MusaeusDB - Von Mono-Table zu Poly-Table

Benedikt G. Kleiner

11. Dezember 2017

Zusammenfassung

Für die Versionierung von Datenbank konnte sich bisher keine de-facto Lösung etablieren. Ein potenzielle Lösung sollte mit bestehenden Systemen kompatibel sein und marginalen operativen Overhead aufweißen. Diese Arbeit stellt mit MusaeusDB ein einfaches System zur Versionierung mehrerer Tabellen unter Verwendung einer herkömmlichen relationellen Datenbank vor.

1 Einleitung

Die Größe, Komplexität und Geschwindigkeit mit der Datenbanken erstellt und verarbreitet werden müssen, steigt ständig. So betrug die Größe der englischen Wikipedia Datenbank im Jahr 2016 10.4 Terabyte [5] und zeigt auch weiterhin eine starke Tendenz zu wachsen [6]. Damit wird die Kooperation von mehren Informatikern, Data Scientist und Systemadministratoren zur Erstellung solcher Systeme unabdingbar.

Für das kollaborative Bearbeiten von Source Code, Textdokumenten oder Grafiken existieren verschiedene Software-Lösungen. Für Datenbanken, eine der Hauptkomponenten vieler Software-Systeme, konnte sich bisher jedoch keine de-facto Lösung etablieren.

Die Versionierung von Datensätzen und Schemata in klassischen Version Control Systemen ist ineffizient und bietet nur beschränkte Möglichkeiten innerhalb des VCS auf den Daten zu operieren [2]. Andere Arbeiten stellen spezialisierte [2] oder vollkommen neue [3] Konzepte vor. Beide Ansätze sind für eine weitläufige Adoption hinderlich, da bestehende Datenbanken unter Umständen nicht in die entsprechende Niche fallen oder auf ein neues DBMS umgestellt werden müssten.

Diese Arbeit versucht eine Lösung im Geiste von Git [4] basierend auf OrpheusDB zu konzeptionieren. Dabei wurde in einem ersten Schritt eine naive Re-Implementierung von OrpheusDB in C++11/14 erstellt und in einem zweiten Schritt um die Verwaltung mehrerer Tabellen erweitert.

2 OrpheusDB

2.1 Übersicht

OrpheusDB bezeichnet sich selbst als eine "bolt-on" Technik. Die Vorgehensweise bleibt diesem Motto auch treu: Die zu verwaltenden Daten werden in einem ersten Schritt, wahlweise aus einer CSV Datei oder aus einer bestehenden Tabelle, in eine herkömmliche relationale Datenbank importiert. Alle darauf folgenden Versionierungs-Operationen werden durch Hinzufügen weiterer Verwaltungs-Tabellen realisiert. Dabei bleibt die Datenbank selbst vollkommen unbehelligt von der Präsenz des Versionierungs-Systems. Voraussetzungen an das DBMS ist die Unterstützung gängiger SQL-Befehle und ein ARRAY Datentyp [7].

