Technologisches Gewerbe Museum

VSDB

Synchronisation von Heretogenen DBs

Author: Schrack Nikolas & Siegel Hannah

November 14, 2014

Contents

1 Arbeitszeit

1.1 Geschaetzte Arbeitszeit

Task	**Person**	**Time in hours**
Einlesen und Vergleich von gaengigen Middleware-produkten	Schrack Siegel	1 1
Implementierung der Middleware	Schrack Siegel	5 5
Erstellen der Tables, inserts	Schrack Siegel	1 1
Test mit mehr als einer Tabelle	Schrack Siegel	2 2
Dokumenation der Funktionsweise, Problematiken und Problemfälle, sowie allgemeine Dokumentation	Schrack Siegel	1 1
Testfaelle	Schrack Siegel	2 2
Total	Schrack Siegel	12 12
Gesammt Team		**24 stunden**

1.2 Tatsaechliche Arbeitszeit

Task	**Person**	**Time in hours**
Einlesen und Vergleich von gaengigen Middleware-produkten	Schrack Siegel	0.5 0.5
Datenmodell und aufsetzten der Datenbank	Schrack Siegel	3 2
Trigger	Schrack Siegel	3 3
RMI	Schrack Siegel	0 2
Mapping	Schrack Siegel	0 2
Testing und Verbesserung der vorlaufigen Loesung	Schrack Siegel	0 2
Einbauen des synchronisations-states	Schrack Siegel	0.5 0.5
Synchronisations Server	Schrack Siegel	0 1
Verhindern des Lost Updates, Auffangen und reverten von Fehlern, Reverten zum Zustand der funktioniert	Schrack Siegel	0 7
Junit Tests	Schrack Siegel	2 4
Dokumenation der Funktionsweise, Problematiken und Problemfälle, sowie allgemeine Dokumentation	Schrack Siegel	1 3
Total	Schrack Siegel	10 23
Gesammt Team		**33 hours**

2 Arbeitsdurchfuehrung

2.1 Entwerfen der Datenbank

Wir haben uns entschieden, eine ganz normale Firma abzubilden, wie sehr oft verwendet bei solchen test beispielen.
Es soll einen Mitarbeiter/Person geben, welcher einen namen hat, welcher sich unterscheiden soll. Weiters soll dieser Mitarbeiter bei einer Abteilung arbeiten (um eine 1:n Beziehung abzubilden), und als Teilnehmer bei einer Veranstaltung teilnehmen (um eine n:m Beziehung abzubilden).
Des weiteren werden wir darauf achten, moeglichst unterschiedliche und komplexe datentypen zu verwenden. Wir haben unter [1] gelesen, dass zum Beispiel boolean unter mysql und pgsql unterschiedlich ist.

