En gammal hund lär sig sitta

2015-12-28

Introduktion

Som en DBA från datacenter 1.0 så håller jag DDL som helig. Den är definitionen av databasens objekt, är det som finns i produktion inte överensstämmande med den så är det produktion det är fel på. Att ändra objekt utan att ha DDL som grund och som vad som versionshanteras är förstås helt galet. Eller?

Jag har själv skrivit program som just kör DDL, håller reda på vad som kör skript för att skapa från grunden och vad som skall uppgraderas inkrementellt liksom att skriptet självt skall veta vad som behövs. Det är inte svårt, faktum är att alla borde haft sin egen version av det. Men om vi börjar ställa nya krav som att alla förändringar skall kunna backas automatiskt, automatisk test, kunna köra mot olika databaser, hantera olika behov i olika miljöer osv som moderna utvecklingsmiljöer i scrummiga projekt kräver, är svaret då samma?

Mitt svar var och förblev att DDL måste vara början och slutet på varje seriös hantering av databasobjekt, allt annat var tvunget att ta hänsyn till det. Men efter att frågan ställts och några uppdrag beslutat att inte göra det på ett sätt som jag tyckte kändes bra så tog jag mig en funderare. Skulle det vara möjligt att få dagens projekt att bättre versionshantera och att få mer tid över till värdefullare arbetsuppgifter än att skriva DDL och skriva skript för att backa ur, för att hantera olika miljöer osv?

Svaret är att trots ett hårdnackat motstånd så får jag nog säga att fördelarna kraftigt överväger. Nej, jag får inte min fina DDL som jag kan formatera minutiöst för att passa precis vad jag tänker. Men jag får så mycket annat på köpet att jag nog kommer rekommendera alla projekt framöver att använda versionshantering via produkter i stil med Liquibase.

För att visa fördelarna går vi igenom en massa olika förändringar som kan genomföras och ser på hur de skulle införas. Det är de jag skrev ner för vad som vore viktigt och för att bevisa att Liquibase inte dög. Nu blev utfallet ju då inte riktigt så.

Changelog genomgång

Här följer en genomgång av en massa ändringar man kan tänkas vilja göra med Liquibase och hur de görs. Se det som en referens för hur mycket som stöds och som en hjälp då en ändring skall införas. Testkörning av det kommer i avsnittet efter och där används samma ändringar så det finns inget behov att kopiera ihop ändringarna här till en komplett databaseChangeLog att testa med.

Eftersom JSON stöds så ville jag kika på det. JSON är ett betydligt modernare sätt att tagga upp data än XML. Det vinner mark snabbt och har numer starkt stöd i Oracles databas Så därför inleder jag med det och senare använder vi XML så att båda varianterna testas.

Rekommendation: Läs några stycken och få en känsla för att så gott som allt kan göras och gå vidare till stycket efter.

Skapa tabellen Person

```
"databaseChangeLog":[{
    "changeSet": {
        "id": "Createx"
       ,"author": "Mathias"
       ,"changes": [{
        "createTable": {
             "tableName": "person"
            ,"tablespace": "DATA"
            ,"remarks": "Innehåller personer"
            ,"columns": [{
                "column": {
                    "name": "person_id"
                   ,"type": "number(4)"
                   ,"remarks": "Unik identifierare för en person"
              }}
              ,{"column": {
                    "name": "address",
                    "type": "varchar(255)",
                    "remarks": "Adress till en person"
              }}
            ]
          }
        }]
      ,"rollback":[{
            "dropTable": {
                "cascadeConstraints": true
               ,"tableName": "person"
            }
      }]
    }
}
```

Här syns att vi definierar en “databaseChangeLog som är en fil med ändringar som körs i den ordning de finns i filen. Varje ändring är i en “changeSet”. Här ser vi bara en. Det är lämpligt att göra ändringar i atomiska changeset så att varje ändring antingen lyckas helt eller inte alls. Flera i en changeset ger svåra omstarter om något misslyckas mitt i en changeset.

Här ser vi att ändringen getts ett namn, vem som definierat den samt att en tabell skapas med diverse saker taggade:

- Namn på tabellen
- Tabespace
- Kommentar på tabellen
- Två kolumner
- Dataype på kolumnerna
- Kommentar på kolumnerna

Därefter kommer en av de saker som är intressantast. På samma ställe som skapandet av tabellen definieras så kan man också ta med definition av hur ändringen backas ur om det efter att ha infört är viktigt att kunna ta bort igen.