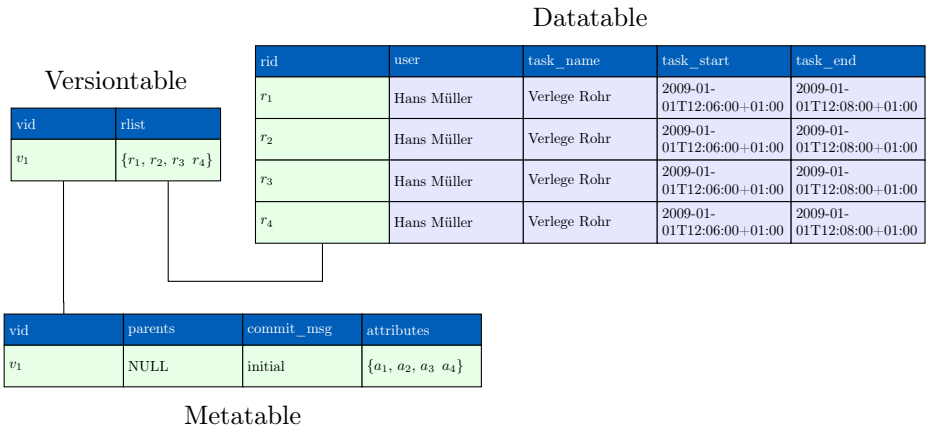

Abbildung 1: Datenbank Schema mit Versionierung

2.2 Features

OrpheusDB wird als ein Git ähnliches Command-Line-Programm implementiert und bietet dem Nutzer folgende Haupt-Operationen: **init** initialisiert einen bestehenden Datensatz als sogenanntes "collaborative versioned dataset" (CVD). **checkout** erlaubt es dem Nutzer, eine bestimte Version eines CVD, entweder als CSV auf die Festplatte oder als Tabelle innerhalb der relationalen Datenbank, zu materialisieren. Mit **commit** kann ein solcher Checkout als neue Version in das System eingepflegt werden.

Betrachten wir nun eine Datenbank (Abb. 1) zum Aufzeichnen von Arbeitszeiten verschiedener Mitarbeiter.

Während der **init** Operation werden die bestehenden Daten (Blau) in eine Datentabelle importiert und mit einer Record ID (`rid`, Grün) versehen. Zusätzlich werden eine Versionstabelle und eine Metadaten-Tabelle angelegt. Die Versionstabelle zeichnet die entsprechenden `rids` für eine Version (`vid`) in einem Feld vom Typ `INT ARRAY` auf. Metadaten, wie die Vorgängerversion, die Commit-Message und die Attribute der Quell-Tabelle, werden in einer zweiten Tabelle hinterlegt.

Listing 1: Checkout

```
SELECT (user, task_name, task_start, task_end)
FROM Datatable
WHERE rid = ANY({r_1, r_2, r_3, r_4}::int[]);
```

Ein **checkout** sammelt zuerst alle `rids` für die angeforderte Version (`vid`) und führt daraufhin eine auf die Felder der Version projizierte Selektion (Listing 1) auf die Datentabelle aus. Die Felder einer Version können über das Attribute-Tupel in der Metadaten-Tabelle in Erfahrung gebracht werden. Anschließend wird das Ergebnis dem Nutzer als CSV oder als reguläre Tabelle zur Verfügung gestellt.

Mit dem **commit** Kommando kann eine neue Version aus einem modifizierten Checkout angelegt werden. Dazu werden die Zeilen des Checkouts mit den Einträgen der Datentabelle auf Gleichheit untersucht.

Listing 2: Commit neuer Einträge

```
INSERT INTO Datatable (user, task_name, task_start, task_end)
(SELECT * FROM Checkout)
EXCEPT
(
  SELECT * FROM Datatable INNER JOIN Versiontable
```

```
  ON rid = ANY(rlist)
  WHERE vid = ANY({v_1, ... v_n}::int[])
)
RETURNING rid;
```

Zeilen des Checkouts, für welche kein bereits bestehender Eintrag gefunden wurde, werden als neue Einträge (Listing 2) in der Datentabelle hinterlegt.

Listing 3: Finden bestehender Einträge

```
SELECT rid FROM
(
  SELECT * FROM Datatable INNER JOIN Versiontable
  ON rid = ANY(rlist)
  WHERE vid = ANY({v_1, ... v_n}::int[])
)
INNER JOIN Datatable
ON user=user
AND task_name=task_name
AND task_start=task_start
AND task_end=task_end;
```

Die so neu entstandenen `rids` werden nun, zusammen mit `rids` der bestehenden, gleichen Einträge (Listing 3), in der Versiontable mit einer neuen `vid` eingetragen. Anschließend speichert das System die eben erstellte `vid`, die übergebene Commit-Message, die verwendeten Attribute und die Eltern-Versionen in der Metadaten-Tabelle.

Falls das Schema des modifizierten Checkouts von den hinterlegten Attributen abweicht, wird dieses neue Tupel an Attributen in der Metatable festgehalten. Die neu dazugekommenen Attribute werden zu der Datentabelle via `ALTER TABLE` hinzugefügt. Entfernte Felder werden allerdings nicht aus der Datentabelle gelöscht, sondern lediglich aus dem Attribute-Tupel der aktuellen Version entfernt. Damit wird sicher gestellt, dass es zu keinem Verlust des Attributes in älteren Versionen kommt.

