssen. Es ist lediglich möglich, dem Benutzer den Zugriff auf die gesamte Datei zu erlauben oder zu verbieten. Die Restriktion auf bestimmte Sensorwerte mag bei den Forschungsdaten des IOW weniger Relevanz haben, muss aber in Betrachtung eines allgemeinen Forschungsdatenmanagements beachtet werden. In einigen Bereichen wie in der Medizin hätte dies eine hohe Relevanz, da der Zugriff auf Patientendaten gegenüber einigen Nutzern möglicherweise pseudonymisiert oder ganz verboten werden muss.

1.2.2 Forschungsdatenmanagement mittels Datenbankentechnologien

Für die Speicherung von großen Mengen an Daten haben sich in der Vergangenheit Datenbanken bewährt [Mey15]. Diese bieten den Vorteil, dass bei einem klassischen relationalen Datenbanken-Managementsystem (DBMS) die Daten in einem einheitlich strukturierten Schema vorliegen. Es lassen sich zudem Beziehungen zwischen verschiedenen Datensätzen verdeutlichen. So ist es beispielsweise oft üblich, dass mehrere Datensätze in einer Beziehung zu einem Metadatensatz stehen. Außerdem ist es möglich auf den gespeicherten Daten effiziente Abfragen und Analysen durchzuführen [Mey15] (vgl. [SHS08, Kap. 1.1]), da die Optimierer-Komponente eines DBMS Anfragen vor der Ausführung hinsichtlich Effizienz algebraisch, intern[3] und kostenbasiert optimiert, ohne dass der Benutzer hier explizit eingreifen muss[4] (vgl. [SHS08]). Weiterhin können mit der Nutzung von Datenbanken einige der oben genannten Probleme, wie das Privacy-Problem durch die Möglichkeit der Vergabe feingranularer Zugriffsrechte oder das Problem des gleichzeitigen Zugriffs auf Daten durch Transaktionssteuerung und Isolation gelöst oder reduziert werden.

Provenance Management Systeme bzw. Provenance in Datenbanken ist ein aktueller Forschungsgegenstand. Daher bietet aktuell noch kein DBMS eine vollumfassende Lösung für die automatisierte Mitführung von Provenance-Informationen, z.B. durch automatische Historisierung von Daten. Es existieren Lösungsansätze wie PERM, allerdings sind auch diese bisher nur experimentell [GA09]. An dieser Stelle ist der Anwender selbst in der Pflicht, diesbezüglich Konzepte zu entwickeln und zu implementieren.

1.2.3 Notwendigkeit von Provenance

In der Provenance ist die Idee der Nachvollziehbarkeit und Reproduzierbarkeit von Ergebnissen aus Rohdaten und Analysefunktionen verankert (vgl. [Heu15]). Es soll sowohl nachvollziehbar sein, wann sich welche Daten wie geändert haben (Daten können in einem DBMS hinzugefügt, ersetzt oder gelöscht werden) als auch auf welchem konkreten Datenbestand welche konkrete Abfrage durchgeführt worden ist und zu welchem Ergebnis die

[3] Die interne Optimierung nutzt Informationen über die konkrete Dateiorganisationsform, Indizes etc. aus.

[4] Ausnahme bilden hier Konstrukte der Data Definition Language (DDL). Schlüssel und Indizes müssen beim Erstellen der Datenbank-Tabellen per Hand angelegt werden. Während der Abfrage muss der Benutzer diese aber nicht explizit beachten.

Abfrage geführt hat. Außerdem soll nachvollziehbar sein, welche Berechnungsvorschriften bei Analysefunktionen genutzt worden sind.

Dass eine solche Speicherung von Provenance-Informationen im wissenschaftlichen Umfeld für die Reproduktion von Ergebnissen notwendig ist, soll folgendes Beispiel verdeutlichen:

Ein Wissenschaftler stellt im Januar 2006 an eine Forschungsdatenbank die Anfrage, wie hoch die Summe aller Distanzen der Planeten in unserem Sonnensystem zur Sonne ist und nutzt das Ergebnis mitsamt seiner Abfragebeschreibung in einem Paper. Als Ergebnis habe er etwa $16\,022 \times 10^6$ km herausbekommen (vgl. [Boe00]). Ein Jahr später will ein weiterer Wissenschaftler diese Angabe nachvollziehen und erhält mit der gleichen Anfrage an die Datenbank das Ergebnis $10\,222 \times 10^6$ km (vgl. [Boe00, Boe00b]). Dieses Ergebnis unterscheidet sich nun trotz gleicher Abfrage signifikant vom Ergebnis des ersten Wissenschaftlers. Es besteht nun entweder die Möglichkeit, dass der erste Forscher falsche Berechnungen getätigt hat oder dass sich die Datensätze in der Forschungsdatenbank signifikant verändert haben. Der Grund in diesem Beispiel liegt in der Tatsache, dass im Juli 2006 von der *International Astronomical Union* entschieden wurde, dass Pluto der Status als Planet aberkannt wurde [Int06].

Der Wissenschaftler, der die Anfrage also ein Jahr später gestellt hat, hätte ohne weiteres Hintergrundwissen kaum die Chance, das Ergebnis des Forscherkollegen nachzuvollziehen. Es muss also eine Umgebung geschaffen werden, die in solchen Fällen die Möglichkeit bietet, ältere Ergebnisse auch bei verändertem Datenbestand nachzuvollziehen und anhand der Rohdaten erneut auf dieses Ergebnis zu kommen. Je nach Grad der Datenspeicherung kann Provenance dabei Plausibilität, Nachvollziehbarkeit oder Reproduzierbarkeit ermöglichen. Eine Vertiefung dazu erfolgt im Kapitel 2.5.

1.3 Zielstellung

Ziel dieser Arbeit ist es, ein Konzept für eine langfristige Forschungsdatenverwaltung für das GODESS-Projekt zu entwickeln und umzusetzen. Dazu muss zuerst analysiert werden, welche Daten in welcher Struktur in den gegebenen Dateien vorhanden sind und wie diese zueinander in Beziehung stehen. Anhand der gewonnenen Informationen soll dann ein Datenbankschema erstellt werden, in das die Quelldaten überführt werden können. Verdeutlicht wurde dieser Prozess in Abb. 2. Als Datenbank soll die quelloffene Version des Datenbankmanagementsystems MySQL verwendet werden.

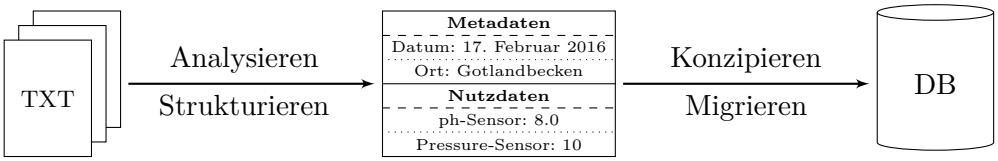

Abb. 2: Grober Plan der Schritte zur Datenüberführung

Bei der Datenbankkonzeption soll Rücksicht auf die Anforderungen der Provenance genommen werden. Prototypisch soll an einer Idee gezeigt werden, wie Provenance in einer Datenbank realisiert werden kann. Außerdem soll weiterhin überprüft werden, wie bestehende Analyseprozesse auf die Datenbankebene verlagert werden können.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in folgende Teile: Neben der erfolgten Einleitung, die die Problemstellung und die Motivation hinter einem Forschungsdatenmanagement und hinter Provenance verdeutlichen soll, wird im Kapitel 2 ein Überblick über die notwendigen Grundlagen für die verwendeten Konzepte gegeben. Dabei wird das GODESS-Projekt technisch und hinsichtlich der Datengewinnung und -speicherung näher betrachtet. Anschließend wird Grundwissen über Datenbanken und Provenance Management vermittelt.

Im dritten Kapitel dieser Arbeit wird auf die theoretische Konzeption zur Lösung der Problemstellung eingegangen. Aus erbrachten Grundlagen werden geeignete Methoden und Konzepte ausgewählt. An dieser Stelle werden Konzepte von Forschungsdatenmanagementsystemen im Generellen betrachtet und neben Umsetzungen auf Basis des SQL:2011-Standards auch Alternativlösungen – z.B. durch NoSQL-Systeme – beschrieben.

In Kapitel 4 wird dann auf zusätzliche Konzepte eingegangen, die für das Forschungsdatenmanagement des IOW notwendig sind. Dabei wird der Fokus umfänglich auf die konkrete Problemstellung des IOW sowie auf die Umsetzung der Konzepte mit dem geforderten MySQL-System gelegt.

Im letzten Kapitel werden dann die gewonnenen Ergebnisse zusammengefasst und die Umsetzung bewertet. Außerdem wird ein Ausblick auf eine zukünftige mögliche Forschungsdatenverwaltung am IOW gegeben, die nicht in die Bachelorarbeit Einzug gefunden hat.

2 Theoretische Grundlagen

In diesem Kapitel soll die technische Funktionsweise der GODESS hinsichtlich Sensorik und Datenspeicherung vertieft werden. Weiterhin werden Grundlagen in den Bereichen Datenbankenmodellierung, temporale Datenbanken und Provenance vorgestellt.

2.1 Technische Grundlagen der GODESS

Die Sensorik der GODESS befindet sich auf einer sogenannten *„Profiling Instrumentation Platform (PIP)“*. Die PIP ist ein Titan-Rahmen, der ursprünglich für die Sonde *CTD 90 M* der Sea & Sun Technology GmbH entworfen und vom IOW unter anderem mit Auftriebsplatten modifiziert wurde [PSB15, IOW01]. Die Sensorik besteht aktuell aus einer Sea & Sun Technology CTD 90 M Multiparametersonde, die Druck, elektrische Leitfähigkeit, Lichtdurchlässigkeit und weitere physikalische und chemische Parameter messen kann. Weiterhin ist auf der PIP eine mit der CTD verbundene Rinko Sauerstoff-Optode zur Messung des Sauerstoffgehaltes, ein Aquadopp Strömungsmesser, ein Rinko Sauerstoffsensor und ein Trios ProPS Sensor zur Messung des Nitratgehaltes aus Spektraldaten verbaut [IOW01, Sea14].

Alle Sensoren sind dabei an einem zentralen Datenlogger angeschlossen. Der Logger dient hier als Bauteil, das kontinuierlich die Messdatenströme der angeschlossenen CTD-, ProPS- und Aquadopp-Sensoren erhält und mit einer Rate von 1 Hz abspeichert. Die CTD 90 M Multiparametersonde verfügt zusätzlich über einen internen Logger und könnte bei einem genannten Messdaten-Erhebungsintervall von 4 Hz aufzeichnen [PSB15]. Da dieser Logger für aktuelle Messungen nicht mehr verwendet wird, ist er insofern nur relevant, wenn es um die Migration von älteren Datenbeständen geht.

Die Datenspeicherung der durch die Sensoren gewonnenen In-Situ-Daten erfolgt in mehreren Textdateien. Dabei werden sowohl – falls vorhanden – die Metainformationen als auch die Nutzdaten („Payload“) in der gleichen Datei gespeichert. Die Daten liegen in keiner standardisierten Struktur wie beispielsweise XML vor, sondern bestehen entweder aus Key-Value-Paaren (vgl. Abb. 3) oder durch einen Delimiter[5] getrennte Werte (vgl. Abb. 4).

[5] Trennzeichen

Der Delimiter ist dabei entweder ein Komma oder ein Leerzeichen. Der Logger erstellt jeweils separate Dateien für die CTD 90 M-Sonde, den Strömungssensor und den Spektraldatensensor. Kalibrierungsdaten liegen in einer XML-Datei vor, die vor dem Deployment von Hand erstellt wird.

```
1 @FILETITEL ASCII-DATA
2 @FILENAME  C:\ASCIIDAT\A150602A.TXT
3 @COPYRIGHT
4 @DATETIME  U1433268106  "2015-06-02" "18:01:46"
```

Abb. 3: Beispiel für die Strukturierung von Metadaten

```
1 $PSDA1,2015-06-02,18:01:46,277,10,45230,61899.000,50073.000,15568.000, ...
2 $PSDA1,2015-06-02,18:01:47,664,11,45236,61915.000,49992.000,15566.000, ...
```

Abb. 4: Beispiel für die Strukturierung von Nutzdaten

2.2 Relationale Datenspeicherung

2.2.1 Begriffsbestimmungen

An dieser Stelle sollen grundlegende Datenbankbegriffe erklärt werden. Diese werden zudem grafisch an Abb. 5 auf S. 18 verdeutlicht.

Relationenschema

Die erste Zeile einer Tabelle beinhaltet Informationen zu Anzahl und Benennung der Spalten. Diese Strukturinformationen werden als Relationenschema bezeichnet (vgl. [SHS08, S. 11]).

Relation

Die Relation umfasst alle Tupel (Zeilen) innerhalb einer Tabelle, die syntaktisch nach dem Relationenschema aufgebaut sind.

Tupel

Ein Tupel ist ein einzelner Datensatz der Relation bzw. eine einzelne Zeile der Tabelle (vgl. [SHS08, S. 11]).

Attribut, Attributwert

Ein Attribut ist eine Eigenschaft einer Entität. In einer Tabelle wird ein Attribut durch eine Spalte realisiert, wobei der Spaltenname dem Attributnamen entspricht. (vgl. [SHS08, S. 11]) Die konkrete Wertzuweisung für das Attribut ist der Attributwert

Tabelle, Entität

Die Tabelle umfasst das Relationenschema und die Relation. Häufig wird die Tabelle auf Modellierungsebene auch als Entity/Entität bezeichnet, beispielsweise im *Entity-Relationship (ER)* Diagramm.

Beziehung, Kardinalität

Eine *Beziehung* gibt an, welche Entitäten zueinander einen Bezug haben. Hierfür findet man auf Modellierungsebene den Begriff *Relationship*. Die *Kardinalität* gibt dabei an, in welcher Quantität eine Entität zu welcher Quantität einer anderen Entität in Beziehung steht. Typische Kardinalitäten sind dabei *1:1*, *1:n* und *m:n*-Kardinalitäten

Primärschlüssel

Der Primärschlüssel ist eine Menge an ausgezeichneten Attributen, welche ein Tupel innerhalb einer Tabelle eindeutig identifiziert. Infolgedessen darf jeder erlaubte Attributwert in den Spalten einer Relation kombiniert nur einmal vorkommen, bei denen im Relationenschema eine Primärschlüsseleigenschaft definiert wurde.

Fremdschlüssel

Um eine Beziehung zwischen zwei Relationen aufbauen zu können, kennzeichnet man ein oder mehrere Attribute als Fremdschlüsselattribute. Der Fremdschlüssel besitzt dabei beispielsweise die Werte der Primärschlüsselattribute aus der zu referenzierenden Relation im Falle einer *1:n*-Kardinalität.

Surrogatschlüssel

Ein Surrogatschlüssel ist ein besonderer Primärschlüssel. Er kommt in keiner Spalte des eigentlichen Datensatzes vor – ist also kein nativer Attributwert – sondern wird in einer im Relationenschema zusätzlichen definierten Spalte gespeichert. Klassisches Beispiel wäre eine fortlaufende Nummer zur Identifikation eines Tupels.

Materialisierung

Unter Materialisierung versteht man die persistente Speicherung von Daten.

Routinen, Stored Procedures, UDFs

Stored Procedures (SP) sind Sequenzen von SQL-Anweisungen. (vgl. [SHS08, S. 451 f.]) Sie werden für die Abarbeitung wiederkehrender Aufgaben genutzt und können parametrisiert sein. Ähnlich funktionieren *UDFs* (*User Defined Functions*). Im Gegensatz zu SPs besitzen UDFs einen Rückgabetyp und können direkt in `SELECT FROM WHERE`-Anweisungen verwendet werden. Stored Procedures und UDFs werden unter dem Begriff *Routinen* zusammengefasst.

Trigger

Ein Trigger ist ein Automatismus, der auf bestimmte Ereignisse wie das Einfügen oder Löschen von Datensätzen in eine bzw. aus einer Tabelle reagieren kann.

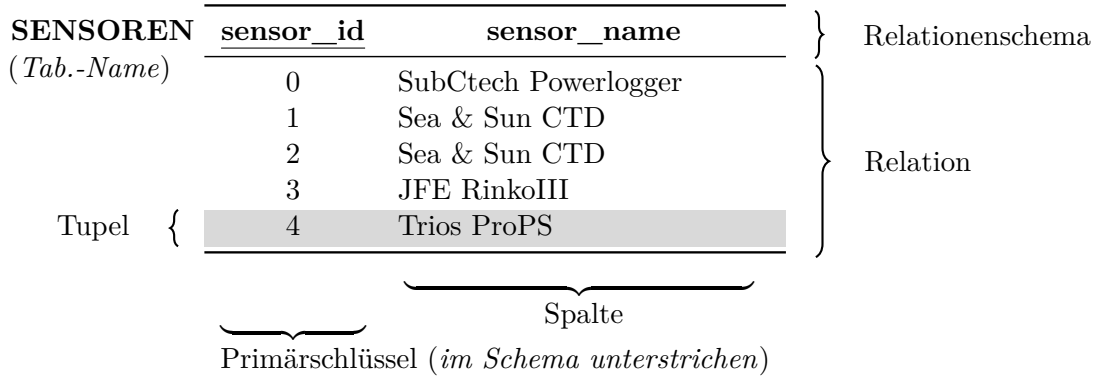

Abb. 5: Grundlegender Aufbau einer Datenbanktabelle

2.2.2 EAV-Speicherung

Die EAV-Speicherung bietet die Möglichkeit einer generischen Speicherung von Daten (vgl. [LKH11]). Nützlich wird diese Art der Speicherung dann, wenn man Daten in die Datenbank aufnehmen will, dessen Datentypen oder Datenstruktur vorher nicht eindeutig bekannt ist. (vgl. [LKH11]) In der klassischen relationalen Speicherung wird ein Objekt in der Datenbank durch eine Entität beschrieben. Attribute werden als Einträge im Relationenschema repräsentiert; eine Menge von Attributwerten wird dann als Tupel in der Relation gespeichert (vgl. Abb. 5).

In der EAV-Speicherung verzichtet man auf diese Art der Repräsentation und nutzt grundlegend drei Tabellen: Die **E**ntitätentabelle, die **A**ttributtabelle und die Attributwerttabelle (**V**alue-Table) (vgl. Abb. 6; vgl. [LKH11]). In der Entitätenrelation wird nur der Name sowie eine identifizierende Entitäten-ID einer Entität gespeichert. Optional kann man in der Relation noch die ID der Eltern-Entität speichern. Angenommen ein Graph besteht aus der Entität „Graph“ bestehend aus zwei Entitäten “Punkt“, so würde jeder „Punkt“ als parent die Elter-ID von „Graph“ bekommen und Graph als parent einen Nullwert.

In der Attributrelation werden die Namen der Attribute der Entität sowie je eine identifizierende Attribut-ID gespeichert.

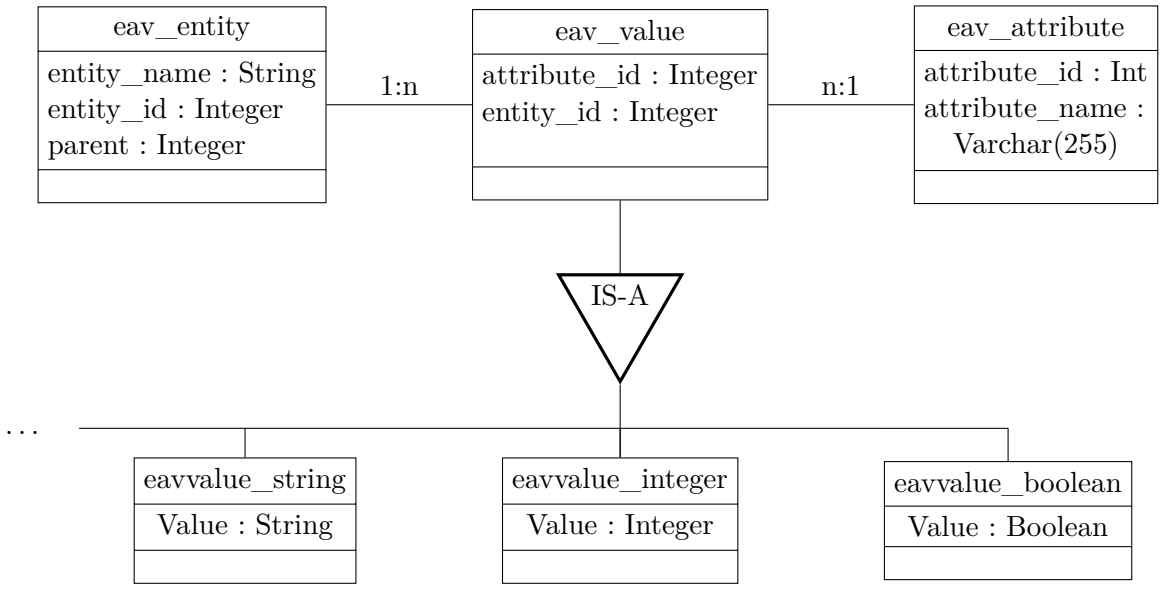

Abb. 6: Entity-Relationship-Diagramm für das EAV-Schema

Die Attributwerte von Attributen einer Entität werden in der Attributwerttabelle gespeichert. Einträge in der Attributwerttabelle verfügen über ein Fremdschlüsselattribut, um eine Beziehung zu einem Tupel in der Attributtabelle herzustellen, sowie über ein Fremdschlüsselattribut, um eine Beziehung zu einer Entität herzustellen. Zudem wird hier der Attributwert gespeichert. Hierbei muss beachtet werden, dass der zu speichernde Datentyp für den Attributwert des Attributes „Attributwert“ der Attributwerttabelle unbekannt ist. Es kann der Fall auftreten, dass ein Integer, ein String, ein Datum oder ein anderer vom DBMS bereitgestellter Datentyp gespeichert werden muss. Daher muss die generalisierte Entity für die Attributwerte in je eine spezialisierte Entity pro genutztem Datentyp aufgebrochen werden, z.B. in $\text{Attributwert}_{\text{String}}$, $\text{Attributwert}_{\text{Integer}}$ etc., wie im ER-Diagramm in Abb. 6 dargestellt.