2.1.1 PgSQL

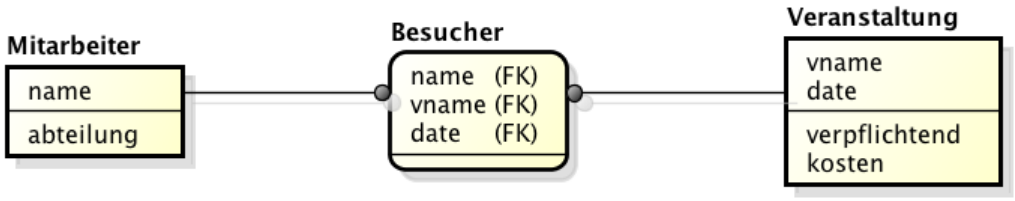


Figure 1: ER fuer pgSQL

2.1.2 Mysql

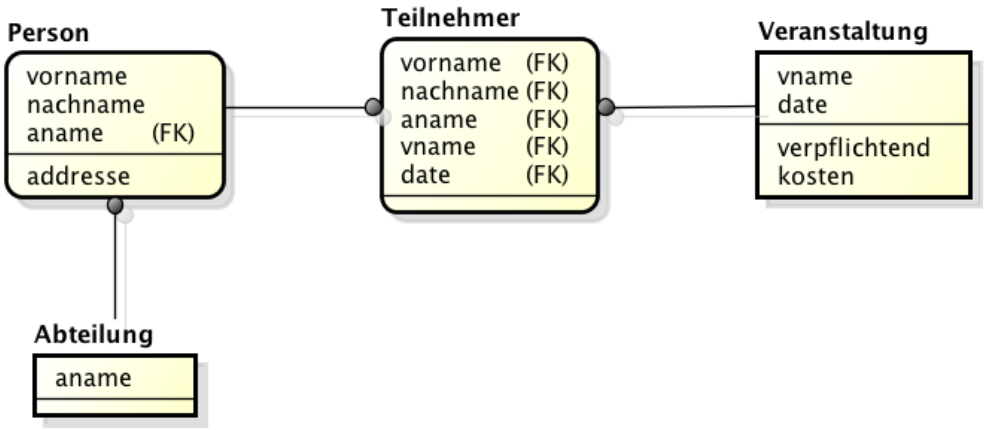
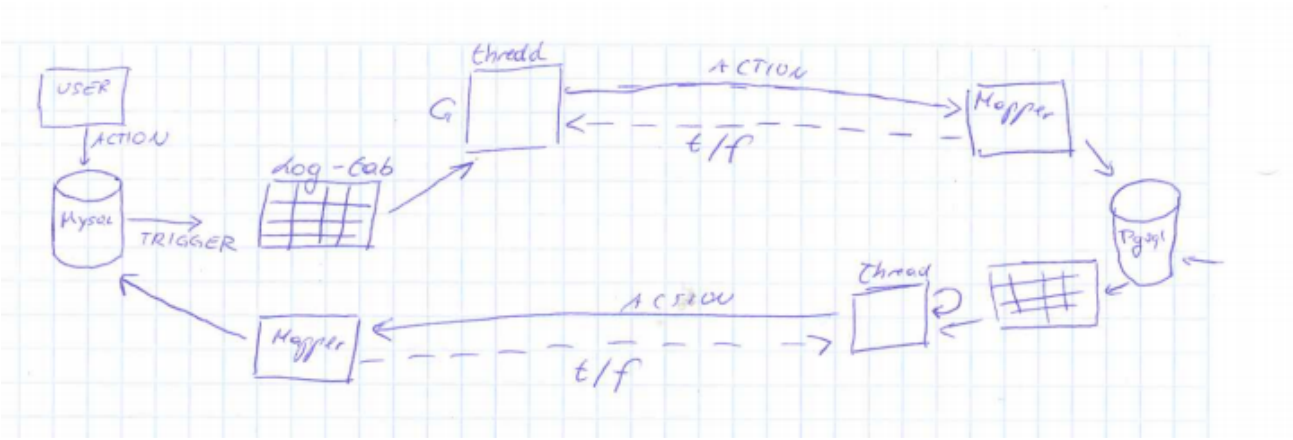


Figure 2: ER fuer mysql

2.2 Aussuchen der Middlewaretechnologien

Es gibt verschiedene Middlewaretechnologien. Da wir fanden, dass die Verwendung von Triggern jedoch nicht schwer sein koennte, allerdings sehr sicher und durchaus effizient und zeitnah, haben wir uns fuer diese Moeglichkeit entschieden. Weiters ist uns bewusst geworden, dass das direkte Mapping mehr oder weniger recht hardgecoded sein muss, und daher haben wir uns auch dafuer entschieden, eine wirklich eigene Middleware zu schreiben.
Das erste Konzept war sehr einfach:



Wir haben uns dafuer entschieden, dass wir mittels JDBC auf die Datenbank zugreifen wuerden. Daher brauchen wir 2 JDBC Verbindungen. Daher brauchen wir 2 Klassen (eine fuer eine Mysql Verbindung und die zweite fuer eine Psql verbindung).
Da wir nicht wollten, dass die Datenbanken sich nicht gegenseitig mit JDBC verbinden muessen, haben wir uns entschieden, RMI fuer die Kommunikation zu verwenden.
Ein Thread wird einfach alle paar sekunden abfragen, ob ein neuer log existiert(von Trigger wird automatisch in die Logged Tabelle geinserted) und sollte dies so sein, muss der Thread den Log auslesen und mittels RMI an den zweiten Peer schicken. Er bekommt dann von diesem eine Rueckmeldung (true od. false) und wenn diese erfolgreich war, dann kann er den log loeschen. Ansonsten muss er es eben in ein paar sekunden nocheinmal probieren.
Das hat auch den Vorteil, dass man theoretisch immer sehen koennte, welche actionen nicht funktioniert haben und diese bei bedarf haendisch nachzubessern.