2.3 Probleme

Während OrpheusDB den import/export in relationale Datenbanken grundsätzlich unterstützt, liegt das Hauptaugenmerk auf dem Arbeiten mit CSV Dateien. Obwohl sich das CSV Format, besonders unter Data Scientists, großer Beliebtheit erfreut, bietet eine relationale Datenbank den Vorteil einer einheitlichen DSL und komplexeren Operationen.

OrpheusDB kann pro CVD nur eine Tabelle verwalten, was besonders für Applikationen außerhalb der Data Science ein Problem darstellt. So wäre es in unserem Beispiel wünschenswert, das Attribut `user` in eine eigene Tabelle zu isolieren um die Datenintegrität sicherstellen zu können. MusaeusDB versucht OrpheusDB um diese Funktionalität zu erweitern.

3 MusaeusDB

3.1 Übersicht

MusaeusDB versucht OrpheusDB um die Unterstützung mehrerer Tabellen zu erweitern, in dem es die zugrundliegende Idee auf mehrere Tabellen anstatt nur einer einzigen anwendet. Schemaänderungen und das Übertragen von Sequences wurden zur Vereinfachung ignoriert, wären aber grundsätzlich möglich.

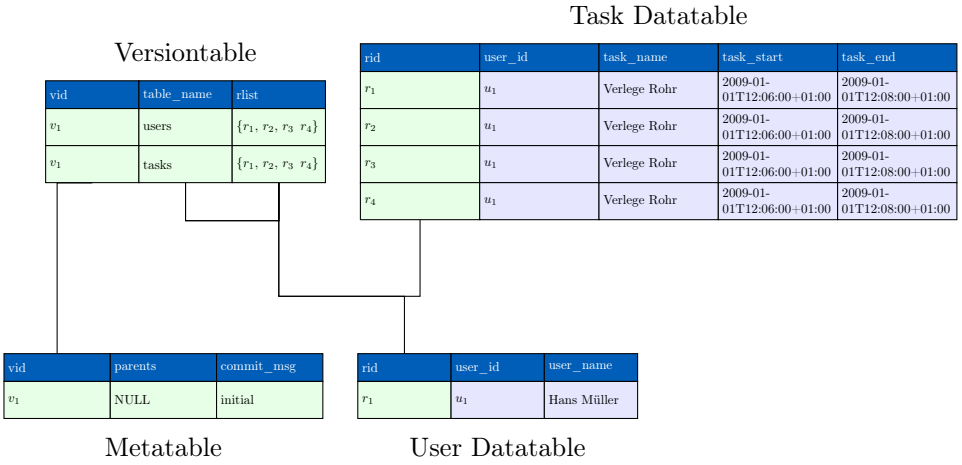

Abbildung 2: Datenbank Schema mit zweiter Tabelle

MusaeusDB wurde in C++11/14, ebenfalls als Git-ähnliches Command-Line Programm, implementiert und verwendet PostgreSQL [8] als relationale Datenbank.

3.2 Features

MusaeusDB bietet dieselben Haupt-Operationen wie OrpheusDB, jedoch mit dem entscheidenden Unterschied, dass nicht mehr auf einzelnen Tabellen, sondern auf ganzen Datenbanken operiert wird. Auf die Verwendung von CSV-Dateien für den Import/Export musste jedoch verzichtet werden, da das Dateiformat keine Relationen abbilden kann. Die Vorgehensweiße als solche bleibt aber bestehen, nach wie vor wird die Versionierung ausschließlich durch das Hinzufügen von Verwaltungs-Tabellen realisiert.

3.3 Implementierung

Betrachten wir nun ein erweitertes Datenbank Schema Abb. 2.

Parallel zu OrpheusDB werden während der **init** Operation die zu versionierenden Tabellen einer Datenbank importiert.