In Tabelle 1 wurde beispielhaft ein Datensatz im EAV-Schema gespeichert. Die Entität ist hier eine Regression, die zwei Attribute „alpha“ und „beta“ besitzt. Die Attributwerttabelle wurde spezialisiert in eine Tabelle, die Fließkommazahlen als Attributwerte halten kann. Die generalisierte Attributwerttabelle entfällt dabei, da alle Attribute auch in der Spezialisierung erhalten sind und bei Beibehaltung eine Redundanz entstehen würde. Es ist an dieser Stelle erlaubt, auf die spezialisierte Entität zu verschmelzen. Über das Attribut `a_id` wurden `alpha` und `beta` mit ihren Werten `0.6666` respektive `42.4711` verknüpft.

ENTITY

e_id	e_name	e_parent
1	Regression	NULL

ATTRIBUTE

a_id	a_name
1	alpha
2	beta

VALUE_FLOAT

a_id	e_id	value
1	1	0.6666
2	1	42.4711

Tabelle 1: Gespeicherte Ergebnisdaten einer linearen Regression in einem EAV-Schema

Die EAV-Speicherung bietet mit dieser Struktur den Vorteil, dass nahezu alle Typen von Ergebnissen in der Datenbank gespeichert werden können. Es können sowohl skalare Ergebnisse[6] als auch Ergebnisse in Tupelform gespeichert werden. Die Speicherungsform bietet also eine hohe Flexibilität gegenüber Tabellen mit einem festen Schema (vgl. [LKH11]).

Die Speicherung von Daten im EAV-Schema birgt allerdings nicht nur Vorteile. Dadurch, dass bei Abfragen auf Daten in diesem Schema mindestens immer zwei `JOIN`s gemacht werden müssen – und damit drei Tabellen in den Hauptspeicher geladen werden müssen – ist die Performance im Gegensatz zu Abfragen auf konventionellen Speicherungschemata geringer. Besonders bei attributzentrierten Anfragen wie Projektionen oder Aggregationen auf großen Datenbanken ist der Performanzverlust spürbar (vgl. [LKH11]). Außerdem werden

[6] Skalare Ergebnisse enthalten nur einen einzelnen Wert

Anfragen im Generellen stark erschwert. Eine einfache `SELECT * FROM ...` Abfrage in einer konventionellen Speicherform wird zu einer Anfrage, die mindestens zwei `JOIN`s beinhaltet und in bestimmten Fällen sogar Rekursion erfordert – nämlich genau dann, wenn eine Entity Kindelemente hat.

Die höhere Komplexität einer Anfrage gegenüber einer konventionellen Speicherung wird an den bereits beschriebenen Beispieltabellen in Tabelle 1, S. 20 deutlich. In einer konventionellen relationalen Form, in der eine Regression durch eine eigene Entität dargestellt wird, die die Attribute `alpha` und `beta` beinhaltet, könnte man das Ergebnis beispielsweise mit `SELECT alpha, beta FROM results WHERE e_name=Regression` auslesen. In der vorliegenden EAV-Form muss dies über eine komplexere Anfrage geschehen:

```
SELECT e.e_name, a.a_name, v.value
  FROM Value_Float v
  JOIN Attribute a
    ON v.a_id = a.a_id
  JOIN Entity e
    ON v.e_id = e.e_id
  WHERE e.e_name = Regression;
```

2.3 Datenspeicherung mit NoSQL-Datenbanken

Neben den klassischen relational arbeitenden Datenbanksystemen gibt es sogenannte NoSQL-Datenbanksysteme („Not only SQL“). Diese verfolgen den Ansatz der nichtrelationalen Speicherung, bei dem das Datenbankschema nicht oder nur teilweise vorhanden ist [Stö14]. Damit ist es beispielsweise möglich, ineinander verschachtelte Datenobjekte wie *Extensible Markup Language (XML)*- oder *JavaScript Object Notation (JSON)*-Dateien zu speichern, ohne dass man dafür ein Schema entwerfen muss. Im Folgenden sollen einige NoSQL-Speichermöglichkeiten vorgestellt werden.

2.3.1 Key-Value-Speicherung mit Postgres H-Store

H-Store ist ein Speichermodul von PostgreSQL, welches es ermöglicht, Key-Value-Paare als Attributwert zu speichern. [BSG] Attribute („Keys“) und Attributwerte („Values“) werden dabei in den Datenbankanfragen syntaktisch durch `Key => Value` beim schreibenden bzw. durch `Key -> Value` beim lesenden Zugriff dargestellt. [BSG] Die Tabelle für die Ergebnisrelationen kann beispielsweise in folgender Definition vorliegen:

```
CREATE TABLE results (
  e_id INT NOT NULL PRIMARY KEY,
  e_name VARCHAR(32),
  e_result HSTORE
);
```

Mit dem Schlüsselwort `HSTORE` wurde hier in der DDL festgelegt, dass `e_result` keinen nativen SQL-Datentypen sondern einen H-Store enthält (vgl. [BSG]). Damit ist es nun möglich, in diese Spalte Key-Value-Paare einzufügen. Das Hinzufügen von Daten in die Tabelle kann nun so aussehen:

```
INSERT INTO results (e_id, e_name, e_result)
  VALUES (1, 'Point', 'X-Coord => 47.11, Y-Coord => 23');
```

Vergleicht man dies nun mit dem EAV-Schema, hat man an diesem Punkt ein Tupel zur Datenbank hinzugefügt, das einen Punkt als Entity gespeichert hat. Die Attribute des Punktes sind „X-Coord“ und „Y-Coord“ und haben die Attributwerte 47.11 bzw. 23. Attribut und Attributwerte sind in einer Zelle vorhanden und müssen nicht in verschiedene Tabellen ausgelagert werden.
Das Auslesen der Daten kann über einen Query wie diesen erfolgen:

```
SELECT e_result FROM results;

e_result
-----------------------------------------
 "X-Coord"=>"47.11", "Y-Coord"=>"23";
```

oder für einen einzelnen Wert:

```
SELECT e_result->X-Coord as X from results;

X
-----------------------------------------
47.11
```

Mit dem H-Store ist es also sehr einfach möglich, Key-Value-Paare zu speichern, ohne dass man sich Gedanken um die zu speichernden Datentypen machen muss. H-Store bietet zudem eine Vielzahl an vorhandenen Funktionen an, wie die Überprüfung, ob ein H-Store einen bestimmten Key oder eine bestimmte Value enthält oder die Konvertierung in Arrays etc. [BSG]. Allerdings hat der H-Store auch einige Nachteile. Durch die Speicherung nichtatomarer Werte wurde hier die erste Normalform der Datenbanknormalisierung verletzt.

Dadurch können Ziele wie Konsistenz in Datenbanken durch Anomalien negativ beeinflusst werden (vgl. [SHS08, Kap. 6.2.2]). Weiterhin ist der Datentyp der Attributwerte unklar, die im H-Store gespeichert werden. Während bei der DDL die Attribute außerhalb des H-Stores eindeutige Datentypen wie `INT` und `VARCHAR` zugewiesen bekommen haben, sind Datentypen innerhalb der H-Stores nicht explizit definiert. Die 47.11 der X-Koordinate kann also – obwohl es aus semantischer Sicht offensichtlich ist, dass es eine Fließkommazahl ist – an dieser Stelle auch ein String sein. Negativ ist außerdem die Tatsache, dass der H-Store nativ keine geschachtelten Attribute erlaubt. Es ist also nativ nicht möglich, beispielsweise einen Graphen etwa über

```
Graph => {Point => {X-Coord => 4, Y-Coord => 7},
          Point => {X-Coord => 1, Y-Coord => 1}}
```

zu definieren und speichern. Es existieren dafür zwar durch Dritte entwickelte Erweiterungen wie „Nested H-Store“[7], allerdings sind Weiterentwicklungen und Korrekturen damit stark vom Entwickler abhängig[8] und bieten keinen offiziellen Support.

2.3.2 Dokumentorienterte Datenbanken

Eine weitere NoSQL-Technologie bilden Dokumentorienterte Datenbanken. Im Gegensatz zu relationalen Datenbanken, bei denen die Daten in Form von Tabellen gespeichert werden, bildet hier ein Dokument die kleinste Speicherform. Dokumente sind dabei entweder nicht strukturiert oder nach einem Key-Value-Schema aufgebaut [Fac13] – beispielsweise in Form von XML oder JSON. Der Value ist dabei kein singulärer Wert, sondern kann beliebig aufgebaut sein. Bei JSON geschieht dies beispielsweise durch die Verwendung von Arrays, Objekten und Zeichenketten. Da der Benutzer die Struktur selbst festlegen kann (und auch muss) [Fac13], ist diese Speicherform beispielsweise dafür geeignet, um Dokumente in der Datenbank abzulegen, ohne dafür vorher ein relationales Schema entwickelt haben zu müssen. Es eignet sich also dafür, wenn man zum Beispiel Forschungsdatendateien schnell bereitstellen möchte, ohne diese erst bezüglich eines Schemas analysieren zu müssen.

Der Nachteil dokumentenorientierter Datenbanken liegt darin, dass keine Beziehungen zwischen Relationen aufgebaut werden können. Zudem können im Gegensatz zu relationalen DBMS keine oder nur bedingt Constraints oder Trigger definiert werden, d.h. der Nutzer muss auf solche Bedingungen, die durch Constraints oder Trigger abgedeckt werden, in den meisten DBMS selbst reagieren [Fac13] (vgl. [KSS14, S. 125]). Dokumentorientierte Datenbanken

[7] https://github.com/tombenner/nested-hstore

[8] Zum Zeitpunkt der Überprüfung der Website (23.11.2015) wurde die letzte Änderung an dieser Erweiterung vor mehr als 9 Monaten vorgenommen.

sind nicht auf Analyseperformance optimiert, sondern für schnellen Im- und Export von Daten. Das bedeutet, dass für Berechnungen diese Art der Datenbanken langsamer sein können, als relationale Datenbanken[9].

2.4 Temporale Datenhaltung (Historisierung)

In diesem Kapitel sollen Aspekte der temporalen Datenhaltung betrachtet werden. Da das Forschungsdatenmanagement der In-Situ-Ostseedaten unter grundlegenden Gesichtspunkten von Data Provenance untersucht werden soll, sind Konzepte temporaler Datenbanken unabdingbar, falls Daten bearbeitet oder gelöscht werden. Bezogen auf Provenance ist das Bearbeiten und Löschen von Daten keineswegs trivial, wie das Einführungsbeispiel im Kapitel 1.2.3 gezeigt hat. Es müssen Informationen mitgeführt werden, wann welche Änderung auf dem Datenbestand vorgenommen wurden. Ohne Historisierung bzw. Archivierung kann das Zustandekommen von z.B. Analyseergebnissen sonst im schlechtesten Fall nicht mehr nachvollzogen oder rekonstruiert werden.

Für das Hinzufügen, das Ändern und das Löschen von Daten sind also Methoden notwendig, die über die eigentliche Operation hinausgehen. Beim Hinzufügen von Daten muss der Zeitpunkt mitgeführt werden, wann der Datensatz in die Relation aufgenommen wurde. Ansonsten kann der Fall auftreten, dass, wenn man eine Berechnung wiederholt, zwischenzeitlich neu hinzugekommene Tupel mit in die Berechnung einbezogen werden, obwohl diese im ersten Berechnungsdurchlauf noch nicht vorhanden waren. Dadurch kann eine Verfälschung von Ergebnissen entstehen. Auch das Ändern und Löschen von Daten muss gesondert behandelt werden. Gelöschte Elemente müssen, anstatt aus der Datenbank entfernt, als gelöscht markiert werden, um die Reproduktion einer früheren Abfrage korrekt zu gewährleisten, die die zu löschenden Elemente z.B. in Berechnungen noch genutzt hat.

An dieser Stelle sollen deshalb Grundbegriffe von temporalen Datenbanken eingeführt werden, die zur anschließenden Erklärung temporaler Konzepte diesen.

2.4.1 Grundbegriffe

DATE

Speichert ein Datum im Format `'YYYY-MM-DD'`. Der mögliche Wertebereich ist dabei `'1001-01-01'` bis `'9999-12-31'` [Orab].

[9] Siehe Besprechung für den Projektantrag auf der DVD unter /Sonstiges/Projektantrag-Bilder.pdf, Folie 6

TIME

Speichert eine Uhrzeit im Format `'HH:MM:SS'`. Der mögliche Bereich ist dabei `'-838:59:59'` bis `'838:59:59'` [Orab]. Der Datenteil für die Stunden bietet mit seinem Wertebereich dabei die Möglichkeit, nicht nur eine bestimmte Tagesstunde zu speichern, sondern z.B. auch eine vergangene Zeit größer 24 h. [Orab, Orac]

DATETIME

Speichert eine Kombination aus Datum und Uhrzeit im Format `'YY-MM-DD HH:MM:SS'`. Dabei ist `'1000-01-01 00:00:00'` bis `'9999-12-31 23:59:59'` der mögliche Wertebereich [Orab]. Im Gegensatz zu `TIME` kann im Datenteil für die Stunde nur eine Tagesstunde gespeichert werden.

TIMESTAMP

Speichert einen 32 bit-Integerwert der angibt, wie viele Sekunden seit dem 01.01.1970 00:00 Uhr vergangen sind. Der mögliche Wertebereich ist dabei `'1970-01-01 00:00:01'` bis `'2038-01-19 03:14:07'`. In MySQL kann auch die Variable `CURRENT_TIMESTAMP` verwendet werden, um die aktuelle Systemzeit zu verwenden [Orab].

Transaktionszeit/Transaction Time

Bei einer Tabelle, die das Konzept der Transaktionszeit nutzt, wird diese um zwei Attribute erweitert. Der erste Attributwert – die Startzeit – enthält dabei Informationen, wann ein Tupel logisch in die Datenbank eingetragen wurde (Startzeit). Der zweite Attributwert – die Endzeit – enthält Informationen, wann das Tupel als ungültig markiert wurde. Mit der Transaktionszeit ist also beantwortbar, ob bzw. zu welchem Zeitpunkt ein Tupel in der Datenbank vorhanden war (vgl. [Mey15]): Wird ein Tupel hinzugefügt, so wird das Startzeitattribut auf die aktuelle Uhrzeit gesetzt. Um ein Tupel als ungültig zu markieren – und damit im Sinne der Provenance zu „löschen“, kann man den Attributwert für die Endzeit auf die aktuelle Zeit zu setzen. Ein nicht gelöschtes aktuelles Tupel kann beispielsweise für den Endzeitpunkt den maximalen Wert besitzen ('9999-12-31 23:59:59').

Gültigkeitszeit/Valid Time

Ähnlich wie bei der Transaktionszeit wird bei Verwendung der Gültigkeitszeit die Tabelle um zwei Spalten erweitert. Diese halten je ein Datum, ab wann ein Datensatz gültig war beziehungsweise ab wann ein Datensatz als ungültig markiert wurde. Es wird also also die Gültigkeit von Fakten der modellierten Realität beschrieben (vgl. [Mey15]).

Soll wie bei der Transaktionszeit ein Endzeitpunkt als nicht gesetzt gelten, so wird der maximal mögliche Attributwert ('9999-12-31 23:59:59') gewählt.

Bitemporale Relation

Eine bitemporale Relation enthält sowohl die Transaktionszeit, als auch die Gültigkeitszeit (vgl. [Mey15]).

2.4.2 Datenmodifikation mit Transaktionszeiten

An der Tabelle 2 soll das Einfügen, Bearbeiten und Löschen unter dem Gesichtspunkt der Nutzung der Transaktionszeit beispielhaft verdeutlicht werden. Die Tabelle soll eine Deployment-ID mit Primärschlüsseleigenschaft, einen booleschen Wert `data_verified` und Transaktionszeitinformationen speichern. Folgende DDL soll zur Erstellung der Relation dienen:

```
CREATE TABLE Deployments (
  d_id INT NOT NULL PRIMARY KEY,
  data_verified BOOLEAN,
  tt_begin DATETIME DEFAULT NOW(),
  tt_end DATETIME DEFAULT ''9999-12-31 23:59:59'
);
```

2.4.2.1 Einfügen in die temporale Tabelle

Das Einfügen ist der einfachste Fall. Es müssen neben dem eigentlichen Datensatz die Transaktionszeitinformationen gespeichert werden. Da die `DEFAULT`-Werte dafür schon in der DDL festgelegt wurden, müssen beim Einfügen diese Informationen nicht explizit angegeben werden. Eine Beispiel-DML für das Einfügen wäre:

```
INSERT INTO Deployments (
  (d_id, data_verified)
  VALUES (14, FALSE)
);
```

Mit der beschriebenen DDL und Data Manipulation Language (DML) wird folgende Tabelle erzeugt:

DEPLOYMENTS			
d_id	**data_verified**	**tt_begin**	**tt_end**
14	FALSE	2015-11-15 12:00:00	9999-12-31 23:59:59

Tabelle 2: Temporale Beispieltabelle nach Einfügen eines Datensatzes

2.4.2.2 Löschen aus der temporalen Tabelle

Das Löschen eines Datensatzes ist nicht so trivial wie das Einfügen, da die klassische Löschoperation mit `DELETE` hier nicht verwendet werden kann. Würde man den Datensatz standardmäßig mit folgendem SQL-Syntax löschen:

```
DELETE FROM Deployments
  WHERE d_id = 14;
```

so hätte man das Problem, dass die Provenance-Möglichkeit verletzt wäre, auch Ergebnisse aus Datenbeständen früherer Zeitpunkte zu rekonstruieren. Berechnungen, die dieses Tupel genutzt haben, können nicht erneut durchgeführt werden. Umgehen kann man dies, indem man nicht das Tupel an sich löscht, sondern die Endzeit `tt_end` in der Relation auf den Zeitpunkt des Löschens setzt und damit das Tupel als ungültig definiert.

```
UPDATE Deployments
  SET tt_end = NOW()
  WHERE d_id = 14;
```

Bei einer Anfrage an die Datenbank ist zu beachten, dass explizit betrachtet werden muss, ob ein Tupel noch gültig ist oder nicht. Für die Selektierung nach nicht gelöschten Tupeln könnte man in diesem Fall in der `WHERE`-Klausel die konjunktive Bedingung hinzufügen, dass `tt_end = '9999-12-31 23:59:59'` erfüllt sein muss. Da `tt_end` in diesem Beispiel nun keinen maximalen Wert mehr trägt, kann dieses Tupel nun als gelöscht angesehen werden. Als wichtiger Hinweis sei gesagt, dass kein Datensatz in die Relation eingefügt werden darf, der den gleichen Primärschlüsselwert hat, wie ein bereits gelöschtes Tupel. Soll dies möglich sein, so muss die erweiterte Schlüsseleigenschaft beachtet werden, die in Kapitel 2.4.2.3 beschrieben wird.

DEPLOYMENTS			
d_id	**data_verified**	**tt_begin**	**tt_end**
14	FALSE	2015-11-15 12:00:00	2015-11-15 12:30:00

Tabelle 3: Temporale Beispieltabelle nach dem „Löschen“ eines Datensatzes

2.4.2.3 Aktualisieren von Datensätzen der temporalen Tabelle

Der komplexeste Fall ist der `UPDATE`-Fall. Soll neben dem Einfügen und dem Löschen auch das Aktualisieren eines Tupels möglich sein, so müssen folgenden Betrachtungen vorab gemacht werden:

- Die Attributwerte eines zu aktualisierenden Tupels werden nicht aktualisiert, sondern das Tupel wird mittels Modifikation von `tt_end` als ungültig markiert.
- Ein neues Tupel wird mit den neuen Attributwerten in die Tabelle eingefügt; dabei ist das neuere Tupel durch die Transaktionszeitinformationen erkennbar.
- Hierdurch entsteht die Möglichkeit einer Schlüsselverletzung.

Es müssen also die Transaktionszeitinformationen mit in den Primärschlüssel aufgenommen werden, damit eine Schlüsselverletzung nicht auftreten kann. Es gilt allerdings die Prämisse, dass nicht zwei Tupel gleichzeitig als aktuell gekennzeichnet sein können. Es ergibt sich daraus nun folgende veränderte DDL:

```
CREATE TABLE Deployments (
  d_id INT NOT NULL,
  data_verified BOOLEAN,
  tt_begin DATETIME NOT NULL DEFAULT NOW(),
  tt_end DATETIME DEFAULT NULL,
  PRIMARY KEY(d_id, tt_begin)
);
```

Betrachtet man nun Tabelle 2 – nun allerdings mit verändertem Primärschlüssel – so muss eine `UPDATE`-Operation beispielsweise so aussehen, falls man den booleschen Wert `data_verified` auf `TRUE` setzen möchte:

```
UPDATE Deployments
  SET tt_end = NOW()
  WHERE d_id = 14;
INSERT INTO Deployments
  (d_id, data_verified)
  VALUES (14, TRUE);
```

Tabelle 4 stellt das Anfrageergebnis dar, das aus diesem Query resultiert. Der alte Datensatz bleibt erhalten, während ein neuer Datensatz hinzugekommen ist: Die grau unterlegte Zeile stellt eine Zeile dar, dessen Transaktions-Endzeit gesetzt wurde. Damit wurde sie als gelöscht markiert. Da die Transaktions-Startzeit nun Teil des Primärschlüssels ist, findet keine Schlüsselverletzung statt. Über die Bedingung `tt_end = '9999-12-31 23:59:59'` lässt sich nun der aktuellste Datensatz einer Deployment-ID herausfiltern.