2.3 Aufsetzen der Datenbanken

Die Create Scripts aus Astah zu exportieren und leicht anzupassen war sehr leicht. Wir haben dann auf jeder Tabelle 10 Datensaetze pro Tabelle (die gleichen Datensaetze) geinserted um einen Untergrund zum Testen zu haben.
Die Create Scripts hierfuer befinden sich in dem Abgabe Ordner.

2.4 Trigger und Logged-Tabellen

Wir mussten dann noch unser Create-script um die Logged Tabelle und Trigger erweitern. Wir haben ein Globales Schema file angelegt, um sich daran halten zu koennen: Dieses ist im Json format verfasst, da man mittels einem Json Parser die Elemente sehr angenehm wieder auslesen kann.

```
Person:
{"name":"<name>", "addresse":"<addresse>","aname":"<aname>"}

Teilnehmer:
{"name":"<name>", "vname":"<vname>", "date":"<date>"}

Veranstaltung:
{"vname":"<vname>",
    "date":"<date>","verpflichtend":<verpflichtend>,"kosten":<kosten>}

Abteilung:
{"aname":"<aname>"}
```

2.4.1 Mysql

Da wir dies im Unterricht mittels Mysql gelernt hatten, haben wir damit begonnen:

```
CREATE TABLE Logged (
 id BIGINT AUTO_INCREMENT PRIMARY KEY,
 action ENUM('insert', 'update', 'delete') NOT NULL,
 tableName VARCHAR(255) NOT NULL,
 old_values VARCHAR(255) NOT NULL,
 new_values VARCHAR(500) NOT NULL,
 date_done TIMESTAMP
);

-- insert teilnehmer
delimiter //
CREATE TRIGGER insertteilnehmer AFTER INSERT ON Teilnehmer FOR EACH ROW
BEGIN

INSERT INTO Logged(action,tableName,old_values,new_values,date_done)
VALUES ('insert','Teilnehmer','{}',CONCAT('{"name":"',NEW.vorname,'
',NEW.nachname,'","vname":"',NEW.vname,'","date":"',NEW.date,'"}'),NOW());

END;//
delimiter ;
```

2.4.2 Postgres

Das erstellen von Triggern mittels Postgres war dann nun auch etwas schwieriger.

```
CREATE OR REPLACE FUNCTION insert_into_Mitarbeiter() RETURNS TRIGGER AS
    $mitarbeiter$
 BEGIN
  INSERT INTO Logged(id,action,tableName,old_values,new_values,date_done)
  VALUES
      (DEFAULT,'insert','Person','{}','{"name":"'||NEW.name||'","addresse":"","aname":"'||NEW.al
  RETURN NEW;
 END;
$mitarbeiter$ LANGUAGE plpgsql;

CREATE TRIGGER insert_into_Mitarbeiter_trigger AFTER INSERT ON Mitarbeiter
FOR EACH ROW EXECUTE PROCEDURE insert_into_Mitarbeiter();
```

Es war ein bisschen schwierig, das Automatische updaten des Primary Keys in Postgres umzusetzten, aber im endeffekt hat es funktioniert.

2.5 RMI

Das Aufsetzen von RMI war (bis auf einen kleinen Bug) recht einfach.
Wir haben zwei interfaces defined, Mapper und Register und diese haben unsere MysqlServer und PostgresServer Klassen implementiert.