Listing 4: Finden der Quell-Tabellen

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema='source';
```

Dazu müssen die in einer Datenbank enthaltenen Tabellen aus dem `information_schema` in Erfahrung gebracht werden Listing 4.

Listing 5: Finden der Quell-Attribute

```
SELECT column_name, data_type, character_maximum_length
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'tasks'
AND table_schema = 'source'
AND column_name NOT IN ('rid');
```

Für jede der gefundenen Tabellen werden Spaltennamen, Datentypen und, im Falle von Zeichenketten, die Länge der Zeichenketten aus dem `information_schema`

Datasettable

dataset_name	tables
d_1	$\{t_1, ... t_n\}$

Checkouttable

checkout_name	dataset_name	parents
c_1	d_1	$\{v_1 ... v_n\}$

Abbildung 3: Zusätzliche Tabellen

abgerufen Listing 5. Diese Information wird anschließend genutzt, um neue Datentabellen mit einem identischen Schema plus einer fortlaufenden Record ID (`rid`) zu erstellen und zu befüllen.

Zusätzlich wird eine Versionstabelle angelegt, die Versionsnummer, Tabellenname und eine Liste der in der Tabelle und Version enthaltenen `rids` beinhaltet. Im Gegensatz zu OrpheusDB bildet hier ein Composite-Key aus `vid` und `table_name` den Primary-Key der Versiontable, um eine beliebige Anzahl von Tabellen pro Version zu erlauben.

Eine Metatable zeichnet die Elternversionen und die Commit-Message einer Version auf. MusaeusDB speichert die Attribute einer Tabelle nicht in der Metatable ab, sondern ruft diese bei Bedarf aus dem `information_schema` ab.

Um Tabellen-Namen nicht jedesmal neu aus dem `information_schema` suchen zu müssen werden diese in einer weiteren Tabelle abgelegt Abb. 3. Außerdem wird eine Tabelle zum Verwalten von verschiedenen Checkouts angelegt. Diese Tabelle ist initial leer.

Beide Tabellen sind vollkommen optional und werden nicht zwingend vom System benötigt.

Listing 6: Finden der Record IDs

```
SELECT distinct rlist, table_name
FROM Versiontable WHERE vid = ANY({v_1, ... v_n}::int[]);
```

Die **checkout** Operation sammelt zunächst für die angeforderten Versionen Listing 6 alle `rids` pro Tabelle.

Listing 7: Erstellen der Checkout-Tabellen