I det här fallet är det bara med för att visa hur det kan skrivas med taggar för vad som skall göras och inte kräva specifik DDL för det heller.

Det skall noteras att “rollback” sektionen här inte ens behöver skrivas, den skulle fungera automatiskt. “createTable” har automatisk rollback hantering liksom de allra flesta taggarna Liquibase stödjer. Det i sig är ett starkt argument för det, det finns ingen egen kod som måste testas och verifieras för den rollback som eventuellt skulle kunna behövas. Genom att kunna införa och backa ur förändringar kan man också i en miljö testa både före och efter versionen av systemet.

För att implementera en ny ändring man har i sin “databaseChangeLog” så är kommandot:

```
liquibase update
```

Har man infört en ändring och man vill backa bara den så är kommandot i stället:

```
liquibase rollback
```

Skapa en Tag

Den changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
    ,{
      "changeSet": {
         "id": "Tag version 1",
         "author": "Mathias",
         "changes": [{
             "tagDatabase": {
                 "tag": "version_1.3"
             }
         }]
      }
    }
```

Allt som görs här är att skapa en tag i en tabell som visar att här är en namngiven nivå på vårt schema. Förändringen definierar bara ett namn på vår "tag". Den kan sedan refereras om en stor uppgradering behöver backas eller för att bara veta vid vilket skript systemet uppgraderades till så att små patchar som lagts till senare lätt kan identifieras.

Lägg till en kolumn

Den changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
    ,{
      "changeSet": {
         "id": "Add kolumn kommentar",
         "author": "Mathias",
         "changes": [{
             "addColumn": {
                 "tableName": "person"
                ,"columns": [{
                     "column": {"name": "kommentar","type":
"varchar(255)","remarks": "Kommentar för personen"}
                 }]
             }
         }]
      }
    }
```

Här definierar ändringen den tabell kolumnen skall läggas till på samt namn, datatyp och kommentar för den nya kolumnen.

Om man efter att ha infört den vill backa till den tag vi skapade nyligen är kommandot:

```
liquibase rollback version_1.3
```

Vill man lägga till kolumnen på nytt därefter så är det som vanligt:

```
liquibase update
```

Ta bort tabellen

Den changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
    ,{
      "changeSet": {
         "id": "Drop table person",
         "author": "Mathias",
         "changes": [{
             "dropTable": {
                 "tableName": "person"
                ,"cascadeConstraints": true
             }
         }]
      }
    }
```

Här definieras som förändring vilken tabell och att eventuella constraints skall följas vid borttaget.

Här skulle det egentligen finnas en rollback som skapar om tabellen ifall droppandet av den misslyckas. Både det och återställandet av data behöver taggas specifikt. Men då den här databaseChangeLoggen avslutas här så har det inte implementerats.

Övergång från JSON till XML

Nu väljer vi XML för fortsatta ändringar. Anledningen är att JSON inte har stöd för taggar som sträcker sig över flera rader. Det är inte ofta det behövs, men för ren DDL/SQL/PL gör det det och att tvingas lägga in stora stycken kod på en enda rad är opraktiskt.

Header i XML

XML har en del setup-taggar för att tala om en del om dokumentet som inte JSON kräver i sin enklaste form. För Liquibase ser den headern i varje databaseChangeLog ut så här (med viss variation).

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
   xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
   xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
 http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.2.xsd
 http://www.liquibase.org/xml/ns/dbchangelog-ext
 http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">
```

Först definieras bara att det är XML och att encoding skall antas vara UTF-8.

Därefter startas databasChangeLog med att uppge referenser till scheman för XML för liquibase core och oracle-utvidgningarna.

Efter det här kommer XML-versioner av den changeSet-tag vi redan sett i JSON-syntax.