DEPLOYMENTS

d_id	data_verified	tt_begin	tt_end
14	FALSE	2015-11-15 12:00:00	2015-11-15 12:30:00
14	TRUE	2015-11-15 12:30:01	9999-12-31 23:59:59

Tabelle 4: Temporale Beispieltabelle nach dem „Aktualisieren“ eines Datensatzes

2.5 Grundlagen der Provenance

Provenance im Kontext der Datenverwaltung bezeichnet die Verfolgbarkeit, wie welche Ergebnisse aus welchen Rohdaten entstanden sind (vgl. [Heu15]). Es werden dadurch Grundlagen für die Authentizität und Vertrauenswürdigkeit von Daten und Ergebnissen geschaffen ([W3C10]). Wie in Tabelle 5 ersichtlich ist, gibt es je nach Grad der Datenhaltung verschiedene Stufen der Provenance. Im schlechtesten Fall kann man gar keine Provenance gewährleisten, im besten Fall kann man Ergebnisse sogar komplett rekonstruieren. Im Folgenden sollen die verschiedenen Stufen der Provenance[10] anhand eines Beispiels erklärt werden. Die Provenance bezieht sich dabei immer auf ein Ergebnis respektive einen Bericht, d.h. wie stark kann man ein Ergebnis bzw. einen Bericht auf Basis des Grades der Datenspeicherung nachvollziehen.

Metadaten	Primärforschungsdaten	Zwischenerg.	Funktion	Dokumentation
✓	✓	bel.	✓	Rekonstruierbar
✓	✓	✓	✗	Nachvollziehbar/Rekonstruierbar
✓	✓	✗	✗	Nachvollziehbar
✓	✗	bel.	bel.	Plausibilität
✗	bel.	bel.	bel.	Keine Provenance

✓= vorhanden, ✗= nicht vorhanden, bel. = beliebig

Tabelle 5: Stufen der Provenance

[10] Die Ideen der vier Stufen der Provenance entstammen dem Vortrag [HB15].

2.5.1 Keine Provenance

Ein Forschungsdatenmanagement kann ohne Metadaten nicht existieren und damit auch kein Provenance Management in dieser Anwendung. Hat ein Forscher ein Ergebnis aus Forschungsdaten abgeleitet und in einem Bericht verwendet, so kann dies kein weiterer Forscher überprüfen, sofern im Bericht z.B. nur die Nutz-, nicht aber die Metadaten vorhanden sind. Verständlich ist dies an einer Datei, die nur reine Messwerte ohne Beschreibungen enthält. Sofern man keinerlei Informationen über die Semantik der Daten hat, also welche Daten welche Bedeutung haben, kann man mit diesen Daten nichts anfangen. Entweder werden Metadaten in die Forschungsdatendateien bzw. in zusätzliche separate Dateien aufgenommen oder es existiert eine Dokumentation, die in diesem Falle die Rolle von statischen Metadaten wie die Einheit von Messwerten übernehmen kann. Dynamische Metadaten wie z.B. Erhebungsort- und zeitraum müssen allerdings zwingend erhoben werden und können aufgrund der Veränderlichkeit nicht durch eine Dokumentation ersetzt werden.

2.5.2 Plausibilität

Der schwächste Grad der Provenance (abgesehen von „Keine Provenance“) bildet die Plausibilität. Plausibilität kommt zustande, wenn zwar Metadaten, aber keine Primärforschungsdaten vorliegen. Liegt in einem Bericht die Angabe vor, dass der Salzgehalt pro Kilogramm Seewasser der Ostsee zu einem beschriebenen Zeitpunkt etwa 7 g beträgt, so kann diese Angabe plausibel sein, da der Salzgehalt der Ostsee im Normalfall etwa in diesem Bereich liegt (vgl. [Fei06]). Durch den Mangel an vorliegenden Primärforschungsdaten kann allerdings nicht nachvollzogen werden, wie die gemessenen Rohdaten des Salzgehaltes aussahen und ob tatsächlich aus den Sensordaten der korrekte Salzgehalt abgeleitet wurde.

2.5.3 Nachvollziehbarkeit

Nachvollziehbar wird das Experiment, wenn zusätzlich zu den Metadaten die Rohdaten zum Bericht veröffentlicht werden. Im Gegensatz zur Plausibilität wird nicht nur ersichtlich, *dass* ein bestimmtes Datum gemessen wurde, sondern auch, *welche* konkreten Werte erfasst wurden. Anhand dessen kann ein Ergebnis oder ein Bericht durch einen Experten nachvollzogen werden. Wenn das Ergebnis durch ein Rechensystem erneut berechnet werden kann, so ist es sogar rekonstruierbar.

2.5.4 Rekonstruierbarkeit

Ein Rohdatum muss nicht zwingend ein konkretes Messergebnis darstellen. Die Daten können auch z.B. in für den Menschen unlesbarer Form, z.B. als hexadezimaler String, vorliegen. In solchen Fällen muss ein Rohdatum noch durch eine Analysefunktion transformiert werden. Um hier die Rekonstruierbarkeit der Ergebnisdaten im Bericht zu gewährleisten, gibt es die folgenden Möglichkeiten:

- Es sind die Analysefunktionen, nicht aber die Zwischenergebnisse im Bericht vorhanden. Dies stellt kein Problem dar, sofern die Analysefunktionen deterministisch sind. Mit Hilfe der Analysefunktionen sind die einzelnen Zwischenschritte rekonstruierbar und somit auch das gesamte Ergebnis.
- Es sind die Zwischenergebnisse, nicht aber die Analysefunktionen im Bericht vorhanden. Dies stellt ein Problem dar, falls die Funktionen sich nicht aus den Primärforschungsdaten und Ergebnissen ableiten lassen. Dann wäre lediglich die Nachvollziehbarkeit gewährleistet, da man zwar alle vorhandenen Daten vorliegen hat, aber nicht den Schritt von den Primärforschungsdaten über die Zwischenergebnisse bis hin zu den Endergebnissen rekonstruieren kann. Es ist nicht klar, ob die Berechnungsfunktionen korrekt gearbeitet haben, da deren Berechnungsvorschriften unbekannt bleiben.
- Der optimale Fall besteht aus der Speicherung von Zwischenergebnissen und Funktionen. In diesem Fall ist sogar neben der Rekonstruierbarkeit eine Fehlernachvollziehbarkeit möglich.

Für alle Fälle der Rekonstruierbarkeit müssen Meta- und Primärforschungsdaten gespeichert sein.

3 Generelle Konzeption

In diesem Kapitel wird das konkrete Framework für die Forschungsdatenverwaltung entwickelt, das die Probleme aus Abschnitt 1.3 löst. Die Kernprobleme, die sich im Abschnitt refsec:zielstellung herauskristallisiert haben, waren einerseits die **Datenhaltung** und andererseits die **Provenance**. Für folgende existente Teilprobleme wird eine Lösung vorgestellt:

- Speicherung der Forschungsrohdaten
- Speicherung von Analysefunktionen im DBMS
- Materialisierung von Analyseergebnissen
- Speicherung von Metadaten, welche für die Provenance benötigt werden

Die Konzeptideen basieren dabei weitestgehend auf dem SQL:2011-Standard, nutzen aber auch Lösungsideen neuerer Datenbanktechnologien, wie zum Beispiel der NoSQL-Systeme. Zusätzliche Konzepte konkret für das GODESS-Projekt werden im Kapitel 4 ab Seite 48 beschrieben.

3.1 Komponenten des Frameworks

Das hier beschriebene Framework für das Forschungsdatenmanagement soll sowohl die *un- bzw. semistrukturierten Datenhaltung* als auch die *strukturierten Datenhaltung* ermöglichen. Grafisch dargestellt ist das Framework mit seinen Komponenten in Abb. 7.
Die *un- bzw. semistrukturierte Datenhaltung* bietet Möglichkeiten zur Speicherung von Daten in einem *Key-Value-Store*, einem *EAV-Store* und in einem *Document Store*, während bei der *strukturierten Datenhaltung* die Möglichkeiten der Speicherung in einer relationalen zeilenorientierten Datenbank (*Row Store*) bzw. in einer relationalen spaltenorientierten Datenbank (*Column Store*) zur Verfügung stehen. Die Komponenten werden ab Abschnitt 3.1.1 näher erläutert.

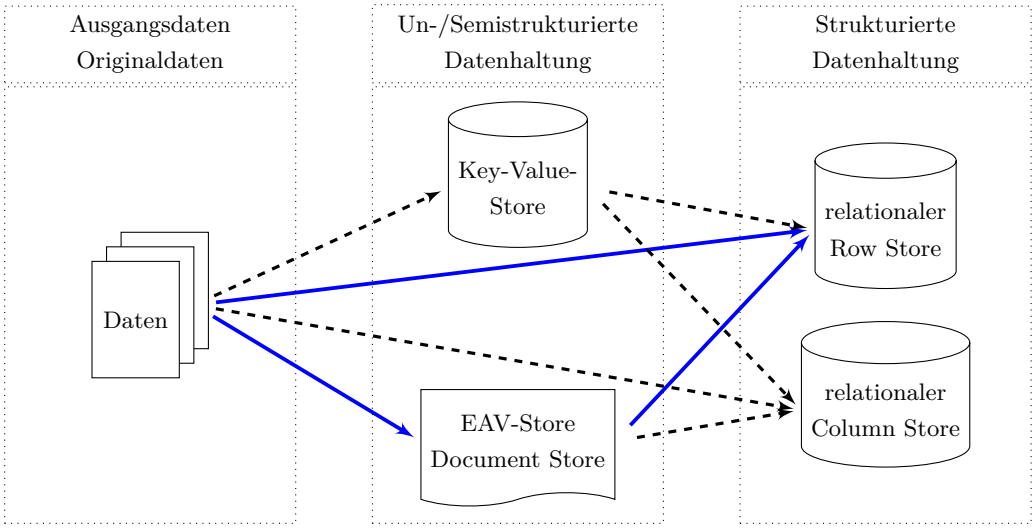

Abb. 7: Möglichkeiten der Datenhaltung im Framework

Je nach Ziel der Speicherung wird eine Art der Datenhaltungsvariante gewählt werden. Voraussetzung für die Entscheidung der Speicherart ist das Hinterfragen und Klären der folgenden Punkte:

- Speicherziel – Sollen die Daten zum Zeitpunkt des Importes nur abgelegt werden oder in der vorliegenden Form auch weiterverarbeitet bzw. analysiert werden?
- Frequenz der zu importierenden Daten – Werden kontinuierlich neuen Daten importiert (hohe I/O-Rate) oder nur zu bestimmten Zeitpunkten?
- Komplexität der Analysefunktionen – Sollen Berechnungen auf der DBMS-Ebene erfolgen oder extern durch eine andere Programmiersprache mit Anbindung zur Datenbank?
- Muss eine Datenbereinigung wie beispielsweise Duplikateliminierung stattfinden?
- Müssen Restriktionen bezüglich einiger Datensätze gelten? (Zugriffsregelungen auf bestimmte Datensätze)

Im Folgenden werden die einzelnen Komponenten aus Abb. 7 mit ihren konkreten Aufgaben im Framework sowie deren Eigenschaften vorgestellt. Dabei wird der Fokus auf den Datenfluss gelegt, der in Abb. 7 blau gekennzeichnet ist: Die Speicherung im *EAV-Store*, im *Document Store* und in einem relationalen *Row Store*. Die weiteren Komponenten wie *Key-Value-Store* und der relationale *Column Store* werden als alternative Komponenten für bestimmte Anwendungszwecke angesehen und deren Zweck im Forschungsdatenmanagementsystem erläutert, spielen aber eher eine untergeordnete Rolle.

3.1.1 Document Store

Der Document Store dient im Framework als Komponente, dessen primärer Einsatzzweck das schnelle Speichern und Bereitstellen von Daten im Forschungsdatenmanagementsystem ist. Der Document Store ist bezüglich hoher I/O-Raten optimiert und speichert die Daten auf feingranularster Ebene als Datei. (vgl. [McN14]) Daher können hier Daten schnell gespeichert werden, ohne, dass vorher dafür ein Datenbankschema entworfen werden muss. Er hat aber den Nachteil, dass Abfragen nicht so effizient wie bspw. in relationalen DBMS umgesetzt werden können. Ebenfalls nachteilig ist die fehlende Möglichkeit, Beziehungen zwischen Entitäten herzustellen und feingranulare Restriktionen zu realisieren. Der Document Store ist also die Komponente im Framework, die es erlaubt, Daten schnell zu speichern.

3.1.2 EAV-Store

Die Grundidee hinter dem EAV-Store wurde in Abschnitt 2.2.2 auf Seite 19 beschrieben. Er dient im Framework einerseits zur Speicherung von Daten, dessen Schema vorher nicht bekannt oder veränderlich ist (z.B. wenn Daten eine dynamische Anzahl an Attributen haben). Wie in Abschnitt 2.2.2 angemerkt, geht diese Art der Speicherung zu Ungunsten der Performance. Möchte man effiziente Analysen auf den Daten durchführen, muss ein relationales Datenbankschema entwickelt und genutzt werden.

3.1.3 Key-Value-Store

Als eine weitere un- bzw. semistrukturierte Speichermöglichkeit steht der Key-Value-Store zur Verfügung. Dieser wurde in Abschnitt 2.3.1 auf Seite 21 beschrieben und dient als alternative Speichermöglichkeit zum EAV- oder Document Store. Er soll an dieser Stelle nicht weiter beschrieben werden. Auch diese Speichermöglichkeit eignet sich für das Speichern von Daten ohne festes Schema, da die Anzahl der Keys und Values pro Datensatz variieren kann.

3.1.4 Row Store

Der Row Store als zeilenenorientierte relationale Speicherform stellt eine der beiden hier beschriebenen Möglichkeiten zur strukturierten Datenhaltung dar. Im Framework dient er als primäre Speicherform für die Speicherung von Daten, für die ein Schema vorliegt. Genutzt wird diese für Daten, bei denen die Selektion hauptsächlich über alle Attribute erfolgt. Durch die zeilenweise Speicherung erreicht man dabei eine hohe Performance, da die Daten beim Auslesen effizient abgerufen werden können. Der Row Store eignet sich, wenn man Beziehungen zwischen Entitäten aufbauen möchte, Duplikate können beim Einfügen in die Datenbank

erkannt werden und feingranulare Restriktionen sind möglich. Der Nachteil an dem Row Store bzw. der strukturierten Datenhaltung im Allgemeinen ist, dass das Schema zum Speichern von Daten von Hand erstellt werden muss und dies oft Zeit kostet (vgl. [McN14])[11].

3.1.5 Column Store

Der Column Store als spaltenorientierte Speicherform stellt eine Alternative zum Row Store für eine relationale Speichervariante dar. Durch die Art der Datenhaltung wird der Column Store im Framework zum Speichern von Daten verwendet, auf denen Aggregatfunktionen über eine Spalte wie z.B. die Durchschnittsbildung anwendet werden. Solche Abfragen sind besonders effizient, da die Attributewerte so gespeichert werden, dass diese mit wenigen Plattenzugriffen in den Hauptspeicher geladen werden können. Ansonsten gelten Vor- und Nachteile des Row Stores bezüglich Beziehungen, Duplikateliminierung, Schemaerstellung und Möglichkeiten zur Restriktion entsprechend.

Verdeutlicht an Tab. 6 liegen die Daten wie in Zeile 1 im Row Store bzw. wie in Zeile 2 im Column Store vor:

```
1 January,1.8;February,2.6;March,6.8;
2 January,February,March;1.8,2.6,6.8;
```

Will man eine Aggregatsfunktion über die Spalte `value` ausführen, so liegen die entsprechenden Werte auf logischer Ebene nah beieinander, sodass vom DBMS nur der entsprechende Teil der Datei gelesen werden muss, wodurch die Performanceverbesserung gegenüber dem Row Store resultiert.

AVG_TEMP	**month**	**value**
	January	1.8
	February	2.6
	March	6.8

Tabelle 6: Beispieltabelle für gemessene Werte

[11] Es existieren automatisierte Ansätze, wie die Schemaextraktion aus Daten aus NoSQL-Datenbanken, allerdings sind diese noch Gegenstand der Forschung

3.2 Speicherung von Daten

Nachdem betrachtet wurde, welche Komponenten das Framework für welche Zwecke bietet, wird nun geklärt, welche Daten im Framework gespeichert werden. Insgesamt gibt es dabei vier Arten von Daten, die im Framework festgehalten werden müssen: Die **Nutzdaten**, **Analysefunktionen**, **Analyseergebnisse** und **Metadaten**. Wie diese vier Arten zusammenspielen, ist in Abbildung 8 ersichtlich: Im System existieren Nutzdaten und Analysefunktionen, welche „unabhängig“ von anderen Datensätzen sind, d.h. sie sind nicht aus vorhergehenden Daten generiert worden. Aus dem Zusammenspiel von Nutzdaten und Analysefunktionen sowie ggf. Metadaten werden Analyseergebnisse erzeugt. Diese können entweder virtuell existieren, d.h. das Ergebnis wird einmal berechnet und ist nach dem Ausführen einer anderen Operation im DBMS weg, oder die Ergebnisse können materialisiert werden, d.h. die Ergebnisse werden nach Berechnungen persistent im DBMS gespeichert. Metadaten beschreiben dabei sowohl die Nutzdaten, die Analysefunktionen und -ergebnisse.

Wie genau diese vier Arten von Daten im Framework gespeichert werden, wird nachfolgend erläutert.

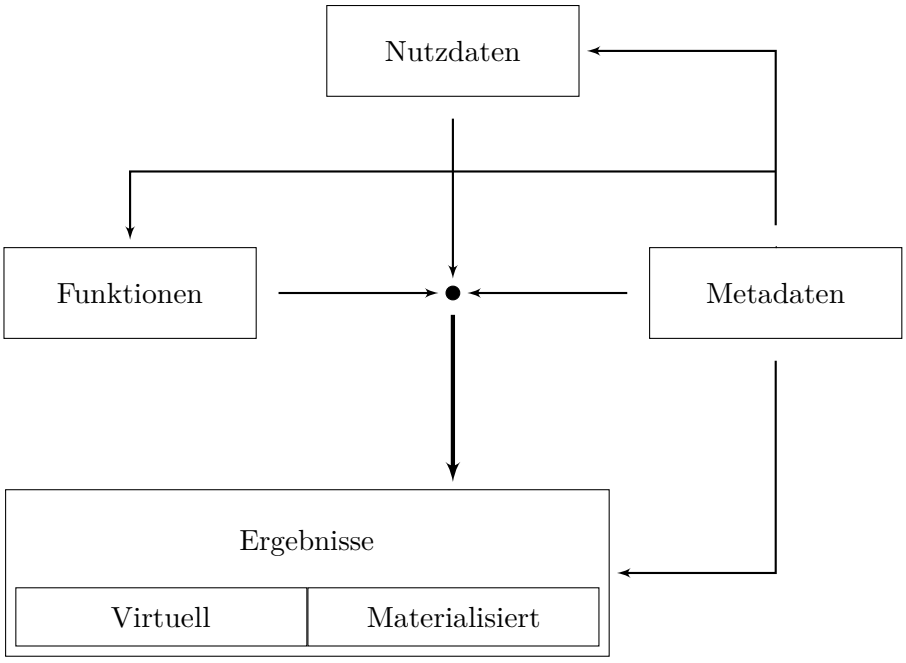

Abb. 8: Einfließende Daten im Framework

3.3 Speicherung von Datenstrukturen und Datentypen

Hinsichtlich der Speicherung von Datenstrukturen und Datentypen, die aus den Forschungsrohdaten resultieren, müssen einige Betrachtungen durchgeführt werden. Darunter fällt unter anderem der Aufbau und die Heterogenität der Daten: Daten können z.B. als eindimensionales Datum, mehrdimensional, in Tabellenform oder als Hierachie auftreten. Je nach Homogenität der Daten werden verschiedene Varianten der Speicherung genutzt. Homogenität bezieht sich an dieser Stelle auf die Gleichheit der Daten bezüglich deren Aufbau und Attributwertdatentypen. Beispielsweise wäre der Datensatz eines Personalausweises homogen, da auf jedem Personalausweis die gleiche Menge an Attributen aufgedruckt sind (Name, Vorname, Geburtsdatum etc.) und diese sich nur in ihren Attributwerten unterscheiden. Bedingt homogen wären möglicherweise Messdatensätze, wenn zwischen den verschiedenen Messreihen einige Sensoren hinzugefügt oder weggelassen oder Daten in unterschiedlichen Formaten gespeichert werden. In den folgenden Abschnitten werden für verschiedene Datenstrukturen Möglichkeiten beschrieben, die sich für die Speicherung eignen. Für das Framework ist ebenfalls von Relevanz, ob die Daten bzw. die Schemata als „Closed World“ oder „Open World“ angesehen werden können. Beim ersten sind die vorhandenen Schemata über die Laufzeit des Systems hinweg gleich. Es wird also initial ein Schema erstellt werden, in das die Daten dann eingefügt werden. Ist das Forschungsdatenmanagement „Open World“, so ist davon auszugehen, dass für die Datenhaltung nicht zwingend existente Schemata genutzt werden können und vor der Datenspeicherung ein neues Schema entwickelt werden muss, sofern eine strukturierte Datenhaltung genutzt wird.

3.3.1 Eindimensionale Werte

Für die Speicherung von eindimensionalen Werten gibt es mehrere Ansätze. Ist der Datentyp des Ergebnisses unbekannt, so materialisiert man das Ergebnis als einzelne Tabelle. Aufgrund der daraus resultierenden Unübersichtlichkeit und der Tatsache, dass weiterführende Berechnungen inperformant werden können, wenn eine Vielzahl an Tabellen in einen Verbund aufgenommen werden muss, sollte man diese Methode nicht nutzen, sofern sich eine andere Lösung anbietet.

Ist der Datentyp des Ergebnisses bekannt, so kann das Ergebnis zu eine Tabelle hinzugefügt werden, die Werte dieses Datentypes hält. Die Tabelle besteht dabei aus einer Spalte für eine ID, die diesen Datensatz für weitere Berechnungen eindeutig identifizierbar macht, einer typisierten Spalte für ebendiesen Datentyp und zwei Spalten für Transaktionszeitinformationen.