Das exportieren hat folgendermassen funktioniert:

```
//exporting the object
MapperRegister x = (MapperRegister) UnicastRemoteObject.exportObject(new
    PostgresMapper(), 0);

//getting the registry
Registry registry = LocateRegistry.createRegistry(1099);

//binding
registry.bind("Mysql1", x);
```

Und Sobal ein neuer Datensatz in der Logged Tabelle von Mysql war, musste man folgenden code verwenden:

```
Registry registry = LocateRegistry.getRegistry(registryIp,1099);

m_mapper = (Mapper) registry.lookup("Postgres1");

m_mapper.excecute("insert","Person","{....}","{}");
```

2.6 Mapping

Die execute Methode wird nun aufgerufen, und die eine Datenbank erhaelt daten von der anderen. Dann muessen diese Daten jeweils auf die Datenbank gemappt werden.
Bevor wir uns dafuer entschieden haben, ein Globales Schema zu verwenden, mussten wir jedoch Um ein Mysql Mapping zu machen wissen, von welchem Peer die daten stammten (um das mapping und das fetching aus dem Json Object auch richtig zu machen). Zum Beispiel schickt Mysql den abteilungsnamen ja mit aname und Postgres mit abteilug. Daher musste Mysql den abteilungs string verwenden und Postgres den anamen, was natuerlich so nicht funktionieren kann, da sobal man eine weitere Datenbank hinzufuegen moechte, alles wieder von neu gemacht werden muss. Daher haben wir nun das Globale Schema (wie oben beschrieben) verwendet. Die Daten welche aus dem Aufruf der excecute Methode als Parameter zur Verfuegung stehen mussten nur noch in den richtigen String umgewandelt werden. Das mussten wir leider Hardcoden, und wir sind uns sicher, dass es um einiges schoener gegeangen waere.

2.6.1 Mysql Person Insert Beispiel

Bei diesem Beispiel, ist es sehr einfach zu sehen, dass das Mapping sehr schwierig sein kann. Da im globalen Schema (sowie in der Pg Datenbank) der Name zusammen geschrieben wird, also vorname und nachname, muss man hierbei nun den namen splitten (Methode processName). Weiters muss abgefragt werden, ob die Abteilung schon besteht, und wenn nicht muss sie ebenfalls geinserted werden, da in unserem Beispiel nur in Mysql die Abteilung als eigene Tabelle abgelegt ist.

```
public boolean executePerson(String action, JsonObject pks, JsonObject values){
  String[] splited=null;
  String name = values.getString("name");
  splited = processName(name);
  new_nachname = splited[0];
  new_vorname = splited[1];
  new_abteilung = values.getString("aname");
  new_addresse = values.getString("addresse");

if(!(m_connection.isInDB("Abteilung", "aname" , new_abteilung)){
       m_connection.execUpdate("INSERT INTO Abteilung VALUES
           ('"+new_abteilung+"','current')");
}

String sql_string="";

if(action.equalsIgnoreCase("insert"))
 sql_string = "INSERT INTO Person VALUES('"+new_vorname+"','"+new_nachname+"',
 '"+new_abteilung+"','"+new_addresse+"')";

else if ...

return m_connection.execUpdate(sql_string);
```

2.7 Testing

Die ersten Tests waren recht erfolgreich. Wir konnten bereits nach kurzer zeit und ausmerzen der Fehler synchronisieren, in beide Richtungen versteht sich.
Es gab einige kleine Probleme mit den Datentypen, aber wir konnten diese durch anpassen des codes leicht beheben (zu Beispiel ist die Verwendung eines Booleans (true/false) in Postgres und eines Tinyints(0/1) in Mysql).