Skapa en partitionerad tabell

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Create tabell kund" author="Mathias"
runAlways="false" failOnError="true">
    <createTable tableName="kund"
                 tablespace="data"
                 remarks="Aktiva kunder">
      <column name="kund_id"
              type="number(4)"
              remarks="Unik identifierare för en person">
        <constraints nullable="false"/>
      </column>
      <column name="address"
              type="varchar(255)"
              remarks="Adress till en person"           >
        <constraints nullable="true" />
      </column>
      <column name="skapad"
              type="date"
              remarks="Datum då kunden skapades"        >
        <constraints nullable="false"/>
      </column>
    </createTable>
    <modifySql context="prod">
      <replace replace="data"
               with="data partition by range (skapad)
                       interval (numtodsinterval(1,'DAY'))
                       (PARTITION MINVAL  VALUES LESS THAN
                           (TIMESTAMP' 2010-01-01 00:00:00')
                         SEGMENT CREATION DEFERRED TABLESPACE data)
                       ENABLE ROW MOVEMENT" />
    </modifySql>
  </changeSet>
```

Som synes så blir det lite längre definition med XML, men bortsett från att tabell och kolumner definieras med XML-taggar så är inget nytt där. Det nya är själva partitioneringen.

Liquibase har inget direkt stöd för partitionerade tabeller efter som det är väldigt Oracle-specifikt. Dock finns det sätt att hantera det på som är väldigt kraftfullt. Ett av dem är modifySql som ger olika sätt att ändra den genererade DDLen. Här använder vi tablespacets namn “data” som en indikation på var vi vill lägga till lite egen kod.

Det görs genom att lägga med kod för att göra tabellen intervall-partitionerad. Dvs, tabellen kommer själv skapa partitioner som behövs baserat på den data som skrivs till den.

Det finns ytterligare en trevlig finess i changesetet. Det är att modifySql har en “tag” “context”. Det gör att vi kan få samma skript att anpassa DDL efter den miljö det körs i. Kör vi i en miljö där vi får, kan och vill ha partitionerade tabeller så anger vi att vi kör med det contextet påslaget, annars anger vi ett annat context.

Med kommandot `liquibase --contexts=prod update` så skapas tabellen partitionerad och med `liquibase --contexts=test update` så skapas den opartitionerad.

Lägg till en virtuell kolumn

Det changeset vi nu lägger till i databaseChangeLog ser ut så här:

```
  <changeSet id="Add virt col nasta_kundid (kundid + 1)"
author="Mathias" runAlways="false" failOnError="true">
    <addColumn tableName="kund">
      <column name="nasta_kundid" type="number(4)"
defaultValueComputed="generated always as (kund_id + 1) virtual"
remarks="Ett högre än kundid"/>
    </addColumn>
    <modifySql>
      <replace replace="DEFAULT"
               with=""/>
    </modifySql>
  </changeSet>
```

Här lägger vi till en kolumn vars värde inte lagras utan i stället beräknas när kolumnen läses (en sk virtuell kolumn i Oracle. Själv värdet räknas bara ut som kundid + 1. ModifySql används även här, men nu för att ta bort ett keyword i den genererade DDLen som Oracle inte accepterar.

Skapa ett unikt index

Det changeset vi nu lägger till i databaseChangeLog ser ut så här:

```
  <changeSet id="add unique ix kund_unq" author="Mathias"
runAlways="false" failOnError="true">
    <createIndex tableName="kund"
                 indexName="kund_unq"
                 unique="true"
                 tablespace="data">
      <column name="address"/>
    </createIndex>
  </changeSet>
```

Här uppger vi bara vilken tabell, namn på index att skapa och att det skall vara unikt samt vilket tablespace som skall användas. Därtill används columntaggen nu för att bara referera kolumn med namn som skall vara med i indexet.

Skapa ett partitionerat index

Det changeset vi nu lägger till i databaseChangeLog ser ut så här:

```
  <changeSet id="add partitioned ix kund_skapad" author="Mathias" runAlways="false"
failOnError="true">
    <createIndex tableName="kund"
                 indexName="kund_skapad"
                 unique="true"
                 tablespace="data">
      <column name="skapad"/>
    </createIndex>
    <modifySql context="prod">
      <replace replace="TABLESPACE data"
               with="tablespace data local"/>
    </modifySql>
  </changeSet>
```

Även detta indexet är unikt. Skillnaden från det förra är att vi gör det partitionerat genom att använda `modifySql`. Vi lägger helt enkelt till "local" för att ange att vi vill ha lokala partitioner baserat på tabellens partitionering.

Det är värt att notera att ändringar som förändras med modifySql har automatisk rollback. Det här indexet tas bort trots att det är partitionerat så länge det fortfarande är ett index med namnet kund_skapad på tabellen kund.

Skapa en sekvens

Det changeset vi nu lägger till i databaseChangeLog ser ut så här:

```
  <changeSet id="Create sequence kund_seq" author="Mathias"
runAlways="false" failOnError="true">
    <createSequence sequenceName="kund_seq"
                    startValue="1"
                    incrementBy="1"
                    minValue="1"
                    maxValue="10000"
                    ordered="true"
                    cycle="true"/>
  </changeSet>
```

Här uppges alla de saker vi är vana vid för sekvenser som metadata så sekvensen kan sedan skapas baserat på det.

Ändra en sekvens

Det changeset vi nu lägger till i databaseChangeLog ser ut så här:

```
  <changeSet id="Alter sequence kund_seq" author="Mathias"
runAlways="false" failOnError="true">
    <alterSequence sequenceName="kund_seq"
                   incrementBy="10"
                   maxValue="50000"
                   ordered="false"/>
  </changeSet>
```

Relativt standardmässigt - taggar för de saker men vill ändra på en skapad sekvens.

Skapa underordnad tabell kund_order till kund

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Create depending table kund_order" author="Mathias"
runAlways="false" failOnError="true">
    <createTable tableName="kund_order"
                 tablespace="data"
                 remarks="Orders from kunder">
      <column name="kund_id"  type="number(4)"
              remarks="Unik identifierare för en person">
        <constraints nullable="false"/>
      </column>
      <column name="order_id" type="number(4)"
              remarks="Unik identifierare för en order" >
        <constraints nullable="false"/>
      </column>
      <column name="skapad"   type="date"
              remarks="Datum då kunden skapades"        >
        <constraints nullable="false"/>
      </column>
    </createTable>
  </changeSet>
```

Inget nytt här från tidigare tabeller, Tabellen har bara en definierad referens som hänger ihop med tabellen kund. Kopplingen är ännu inte skapad.

Skapa en primär nyckel för kund och kund_order

Det två changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add primary key for kund" author="Mathias"
runAlways="false" failOnError="true">
    <addPrimaryKey tableName="kund"
                   columnNames="kund_id"
                   constraintName="kund_pk"
                   tablespace="data"/>
  </changeSet>
  <changeSet id="Add primary key for kundorder" author="Mathias"
runAlways="false" failOnError="true">
    <addPrimaryKey tableName="kund_order"
                   columnNames="order_id"
                   constraintName="kund_order_pk"
                   tablespace="data"/>
  </changeSet>
```

Här definieras att tabellen kund skall ha en primärnyckel som består av kolumnen kund_id och tabellen kund_order skall ha en bestående av order_id.

Skapa en främmande nyckel

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add foreign key for kundorder to kund"
author="Mathias" runAlways="false" failOnError="true">
    <addForeignKeyConstraint baseTableName="kund_order"
                             baseColumnNames="kund_id"
                             constraintName="kund_order_fk_kund"
                             deferrable="false"
                             onDelete="RESTRICT"
                             onUpdate="RESTRICT"
                             referencedTableName="kund"
                             referencedColumnNames="kund_id"/>
  </changeSet>
```

Här definieras referentiell integritet bestående av en främmande nyckel mellan kund och kund_order. Vilka kolumner som kopplas ihop liksom vad som skall ske vid uppdatering eller borttag definieras.

Skapa en unik constraint

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add unique key for kundorder on kund_id"
author="Mathias" runAlways="false" failOnError="true">
    <addUniqueConstraint tableName="kund_order"
                         columnNames="kund_id"
                         constraintName="kund_order_uc_kund_id"
                         deferrable="false"
                         disabled="false"
                         tablespace="data"/>
  </changeSet>
```

Här definieras vilken kolumn som skall vara unik på vilken tabell.

Sätt en kolumn till not null

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add not null constraint on kund.skapad"
author="Mathias" runAlways="false" failOnError="true">
    <addNotNullConstraint tableName="kund"
                          columnName="address"
                          defaultNullValue="Ingen Adress"/>
  </changeSet>
```

Här definieras vilken kolumn det är som skall ändras till att vara not null. Man kan också ange det värde som existerande NULL skall ändras till.