3.3.2 Tabellen

Ergebnisse einer Tabellenfunktion („Table Function“) in Form einer UDF besitzen als Rückgabetyp Tabellen mit typisierten Spalten. Führt man eine solche UDF aus und soll die daraus resultierende Sicht materialisiert werden, so speichert man diese in einer neuen Tabelle. Sei also `udf()` eine Funktion, die eine Tabelle zurückgibt, so lässt sich das Ergebnis in SQL mit `CREATE TABLE persistent_udf SELECT * FROM udf()` persistent speichern. Die Tabelle `persistent_udf` wird dabei nach dem gleichen Schema aufgebaut, welches die zurückgegebene Tabelle der UDF `udf()` besitzt und die zurückgegebenen Datensätze werden in der neuen Tabelle eingefügt. Diese Tabellen eignen sich auch für die Speicherung von heterogenen Werten, da bei jeder Erzeugung das Tabellenschema neu definiert wird.

Existieren Tabellen, die nach dem gleichen Schema aufgebaut sind und die gleichen semantischen Informationen enthalten (z.B. weil die Berechnung zyklisch ausgeführt wird), so kann man die Ergebnistabelle mit der schon existenten Tabelle vereinen, z.B. durch die Nutzung von `UNION`.

3.3.3 Wiederkehrende Datenstrukturen

Für wiederkehrende Datenstrukturen (*Structs*) werden im Framework eigene Relationen angelegt. Voraussetzung dabei ist eine hohe Homogenität der Daten, da die Spalten der Tabelle typisiert sind. Daten der gleichen Datenstruktur werden in dieselbe Tabelle aufgenommen. Je nach Weiterverarbeitungszweck wird dabei die Komponente des Row Stores oder des Column Stores gewählt.

3.3.4 Hierarchien

Die Speicherung von Hierarchien kann komplex sein. Hier gibt es je nach Art der vorliegenden Daten die Möglichkeit zur Nutzung von Adjaszenzlisten, Path-Enumeration, Nested-Sets, Closure-Tables etc. [Mey15]. Je nach Struktur der Hierarchie wird eine andere Komponente im Framework gewählt. Liegt diese beispielsweise als JSON-Dokument vor, so wird ein Document Store verwendet. Andersfalls kann man die Hierarchie in einem Row Store aufbauen. An dieser Stelle soll nicht weiter auf die Speicherung von Hierarchien eingegangen werden.

3.3.5 Spaltenbezeichnungen

Um ein relationales Schema anzulegen, sind nach dem SQL-Standard Spaltennamen notwendig. Es existiert im Framework eine einheitliche Vorgehensweise, um die Spaltennamen zu vergeben.
Es werden zuerst die Metadaten betrachtet. Gibt es in den Metadaten explizit eine Information dazu, wobei es sich um die Attributwerte einer Spalte handelt (z.B. „Pressure“), so wird

diese Beschreibung vorrangig verwendet. Falls es keine Informationen in den Metadaten dazu gibt, semantisch aber ersichtlich ist, worum es sich handelt (z.B. eine Liste der Titel von Research Papern), wird anhand dieser Information vom Nutzer der Spaltenname bestimmt. Ist keinerlei Informationen über die Bedeutung einer Spalte vorhanden, so wird der Datentyp in Kombination mit einer laufenden Nummer als Spaltenname gewählt, beispielsweise `Datetime1`, `Datetime2` und `Blob1`.

Bei automatisch generierten Tabellen – beispielsweise durch Materialisierung von Ergebnissen – übernimmt das DBMS die Spaltenbenennung, sofern nicht noch manuell auf diese Generierung Einfluss genommen wird.

3.3.6 Schlüsselbestimmung

Wie im Kapitel 2.2.1 beschrieben, ist ein Schlüssel notwendig, um ein Tupel eindeutig zu identifizieren. Dazu muss man die kleinste eindeutige Teilmenge von Attributen bestimmen, deren Attributwerte gemeinsam einmalig in der Relation vorkommen. Häufig sind dies IDs oder Zeitstempel. Weiterhin muss entschieden werden, ob ein Tupel doppelt in der Relation vorkommen darf. Sollen doppelte Tupel explizit zugelassen werden – beispielsweise, weil mit dem Weglassen eines Tupels sonst ein Aggregat wie der Durchschnitt verfälscht wird – existiert kein eindeutiger Schlüsselkandidat. In so einem Fall wird ein Surrogatschlüssel eingeführt. Sollen doppelte Tupel verboten werden, so werden diese beim Einfügen ignoriert, z.B. durch die Nutzung von `INSERT IGNORE`. Die Entscheidung, ob ein Tupel doppelt existieren darf oder nicht, obliegt dem Datenbanknutzer und ist vom DBMS nicht bestimmbar.

3.4 Analysemethoden

Um Analysefunktionen auf Datenbestände anzuwenden, so müssen Vorüberlegungen hinsichtlich der Speicherung der Funktion an sich und der Ergebnisse gemacht werden. Es muss entschieden werden, welche Funktionen als Stored Procedure im DBMS gespeichert werden oder durch anderweitige Programmiersprachen abgedeckt werden (s. Kapitel 3.4.1). Weiterhin werden bezüglich der Provenance Einschränkungen hinsichtlich der Rechtevergabe gemacht, die ab Abschnitt 3.4.3 beschrieben werden. Für die Provenance existiert eine Metadatenrelation, die Informationen darüber hält, welche Berechnungsergebnisse durch welchem Datenbestand und welche Analysefunktion zustande kam, da ansonsten die Zwischenergebnisse nicht mehr zu rekonstruieren sind. Dieses Thema wird im Abschnitt 3.6.2 erläutert.

An dieser Stelle sei darauf hingewiesen, dass nicht Implementierungskonzepte, sondern das Konzept für die Provenance von Analysefunktionen im Vordergrund steht.

3.4.1 Externe Berechnungen vs. Routinen

Einige Berechnungen sind in SQL nicht oder nur schwer realisierbar. Als Beispiel dafür seien Lambda-Funktionen angeführt. Daher werden für solche Berechnungen die Berechnungsvorschriften nicht in SQL, sondern in einer anderen Programmiersprache realisiert. In der Programmiersprache selbst wird dann die Verbindung zur Datenbank aufgebaut, Berechnungen getätigt und Ergebnisse zurückgeschrieben. Manche DBMS wie DB2 bieten auch die Möglichkeit der Nutzung von Programmiersprachen wie R oder Java, ohne dass die Analysefunktion über einen externen Prozess berechnet wird. Dabei wird eine Funktion der jeweiligen Programmiersprache im DBMS registriert, die dann als UDF aufrufbar ist.

Bei der Implementierung einer Routine muss abgewogen werden, ob und mit welchem Aufwand sich eine Berechnungsfunktion im DBMS implementieren lässt, oder ob man dafür eine andere Programmiersprache nutzt.

3.4.2 Temporale Informationen für Analysefunktionen

Bei Routinen wird die Gültigkeitszeit und die Transaktionszeit verwendet. Die Rohdaten und die Ergebnisse verwenden dagegen nur die Transaktionszeit. Das hat folgenden Hintergrund: Für die Rohdaten ist relevant, wann die Daten zur Datenbank hinzugefügt wurden. Nur mit diesem Wissen ist entscheidbar, ob bei der Rekonstruktion von Rohdaten aus einem Ergebnis ein Rohdatum mit in die Berechnung einbezogen wurde oder nicht. Ansonsten werden Roh- und Ergebnisdaten aber nicht berührt. Sie sind von ihrem Attributwert her über die gesamte Laufzeit des Forschungsdatenmanagementsystems unveränderlich.
Routinen dagegen können sich im Laufe der Zeit in ihrer Definition ändern, z.B. weil andere Berechnungsvorschriften gelten. Dafür wird zusätzlich die Gültigkeitszeit verwendet, die angibt, in welchem Zeitraum eine Routine Gültigkeit hatte. Damit ist es auch möglich, zukünftige Berechnungsvorschriften zum implementieren oder Routinedefinitionen der Vergangenheit ins DBMS hinzuzufügen. Mit der Kombination von Transaktionszeit und Gültigkeitszeit wird eine bitemporale Relation erzeugt.

3.4.3 Archivierungskonzept für Analysefunktionen im DBMS

Um Provenance zu gewährleisten, gelten bestimmte Bedingungen bei der Rechteverwaltung. Im Sinne der Archivierung von Routinen dürfen diese nicht gelöscht werden. Sollte ein Aufruf von `DROP FUNCTION` oder `DROP PROCEDURE` uneingeschränkt für den Datenbankbenutzer

ausführbar sein, so kann es passieren, dass eine lückenlose Archivierung nicht eingehalten wird, falls eine Routine ohne vorherige Vorbetrachtung gelöscht wird. Um diesem Problemfall entgegenzuwirken, existiert eine Historie der Routinen. Folgende Informationen werden dabei erfasst:

- Die Routinedefinition.
- Die Gültigkeitszeitpunkte (Valid Time), zu welchem Zeitpunkt die Routine beginnend und endend Gültigkeit hatte.
- Transaktionszeitinformationen, die Aufschluss darüber geben, wann eine Routine zum DBMS hinzugefügt und gelöscht worden ist.

Wird nun eine Routine im DBMS angelegt, so wird ihre Definition zusätzlich in die Archivierungsrelation eingetragen. Außerdem wird der Gültigkeitszeitpunkt entsprechend gesetzt. Im Beispiel soll die Routine ab dem jetzigen Zeitpunkt gültig sein:

```
CREATE PROCEDURE dummyProcedure ()
  BEGIN END;
INSERT INTO routines_history (specificName, definition,
vt_begin, vt_end, tt_begin, tt_end)
  VALUES ('dummyProcedure', 'BEGIN END', NOW(), '9999-12-31 23:59:59'
NOW(), '9999-12-31 23:59:59');
```

Nach Ausführung der DML für das Hinzufügen der entsprechenden Werte in die Archivierungsrelation ist in dieser der Eintrag hinzugefügt worden, der in Tabelle 7 ersichtlich ist. Besitzt eine Routine Parameter, werden diese zusätzlich in eine weitere Archivierungsrelation aufgenommen. Auch diese werden mit Gültigkeitsinformationen versehen.

ROUTINES_HISTORY

specificName	definition	vt_begin	vt_end	tt_begin	tt_end
dummyProcedure	BEGIN END;	2016-01-15 12:34:56	9999-12-31 23:59:59	2016-01-15 12:34:56	9999-12-31 23:59:59

Tabelle 7: Beispielaufbau Archivierungstabelle

Soll eine Routine gelöscht werden, so kann dies rein konzeptionell über eine Prozedur geschehen, die den Namen der zu löschenden Routine als Eingabeparameter annimmt, die Historisierung entsprechend anpasst und die Routine letztendlich aus dem DBMS löscht. Folgende Statements müssen dabei innerhalb der Routine ausgeführt werden:

```
UPDATE routines_history SET vt_end = NOW (), tt_end = NOW()
WHERE specificName = 'dummyProcedure'
AND tt_end = '9999-12-31 23:59:59';
DROP FUNCTION dummyProcedure;
```

ROUTINES_HISTORY

specificName	definition	vt_begin	tt_end	tt_begin	tt_end
dummyProcedure	BEGIN END;	2016-01-15 12:34:56	2016-01-20 20:00:00	2016-01-15 12:34:56	2016-01-20 20:00:00

Tabelle 8: Archivierungstabelle nach dem Löschen einer Prozedur

Die als invalide und gelöscht gekennzeichnete Routine ist in Tabelle 8 ersichtlich. In dieser Tabelle liegt der Sonderfall vor, dass die Beginn- und Endzeiten der Transaktions- und Gültigkeitszeiten identisch sind. Semantisch bedeutet dies, dass die Funktion zum Zeitpunkt des Anlegens Gültigkeit erhalten hat und mit Ende ihrer Gültigkeit als gelöscht gekennzeichnet wurde. Das muss nicht zwingend der Fall sein, es kann z.B. einer Routine auch eine Gültigkeitsende hinzugefügt werden, ohne gleich gelöscht zu werden, z.B. wenn erreicht werden will, dass eine Routine ab einem bestimmten Zeitpunkt nicht mehr verwendet werden darf. Allerdings muss darauf geachtet werden, dass das Ende der Gültigkeitszeit zeitlich vor dem Ende der Transaktionszeit liegt. Ist dies nicht der Fall, würde es bedeuten, dass die Funktion gültig ist und angewendet werden darf, obwohl sie bereits als gelöscht markiert wurde. Zudem muss beachtet werden, ob die Funktion noch benutzt werden darf, sprich, das Ende der Gültigkeitszeit noch nicht erreicht wurde.

Nach SQL-Standard ist es nicht möglich, eine Routine innerhalb einer Routine zu löschen (`DROP`). Auch lassen sich keine Trigger erstellen, die auf das Löschen von Routinen reagieren. Daher existiert an dieser Stelle keine Möglichkeit einer Automatisierung, die sicherstellt, dass vor dem Löschen einer Routine eine Kopie in die Archivierungstabelle aufgenommen wurde. Daher ist es nicht möglich, das DBMS den Historisierungsprozess vor Löschung abarbeiten zu lassen, falls dies vergessen wurde. Um zu verhindern, dass der Forscher im generellen Arbeitsablauf in der Programmiersprache versehentlich ein `DROP PROCEDURE` bzw. `DROP FUNCTION` ausführt, ohne vorher zu archivieren, wird für ebendiesen Zweck des Löschens dem Datenbankbenutzer generell das Löschen für Prozeduren verboten und ein separater Datenbankbenutzer mit dem `DROP`-Recht erstellt und genutzt. Es muss der Implementierung der Löschfunktion zu beachtet werden, dass je eine separate Funktion für das Löschen von UDFs und SPs benötigt wird, da der Syntax zum Löschen variiert. Das Löschen einer Routine geschieht mittels einer Programmiersprache wie Java, die die Invalidierungen vornimmt und und das Löschen ausführt. Dies wird durch eine Java-Funktion

realisiert, die als Parameter den Namen der Routine übergeben bekommt, basierend auf dem Parameter die Archivierung durchführt und die Routine löscht.

Einige DBMS wie MySQL bieten die Möglichkeit, die Routineinformationen über das Informationsschema der Datenbank auszulesen, sodass der Benutzer nicht wie im vorangegangenen Beispiel die Prozedurdefinition manuell in die Historisierungstabelle eintragen muss. Konkret eingegangen wird darauf im Abschnitt 4.2.

3.5 Speicherung von Analyseergebnissen

Bei der Speicherung von Analyseergebnissen treten ähnliche Probleme auf, wie bei der Speicherung der Forschungsrohdaten. Zuerst muss entschieden werden, ob die Analyseergebnisse materialisiert werden müssen oder nicht. Anschließend wird, wie bei den Rohdaten, überprüft, in welcher Form bzw. mit welchem Schema eine Materialisierung stattfinden kann. Letztendlich wird auf Metadaten-Ebene festgehalten, welche Ergebnisse durch welche Berechnungen und welche Rohdaten zustande gekommen sind.

3.5.1 Materialisierungskriterien

Sofern ein Ergebnis berechnet wurde, ist zu entscheiden, ob es wichtig ist, dies zu materialisieren. Bei folgenden Beispielen ist die Materialisierung hinsichtlich der Provenance notwendig, sofern man Rekonstruierbarkeit gewährleisten möchte:

- Die Analyseergebnisse werden direkt (als Wert) oder indirekt (z.B. textuelle Auswertung auf Basis der Ergebnisse) in Publikationen verwendet.
- Die Analyseergebnisse sind Zwischenergebnisse für andere Analysefunktionen. Die Endergebnisse werden in Publikationen verwendet.

In folgenden Fällen ist eine Materialisierung sinnvoll, nicht aber zwingend notwendig:

- Ermitteln von Ergebnissen, die nicht weiterverwendet werden.
- Ausprobieren neuer Berechnungsfunktionen mit der Prämisse, dass das Ergebnis an dieser Stelle keine Relevanz hat.
- Speicherung von Ergebnissen, die aktuell nicht benötigt werden, aber einen hohen Verbrauch an Ressourcen (Rechenleistung oder Speicher) gefordert haben.

- Trivial **deterministische** Zwischenergebnisse, die keine weitere Relevanz haben (z.B. Funktion, die die aktuellen Werte mit zwei multipliziert), sofern der Determinismus sichergestellt wird. Eine Multiplikation mit dem aktuellen Mehrwertsteuersatz von 19% wäre dagegen nichtdeterministisch, da sich der Mehrwertsteuersatz ändern kann. An dieser Stelle müsste materialisiert werden.

3.5.2 Möglichkeiten der Materialisierung

Zur Materialisierung von Analyseergebnissen existieren verschiedene Möglichkeiten. Ist das Schema eines Ergebnisses unbekannt, so besteht die Möglichkeit, dass, wie im Kapitel 3.3.2 beschrieben, das Ergebnis als Tabelle materialisiert wird. Dies ist der trivialste Weg, der immer funktioniert, für weiterführende Berechnungen aber nicht immer die optimale Speichervariante darstellt.

Ist das Tabellenschema des Ergebnisses bekannt und tritt dieses Schema bei Berechnungen nicht nur einmalig auf, wird eine Tabelle mit diesem Schema erstellt und die Ergebnisse werden dort eingefügt. Möchte man beispielsweise den täglichen Durchschnittswert der Tagestemperatur berechnen und zu einer Relation hinzufügen, so beinhaltet das Schema vermutlich immer ein Datum und eine Fließkommazahl für die Temperatur. Es lohnt sich in diesem Fall, die Ergebnisse immer zu dieser Tabelle hinzuzufügen, statt jeden Tag eine neue Tabelle hinzuzufügen, da das Schema gleich aufgebaut und die Informationen semantisch ähnlich sind. Möchte man nun Berechnungen beispielsweise zur Jahresdurchschnittstemperatur machen, so kann man die Aggregatsfunktion `AVG()` über diese Tabelle ausführen, ohne 365 Einzeltabellen in einen Verbund aufnehmen zu müssen.

An dieser Stelle sollen die Möglichkeiten der Materialisierung nicht weiter vertieft werden, da die Methoden und Vorgehensweisen aus Abschnitt 3.3 entsprechend angewendet werden können. Es muss allerdings festgehalten werden, welches Ergebnis wie entstanden ist. Das Konzept dazu wird in Abschnitt 3.6.2 erläutert.

3.6 Speicherung von Metadaten

Metadaten sind Informationen, die in strukturierter Form Forschungsdaten beschreiben. „*[Sie helfen,] Forschungsdaten zu managen, zu erschließen, zuverstehen und zu benutzen*“ [BHM11, S. 83]. Wie im Abschnitt 2.5 erklärt, kann keine Provenance ohne Metadaten existieren. An dieser Stelle wird das Metadatenmanagement im Framework betrachtet. Dabei wird Fokus auf die Speicherung der Metadaten von Forschungsrohdaten und Ergebnissen gelegt

und Analysemethoden vernachlässigt, da von diesen die Metadaten wie Beschreibung oder Erstellungsdatum durch das DBMS verwaltet werden. Lediglich temporale Informationen werden ergänzt.

3.6.1 Metadaten der Forschungsrohdaten

Liegen die Metadaten der Forschungsrohdaten nach jedem Messdurchlauf in der gleichen Struktur vor, so baut man, wie in Abschnitt 3.3.3 auf Seite 38 beschrieben, eine Tabelle mit ebendieser Struktur auf. Die Spaltennamen werden dabei vom Nutzer gewählt, da es häufig keine Metadaten über Metadaten gibt. Dabei wird als Spaltenname eine semantische Beschreibung des Metadatums gewählt. Liegt beispielsweise um eine Liste mit °C, m/s^2, ..., vor so wird z.B. „Unit“ als Spaltenname gewählt, da es sich hierbei offensichtlich eine Liste mit Messeinheiten handelt.

Liegen die Metadaten nicht in gleicher Struktur vor, beispielsweise weil die Metadaten per Hand geschrieben wurden und sich im Laufe der Zeit änderten, so wird im Framework dafür eine EAV-Schema verwendet. Dieses bietet den Vorteil, dass die Metadaten ergänzt werden können, ohne dass eine Schemaänderung stattfinden muss. Das neue Metadatum wird einfach als Attribut mit entsprechendem Wert ergänzt, wie beispielhaft in Tabelle 9 dargestellt.

ENTITY

e_id	e_name	e_parent
1	Project	NULL
2	MetaObject	1

ATTRIBUTE

a_id	a_name
1	createdByResearcher
2	createdByCompany
3	createdAtLocation
4	generatedAt
5	importedAt

VALUE_TEXT

a_id	e_id	value
1	1	Mark Lukas Möller
2	1	Universität Rostock
3	1	Hansestadt Rostock

VALUE_DATE

a_id	e_id	value
4	1	2016-01-27 01:02:03
5	1	2016-02-06 04:05:06

Tabelle 9: Metadaten in einem EAV-Format

Dabei wird „MetaObject“ als ein Kindobjekt des zu beschreibenden Nutzdatums „Project“ angesehen. Die Metadaten für das Nutzdatum werden dann als Attribute und Attributwerte für das „MetaObject“ realisiert. Auch die Metadaten im EAV-Schema besitzen temporale

Informationen im Sinne der Transaktionszeit (`tt_begin`, `tt_end`), diese wurden der Übersichtlichkeit halber hier nicht mit aufgeführt. Dies funktioniert so wie beschrieben nur, sofern auch die Nutzdaten in einem EAV-Schema vorliegen. Liegen die Daten aber in einem relationalen Schema vor, so wird die Beziehung zwischen dem Nutzdatenobjekt und dem Metadatenobjekt anders aufgebaut. Der oder die zu beschreibenden Nutzdatensätze bekommen dann einen im gesamten Forschungsdatenmanagement einmaligen Surrogatschlüssel[12], der dann in die `e_parent`-Spalte der Entity-Tabelle eingetragen wird.