2.7.1 Synchronisation geht ewig

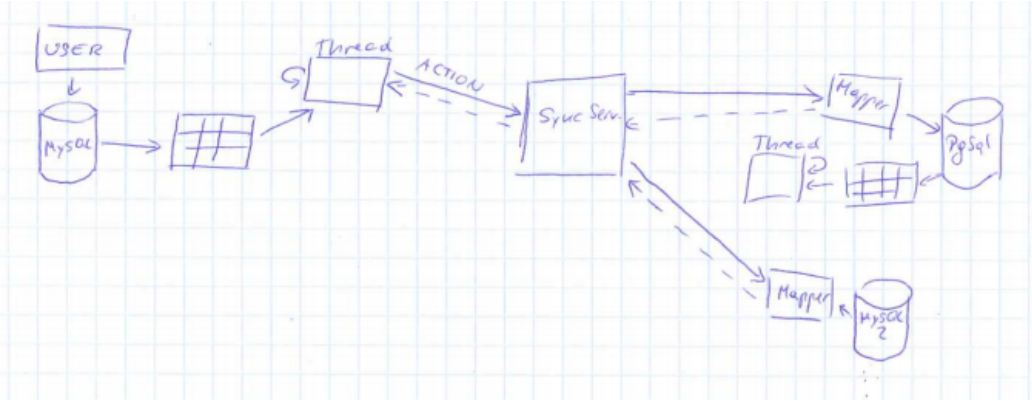
Etwas was wir in dem ersten Entwurf nicht bedacht hatten, war dass wenn zB ein Insert in Datenbank1 passiert, dieser wird gelogged und an Datenbank2 geschickt, diese Inserted die Daten ebenfalls, und loggt dann (weil sie ja auch einen After insert trigger hat) den insert, dieser wird wieder weiter geschickt u.s.w. Es ergibt sich daher ein Teufelskreis.
Wir haben daher in jede Tabelle eine spalte sync_state hinzugefuegt.
In den Triggern wird abgefragt, ob der sync_state von dem Datensatz richtig ist, und nur wenn ja wird geloggt.
In dem Programm muss nun immer darauf geachtet werden, den sync_state wieder auf current zu setzten.
Das funktioniert zwar theoretisch sehr gut, allerdings kann dies zu entweder einem Umgehen des synchronisierens bei setzten eines falschen wertes oder zu Fehlern fuehren, wenn diese synchronisationsabfrage irgendwo vergessen wird.

2.8 Vorteile und nachteile der Vorlaeufigen Loesung

Diese Loesung war, bis auf einige winzige bugs eventuell sehr lauffaehig. Jedoch ist sie sehr star, und das hinzufuegen neuer Datenbanken ist nur bedingt moeglich.
Weiters wird die Datenbankstruktur sehr stark im Code wieder gespiegelt. Das Programm ist daher sehr schwer an andere Datenbankinhalte anzupassen, aber das ist nun mal so.