Skapa en vy

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add view kundvy" author="Mathias" runAlways="false"
failOnError="true">
    <createView viewName="kundvy"
                replaceIfExists="false">
      select a.kund_id
            ,a.address
            ,a.skapad       kund_skapad
            ,a.nasta_kundid
            ,b.order_id
            ,b.skapad       kund_order_skapad
       from       kund       a
      inner join kund_order b
         on a.kund_id = b.kund_id
    </createView>
  </changeSet>
```

För en vy anger man bara namn och om det skall vara "or replace" i DDLen. Själva frågan skriver man mellan start och slut-tag för `createView`. Det innebär att vyn kan vara specialiserad per databas eller miljö samtidigt som automatisk rollback stöds.

Skapa en synonym

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add synonym kundsyn" author="Mathias"
runAlways="false" failOnError="true">
    <sql endDelimiter=";"
         splitStatements="true"
         stripComments="true">
         <comment>Ingen standard-tag för synonymer än</comment>
      create synonym kundsyn for kund;
    </sql>
    <rollback>
      drop synonym kundsyn;
    </rollback>
  </changeSet>
```

Synonymer stöds inte i Liquibase utan här måste man använda en SQL-tag för att skriva sin egen DDL. Man anger vad som är slut på ett kommando (;), om kommentarer skall tas bort och om olika kommandon skall köras ett i taget. Efter det får man lägga in sin DDL som man vill ha den.

Det betyder också att automatisk rollback inte kan stödjas då Liquibase inte vet vad som görs i det här changesetet utan det får man komplettera med i en rollback-sektion.

Skapa en logg för en materialiserad vy

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add materialilized view log on kund"
author="Mathias" runAlways="false" failOnError="true">
    <sql endDelimiter=";"
         splitStatements="true"
         stripComments="true">
         <comment>Ingen standard-tag för loggar som stödjer
materialiserade vyer</comment>
      create materialized view log on kund tablespace data with rowid
including new values;
    </sql>
    <rollback>
      drop materialized view log on kund;
    </rollback>
  </changeSet>
```

Loggar för materialiserade vyer har heller inget specifikt stöd utan de skriver man också själv. Givetvis krävs då också en egen rollback.

Skapa en materialiserad vy

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add materialilized view kundmv" author="Mathias"
runAlways="false" failOnError="true">
    <ext:createMaterializedView viewName="kundmv"
                                tablespace="data"
                                queryRewrite="true"
                                subquery="select count(*) from
kund"/>
  </changeSet>
```

Här ser vi en Oracle-utvidgning av Liquibase standard funktionalitet. Taggens namn inleds med ext: och det är en referens till den header-XML som visas i sektion Header i XML. Där definieras schemat för utvidgningar till att refereras med ext.

Så med Oracle-utvidgningarna får vi fullt stöd för materialiserade vyer så de kan rullas tillbaka utan att skriva egen logik för det. Själva frågan skrivs förstås in som vanlig SQL.

Skapa en databaslänk

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Add database link dbl" author="Mathias"
runAlways="false" failOnError="true">
    <sql endDelimiter=";"
         splitStatements="true"
         stripComments="true">
         <comment>Ingen standard-tag för databaslänkar</comment>
      create database link dbl connect to dbuser identified by
iqwaszx using 'otherdb';
    </sql>
    <rollback>
      drop database link dbl;
    </rollback>
  </changeSet>
```

Inte heller databaslänkar är standardfunktionalitet i alla databaser så det finns ingen specifik tag för det. Därför används återigen SQL-taggen och egen rollback måste skrivas. Just den här länken går förstås inte någonstans utan skulle den användas efter att Liquibase skapat den skulle man få ett fel returnerat.

Insert i kund

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Insert row into kund" author="Mathias"
runAlways="false" failOnError="true">
    <insert tableName="kund">
      <column name="kund_id" valueNumeric="128"             />
      <column name="address" value="St Eriksgatan 117"      />
      <column name="skapad"  valueDate="2015-12-13 13:14:15"/>
    </insert>
    <rollback>
      <delete tableName="kund">
        <where>kund_id = 128</where>
      </delete>
    </rollback>
  </changeSet>
```

Även vanliga inserter kan man tagga upp väldigt deklarativt. Det här är bra för data man har lite av, men stora mängder kommer förstås gå långsamt. Men det man vill göra med Liqubase är förstås hantera data som man behöver versionshantera, inte operativt data i ett system.