3.6.2 Metadaten der Analyseergebnisse

Für die Analyseergebnisse werden ebenfalls Metadaten gespeichert. Im Sinne der Provenance muss festgehalten werden, welche Ergebnisse durch welche Funktionen aus welchen Datenbeständen entstanden sind. Sollte mehr als ein Berechnungsschritt gemacht werden, so ist es notwendig, die Zwischenergebnisse zu materialisieren, um den höchsten Grad von Provenance zu erreichen. Es wird gespeichert, zu welchem Zeitpunkt die Berechnung stattgefunden hat und welche Abfrage ausgeführt wurde, um die Ausgangsdaten zu rekonstruieren. Außerdem muss eine Metainformation darüber gehalten werden, wo das Ergebnis abgespeichert wurde. Dabei wird der Name der Tabelle gespeichert, in der die Ergebnistupel eingefügt wurden respektive die Sicht, die entstanden ist, sowie eine Surrogatschlüssel, mit dem sich entstandene Tupel referenzieren lassen. Wie die konkrete Umsetzung dafür aussieht, wird im Abschnitt 4.3 auf Seite 66 erläutert und mit einem Bespiel unterlegt.

3.7 Bewertung des Frameworks

In diesem Kapitel wurde das Konzept des Frameworks für das Forschungsdatenmanagement beschrieben. Es wurde erläutert, welche Komponenten zur Speicherung existieren und für welche Anwendungsfälle welche Komponenten Verwendung finden sollten.

Das Framework bietet die Möglichkeit, Daten in den verschiedensten Formen abzulegen. Sowohl un- bzw. semistrukturierte als auch strukturierten Daten können gespeichert werden. Es wurde ein Konzept für die Provenance vorgestellt, das auf temporalen Aspekten und Metadaten beruht und versucht, einen hohen Grad an Provenance zu realisieren.

Die möglichen Stufen der Provenance wurden in Tabelle. 5 auf Seite 29 veranschaulicht. Die erste Stufe der Provenance, die mindestens zur Plausibilität führt, wird durch die Speicherung von Experimentmetadaten realisiert. Metadaten werden dabei im Forschungsdatenmanage-

[12] Über die Tabelle hinweg eindeutig, z.B. durch eine GUID (Globally Unique Identifier) pro zu beschreibende Nutzdatensätze.

ment auf allen Ebenen realisiert: Sowohl bei der Speicherung von Forschungsrohdaten, als auch bei Analysefunktionen und -ergebnissen.

Da die Primärforschungsdaten im Forschungsdatenmanagement gespeichert werden, ist mindestens auch die zweite Ebene der Provenance, die Nachvollziehbarkeit, abgedeckt. Der weitere Grad der Provenance hängt von dem Nutzer des Frameworks ab. Beachtet man konsequent die Materialisierungskriterien, die im Abschnitt 3.5.1 beschrieben werden, so kann Provenance bis zum Grad der Rekonstruierbarkeit gegeben sein.

Nicht beachtet wurden zum jetzigen Zeitpunkt Aspekte zur Provenance von externen Prozeduren, Datenbereinigung z.B. durch Sichten sowie konkrete Konzepte zur Datenrestriktion.

4 Konzeption und Umsetzung für GODESS

In diesem Kapitel werden zu den generellen Konzepten aus dem vorherigen Kapitel zusätzliche Konzepte für das Forschungsdatenmanagement der GODESS-Forschungsdaten sowie deren Umsetzung vorgestellt. Es sollen die Teilprobleme, die sich im Kapitel 3 herausgestellt haben, konkret für GODESS gelöst werden. Die Umsetzung wird mit MySQL realisiert.

4.1 Speicherung der Forschungsrohdaten

4.1.1 Überführung von Forschungsdaten in Datenbanken

Für das Überführen der Daten der GODESS in das DBMS werden drei Schritte vorgenommen.

Der erste Schritt besteht aus der Analyse der Rohdaten. Es wird betrachtet, was für Daten vorliegen und ob es Schemata gibt, auf denen man diese Daten abbilden kann (vgl. Abschnitt 3.3). Es müssen Datentypen bestimmt werden sowie überprüft werden, ob zwischen Datensätzen Abhängigkeiten existieren. Es wird geprüft, ob Metadaten zur Beschreibung der Datensätze existieren.

Darauf aufbauend wird im nächsten Schritt untersucht, wie ein Datenbankschema für die Speicherung der Daten aussehen kann. Dabei werden die aus dem ersten Schritt gewonnenen Informationen maßgeblich mit einbezogen, um Entitäten und Relationen zu identifizieren. Parallel dazu wird untersucht, ob und welche Informationen zusätzlich mitgeführt bzw. gespeichert werden müssen, um die Anforderungen an Provenance zu erfüllen.

Der dritte Schritt besteht aus der Datenmigration. Mithilfe eines Java-Programms werden die Dateien mit den Rohdaten eingelesen und geparst. Anschließend wird eine Verbindung zur Zieldatenbank aufgebaut und die Daten in das definierte Datenbankschema übertragen.

Für die Erstellung des Datenbankschemas ist als erster Schritt eine Vorbetrachtung der Dateien mit den Rohdaten nötig. An dieser Stelle soll daher eruiert werden, welche Datenbankschemata zur Speicherung genutzt werden können, welche Abhängigkeiten existieren

und ob Schlüsselkandidaten in den Daten ausfindig gemacht werden können.

Es existieren drei Arten von Nutzdaten, die analysiert werden müssen: Die Dateien des Powerloggers, die des Aquadopp-Strömungsmessers und die des Trios-PROPS-Spektralanalysegerätes der GODESS. Zusätzlich müssen pro Deployment Korrekturdatensätze sowie eine Deployment-Informationsdatei in der Datenbank gespeichert werden.

4.1.2 Analyse der Powerloggerdaten

Die Struktur der Daten des Powerloggers wurde schon einmal in den Abbildungen 3 und 4 auf Seite 16 vorgestellt. Ausführlichere Beispiele werden in A.1 und A.2 im Appendix auf Seite 78 bzw. Seite 79 aufgeführt; ein komplettes Beispiel ist auf der beiliegenden CD enthalten.

Die Daten des Powerloggers liegen in einer Textatei vor, die als Titel das aktuelle Datum mit voran- und nachgestelltem „A“ trägt, also zum Beispiel “A150602A.TXT“. Die Datei beinhaltet drei verschiedene Datensätze: Den Metadatenteil, der die Datei beschreibt und zeitliche Informationen zum Messzyklus beinhaltet (vgl. Abb. A.1, Z. 2-8), einen weiteren Metadatenteil, der die Nutzdaten beschreibt (vgl. Abb. A.1, Z. 10-17) und einen Nutzdatenteil (vgl. Abb. A.2 ab Z. 2). Jeder dieser Teile kann mehrfach vorkommen. In einer Datei können beispielsweise 20 Metadatenteile zur Datei, 20 Metadatenteile zu den Nutzdaten und tausende Nutzdatenteile vorhanden sein.

4.1.2.1 Metadaten der Powerloggerdatei

Die Metadaten, die die Datei beschreiben, liegen in einem Key-Value-Format vor, wobei die Keys und die Values alle als Datentyp String behandelbar sind. Entsprechend können die auf `CHAR` oder `VARCHAR` abgebildet werden. Zusammengesetzte Values wie z.B. *U1433268106 "2015-06-0218:01:46"*, die man auch auf `TIMESTAMP` und `DATETIME` abbilden könnte, werden dennoch als String behandelt. Die Verletzung der ersten Normalform bei der Datenbanknormalisierung wird bewusst in Kauf genommen. Der Hintergrund für diese Entscheidung ist die, dass bereits Scripte mit Berechnungsvorschriften existieren, die diese Daten genauso wie vorliegend erwarten. Auch wenn es z.B. sinnvoll wäre, den genannten Attributwert in zwei separate Werte aufzuspalten, so wird an dieser Stelle darauf verzichtet, damit man in den Scripten im ersten Schritt nur die Datenquellen anpassen muss und nicht die Logik editieren muss, die diese Daten nutzt. In einem weiteren Schritt lässt sich dann eine normalisierte Sicht erstellen.

In folgendem Format liegen die Metadaten der Powerloggerdatei vor:
$@key_1 \quad value_{1,1}$

$@key_2 \quad value_{2,1}$

$\vdots$

$@key_m \quad value_{m,1}$

Beispiel:

```
1 @FILETITEL    ASCII-DATA
2 @FILENAME     C:\ASCIIDAT\A150602A.TXT
3 @DATETIME     U1433268106"2015-06-02"  "18:01:46"
```

Da es zwischen den verschiedenen Metadatensätzen eine gleichbleibende Anzahl an homogenen Metadatenattributen gibt, eignet es sich, eine Tabelle, zu entwerfen, bei der die Keys als Spaltennamen gewählt werden und die Tupel als Values behandelt werden. Neben den eigentlichen Metadaten werden, wie im Kapitel 3 beschrieben, Transaktionszeiten eingeführt, um Daten analysieren zu können, die in einem bestimmten Zeitraum vorhanden waren. Das resultierende Datenbankschema für die Datei-Metadaten ist in der Abb. 9 ersichtlich.

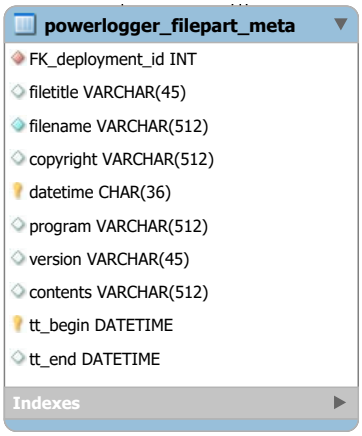

Abb. 9: Metadaten-Entität für die Dateien des Powerloggers.

Daraus abgeleitet wird folgende DDL zum Erzeugen dieser Tabelle ausgeführt[13]:

```
CREATE TABLE `godess`.`powerlogger_filepart_meta` (
  `FK_deployment_id` INT,
  `filetitle` VARCHAR(45),
  `filename` VARCHAR(512) NOT NULL,
  `copyright` VARCHAR(512),
  `datetime` CHAR(36) NOT NULL,
  `program` VARCHAR(512),
  `version` VARCHAR(45),
  `contents` VARCHAR(512),
  `tt_begin` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tt_end` DATETIME NULL DEFAULT '9999-12-31 23:59:59' ,
  PRIMARY KEY (`datetime`, `tt_begin`)
  CONSTRAINT `file_belong_to_deployment`
    FOREIGN KEY (`FK_deployment_id`)
    REFERENCES `godess`.`deployments` (`deployment_id`)
);
```

Die Datentypen werden an dieser Stelle alle als String behandelt, da kein anderer Datentyp abgeleitet werden kann. In SQL werden diese also auf `CHAR` oder `VARCHAR` abgebildet. Die Spalte `datetime` wird als `CHAR` behandelt, da der Attributwert kein reines Datum sondern noch zusätzlich das Datum als UNIX-Timestamp enthält. Als Schlüsselkandidaten wird das eindeutige Attribut `datetime` in Kombination mit `tt_begin` verwendet. Durch die Einführung von `tt_end` wird ein gewisser Grad an Fehlertoleranz sichergestellt: Generell werden Nutzdaten nicht aus dem System gelöscht. Abgebrochene oder teilweise fehlgeschlagene Importe können invalidiert und neu initiiert werden. Dies ist besonders dann relevant, wenn falsch importierte Daten bereits genutzt und publiziert wurden. Bei Löschung würden Provenancekriterien verletzt werden. Es existiert ein Fremdschlüssel zur Deployment-Relation, die später beschrieben wird. Dabei handelt es sich um eine *1:n*-Beziehung, wobei die Metadatentabelle für die Powerloggerdatei auf der 1-Seite steht.
Ein Beispiel für in das Schema überführte Daten ist in der Beispieltabelle A.1 auf Seite 81 ersichtlich.

[13] Der Defaultwert `CURRENT_TIMESTAMP` ist erst mit MySQL 5.6.5 für `DATETIME` verfügbar.
Siehe dazu auch `http://stackoverflow.com/a/168832/2682923`. Für frühere Versionen eignet es sich, keinen Defaultwert zu verwenden und beim Einfügen von Daten das `DATETIME`-Feld mit `SELECT NOW()` als Value zu belegen.

4.1.2.2 Metadaten der Powerlogger-Nutzdaten

Für die Metadaten der Nutzdaten wird aufgrund der Struktur der Daten eine ähnliche Herangehensweise wie bei der Powerloggerdatei genutzt. Die Daten liegen schematisch, wie die Metadaten der Powerloggerdatei, im Key-Value-Format vor, besitzen aber mehrere Values (vgl. A.1):

$$
\begin{aligned}
&@key_1, value_{1,1}, value_{1,2}, \ldots, value_{1,n} \\
&@key_2, value_{2,1}, value_{2,2}, \ldots, value_{2,n} \\
&\vdots \\
&@key_m, value_{m,1}, value_{m,2}, \ldots, value_{m,n}
\end{aligned}
$$

Beispiel:

```
1 @CHANNEL,_,_,_,_,0 ,1 ,2 ,3 ,4 ,5 , ...
2 @NAME,DATE,TIME,FRAC,SEC, VBatt , Press , Temp, Cond , RawO2, ...
3 @UNIT , YY-MM-DD, HH:MM: SS , MS, RUNSEC, V, dBar , ∅C, mS/cm, , ...
```

Dabei stehen $value_{m,x}$ und $value_{n,x}$ im kausalen Zusammenhang, da sie das gleiche Datum des Nutzdatensatzes beschreiben, der sich im gleichen Format an Stelle x befindet. Beispielsweise wird auf Channel 0 der Batteriestatus *VBatt* in der Einheit V gemessen. Diese Informationen stehen in jeder Zeile an sechster Stelle. Bei den in ähnlicher Struktur vorliegenden Nutzdaten wird der an sechster Stelle stehender Nutzdatenwert beschrieben.
Auch in dieser Tabelle werden, wie in Abb. 10 abgebildet, temporale Informationen mitgeführt. Analog zu den Metadaten der Datei entsprechen auch hier die Keys den Spaltennamen und die Values den Tupeln. Die Kombination aus dem Attribut `name` und `tt_begin` ist eindeutig und wird als Schlüssel verwendet.

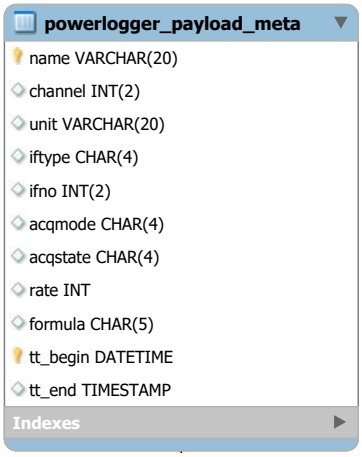

Abb. 10: Metadaten-Entität für den Payload des Powerloggers.

Ausgehend davon ergibt sich folgende DDL zur Erzeugung der Metadatentabelle:

```
CREATE TABLE `godess`.`powerlogger_payload_meta` (
  `name` VARCHAR(20) NOT NULL,
  `channel` INT(2),
  `unit` VARCHAR(20),
  `iftype` CHAR(4),
  `ifno` INT(2),
  `acqmode` CHAR(4),
  `acqstate` CHAR(4),
  `rate` INT,
  `formula` CHAR(5),
  `tt_begin` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tt_end` DATETIME DEFAULT  '9999-12-31 23:59:59',
  PRIMARY KEY (`name`, `tt_begin`)
);
```

4.1.2.3 Nutzdaten der Powerloggerdatei

Ähnlich wie die die Metadaten der Powerloggernutzdaten liegen die Nutzdaten in einem kommaseparierten Format mit mehreren Values vor. Der Key, der bei den Powerlogger-Metadaten den Spaltennamen gebildet hat, ist bei den vorliegenden Daten nicht vorhanden. Die Nutzdaten des Powerloggers bestehen nur aus kommaseparierten Values. Daher steht das Problem der Spaltenbenennung im Raum. Entsprechend der in Abschnitt 3.3.5 beschriebenen Vorgehensweise zur Benennung der Spalten wird zuerst in den Metadaten geschaut, ob sich Informationen bezüglich der Spaltennamen finden lassen. Da die Metadatenzeile beginnend mit `@NAME` Informationen darüber enthält, welche Semantik sich in den Nutzdatensätzen verbirgt, werden die ebendiese Informationen als Spaltennamen wie z.B. `date`, `time`, `press`, `cond`, etc. verwendet (vgl. Abb. A.1, Z. 10).

Aufgrund des Umfangs befindet sich die DDL im Appendix im Abschnitt A.7 auf Seite 83. Die Entität ist im kompletten Entity-Relationship-Diagramm in Abb A.3 auf Seite 80 ersichtlich.

4.1.2.4 Verknüpfung von Nutzdaten und Metadaten

Aufgrund des Schemas von den Metadaten der Nutzdaten und den Nutzdaten selbst ist davon ausgehen, dass ein Metadatensatz einem Nutzdatensatz mehrmals zugeordnet werden kann und umgekehrt[14]. Da die Metadaten der Nutzdaten Informationen enthalten, die über

[14]Da die Nutzdaten zeilenweise als n-Tupel gespeichert werden und jede **Zelle** durch ein Metadatum beschrieben wird, besitzt jedes Tupel in der Nutzdatentabelle n Metadaten

mehrere Messzyklen identisch bleiben, wie z.B. Messeinheit oder der Name des Messdatums, ist es an diese Stelle möglich, eine *m:n*-Beziehung aufzubauen. Daher werden die Daten in einer dritten Tabelle festgehalten, die die Informationen über die Beziehung speichert.

```
CREATE TABLE `godess`.`powerlogger_payload_has_meta` (
  `P_date` DATE NOT NULL,
  `P_time` TIME NOT NULL,
  `P_frac` INT NOT NULL,
        `P_tt_begin` DATETIME NOT NULL,
  `PM_name` VARCHAR(20) NOT NULL,
  `PM_tt_begin` DATETIME NOT NULL,
  `tt_begin` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tt_end` VARCHAR(45) NOT NULL DEFAULT '9999-12-31 23:59:59',
  PRIMARY KEY (`P_date`, `P_time`, `P_frac`, `P_tt_begin`,
  `PM_name`, `PM_tt_begin`, `tt_begin`),
  CONSTRAINT `payload_references`
    FOREIGN KEY (`P_date` , `P_time` , `P_frac`, `P_tt_begin`)
    REFERENCES `godess`.`powerlogger_payload`
      (`date`, `time`, `frac`),
  CONSTRAINT `payloadmeta_references`
    FOREIGN KEY (`PM_name` , `PM_tt_begin`)
    REFERENCES `godess`.`powerlogger_payload_meta`
      (`name`, `tt_begin`)
);
```

Die Tabelle enthält dabei die Primärschlüsselattribute der Nutzdatentabelle und der Metadatentabelle für Nutzdaten. Weiterhin wurden die Transaktionszeitenattribute `tt_begin` und `tt_end` ergänzt. Der Primärschlüssel ist dabei ein zusammengesetzter Primärschlüssel aus den Primärschlüsselattributen der referenzierten Tabellen sowie dem Beginn der Transaktionszeit.

4.1.3 Analyse der Spektralanalysedaten

Ähnlich den Powerloggerdaten bestehen die Spektralanalysedaten aus zwei Metadatenteilen sowie aus einem Nutzdatenteil. Die Daten liegen in Textdateien vor, die im Namen das Präfix „PC“ bzw. „PM“ tragen, gefolgt von dem Tag, der Stunde und der Minute des Messens, sowie einer laufenden Nummer nach einem Punkt. Beispielhaft ist die Datei „PC291758.0“ am 29.06.2015 um 17:58 erstellt worden. Der Monat und das Jahr des Messvorgangs gehen nicht aus dem Titel hervor.

4.1.3.1 Metadaten der Spektralanalysedaten

Die zwei verschiedenen Metadatenteile der Datei liegen in einem Key-Value-Format vor. Allerdings bestehen diese, im Gegensatz zu den Metadaten der Powerlogger, aus je einem Key und je einer Value (und nicht etwa mehreren Values). Da pro Datei jeder Metadatenteil jeweils genau einmal vorkommt, ist es möglich, die beiden Teile in einer Relation unterzubringen. Wie folgt liegen die Daten vor:

```
1  [Spectrum]
2  Version=1
3  IDData=2015-06-29_17-58-34_264
4  IDDevice=ProPS_D016
5  IDDataType=SPECTRUM
6  ⋮
7  [Attributes]
8  CalFactor=1974
9  IntegrationTime=512
10 Unit1=$05 $00 Pixel
11 Unit2=$03 $05 Intensity Counts
12 Unit3=$f0 $05 Error counts
13 Unit4=$f1 $00 Status
```

Wie in Abschnitt 4.1.2.1 beschrieben, eignet es sich auch hier, eine Tabelle zu entwickeln, die die Keys als Spaltennamen besitzt und die Values entsprechend als Tupel. Auch hier werden wieder Transaktionszeitattribute ergänzt. Folgende DDL wird zur Erstellung der Tabellen genutzt:

```
CREATE TABLE `godess`.`spectrum_meta_and_attributes` (
  `FK_deployment_id` INT,
  `version` INT,
  `idData` VARCHAR(23),
  `idDevice` VARCHAR(45),
  `idDatatype` VARCHAR(32),
  `idDatatypeSub1` VARCHAR(32),
  `idDatatypeSub2` VARCHAR(32),
  `dateTime` CHAR(19),
  `comment` VARCHAR(45),
  `commentSub1` VARCHAR(45),
  `commentSub2` VARCHAR(45),
  `commentSub3` VARCHAR(45),
  `calFactor` INT,
  `integrationTime` INT,
  `unit1` VARCHAR(32),
  `unit2` VARCHAR(32),
  `unit3` VARCHAR(32,
  `unit4` VARCHAR(32),
  `tt_begin` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tt_end` DATETIME NULL DEFAULT  '9999-12-31 23:59:59',
  PRIMARY KEY (`idData`, `tt_begin`) ,
  CONSTRAINT `FK_deployment_id`
    FOREIGN KEY (`FK_deployment_id`)
    REFERENCES `godess`.`deployments` (`deployment_id`)
);
```

Die Kombination aus dem Attribut `idData`, dessen Attributwert ein Datum und eine Uhrzeit enthält, sowie dem Beginn der Transaktionszeit ist eindeutig und wird als Primärschlüssel verwendet.