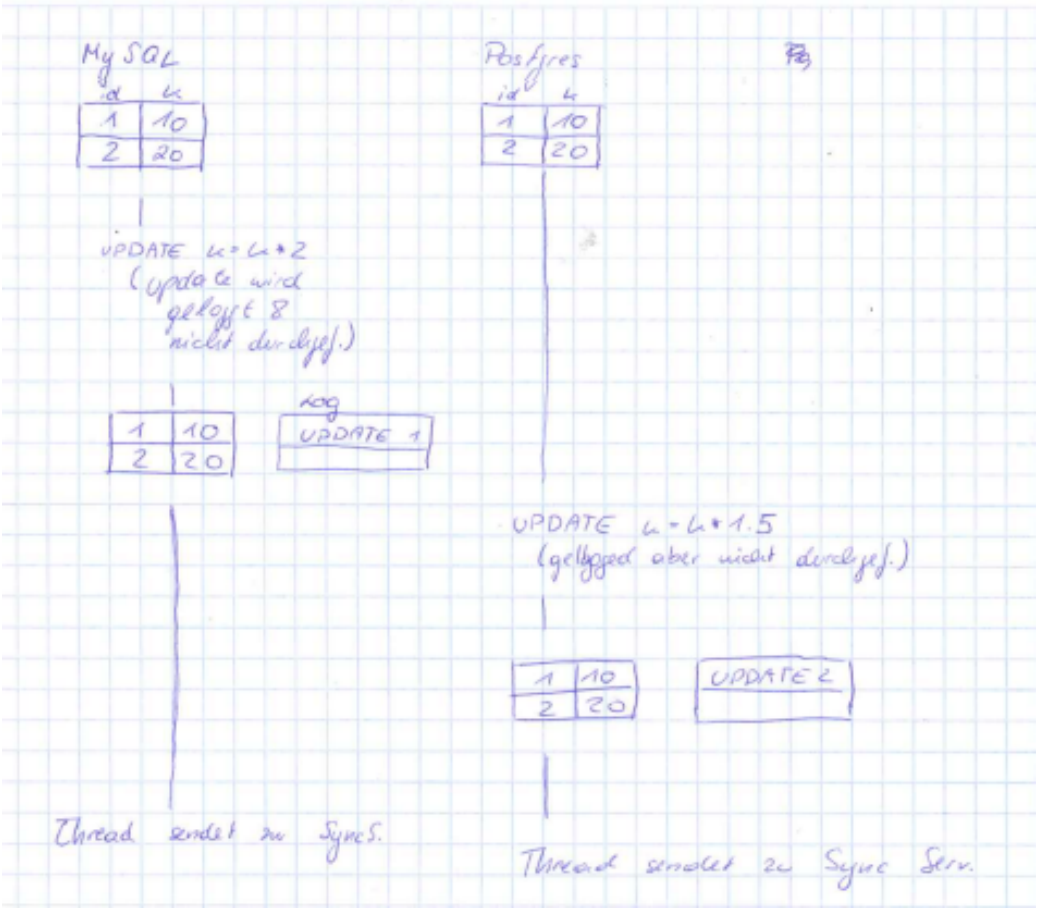
2.9 Synchronisation Server

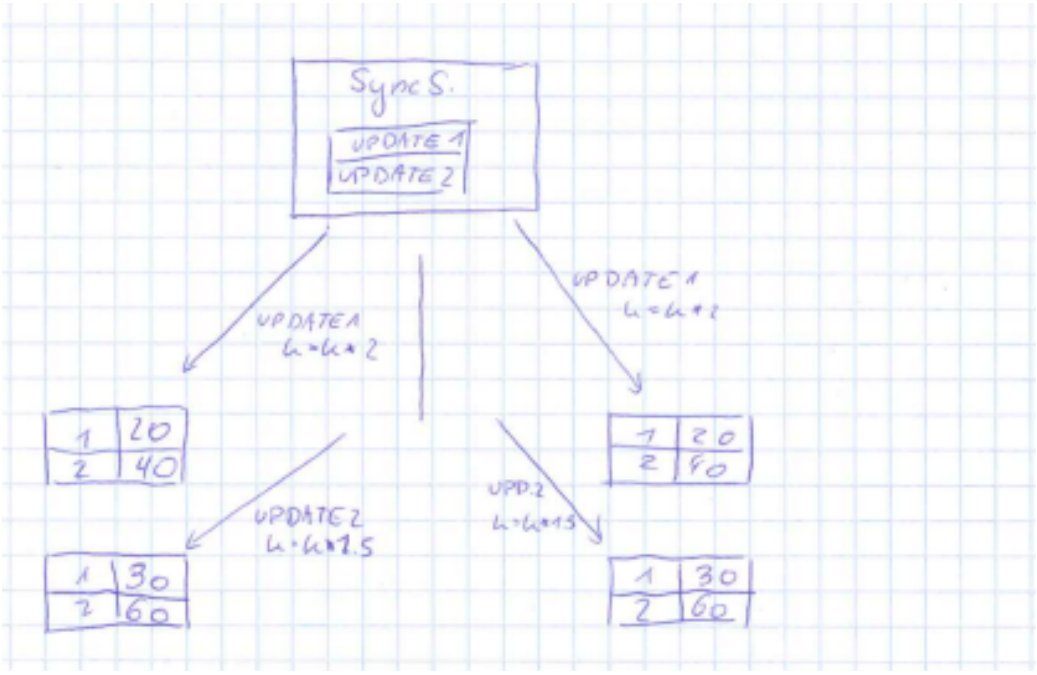
Um mehr kontrolle und vorallem ein etwas weniger statisches setting zu haben, haben wir anschliessend einen synchronisations server implementiert. Er ist nun der einzige der eine Registry aufspannt und in welcher er sich registriert. Alle Datenbanken koennen sich bei ihm anmelden und abmelden, was das dynamische hinzufuegen und auch wegnehmen von Datenbanken ermoeglichen soll. Man koennte nun im Synchronisations Server auch kontrolle von Daten vornehmen und wir wollten ihn auch verwenden, um das Lost Update zu vermeiden.



2.10 Lost Update

Die Vermeidung eines Lost Updates war nicht so einfach. Da die Synchronisation nicht sofort geschieht, sondern zeitgesteuert, muss ein Lost Update verhindert werden, in dem man die Middleware damit beauftragt. Unsere erste idee war es, einfach ein Update zu stoppen, das heisst, dass ein Update zwar geloggt wird allerdings eigentlich gar nicht passiert, um dieses dann mittels Hilfe des Synchronisationsservers gleichzeitig an alle peers weiterzugeben.
Diese Idee schien uns die einzig moegliche und wir haben daher versucht sie umzusetzten.