Här används delete-taggen för att ta bort. Anledningen är att Liquibase i vissa fall inte kan avgöra om information i insert är tillräcklig för att unikt identifiera en rad som skall tas bort.

Ladda data

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
<changeSet id="Load data into kund" author="Mathias" runAlways="false" failOnError="true">
  <loadData tableName="kund"
            encoding="UTF-8"
            file="single/demo26.csv"
            quotchar="'"
            separator=",">
    <column name="kund_id" type="numeric"/>
    <column name="address" type="string"/>
    <column name="skapad"  type="date"/>
  </loadData>
  <rollback>
    <delete tableName="kund">
      <where>kund_id between 200 and 202</where>
    </delete>
  </rollback>
</changeSet>
```

Här uppges en fil, vilken encoding den har samt hur den skall parsas. Därtill de kolumner som träffas på i filen och vilken datatyp de har. Datatypen kan ju i teorin skilja mellan filen och tabellens.

Filen innehåller detta:

```
kund_id,address,skapad
200,'St Eriksgatan 115','2015-01-01 11:12:13'
201,'Odenplan 1','2015-07-01 11:12:13'
202,'Vanadisväger 12','2005-12-31 11:12:13'
```

Tre rader kommer alltså läggas till i tabellen kund. Notera att rollback är skriven för att ta bort just de raderna ifall man vill backa ur förändringen.

Ladda/Uppdatera data

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Load and replace data into kund" author="Mathias" runAlways="false"
failOnError="true">
    <loadUpdateData tableName="kund"
                    primaryKey="kund_id"
                    encoding="UTF-8"
                    file="single/demo27.csv"
                    quotchar="'"
                    separator=",">
      <column name="kund_id" type="numeric"/>
      <column name="address" type="string"/>
      <column name="skapad"  type="date"/>
    </loadUpdateData>
    <rollback>
      <delete tableName="kund">
        <where>kund_id between 203 and 204</where>
      </delete>
      <loadUpdateData tableName="kund"
                      primaryKey="kund_id"
                      encoding="UTF-8"
                      file="single/demo26.csv"
                      quotchar="'"
                      separator=",">
        <column name="kund_id" type="numeric"/>
        <column name="address" type="string"/>
        <column name="skapad"  type="date"/>
      </loadUpdateData>
    </rollback>
  </changeSet>
```

Filen innehåller:

```
kund_id,address,skapad
200,'St Eriksgatan 112','2015-01-01 01:12:13'
201,'Odenplansgatan 1','2015-07-01 11:12:13'
202,'Vanadisvägen 12','2005-12-31 20:21:22'
203,'Thorildsplan 28','2015-05-01 13:14:15'
204,'Kungsgatan 10','2016-01-01 00:02:28'
```

Taggen för att ladda är i det närmaste identisk, enda skillnaden är att primärnyckeln definieras. Det för att nu skall det för varje rad kollas om raden redan finns innan den antingen läggs till via insert eller tas bort via delete.

Det här är givetvis en mycket långsam process som utan index dessutom kan bli långsammare och långsammare ju mer data som läggs till i tabellen. För några få rader är det inget problem, men för mycket data kommer det ta enormt lång tid.

Rollback här är det som upptar mest plats. Den tar först bort de nya raderna och sedan används filen från förra stycket till att uppdatera den de raderna som då lades till så att de återställs till före den här uppdateringen.

Ändra datatyp på en kolumn

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Change datatype on skapad to timestamp"
author="Mathias" runAlways="false" failOnError="true">
    <modifyDataType tableName="kund_order"
                    columnName="skapad"
                    newDataType="timestamp(6)"/>
    <rollback>
      <modifyDataType tableName="kund_order"
                      columnName="skapad"
                      newDataType="date"/>
    </rollback>
  </changeSet>
```

För att ändra datatypen anges tabell och kolumn samt vilken datatyp man vill att kolumnen skall ändras till. För rollback görs samma sak fast med förändring till den datatyp den hade från början.

Ändra namn på en kolumn

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Rename column kund.skapad" author="Mathias"
runAlways="false" failOnError="true">
    <renameColumn tableName="kund"
                  oldColumnName="skapad"
                  newColumnName="skapad_ts"
                  remarks="Timestamp då kunden skapades"/>
  </changeSet>
```

Tabell, gammalt namn, nytt namn och även kommentr ifall man vill ändra den är vad som anges för att ge en kolumn ett nytt namn.