4.1.3.2 Nutzdaten der Spektralanalysedaten

Die Nutzdaten der Spektralanalysedaten liegen in einem Format vor, das einer Matrix gleichkommt:

```
1 [DATA]
2 0 9 0 0
3 1 1357 0 0
4 2 1321 0 0
5 3 1305 0 0
6 4 1325 0 0
```

Dabei werden pro Datei 256 Nutzdatentupel mit je vier Attributen gespeichert. Die semantische Bedeutung erschließt sich aus dem `[Attributes]`-Abschnitt der Metadaten. Die vier Keys `Unit1` bis `Unit4` enthalten die Beschreibung für die vier Spalten. Es handelt sich dabei um den Pixel (*Pixel*), die Intensität, die bei dem entsprechenden Pixel registriert wurde (*Intensity Counts*), die Anzahl der aufgetretenen Fehler (*Error Counts*) und den Status (*Status*). Entsprechend werden diese für die Spaltennamen verwendet. Neben den genannten vier Spalten existiert ein Fremdschlüsselattribut, dass den eindeutigen Dateinamen enhält und die Metadatentabelle referenziert. Weiterhin werden die Transaktionszeitinformationen mitgeführt. Die Kombination aus den referenzierten Werten der Metadatentabelle, der Pixelnummer sowie dem Attribut für den Beginn der Transaktionszeit ist eindeutig und wird daher als Primärschlüssel gewählt. Die Primärschlüsselattributwerte (`idData` und `tt_begin`) der Metadaten dienen in dieser Relation die Fremdschlüsselattributwerte zur Referenzierung dieser.

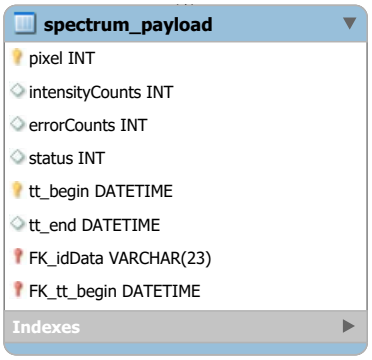

Abb. 11: Nutzdaten-Entität für die Daten des Spektralanalysegerätes

Folgende DDL wird zum Erzeugen der Tabelle verwendet:

```
CREATE TABLE `godess`.`spectrum_payload` (
  `pixel` INT NOT NULL,
  `intensityCounts` INT,
  `errorCounts` INT,
  `status` INT,
  `tt_begin` DATETIME NOT NULL,
  `tt_end` DATETIME NULL DEFAULT  '9999-12-31 23:59:59',
  `FK_idData` VARCHAR(23) NOT NULL,
  `FK_tt_begin` DATETIME NOT NULL,
  PRIMARY KEY (`pixel`, `tt_begin`, `FK_idData`, `FK_tt_begin`),
  CONSTRAINT `spectrum_payload_has_meta`
    FOREIGN KEY (`FK_idData` , `FK_tt_begin`)
    REFERENCES `ba`.`spectrum_meta_and_attributes` (`idData` , `tt_begin`)
);
```

4.1.4 Analyse der Strömungsmessdaten

Die Datei für die Daten des Strömungsmessers ist im Gegensatz zu den Powerlogger- und Spektraldaten unkomplizierter aufgebaut. Sie enthält keine Metadatenattribute, sondern nur drei Nutzdatenattribute: Ein Datum, eine Uhrzeit und einen hexadezimalen String. Der Dateiname besteht dabei aus dem Präfix „NO“, gefolgt von der zweistelligen Jahreszahl, dem Monat und dem Tag der Datenerhebung, z.B. „NO150602“ als Dateiname für die Datei, die am 02.06.2015 erstellt wurde. Die Daten liegen wie folgt in der Datei vor:

```
1 2015-06-02 18:01:46 A521C400294612070201000000007300A0380E05F5FF1500031 ...
2 2015-06-02 18:01:48 A521C400294712070201000000007300A0380C05F7FF1500031 ...
3 2015-06-02 18:01:49 A521C400294812070201000000007400A0380705F8FF1600031 ...
```

In das Schema werden zusätzlich zu den drei Nutzdaten eine Fremdschlüsselspalte für die zugehörige Deployment-ID, sowie Transaktionszeitinformationen aufgenommen. Die Kombination aus dem Datum, der Zeit und dem Beginn der Transaktionszeit bildet den Primärschlüssel. Die Spalte Deployment-ID dient als Fremdschlüssel zur Referenzierung der zugehörigen Deployment-Relation. Entsprechend wird wie in Abb. 12 dargestellt die Entität gebildet. Nach Konvention zur Vergabe der Spaltennamen, welche in Abschnitt 3.3.5 beschrieben wurde, wird zuerst geschaut, ob es beschreibende Metadaten gibt. Das ist hier nicht der Fall. Es wird geprüft, ob weitere Informationen zur Semantik existieren. Zumindest bei den ersten beiden Spalten ist offensichtlich, dass es sich hierbei um ein Datum sowie eine Uhrzeit handelt; entsprechend werden die beiden Spaltenbezeichungen „date“ respektive „time“ gewählt. Bei der dritten Spalte handelt es sich um den Payload des Strömungsmessers in hexadezimaler Darstellung. Hierfür wird „hex_payload“ als Spaltenbezeichnung gewählt.

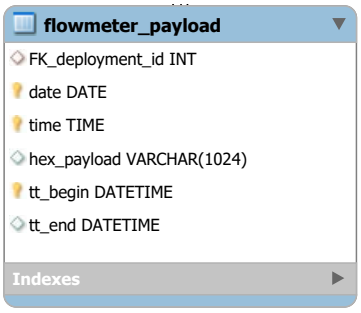

Abb. 12: Entität für die Daten des Spektralanalysegerätes

Folgende DDL wird zum Erzeugen der Tabelle verwendet:

```
CREATE TABLE IF NOT EXISTS `godess`.`flowmeter_payload` (
  `FK_deployment_id` INT,
  `date` DATE NOT NULL,
  `time` TIME NOT NULL,
  `hex_payload` VARCHAR(1024),
  `tt_begin` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tt_end` DATETIME DEFAULT  '9999-12-31 23:59:59',
  PRIMARY KEY (`date`, `time`, `tt_begin`),
  CONSTRAINT `deployment_has_hexdata`
    FOREIGN KEY (`FK_deployment_id`)
    REFERENCES `godess`.`deployments` (`deployment_id`)
);
```

4.1.5 Speicherung der Deployment-Metadaten

An dieser Stelle soll untersucht werden, wie die Metadaten zum konkreten Deployment – Daten zu verwendeten Sensoren, Länge- und Breitengrad, etc. – gespeichert werden, die in der Textdatei `SUMMARY_GODESS_DEPLOYMENTS.txt` erfasst wurden. Trotz dessen, dass die vorliegende Datei händisch geschrieben wurde, wird auf ein EAV-Schema, wie in der Vorgehensweise in Abschnitt 3.6.1 beschrieben, verzichtet, da die gleichen homogenen Daten über alle Messdurchläufe hinweg vorliegen. Es wird ein relationales Schema verwendet.

Die Datei enthält zuerst Informationen über alle Instrumente, die je zum Messen verwendet wurden. Zusätzlich sind Parameter zu den einzelnen Sensoren erfasst, z.B. Werte für den Drucksensor, Temperatursensor oder pH-Sensor.

```
1 0a: SubCtech Powerlogger, S/N 1 (according to cap label)
2 0b: SubCtech Powerlogger, S/N 2 (according to cap label)
3 1a: Sea & Sun CTD, CTM468 :
4 Pressure Sensor: 4681   Temperature Sensor: 4681
    Conductivity Sensor: 4681       Seapoint Turbidity
    Sensor: 12023       AMT pH Sensor:  4681    AMT ORP
    Sensor: 4681    Turner Chla Sensor: 2101167
5 1b: Sea & Sun CTD, CTM468 :
6 Pressure Sensor: 4681   Temperature Sensor: 4681
    Conductivity Sensor: 4681       Seapoint Turbidity
    Sensor: 12023       AMT pH Sensor:  4682    AMT ORP
    Sensor: 4682    Turner Chla Sensor: 2101167
```

Anschließend werden Information darüber gehalten, welche Sensoren bei welchem Deployment zum Einsatz kamen. Zusätzlich werden weitere Metadaten wie Längen- und Breitengrad gespeichert. Die Daten liegen in Form einer Tabelle vor, deren Einträge mit Leerzeichen getrennt wurden.

Number	Date Depl.	Date rec.	Winch Number/PIP	Profile Interval	Winch/Logger(CTD)	Sensors	Lon	Lat	Notes
01	09-05-2010	10-05-2010	06/1	1 h	17/17	1a, 3b	57,3203	20,137	Test Gotland basin

Tabelle 10: Aufbau der Deployment-Metadaten

Letztendlich befinden sich optionale Kommentare zum Deployment in der Tabelle (zusätzlich zu denen in der vorhandenen Spalte `Notes`). Die Kommentare werden durch die Deployment-ID eingeleitet und enthalten eine textuelle Beschreibung:

```
1 Deployment 03:
2 CTD ran out of power due to unknown error
3 PIP was floating on sea surface because the winch ran out of power and let
      loose during profiling (after about 220 profiles).
```

Die Deployment-Informationen in der obigen Tabelle stehen offensichtlich in einer *m:n*-Relation mit den verwendeten Messinstrumenten: Bei einem Messzyklus wurden mehrere Instrumente verwendet, die Instrumente selbst kamen bei mehreren Messzyklen zum Einsatz. Entsprechen sehen die Tabellen für die Instrumente, die Deployments und die Verknüpfung zwischen beiden Tabellen wie folgt aus:

Die Instrumente werden in einer eigenen Relation abgelegt. Die Datensätze besitzen neben der ID des entsprechenden Instrumentes, welche aus einer Ziffer und einem Buchstaben besteht, die genannten Attribute zum Drucksensor, Temperatursensor etc. Als Primärschlüssel. wird dabei die Sensor-ID in Kombination mit dem Beginn der Transaktionszeit gewählt. Die resultierende Relation ist in Abb.14 ersichtlich. Die Informationen zum Deployment werden in der Relation gespeichert, die in Abb. 13 ersichtlich ist.
Verknüpft werden die beiden Informationen dann über eine Tabelle, die die Primärschlüssel der beiden Tabellen enthält, wie in Abb. 15 dargestellt.

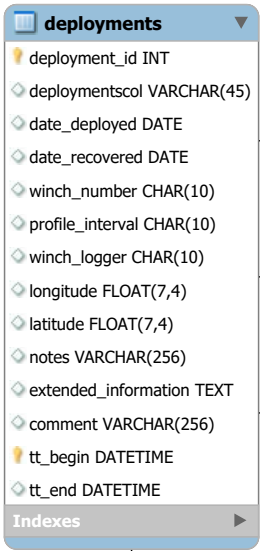

Abb. 13: Entität für die Daten der Deployments

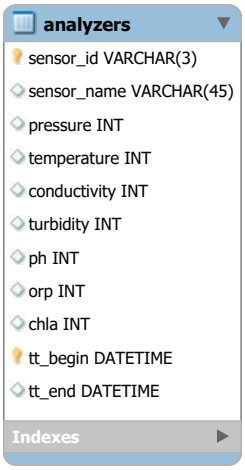

Abb. 14: Entität für die Daten der Messgeräte

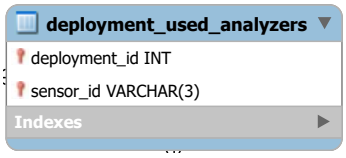

Abb. 15: Entität für die Verknüpfung von Deploymentdaten und Messgeräten

Die DDL zum Erstellen der Deployment-Tabelle ist folgende:

```
CREATE TABLE `ba`.`deployments` (
  `deployment_id` INT NOT NULL,
  `deploymentscol` VARCHAR(45),
  `date_deployed` DATE,
  `date_recovered` DATE,
  `winch_number` CHAR(10),
  `profile_interval` CHAR(10),
  `winch_logger` CHAR(10),
  `longitude` FLOAT(7,4),
  `latitude` FLOAT(7,4),
  `notes` VARCHAR(256),
  `extended_information` TEXT,
  `comment` VARCHAR(256),
  `tt_begin` DATETIME NOT NULL,
  `tt_end` DATETIME NULL DEFAULT '9999-12-31 23:59:59' COMMENT '',
  PRIMARY KEY (`deployment_id`, `tt_begin`)
);
```

In der Tabelle für die Daten des Deployments ist die Spalte `extended_information` vorhanden (vgl. Abb. 15). Diese Spalte ist als CLOB (in MySQL „`TEXT`“ typisiert und enthält die Korrekturdaten, die im XML-Format vorliegen. Das gesamte XML-Dokument wird dabei als Attributwert gespeichert. Ein Ausschnitt des XML-Dokumentes sei der Folgende.

```
1 <GODESS_deployment>
2   <Metadaten>
3     <deployment>
4       <deployment_number>12</deployment_number>
5     </deployment>
6   </Metadaten>
7 </GODESS_deployment>
```

MySQL bietet die Möglichkeit, wie mit XPath auf beliebige XML-Knoten zuzugreifen. Möchte man nun auf den Wert des Knotens `<deployment_number>` zugreifen, so ist dies mit folgender Abfrage möglich:

```
SELECT EXTRACTVALUE(extended_information,
  '//GODESS_deployment/Metadaten/deployment/deployment_date'
) FROM ba.deployments;
```

Da die eben beschriebenen Deployment-Metadaten von Hand geschrieben werden und nicht in einer eindeutigen Form vorliegen, sollten diese Daten manuell in die Datenbank geschrieben werden, also nicht durch ein Programm geparst und übertragen werden. Automatisierbar ist dieser Prozess, wenn die Deployment-Metadaten ähnlich den Korrekturdaten in einer (semi-)strukturierten Form gespeichert werden.

4.1.6 Migration der Rohdaten

Zum Migrieren der Nutzdaten wird Java genutzt. Nachdem die Struktur der Daten herausgestellt und entsprechende Datenbankschemata angelegt wurden, werden von einem Java-Programm die Dateien eingelesen. Der erste Schritt besteht dabei aus dem parsen der Daten. Dafür wird pro Datei ein Datenstrom geöffnet, aus dem die Daten zeilenweise gelesen werden. Der Parser entscheidet, ob es sich um ein Metadaten- oder Nutzdatenattribut handelt und leitet es an einen weiteren Parser weiter. Dieser wiederum entscheidet, um welches Attribut es sich gerade handelt und speichert es entsprechend in einer dafür definierten Java-Klasse ab.

Nachdem alle Dateien eingelesen und geparst worden sind, wird mittels Java Database Connectivity (JDBC) der Datenbanktreiber geladen, eine Datenbankverbindung aufgebaut und die Daten in die entsprechenden Tabellen übertragen. Der entsprechende Quellcode wird bereitgestellt.

4.2 Speicherung von Analysemethoden

4.2.1 Speicherung als Routine

Wiederkehrende Analysemethoden sollen – sofern mit MySQL realisierbar – auf Datenbankebene als Routine gespeichert und ausgeführt werden. Neben der eigentlichen Speicherung steht hier die Problematik der Historisierung von Analysemethoden im Vordergrund. Es muss davon ausgegangen werden, dass sich die Analysemethoden bzw. deren Berechnungsvorschriften im Laufe der Zeit ändern. Für die Provenance ist es notwendig, auch auf mittlerweile ungültige Analysemethoden Zugriff zu haben, um Berechnungen rekonstruieren zu können. Ein Lösungsansatz wird erbracht, um die Definitionen der SPs und der UDFs so zu speichern, dass deren Definitionen auch nach dem Löschen oder nach dem Aktualisieren der Methodendefinition weiterhin zur Verfügung stehen, wie sie vor dem Löschen bzw. vor dem Aktualisieren waren.

4.2.2 Provenance der Analysemethoden

Beim Anlegen von Routinen werden diese im DBMS gespeichert. Informationen wie beispielsweie Erstellungsdatum, Routinedefinition und -typ (UDF, SP) sind bei MySQL über die Sicht `ROUTINES` der Datenbank `INFORMATION_SCHEMA` einsehbar. Dabei handelt es sich um eine vom DBMS erzeugte read-only Sicht, d.h. MySQL stellt hier nur Informationen dar. Änderungen z.B. an Routinedefinitionen können nicht über Änderungen an dieser Sicht vorgenommen werden. Routinen werden von MySQL nur in der aktuellen Version gespeichert und sind somit in der Sicht auch nur in der aktuellen Version hinterlegt. Das bedeutet, dass beim Editieren einer Routine diese in der alten Version komplett gelöscht und neu angelegt wird. Das DBMS legt keine Historisierungsinformationen der Routinen an, weswegen es nicht möglich ist, auf ältere Definitionen einer Routine zuzugreifen.

Es soll eine bitemporale Relation existieren, die die Informationen der `ROUTINES` Relation persistent halten kann und zusätzlich dazu Gültigkeitsinformationen speichert. Weiterhin müssen auch die Einträge in der `PARAMETERS` Relation archiviert werden, die die Informationen über die In-/Out-/InOut-Parameter der Routinen halten. Die DDL für die Erstellung solcher Tabellen ist auf Seite 82 ersichtlich.

Die Tabellen `routines_history` und `routines_parameter_history`, die nach Ausführung der DDL erstellt wurden, sind nun nach dem selben Schema wie die `INFORMATION_SCHEMA`-Sicht für Routinen bzw. Parameter aufgebaut und enthalten jeweils zusätzlich die Spalten `vt_begin` und `vt_end` für die Gültigkeitsinformationen. Beim Anlegen der Tabellen werden die Tupel der `ROUTINES` und `PARAMETERS`-Informationsschemasichten gleich mit in die entsprechenden Tabellen eingefügt. Der Beginn des Transaktionszeitpunktes wird dabei standardmäßig auf den Zeitpunkt gesetzt, an dem die Routine im DBMS angelegt wurde und das Ende des Transaktionszeitpunktes auf den maximal möglichen Wert `'9999-12-31 23:59:59'`. Sollten im DBMS schon SPs oder UDFs für das GODESS Projekt existieren, so müssen in den Historisierungstabellen die Gültigkeitsinformationen entsprechend angepasst werden, da diese beim Übernehmen aus der Informationsschemasicht in die Historisierungstabellen mit `NULL` belegt werden.

Um zukünftige Routinen zu historisieren, wird die Prozedur `materializeLatestRoutine` verwendet, die sich im Abschnitt A.8 auf Seite 84 befindet. Diese Prozedur verwendet die Tabelle `ROUTINES` der Informationsschemasicht und selektiert die jüngste Prozedur, die noch nicht in der `routines_history` Relation vorhanden ist. Es werden dabei Routinen ignoriert, die `DoNotMaterialize` im Routine-Kommentar haben, damit reine Hilfroutinen wie `materializeLatestRoutine`, die irrelevant für die Provenance sind, nicht erfasst werden. Anschließend werden – falls vorhanden – die Routineparameter ebenfalls in die dafür vorge-

sehene Archivierungsrelation übernommen. Wichtig ist, dass die Prozedur jedes Mal nach dem Anlegen einer Routine aufgerufen wird, da wie beschrieben nur die jüngste, noch nicht vorhandene Routine übernommen wird und nicht etwa alle noch nicht archivierten Routinen.

Folgende Schritte werden abgearbeitet, um eine Routine persistent zur Provenance-Tabelle hinzuzufügen:

- Erstellen der Routine auf gewöhnlichem Weg (`CREATE PROCEDURE ...` respektive `CREATE FUNCTION ...`).
- Übernahme in den Historisierungs-Relation durch Aufruf von `CALL materializeLatestRoutine(<vt_begin>, <vt_end>)`. Die Historisierung wird direkt nach dem Erstellen der Routine durchgeführt.

Soll nun eine Routinedefinition durch eine neue Definition abgelöst oder ganz gelöscht werden. So werden folgende Schritte durchgeführt:

- Invalidieren der alten Routine in der Provenance-Relation durch Setzen der `tt_end`-Zeit. Mit dem Setzen der `tt_end`-Zeit wird die Funktion als gelöscht markiert und kann auch physisch gelöscht werden.
- Gegebenenfalls anpassen der `vt_end`-Zeit, falls dieses Datum vorher unbekannt war und auf den Maximalwert gesetzt war.
- Löschen der alten Routinedefinition aus dem DBMS. Da – im Gegensatz zu Tupeln – zwei Prozeduren mit gleichem Namen aber unterschiedlichen Gültigkeitszeiten nicht koexistieren können (der Name einer Prozedur muss immer eindeutig sein), muss eine Prozedur nach dem invalidieren zwingend aus dem DBMS (`nicht` aber aus der Historisierungs-Relation) gelöscht werden, um eine gleichnamige Routine mit anderer Definition erstellen zu können.
- Bei Ersetzung: Definieren der neuen Routine und Übernahme in den Archivierungs-Relation durch Ausführung des `CALL materializeLatestRoutine(<vt_begin>, <vt_end>))` Statements.

Der erste Schritt lässt sich durch eine Prozedur bewerkstelligen. Die Definitionen der Prodezuren für das invalidieren einer SP bzw. einer UDF lassen sich im Abschnitt A.9 auf Seite 85 finden. Die Prozeduren nehmen dabei jeweils als Parameter den Namen der **spezifischen** Prozedur bzw. der spezifischen UDF an. Der spezifische Name ist von Relevanz, da verschiedenen Routinen unter dem gleichen Namen existieren können, sofern sie eine andere Signatur besitzen (andere In/Out/Inout-Parameter). Diese werden dann durch den spezifischen Namen unterschieden.