Aus mangelder Erfahrung mit Triggern haben wir allerdings sehr lange dafuer gebraucht.

Wir haben es aber geschafft, ein Update zu verhindern. Dies ging eher einfach in Postgres, wo man einfach return null anstatt return new schreiben muss, doch in msyql war die einzige moeglichkeit welche wir gefunden hatten, einen error zu werfen. Voller Freude, dass das nun endlich funktioniert haben wir ganz uebersehen und vergessen zu testen, ob 'bulk-updates' wie zum beispiel `UPDATE tab SET a=a*2` nicht mehr funktionieren, da Mysql nach dem ersten error damit aufhoert.

Weiters haben wir den Sync Server angepasst, um das Lost-Update aufzufangen, indem er im Falle eines updates auch an den sender das update sendet.

Im endeffekt, war alles soweit fertig doch uns ist aufgefallen, dass ein lost update sehr schwer zu simulieren ist, weil wir ja unsere werte hardcoden in die log tabelle. Das heisst, wir haben gar nicht kosten=kosten*2 geschrieben sonder kosten=10. Um ein Lost update zu vermeiden haette daher der Befehl und nicht die daraus resultierenden Werte geloggt werden muessen. Weiters hat das Updaten in Postgres nicht funktioniert und wir konnten den Fehler nicht sofort finden, und wir hatten kaum mehr zeit dazu. Wir haben uns daher entschlossen, den Lost Update einfach zu ignorieren und sind dennoch mit mehr Erfahrung aus diesem kleinen Irrweg herausgegangen.

Die Folgenden beiden Tabellen zeigen, was wo funktioniert hat:

Mysql

	Person	Veranstaltung	Teilnehmer	Abteilung
insert	yes	yes	yes	yes
update	yes	no (pg doesnt update)	no (pg doesnt update)	yes
bulk - update	no	no	no	
delete	yes	yes	yes	yes

Postgres

	Person	Veranstaltung	Teilnehmer
insert	yes	yes	yes
update	yes	no (pg doesnt update)	no (pg doesnt update)
bulk - update	yes	no (pg doesnt update)	no (pg doesnt update)
delete	yes	yes	yes

2.11 Konfliktlösung bei Zeitüberschneidung bzw. Datenproblemen (Log)

Da sich unser Synchronisationsserver (zwar nur zur laufzeit aber trotzdem) merkt, bei welcher Datenbank welche Aktion erfolgreich war, und ein Log nur geloescht wird, wenn es ueberall erfolgreich funktioniert hat, ist unsere Konfliktloesung einfach, dass das Problem nicht vergessen wird und bei bedarf von einem Administrator gesehen und geloest werden kann.

2.11.1 Commit Protokoll

Weiters haben wir uns ueberlegt, dass man mit Hilfe des Synchronisationsservers eine Art von einem Commit Protokoll implementieren koennte. Wir haben dies ausprobiert und sind zu dem schluss gekommen, das dies bei einer Loesung wie der unseren technisch moeglich waere.
Eine Zeichnung zu dieser Idee findet sich als PDF in unserer Abgabe wieder.

3 Testing

3.1 Testing per Hand

Das Testen per hand (also eine aktion auf der einen Datenbank ausfuehren und checken ob sich die zweite Datenbank danach gerichtet hat) war sehr einfach und auch durchaus gut am Anfang, wo wir die grundsaetzliche Synchronisation testen wollten. Spaeter jedoch war dieser Weg sehr kompliziert und zunehmend anstrengend. Des weiteren, wenn man keine automatisierten Testablaeufe hat, ist es sehr oft passiert, dass man inmitten eines Testes einfach angefangen

hat, die Fehler auszubessern anstatt fertig zu testen, und da bei aender Veraenderung des codes eventuell vorher positiv getestete Ergebnisse wieder nicht funktionieren koennten, hat man auf diese art und weise eigentlich nie fertig testen koennen.