```
SELECT (user_id, task_name, task_start, task_end)
INTO checkout.tasks FROM datatable.tasks
WHERE rid = ANY({r_1, ... r_n}::int[]);
```

Anschließend wird über diese `table_name`, `rlist` Paare iteriert und die Daten, exklusive der `rid`, in eine entsprechende Checkout-Tabelle eingefügt Listing 7. Die zu einzufügenden Attribute werden dabei wie in Listing 5 gefunden. Nachdem dieser Prozess abgeschlossen ist, wird ein Eintrag in der Checkouttable erstellt, um später einen Checkout wieder einem Dataset und einer Version zuordnen zu können.

Der überarbeitete Checkout kann mit **commit** wieder in das System eingebracht werden. Dabei wird der jeweilige Checkout über die Checkout-Table wieder einem Datensatz zugeordnet. Über die Datasettable werden die enthaltenen Tabellen abgerufen. Für jede dieser Tabellen wird derselbe Prozess ausgeführt wie für die Commit-Operation in Unterabschnitt 2.2 mit dem einzigen Unterschied, dass in die Versionstabelle der Tabellenname mit eingetragen wird. Falls durch diesen Prozess eine neue Version entstanden ist, wird diese als `parent` in der Checkouttable eingetragen.

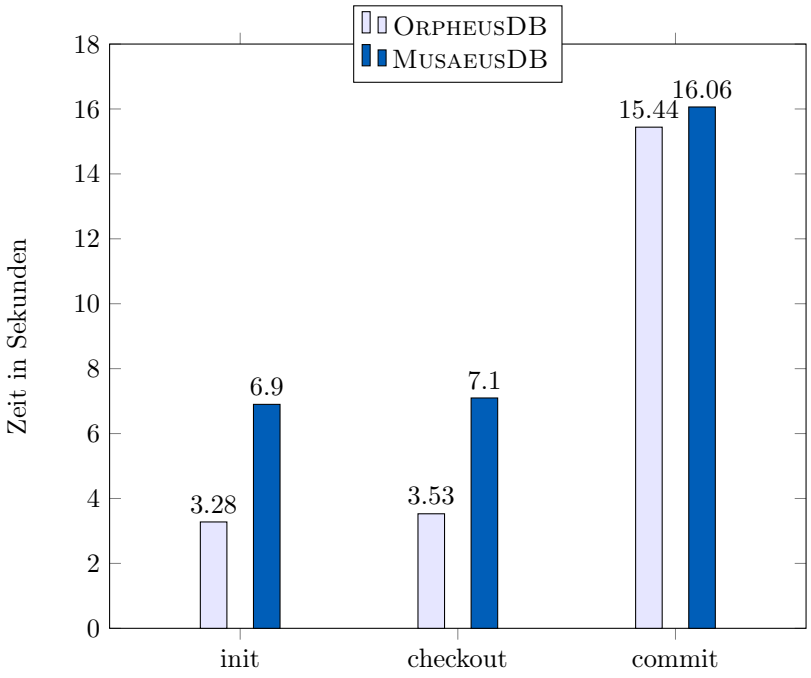

Abbildung 4: Laufzeit der Haupt-Operationen bei 1M Einträgen

3.4 Evaluation

Um die Auswirkungen der vorgenommen Änderungen auf die Performance-Charakteristiken zu untersuchen werden die Laufzeiten der Haupt-Operationen für OrpheusDB und MusaeusDB miteinader verglichen.

Dazu wurden Datensätze mit 1 Million Einträgen pro Tabelle in den beiden vorgestellten Schemata (Abb. 1, Abb. 2) erstellt. Der Datensatz für den MusaeusDB Test enthält damit in Summe 2 Millionen Einträge, jeweils 1 Million Zeilen für die Task und User Tabellen.

Für beide Implementierungen wird der Datensatz via `init` in das System geladen, mit `checkout` dem Nutzer zur Verfügung gestellt und per SQL `UPDATE` modifiziert. Diese Änderung wird auf alle Einträge angewandt um ein Worst-Case Szenario für einen darauf folgenden `commit` Befehl zu konstruieren.

Dieser Benchmark wurde mit 50 Wiederholungen auf einer Maschine mit einem Intel i7 6800K Prozessor, 32GB DDR4 RAM und einer NVMe SSD ausgeführt. Als relationale Datenbank für den Test diente PostgreSQL 9.6.3 und die MusaeusDB bzw. OrpheusDB Implementierung wurde mit GCC 7.2.1 kompiliert.

Wie die Ergebnisse des Benchmarks Abb. 4 zeigen haben die vorgestellten Änderungen kaum Einfluss auf die Leistungs-Merkmale. Die längeren Laufzeiten für `init` und `checkout` sind zu erwarten, da hier jeweils die doppelte Menge an Daten, für die Task und die User Tabellen, in bzw. aus dem System geladen werden.

4 Zusammenfassung

Durch seinen simplen Aufbau ließ sich OrpheusDB sehr leicht auf mehrere Tabellen erweitern.

Die fehlende Unterstützung für Sequences und Schema-Änderungen macht MusaeusDB aktuell wenig nützlich in einer Echtwelt-Anwendung. Beides sind jedoch Features, die ohne größeren Aufwand hinzugefügt werden können und sich bereits Planung befinden.

Zusätzlich sollte eine Implementierung als PostgreSQL Extension [9] für bessere Performance und eine tiefere Integration in Betracht gezogen werden.

Literatur

[1] Silu Huang and Liqi Xu and Jialin Liu and Aaron J. Elmore and Aditya G. Parameswaran, OrpheusDB: Bolt-on Versioning for Relational Databases, 2017

[2] A. Bhardwaj et al. Datahub: Collaborative data science & dataset version management at scale. CIDR, 2015.

[3] M. Maddox et al. Decibel: The relational dataset branching system. VLDB, 9(9).

[4] Git, `https://git-scm.com/`

[5] How big are the en wikipedia dumps uncompressed?, `https://meta.wikimedia.org/wiki/Data_dumps/FAQ`

[6] Size of the English Wikipedia database, `https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia`

[7] PostgreSQL ARRAY, `https://www.postgresql.org/docs/9.1/static/arrays.html`

[8] PostgreSQL, `https://www.postgresql.org/`

[9] PostgreSQL Extensions, `https://www.postgresql.org/docs/9.5/static/external-extensions.html`