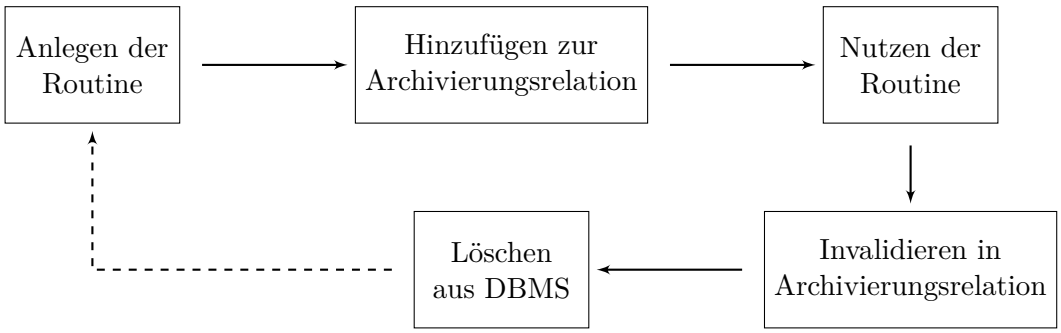

Abb. 16: Lifecycle der Routinen

Möchte man das Geschehen von Invalidierung und Löschung komplett automatisieren, so stößt man auf das beschriebene Problem, dass man in MySQL keine Prozeduren innerhalb anderer Prozeduren löschen kann. Als Abhilfe kann ein Java-Programm verwendet werden, um das Löschen einer Routine dennoch zu automatisieren. Durch die Java-Datei wird mittels *JDBC* eine Verbindung zur Datenbank hergestellt, die alte Routinedefinition invalidiert und die Routine gelöscht. Ein Beispiel-Javacode liegt auf der beiliegenden CD bereit.

Der beschriebene Lifecycle der Analysefunktionen wurde durch Abb. 16 visualisiert. Die konkreten Schritte waren dabei das **Anlegen** der Routine durch Ausführen einer DDL, das **Historisieren** der Routine durch Aufrufen der Historisierungsprozedur, das **Nutzen** der Routine, das **Invalidieren** der Routine durch das Setzen entsprechender temporaler Attribute mittels einer Prozedur und das **Löschen** der Routine. Optional war dabei der Schritt, der vom Löschen zum Anlegen geführt hat, falls eine Ersetzung gewünscht war.

4.3 Speicherung von Analyseergebnissen

Neben der Speicherung der Ergebnisse von Analysefunktionen müssen Informationen darüber gehalten werden, wie welche Ergebnisse entstanden sind. Im Abschnitt 3.6.2 wurden bereits die Informationen herausgestellt, die gespeichert werden müssen:

- Das Datum und die Uhrzeit der Berechnung.
- Die verwendete SQL-Abfrage der Berechnung. Indirekt werden damit auch die verwendeten Relationen der Rohdaten und benutzte Routinen gespeichert, da diese in der Abfrage auftauchen.
- Eine Referenz, die angibt, in welcher Tabelle das Ergebnis zu finden ist.
- Eine Referenz, die angibt, welche ID die Ergebnistupel besessen haben.

Die beschriebene Metadatenrelation für Analyseergebnisse wird wie folgt aufgebaut:

```
CREATE TABLE `godess`.`result_meta` (
  `datetime` DATETIME NOT NULL,
  `query` VARCHAR(2014) NOT NULL,
  `table_name` VARCHAR(64) NOT NULL,
  `result_id` INT NOT NULL UNIQUE,
  PRIMARY KEY (`result_id`)
) ENGINE = InnoDB;
```

Die Vorgehensweise zur Nutzung dieser Metadatentabelle soll beispielhaft an einem Beispiel erläutert werden, welches Daten aus Tabelle 11 nutzt. In besagter Tabelle sind Werte für Temperaturen in Kelvin gespeichert, die ein mal pro Tag aufgezeichnet wurden. Es soll die Durchschnittstemperatur der im Januar gemessenen Werte errechnet werden. Weiterhin sollen die Werte der `temperature`-Spalte von Kelvin in Grad Celsius umgerechnet werden. Eine entsprechende Funktion `celsiusToKelvin` sei dafür definiert. Die Berechnung findet am 31.01.2016 um 18:00 Uhr statt. Das Ergebnis wird in der dafür angelegten Tabelle `avg_temp` materialisiert. Folgende DML wird ausgeführt,um dieses Ziel zu erreichen.

DAILY_TEMP

date	temperature	tt_begin	tt_end
2016-01-30	274.15	2016-02-10 12:36:45	9999-12-31 23:59:59
2016-01-31	275.55	2016-02-10 12:36:45	9999-12-31 23:59:59
2016-02-01	277.15	2016-02-10 12:36:45	9999-12-31 23:59:59
2016-02-02	277.15	2016-02-10 12:36:45	9999-12-31 23:59:59

Tabelle 11: Beispieltabelle mit tageweisen Durchschnittstemperaturen

AVG_TEMP

result_id	month	temperature	tt_begin	tt_end
1	January	2	2016-02-10 18:00:00	’9999-12-31 23:59:59’

Tabelle 12: Beispieltabelle mit tageweisen Durchschnittstemparaturen

```
CREATE TABLE avg_temp (
  result_id INT NOT NULL,
  month varchar(9),
  temperature float(4,2),
  tt_begin DATETIME DEFAULT CURRENT_TIMESTAMP,
  tt_end DATETIME DEFAULT '9999-12-31 23:59:59'
);

INSERT INTO avg_temp
(result_id, month, temperature, tt_begin, tt_end)
VALUES ('January',
  1,
  (SELECT celsiusToKelvin(AVG(temperature)) FROM daily_temp
  WHERE (date BETWEEN '2016-01-01' AND '2016-01-31 23:59:59')),
  NOW(),
  '9999-12-31 23:59:59'
);
```

Nach Ausführung der DML wurde der Tabelle `avg_temp` der Eintrag hinzugefügt, der in Tab. 12 sichtbar ist. In dieser Tabelle kann nun auf die Metadatenrelation referenziert werden, indem in dieser entsprechend der Name der Tabelle `avg_temp` und die Ergebnis-ID `1` in die entsprechenden Spalten eingetragen wird:

```
INSERT INTO result_meta
(datetime, query, result_type, table_name, result_id)
VALUES(NOW(),
  'INSERT INTO avg_temp VALUES ('January',  1,
    (SELECT celsiusToKelvin(AVG(temperature)) FROM daily_temp
    WHERE (date BETWEEN '2016-01-01' AND '2016-01-31 23:59:59')
                AND tt_end = '9999-12-31 23:59:59';
                ),
    NOW(),
  '9999-12-31 23:59:59',
  'avg_temp',
  1
);
```

Entsprechend wurde in der Metadatentabelle der in Tab. 13 vorhandene Eintrag hinzugefügt.

Die Information ist nun verfügbar, dass das Ergebnis der ausgeführten Anfrage in der Tabelle `avg_temp` vorhanden ist. Alle Ergebnisse, die die Ergebnis-ID `1` beinhalten. sind Teile des Ergebnisses. Diese Information ist notwendig, da Analyseergebnisse gleicher Semantik unter einer anderen Ergebnis-ID zu dieser Tabelle `avg_temp` hinzugefügt werden können. Die Art

RESULT_META

datetime	query	table_name	result_id
2016-02-10 18:00:00	'INSERT INTO avg_temp VALUES ('January' ...	avg_temp	1

Tabelle 13: Metadatentabelle für Analyseergebnisse

des Ergebnisses – ob singulärer Wert, Tupel oder Tabelle – spielt dabei keine Rolle. In allen Fällen wird für alle Ergebnistupel die `result_id` gesetzt.

Im genannten Beispiel wurde die `result_id` manuell generiert. Um die nächste verfügbare ID automatisch zu wählen, wird im Query `SELECT MAX(result_id) + 1 FROM result_meta` verwendet, sofern mindestens ein Tupel in der Metadatenrelation für Ergebnisse existent ist (Ansonsten liefert `MAX(result_id)+1` einen Nullwert zurück).

Folgende Regeln gelten im Framework bei dieser Art der Speicherung:

- Auf Ergebnistabellen dürfen keine Updates vorgenommen werden.
- Für größtmögliche Rekonstruierbarkeit müssen alle Zwischenergebnisse gespeichert werden
- Datum und Uhrzeit der Berechnung müssen gespeichert werden, um verwendete Analysefunktionen rekonstruieren zu können. Die verwendete Analysefunktion kann zum Zeitpunkt der Rekonstruktion gelöscht und mit einer anderen Berechnungsvorschrift erneut angelegt worden sein.

4.4 Provenance im Framework

Um nachzuweisen, dass die beschriebenen Konzepte Provenance sicherstellen, soll dies an dem Beispiel aus dem vorherigen Abschnitt gezeigt werden. Soll nun ein Rückschluss darauf gezogen werden, wie der konkrete Wert für den Januar in der Tabelle 12 „avg_temp“ zustande gekommen ist, so wird zuerst die Metadatentabelle abgefragt:

```
SELECT * FROM result_meta
  WHERE TABLE = avg_temp
  AND result_id = 1;
```

Das zuvor in die Metadatentabelle eingefügte Tupel wird dabei als Ergebnis zurückgegeben. Anhand dieses Tupels ist folgendes ersichtlich:

- Das Datum der Abfrage.
- Die Abfrage der Berechnung. Anhand dieser insbesondere (durch die „`query`“ sichtbar:
  - Die verwendeten Nutzdatenrelationen in der `FROM`-Klausel.
  - Verwendete UDFs.

Um nun auf den Nutzdatenbestand zuzugreifen, der für die Berechnung verwendet wurde, werden aus dem Ergebnistupel der Metadatenrelation die verwendeten Tabellen bestimmt. In diesem Fall ist dies nur die Tabelle `daily_temp`. Dabei wurde die Selektion auf `(date BETWEEN "2016-01-01" AND "2016-01-31")` beschränkt. Mit der Abfrage werden alle Tupel aus `daily_temp` selektiert, die der Selektion in der Originalanfrage entsprachen. Weiterhin ist bekannt, dass die Abfrage am 10.02.2016 um 18:00 Uhr gestellt wurde. Anhand dieser Informationen kann nun der ursprüngliche Datenbestand rekonstruiert werden:

```
SELECT * FROM daily_temp
  WHERE (date BETWEEN "2016-01-01" AND "2016-01-31")
  AND tt_begin <= '2016-02-10 18:00:00'
  AND tt_end > '2016-02-10 18:00:00';
```

Es werden zudem temporale Aspekte betrachtet: Der Attributwert von `tt_begin` muss kleiner oder gleich dem Datum sein, an dem die Abfrage getätigt wurde. Damit geht man sicher, dass das Tupel zu diesem Zeitpunkt schon in der Datenbank vorhanden war. Weiterhin muss der `Attributwert` bezogen auf das Datum der Abfrage in der Zukunft liegen. Damit wird beachtet, dass keine Tupel rekonstruiert werden, die zum Zeitpunkt der Datenbankabfrage schon als gelöscht markiert wurden.

Analog funktioniert das Prinzip für die Rekonstruktion der benutzten UDFs. Dabei werden die Anfragen an die Tabellen `routines_history` und `routines_parameter_history` gestellt:

```
SELECT * FROM routines_history
  WHERE ROUTINE_NAME = celsiusToKelvin
  AND ROUTINE_TYPE = 'FUNCTION'
  AND vt_begin <= '2016-02-10 18:00:00'
  AND tt_begin <= '2016-02-10 18:00:00'
  AND vt_end > '2016-02-10 18:00:00'
  AND tt_end > '2016-02-10 18:00:00';
```

Es müssen hierbei alle vier temporale Werte abgefragt werden: Die Funktion muss zu dem Berechnungszeitpunkt **gültig** gewesen sein (also zwischen `vt_begin` und `vt_end` liegen), die Funktion muss zu dem Zeitpunkt schon existiert haben, der Zeitpunkt der Ausführung liegt also zeitlich nach `tt_begin` und die Funktion darf nicht als gelöscht gekennzeichnet gewesen sein, muss also zeitlich vor `tt_end` ausgeführt worden sein.

Eine durchgängig automatisierte Rekonstruktion von Daten und Funktionen ist zum jetzigen Zeitpunkt nicht möglich.

5 Zusammenfassung, Bewertung und Ausblick

Für die Speicherung von Primärforschungsdaten, Analysemethoden und Analyseergebnissen wurde ein Framework entwickelt, das Rücksicht auf die Anforderungen an Provenance nimmt. Insbesondere temporale Aspekte zur Historisierung von Analysefunktionen und Konzepte zur Rekonstruktion von Originaldaten und Ergebnissen wurde betrachtet.

Für Rohdaten gibt es im Framework verschiedene Möglichkeiten der Datenhaltung. Für eine un- bzw. semistrukturierte Datenhaltung steht ein Key-Value-Store, ein EAV-Store sowie ein Document Store zur Verfügung. Für eine strukturierte Datenhaltung stehen relationale Datenbanken in Form von Row Stores und Column Stores zur Verfügung. Die konkrete Umsetzung für das GODESS-Projekt wurde mit dem relationalen Row Store von DBMS MySQL realisiert. Dabei wurden, sofern möglich, die im Konzeptionskapitel beschriebenen Vorgehensweisen genutzt.

Während der Arbeit hat sich herausgestellt, dass MySQL sich nicht als optimale Plattform für ein professionelles Forschungsdatenmanagement anbietet. Insbesondere die fehlende Möglichkeiten wie die Nutzung von Table Functions, wie beispielsweise IBM DB2 es anbietet, fehlen vollständig. Die Verlagerung bestehender Analyseprozesse auf Datenbankebene werden dadurch deutlich erschwert. Auch werden durch MySQL keine Sprachen wie R oder Java in Stored Procedures unterstützt[15], wie es in kommerziellen DBMS wie DB2 der Fall ist. Schlussendlich lässt sich festhalten, dass MySQL weniger für die Nutzung komplexer Analysefunktionen geeignet ist, sondern eher für die Datenhaltung. Analysen können durch die Anbindung externer Programme realisiert werden. Soll die Nutzung über die Datenhaltung hinausgehen, wird ein (kommerzielles) DBMS mit erweitertem Umfang empfohlen.

Auf konzeptueller Ebene wurde eruiert, wie man Provenance von Routinen gewährleisten kann. Dazu wurden die Routinen historisiert und mit Gültigkeitszeitinformationen versehen. Nicht beschrieben wurden bisher Konzepte, wie man externe Analysefunktionen, z.B. in Programmen mit Datenbankanbindungen historisieren kann. Es wurden im Abschnitt 4.4 grundlegende Schritt-für-Schritt Lösungen zur Rekonstruktion von Originaldaten

[15] *„The LANGUAGE characteristic indicates the language in which the routine is written. The server ignores this characteristic; only SQL routines are supported“* [Oraa].

betrachtet. Eine Herausforderung in der Zukunft bildet die Automatisierung dieser Prozesse. Zum jetzigen Zeitpunkt keine Implementierung für eine durchgängig automatisierte Lösung zur Datenrekonstruierung.

6 Literaturverzeichnis

[BHM11] Büttner, Stephan; Hobohm, Hans-Christoph; Müller, Lars: *Handbuch Forschungsdatenmanagement*. Bad Honnef: Bock + Herchen, 2011. – ISBN 978–3–88347–283–6

[Boe00] Böhm-Schweizer, Denise: *Planeten Tabelle*. `http://astrokramkiste.de/planeten-tabelle`, Abruf: 23.10.2015

[Boe00b] Böhm-Schweizer, Denise: *Pluto*. `http://astrokramkiste.de/pluto`, Abruf: 23.10.2015

[BSG] Bartunov, Oleg; Sigaev, Teodor; Gierth, Andrew: *PostgreSQL 9.0.23 Documentation: hstore: (o. J.)*. `http://www.postgresql.org/docs/9.0/static/hstore.html`, Abruf: 10.12.2015

[Fac13] Fachhochschule Köln: *Datenbanken Online Lexikon - Dokumentenorientierte Datenbank*. `http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/DokumentenorientierteDatenbank`. Version: 03/2013, Abruf: 19.01.2016

[Fei06] Feistel, Rainer: *Leibniz-Institut für Ostseeforschung Warnemünde – Wird die Ostsee zum Süßwassermeer?* `http://www.io-warnemuende.de/wird-die-ostsee-zum-suesswassermeer.html`. Version: 2006, Abruf: 25.02.2016

[GA09] Glavic, Boris; Alonso, Gustavo: The PERM Provenance Management System in Action. In: Çetintemel, Uğur (Hrsg.); Zdonik, Stan (Hrsg.); Kossmann, Donald (Hrsg.): *The 35th SIGMOD International Conference on Management of Data*. New York: ACM, 2009, S. 1055–1058

[HB15] Heuer, Andreas; Bruder, Ilvio: *Herausforderungen, Konzepte und Lösungen zur nachhaltigen und nachvollziehbaren Verwaltung von Forschungsdaten: (Vortrag am Leibniz Institut für Ostseeforschung Warnemünde)*. Warnemünde, 2015

[Heu15] Heuer, Andreas: METIS in PArADISE: Provenance Management bei der Auswertung von Sensordatenmengen für die Entwicklung von Assistenzsystemen. In: Ritter, Norbert (Hrsg.); Henrich, Andreas (Hrsg.); Lehner, Wolfgang (Hrsg.) et al.: *Datenbanksysteme für Business, Technologie und Web (BTW 2015), Workshopband: Hamburg, Germany, 02-03.03.2015* Bd. 242. Bonn: Gesellschaft für Informatik, 2015. – ISBN 9–783–88579636–7, S. 131–136

[Int06] International Astronomical Union IAU; IAU (Hrsg.): *Resolution B5: Definition of a Planet in the Solar System; Resolution B6: Pluto*. `https://www.iau.org/static/resolutions/Resolution_GA26-5-6.pdf`. Version: 2006, Abruf: 23.10.2015

[IOW01] Institut für Ostseeforschung Warnemünde: *Profilierende Verankerung im Gotlandbecken*. `http://www.io-warnemuende.de/Profilierende_Verankerung.html`, Abruf: 22.10.2015

[KSS14] Klettke, Meike; Scherzinger, Stefanie; Störl, Uta: Datenbanken ohne Schema? In: *Datenbank-Spektrum* 14 (2014), Nr. 2, S. 119–129. – DOI 10.1007/s13222–014–0156–z. – ISSN 1618–2162

[LKH11] Löper, Dortje; Klette, Meike; Heuer, Andreas: Das Entity-Attribute-Value-Konzept als Speicherstruktur für die Informationsintegration in der ambulanten Pflege. In: Heiss, Hans-Ulrich (Hrsg.): *Informatik 2011* Bd. 192. Bonn: Ges. für Informatik, 2011. – ISBN 978–3–88579286–4, S. 397

[McN14] McNulty, Eileen: *SQL vs. NoSQL - What you need to know*. `http://dataconomy.com/sql-vs-nosql-need-know/`. Version: 2014, Abruf: 08.01.2016

[Mey15] Meyer, Frank: *Aufbau einer Artenlistenverwaltung im Benthos-Projekt*. Rostock, Universität Rostock, Bachelorarbeit, 2015. `http://www.io-warnemuende.de/tl_files/bio/ag-benthische-organismen/pdf/Bachelorarbeit-Frank%20Meyer.pdf`, Abruf: 23.10.2015

[Oraa] Oracle Corporation: *CREATE PROCEDURE and CREATE FUNCTION Syntax - MySQL 5.7 Reference Manual: (o. J.)*. `http://dev.mysql.com/doc/refman/5.7/en/create-procedure.html`, Abruf: 17.02.2016

[Orab] Oracle Corporation: *Date and Time Type Overview - MySQL 5.7 Reference Manual: (o. J.)*. `http://dev.mysql.com/doc/refman/5.7/en/date-and-time-type-overview.html`, Abruf: 15.11.2015

[Orac] Oracle Corporation: *The TIME Type - MySQL 5.7 Reference Manual: (o. J.)*. `http://dev.mysql.com/doc/refman/5.7/en/time.html`, Abruf: 23.01.2016

[PFPSB15] Prien, Ralf D.; Floth-Peterson, Mareike; Schulz-Bull, Detlef E.: *Profiling Mooring Godess - The First 100 km of Profiles from the Gotland Basin.* 10th Baltic Sea Science Congress, Riga, 2015

[PSB11] Prien, Ralf D.; Schulz-Bull, Detlef E.: The Gotland Deep Environmental Sampling Station in the Baltic Sea. Version: 2011. In: IEEE (Hrsg.): *Oceans 2011 (Santander, Spain).* New York: Curran Associates, Inc., 2011. – DOI 10.1109/Oceans–Spain.2011.6003606. – ISBN 9–781–45770086–6, S. 1624–1629

[PSB15] Prien, Ralf D.; Schulz-Bull, Detlef E.: *Prien - Technical note GODESS - A profiling mooring in the Gotland Basin.* Nichtveröffentliches Manuskript, 2015

[Sea14] Sea & Sun Marine Tech: *CTD 90 / CTD 90 M Multi Parameter Probe.* `http://www.sea-sun-tech.com/fileadmin/img/pdf_sea/CTD90.pdf`. Version: 01/2014, Abruf: 03.11.2015

[SHS08] Saake, Gunter; Heuer, Andreas; Sattler, Kai-Uwe: *Datenbanken: Konzepte und Sprachen.* 3., aktualisierte und erw. Aufl. Heidelberg: Mitp bei Redline, 2008. – ISBN 9–783–82661664–8

[Stö14] Störl, Uta: *NoSQL-Datenbanksysteme: Revolution oder Evolution?* Rostock, 24.01.2014