3.2 Testing durch automatisierte Inserts

Beim Ausfuehren des Create Scriptes haben wir ganz am Ende einfach ein paar inserts dazu gemacht, mit welchen das testen einfacher war, denn man hat immer mit den selben Daten getestet und sofort eine Rueckmeldung bekommen. Jedoch war es auch hier schwer zu ueberpruefen ob der Test wirklich funktioniert hat, nur anhand einer positiven Ausgabe.

3.3 Junit

Am Ende haben wir uns entschieden, dass wir eigentlich Junit verwenden sollten.
Wir haben nicht sonderlich viel Erfahrug mit testing gehabt, trotzdem hat es so relativ gut funktioniert.
Das Code Coverage Tool, EclEmma hat anfaenglich nicht funktioniert doch nach einem Update hat es sehr gut getestet.

3.3.1 Testreport

4 Vorteile sowie Nachteile der Umgesetzten Synchronisationsart

4.1 Vorteile

- Es ging recht schnell
- Wenn man es richtig umsetzt, mit etwas mehr Know-how und eventuell mehr zeit, kann es auf jeden fall gut funktionieren
- Mit dem SynchronisationsServer schafft man sich einen guten zentralen Punkt, auch um zB commit-protokolle umzusetzen
- Eigentlich muss nur die Mapper klasse geaendert werden, wenn man andere Daten verwenden will
- Recht sicher
- Man koennte sehr viel umsetzten, da man mittels Triggern und einer hoeheren Programmiersprache viele moeglichkeiten hat

4.2 Nachteile

Die Nachteile moegen auch teilweise durch fehlendes know-how oder zeit zustande kommen

- Lost Update extrem schwierig umzusetzen
- Mapping sehr statisch, hardgecoded
- unuebersichtlich
- schwer zu debuggen (viele Fehlerquellen)
- User muss synchronisation angeben
- wenn synchronisation vergessen oder fehlerhaft umgesetzt wird, funktioniert sie nicht
- Nicht extrem aktuell, dadurch dass der Thread schlaeft, wird die synchronisation nicht sofort umgesetzt
- Wenn SynchronisationsServer nicht vorhanden, geht die synchronisation nicht mehr

5 Derzeitiger Stand des Beispieles

Die meisten funktionen funktionieren. Lost update funktioniert nicht.

6 Lessons Learned

Durch diese Uebung, wenn sie auch eigentich nicht sehr erfolgreich verlaufen ist, konnten wir an routine gewinnen, alte technologien wie JDBC und RMI nocheinmal wiederholen und neue Dinge verstehen.
Es gibt vieles, was wir das naechste mal anders machen wuerden:

1. Designueberlegung etwas genauer (Von Anfang an daran denken, dass man den Trigger nur einmal starten darf, ...)
2. Nur ids als Primary keys oder einzelne Spalten
3. Mapping dynamisher und Globale Schema in code nicht nur designueberlegung miteinbeziehen
4. Frueher mit der Verwendung von Testtools (e.g. JUnit) beginnen, mehr in Richtung Feature Driven Development
5. Sich von Anfang an gedanken ueber erweiterbarkeit und ausfallssicherheit machen
6. Andere Moeglichkeiten besser evaluieren
7. Schreiben von Zeitnaher Dokumentation, verwenden von git hub issue tools
8. Oefters, in kuerzeren Zyklen, Refactoring betreiben
9. Frueher merken, wenn eine Problemstellung so nicht loesbar ist

List of Figures

Glossary

ER Entity Relationship. 4, 15

References

[1] **Roland Bouman**, Dec 21 '09 um 21:36
http://stackoverflow.com/questions/1942586/comparison-of-database-column-types-in-mysql-postgresql-and-sqlite-cross-map
zuletzt abgerufen: 2014-10-24 09:00

[2] **Mysql Documentation**
http://dev.mysql.com/doc/refman/5.6/en/
zuletzt abgerufen: 2014-11-13 16:50

[3] **Psql Documentation**
http://www.postgresql.org/docs/
zuletzt abgerufen: 2014-11-13 16:50

[4] **Psql Documentation**
http://www.postgresql.org/docs/
zuletzt abgerufen: 2014-11-13 16:50

[5] **Various Stackoverflow Questions**
http://stackoverflow.com/
zuletzt abgerufen: 2014-11-13 16:50