[W3C10] W3C Provenance Incubator Group: *What Is Provenance.* `https://www.w3.org/2005/Incubator/prov/wiki/index.php?title=What_Is_Provenance&oldid=2026`. Version: 18.11.2010, Abruf: 18.11.2015

A Appendix

A.1 Beispieldatei Powerlogger (Metadatenteil)

```
|----------------------------------------
@FILETITEL ASCII-DATA
@FILENAME  C:\ASCIIDAT\A150602A.TXT
@COPYRIGHT
@DATETIME  U1433268106  "2015-06-02" "18:01:46"
@PROGRAM   SmartDI v6.13.2010 (2014-10-15) C:\SDI\SDI\SMARTDI.EXE /pc\sdi\data\SDIIOW2
@VERSION   DPMI  $FC303DA9
@CONTENTS
|----------------------------------------
@METASTART
@CHANNEL,_,_,_,_,_,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
    ,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67
@NAME,DATE,TIME,FRAC,SEC,VBatt,Press,Temp,Cond,RawO2,pH,ChlA,Redox,Ti,Turbidity,Polym,Tur_1,Tur_2,Tur_3,
    Tur_4,CTD-Ch15,ProPS-Normalized,ProPS-Lamp,Datetime,Heading,Pitch,Roll,Temp.,Press.,AnaIn1,AnaIn2/SV,
    Error,Battery,Status,IN1-DCD1,RXD1,In2-DSR1,CTS1,RI1,TXD1,DTR1,RTS1,TXD1,StatLED-DTR1,Out2-RTS1,Port0:
    Pin0,Port0: Pin1,Port0: Pin2,Port0: Pin3,Port0: Pin4,Port0: Pin5,Port0: Pin6,Port0: Pin7,PLC-SrvState,
    SystemService,SystemError,In1-DCD2,RXD2,In2-DSR2,CTS2,RI2,TXD2,DTR2,RTS2,TXT2,uRiderOn-DTR2,Out2-RTS2,PCI
    -0,PCI-1,PCI-2,PCI-3,PCI-4,PCI-5
@UNIT,YY-MM-DD,HH:MM:SS,MS,RUNSEC,V,dBar,∅C,mS/cm,,,,,,,,,,,,,,,,deg,deg,deg,∅C,Bar,V,m/s,,V
    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
✂.....
@RATE,_,_,_,_,_,-200,-200,-200,-200,-200,-200,-200,-200,-200,-200,-200,-200,-200,-200,-200,-500,-500,1,1,
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
@FORMULA,_,_,_,_,_,LIN,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,POL3,LIN,LIN,LIN,
    LIN,LIN,LIN,LIN,LIN,LIN,LIN,LIN,LIN,LIN,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,
    RAW,RAW,RAW,SRVST,SYSSR,SYSER,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,RAW,LIN,LIN,LIN,LIN,LIN,LIN
@METAEND
```

Abb. A.1: Beispiel für die Strukturierung von Metadaten des Powerloggers

A.2 Beispieldatei Powerlogger (Nutzdatenteil)

```
@DATASTART "2015-06-02" "18:01:46"
$PSDA1,2015-06-02,18:01:46,277,10,45230,61899.000,50073.000,15568.000,14338.000,37102.000,38172.000,3622.000,24602.000,12761.000,567.000,0.000,0.000,0.000,0.000,0.000,0,0,07:29:46,129.4,-1.1,2.1,6.92,6.108,0.00,1449.60,$0000,11.5,$10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.000,0.000,0.000,0.000,0.000,0.000
$PSDA1,2015-06-02,18:01:47,664,11,45236,61915.000,49992.000,15566.000,14336.000,37084.000,38195.000,3748.000,24613.000,12759.000,543.000,0.000,0.000,0.000,0.000,0.000,0,0,07:29:47,129.2,-0.9,2.1,6.92,5.898,0.00,1449.60,$0000,11.5,$10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.000,0.000,0.000,0.000,0.000,0.000
$PSDA1,2015-06-02,18:01:48,068,11,45237,61901.000,49980.000,15567.000,14336.000,37082.000,38198.000,3764.000,24614.000,12759.000,559.000,0.000,0.000,0.000,0.000,0.000,0,0,07:29:47,129.2,-0.9,2.1,6.92,5.898,0.00,1449.60,$0000,11.5,$10,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0.000,0.000,0.000,0.000,0.000,0.000
$PSDA1,2015-06-02,18:01:48,336,11,45238,61903.000,49967.000,15568.000,14336.000,37079.000,38201.000,3764.000,24616.000,12759.000,527.000,0.000,0.000,0.000,0.000,0.000,0,0,07:29:47,129.2,-0.9,2.1,6.92,5.898,0.00,1449.60,$0000,11.5,$10,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0.000,0.000,0.000,0.000,0.000,0.000
$PSDA1,2015-06-02,18:01:48,336,11,45238,61903.000,49967.000,15568.000,14336.000,37079.000,38201.000,3764.000,24616.000,12759.000,527.000,0.000,0.000,0.000,0.000,0.000,0,0,07:29:47,129.2,-0.9,2.1,6.92,5.898,0.00,1449.60,$0000,11.5,$10,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0.000,0.000,0.000,0.000,0.000,0.000
$PSDA1,2015-06-02,18:01:49,097,12,45241,61901.000,49929.000,15571.000,14336.000,37073.000,38211.000,3787.000,24620.000,12760.000,563.000,0.000,0.000,0.000,0.000,0.000,0,0,07:29:48,128.7,-0.8,2.2,6.92,5.654,0.00,1449.60,$0000,11.6,$10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0.000,0.000,0.000,0.000,0.000,0.000
✂ ............................................................
```

Abb. A.2: Beispiel für die Strukturierung von Nutzdaten des Powerloggers

A.3 Entity-Relationship-Diagramm der GODESS-Rohdaten

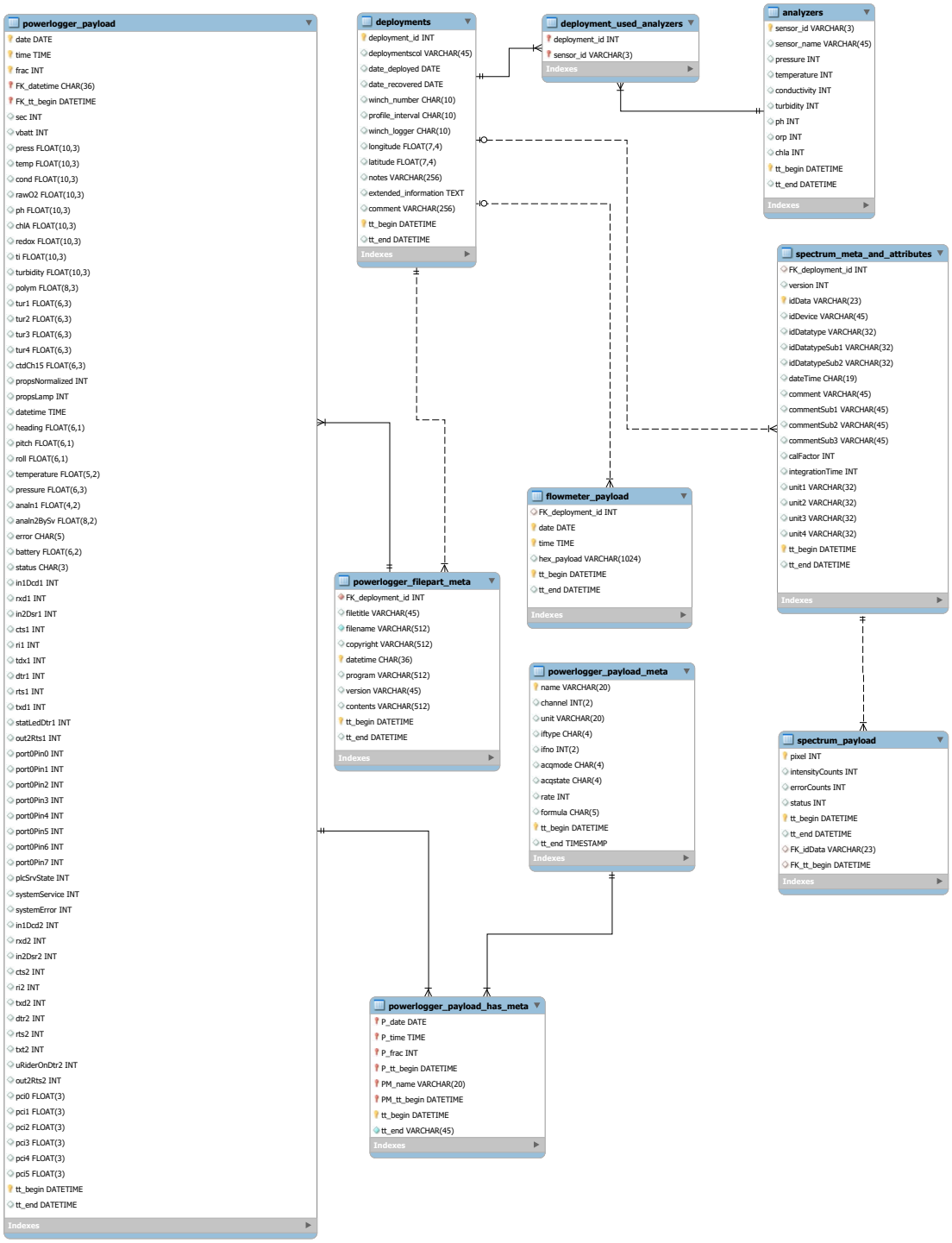

Abb. A.3: Entity-Relationship-Diagramm der GODESS-Rohdaten

A.4 Übeführte Daten in der Powerlogger-Dateien-Metadatentabelle

POWERLOGGER_FILEPART_META

filetitle	filename	copyright	datetime	program	version	contents	tt_begin	tt_end
ASCII-DATA	C:\ASCIIDAT\A1506021.TXT	NULL	U1433268106 "2015-06-0218:01:46“	SmartDI v6.13.2010 (2014-10-15) C:\SDI\SDI\SMARTDI.EXE/pc\sdi\data\SDIIOW2	DPMI $FC303DA9	NULL	2016-01-27 15:16:06	NULL

Tabelle A.1: Übeführte Daten in der Powerlogger-Dateien-Metadatentabelle

A.5 Übeführte Daten in der Powerlogger-Nutzdaten-Metadatentabelle

POWERLOGGER_PAYLOAD_META

FKdatetime	payload_column	channel	unit	iftype	ifno	aqmode	aqstate	rate	formula	tt_begin	tt_end
U1433268106 "2015-06-0218:01:46"	DATE	NULL	YY-MM-DD	NULL	NULL	NULL	NULL	NULL	NULL	2016-01-27 15:16:06	NULL
U1433268106 "2015-06-0218:01:46"	TIME	NULL	HH:MM:SS	NULL	NULL	NULL	NULL	NULL	NULL	2016-01-27 15:16:06	NULL
U1433268106 "2015-06-0218:01:46"	FRAC	NULL	MS	NULL	NULL	NULL	NULL	NULL	NULL	2016-01-27 15:16:06	NULL
U1433268106 "2015-06-0218:01:46"	SEC	NULL	RUNSEC	NULL	NULL	NULL	NULL	NULL	NULL	2016-01-27 15:16:06	NULL
U1433268106 "2015-06-0218:01:46"	VBatt	0	V	COM	10	MEAS	RUN	-200	LIN	2016-01-27 15:16:06	NULL
U1433268106 "2015-06-0218:01:46"	Press	1	dBar	COM	10	MEAS	RUN	-200	Pol3	2016-01-27 15:16:06	NULL
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Tabelle A.2: Übeführte Daten in der Powerlogger-Nutzdaten-Metadatentabelle

A.6 Initiale Provenance-Tabellen für Analysemethoden

```
CREATE TABLE routines_history
(
        tt_begin datetime,
        tt_end datetime,
        vt_begin datetime,
        vt_end datetime
) SELECT * FROM information_schema.ROUTINES;

UPDATE routines_history
  SET tt_begin = created,
           tt_end = '9999-12-31 23:59:59';

CREATE TABLE routines_parameter_history
(
        tt_begin datetime,
        tt_end datetime,
        vt_begin datetime,
        vt_end datetime
) SELECT * FROM information_schema.PARAMETERS;

UPDATE test.routines_parameter_history RHP
  SET tt_begin =
    (SELECT CREATED FROM information_schema.ROUTINES ISR
    WHERE RHP.SPECIFIC_NAME = ISR.SPECIFIC_NAME
    AND RHP.ROUTINE_TYPE = ISR.ROUTINE_TYPE),
    tt_end = '9999-12-31 23:59:59';
```

A.7 DDL für die Tabelle der Powerloggernutzdaten

```
CREATE TABLE IF NOT EXISTS `godess`.`powerlogger_payload`
  ( `date` DATE NOT NULL, `time` TIME NOT NULL, `frac` INT NOT NULL,
     `FK_datetime` CHAR(36) NOT NULL, `FK_tt_begin` DATETIME NOT NULL
     , `sec` INT, `vbatt` INT, `press` FLOAT(10,3), `temp` FLOAT
     (10,3), `cond` FLOAT(10,3), `rawO2` FLOAT(10,3), `ph` FLOAT
     (10,3), `chlA` FLOAT(10,3), `redox` FLOAT(10,3), `ti` FLOAT
     (10,3), `turbidity` FLOAT(10,3), `polym` FLOAT(8,3), `tur1`
     FLOAT(6,3), `tur2` FLOAT(6,3), `tur3` FLOAT(6,3), `tur4` FLOAT
     (6,3), `ctdCh15` FLOAT(6,3), `propsNormalized` INT, `propsLamp`
     INT, `datetime` TIME, `heading` FLOAT(6,1), `pitch` FLOAT(6,1),
     `roll` FLOAT(6,1), `temperature` FLOAT(5,2), `pressure` FLOAT
     (6,3), `analn1` FLOAT(4,2), `analn2BySv` FLOAT(8,2), `error`
     CHAR(5), `battery` FLOAT(6,2), `status` CHAR(3), `in1Dcd1` INT,
     `rxd1` INT, `in2Dsr1` INT, `cts1` INT, `ri1` INT, `tdx1` INT, `
     dtr1` INT, `rts1` INT, `txd1` INT, `statLedDtr1` INT, `out2Rts1`
      INT, `port0Pin0` INT, `port0Pin1` INT, `port0Pin2` INT, `
     port0Pin3` INT, `port0Pin4` INT, `port0Pin5` INT, `port0Pin6`
     INT, `port0Pin7` INT, `plcSrvState` INT, `systemService` INT, `
     systemError` INT, `in1Dcd2` INT, `rxd2` INT, `in2Dsr2` INT, `
     cts2` INT, `ri2` INT, `txd2` INT, `dtr2` INT, `rts2` INT, `txt2`
      INT, `uRiderOnDtr2` INT, `out2Rts2` INT, `pci0` FLOAT(3), `pci1
     ` FLOAT(3), `pci2` FLOAT(3), `pci3` FLOAT(3), `pci4` FLOAT(3), `
     pci5` FLOAT(3),
  `tt_begin` DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `tt_end` DATETIME NULL DEFAULT '9999-12-31 23:59:59',
  PRIMARY KEY (`date`, `time`, `frac`, `FK_datetime`, `FK_tt_begin`,
     `tt_begin`),
  CONSTRAINT `payload_has_filepart_meta`
    FOREIGN KEY (`FK_datetime`, `FK_tt_begin`) REFERENCES
    `ba`.`powerlogger_filepart_meta` (`datetime`, `tt_begin`)
);
```

A.8 Prozedur zum Archivieren von Routinen

```
DELIMITER //
CREATE PROCEDURE materializeLatestRoutine
(in invt_begin DATETIME, in invt_end DATETIME)
COMMENT 'DoNotMaterialize'
BEGIN

  DECLARE latestRoutine VARCHAR(64);
  DECLARE latestRoutineTtBegin DATETIME;
  DECLARE latestRoutineType VARCHAR(9);

  REPLACE INTO test.routines_history
  SELECT ISR.created AS tt_begin, '9999-12-31 23:59:59' AS tt_end,
    invt_begin AS vt_begin, invt_end AS vt_end, ISR .*
  FROM information_schema.ROUTINES ISR
  WHERE ROUTINE_COMMENT NOT LIKE '%DoNotMaterialize%'
  AND NOT EXISTS (
    SELECT tt_begin , SPECIFIC_NAME , ROUTINE_TYPE
    FROM routines_history RH
    WHERE RH.tt_begin = ISR . created
    AND RH.SPECIFIC_NAME = ISR . SPECIFIC_NAME
    AND RH.ROUTINE_TYPE = ISR . ROUTINE_TYPE
  ) ORDER BY CREATED DESC LIMIT 0, 1;

  SELECT RH.SPECIFIC_NAME INTO latestRoutine FROM routines_history RH
  ORDER BY tt_begin DESC LIMIT 0, 1;

  SELECT RH.tt_begin INTO latestRoutineTtBegin FROM routines_history RH
  ORDER BY tt_begin DESC LIMIT 0, 1;

  SELECT RH.ROUTINE_TYPE INTO latestRoutinetype FROM routines_history RH
  ORDER BY tt_begin DESC LIMIT 0, 1;

  INSERT INTO test.routines_parameter_history
  SELECT latestRoutineTtBegin AS tt_begin, '9999-12-31 23:59:59' AS tt_end,
    invt_begin AS vt_begin, invt_end AS vt_end, ISP.*
  FROM information_schema.PARAMETERS ISP
  WHERE ISP.SPECIFIC_NAME = latestRoutine
  AND ISP.ROUTINE_TYPE = latestRoutineType;

END //
DELIMITER ;
```

A.9 Prozeduren zum Invalidieren von Routinen

```
DELIMITER //
CREATE PROCEDURE invalidateProcedure (in specificProcedureName varchar (64))
BEGIN
  UPDATE routines_history SET tt_end = NOW()
  WHERE SPECIFIC_NAME = specificProcedureName
  AND ROUTINE_TYPE = 'PROCEDURE'
  AND tt_end IS NULL;
  UPDATE routines_parameter_history SET tt_end = NOW()
  WHERE SPECIFIC_NAME = specificProcedureName
  AND ROUTINE_TYPE = 'PROCEDURE'
  AND tt_end IS NULL;
END //
DELIMITER ;

DELIMITER //
CREATE PROCEDURE invalidateFunction (in specificFunctionName varchar (64))
BEGIN
  UPDATE routines_history SET tt_end = NOW()
  WHERE SPECIFIC_NAME = specificFunctionName
  AND ROUTINE_TYPE = 'FUNCTION'
  AND tt_end IS NULL;
  UPDATE routines_parameter_history SET tt_end = NOW()
  WHERE SPECIFIC_NAME = specificFunctionName
  AND ROUTINE_TYPE = 'FUNCTION'
  AND tt_end IS NULL;
END //
DELIMITER ;
```

A.10 Java-Programm zum Löschen von Routinen

```
package iowDeleteProcedure;

import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Main {

    public static void main(String[] args) {
        if(args.length != 1 || !args[0].getClass().equals(String.class)){
            System.exit(-1);
        }

        DatabaseHelper db = new DatabaseHelper();
        db.connect();

        try {
            db.getConn().setAutoCommit(false);
            PreparedStatement pstmnt = db.getConn()
              .prepareCall("{CALL invalidateProcedure(?)}");
            pstmnt.setString(1, args[0]);
            pstmnt.executeUpdate();

            PreparedStatement pstmnt2 = db.getConn()
              .prepareStatement(String.format("DROP PROCEDURE %s", args[0]));
            pstmnt2.executeUpdate();

            db.getConn().commit();
            pstmnt.close();
            pstmnt2.close();
            db.close();
        } catch (SQLException e) {
            try {
                if(db.getConn() != null && !db.getConn().isClosed()){
                    db.getConn().rollback();
                    db.close();
                }
            } catch (SQLException subex) {
                subex.printStackTrace();
            }
            e.printStackTrace();
        }
    }
}
```

```
public class DatabaseHelper {
    private Connection conn = null;

    public DatabaseHelper(){
        try{
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException ex){
            System.err.println("Der Datenbanktreiber
              com.mysql.jdbc.Driver wurde nicht gefunden");
            System.exit(-2);
        }
    }

    public void connect(){
        try{
            conn = DriverManager.getConnection("jdbc:mysql://localhost/test
              ?user=bachelor&password=bachelor
              &generateSimpleParameterMetadata=true");
        } catch (SQLException ex) {
            System.err.println("Die Verbindung zur Datenbank
              konnte nicht hergestellt werden. Abbruch");
            System.exit(-3);
        }
    }

    public void close() {
        try {
            conn.close();
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }

    public Connection getConn() {
        return conn;
    }
}
```

A.11 CD

Dieser Arbeit liegt eine CD bei, auf der folgende Inhalte zu finden sind:

- Die Bachelorarbeit als PDF-Dokument
- Verwendete Paper, die im Namen den Zitierschlüssel, den Autor und den Titel des Dokumentes tragen
- Verwendete Webquellen, die im Namen die Zugriffszeit, den Zitierschlüssel und den Titel der Webseite tragen
- Zusätzliche Dokumente aus dem Umsetzungskapitel und Appendix wie beispielsweise SQL-Anweisungen

Selbstständigkeitserklärung zur Bachelorarbeit

Hiermit erkläre ich, Mark Lukas Möller, dass ich die vorgelegte Bachelorarbeit zum Thema „*Aufbau einer Forschungsdatenverwaltung für chemische und physikalische In-Situ-Daten aus der Ostsee*“ selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Verwendete Hilfsmittel und Referenzen in Form von direkten oder sinngemäßen Zitaten, Bild- oder anderen Quellen wurden als solche kenntlich gemacht. Ferner versichere ich, dass diese Arbeit bisher nicht in gleicher oder ähnlicher Form als Prüfungsleistung einer Prüfungskommission vorgelegt wurde.

.....	.....
Ort, Datum	Unterschrift