Ändra namn på en tabell

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Rename table kund_order" author="Mathias"
runAlways="false" failOnError="true">
    <renameTable oldTableName="kund_order"
                 newTableName="kund_order_info"/>
  </changeSet>
```

För att byta namn på en tabell anger man gammalt och nytt namn.

Ändra namn på en vy

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
<changeSet id="Rename view kundmv" author="Mathias" runAlways="false" failOnError="true">
  <renameView oldViewName="kundvy"
              newViewName="kund_info_vy"/>
</changeSet>
```

På samma vis som för tabeller anges bara gammalt och nytt namn.

Dela ut behörigheter - grant

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Run multiple grants" author="Mathias"
runAlways="false" failOnError="true">
    <sqlFile encoding="utf8"
             endDelimiter=";"
             path="single/demo32.sql"
             relativeToChangelogFile="true"
             splitStatements="true"
             stripComments="false"/>
    <rollback>
      <sqlFile encoding="utf8"
               endDelimiter=";"
               path="single/demo32.rbk"
               relativeToChangelogFile="true"
               splitStatements="true"
               stripComments="false"/>
    </rollback>
  </changeSet>
```

Grant är inte heller ett kommando Liquibase stödjer. Därför görs de här genom att köra en fil med flera kommandon. Varje kommando skiljs ut vid ";"

Filen innehåller dessa två grantar:

```
grant select on kund            to system;
grant select on kund_order_info to system;
```

Rollback tar bort samma grantar, med följande innehåll:

```
revoke select on kund            from system;
revoke select on kund_order_info from system;
```

Att grant görs till system är förstås bara för demonstrationens skull.

Uppdatera en rad

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Update a row with kund_id 128" author="Mathias"
runAlways="false" failOnError="true">
    <update tableName="kund">
      <column name="address" value="St Eriksgatan 17" />
      <where>kund_id = 128</where>
    </update>
    <rollback>
      <update tableName="kund">
        <column name="address" value="St Eriksgatan 117" />
        <where>kund_id = 128</where>
      </update>
    </rollback>
  </changeSet>
```

Här uppdateras adressen för den första kunden vi la in i tabellen kund. Man uppger nytt värde för kolumner samt den where-sats som skall tillämpas för att välja ut rader som skall ändras.

Även här behövs en egen rollback då Liquibase inte kan avgöra vad värdet brukade vara.

Använd property och skapa en tabell

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Create table betalning with property for tablespace"
author="Mathias" runAlways="false" failOnError="true">
    <createTable tableName="betalning"
                 tablespace="${ts.betalning}"
                 remarks="Betalningar från kunder">
      <column name="kund_id"     type="number(4)"
              remarks="Unik identifierare för en person">
        <constraints nullable="false"/>
      </column>
      <column name="betal_datum" type="date"
              remarks="Datum då kunden betalde"         >
        <constraints nullable="false"/>
      </column>
      <column name="belopp"      type="number(5,2)"
              remarks="Belopp som kunen betalade"       >
        <constraints nullable="false"/>
      </column>
    </createTable>
  </changeSet>
```

Högst upp i filen före den första `changeSet` läggs också en definition av en property:

```
  <property name="ts.betalning" value="data"/>
```

En property är en variabel eller ersättningssträng så att det kan användas upprepat och få konsekventa effekter. I `createTable` anges `tablespace` som "`{ts.betalning}`" och det kommer vid körning ersättas med vad som anges i `value` för propertyn `ts.betalning`. Dvs, `tablespace` kommer sättas till att vara `data`.

Bortsett från propertyanvändningen är allt som i tidigare exempel då tabeller skapas.

Använd villkorlig changeSet

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Create index betalning.betalning_unq bara om tabell
finns" author="Mathias" runAlways="false" failOnError="true">
    <preConditions>
      <tableExists tableName="betalning"/>
    </preConditions>
    <comment>Index skapas bara om tabellen finns definierad.
             Dessutom är den här kommentaren med för att visa
             kommentering av change set.

             NOTERA: updateSQL gör inget med precondition, de kollas
                     bara vid live exekvering.
    </comment>
    <createIndex tableName="betalning"
                 indexName="betalning_unq"
                 unique="true"
                 tablespace="${ts.betalning}">
      <column name="kund_id"/>
      <column name="betal_datum"/>
    </createIndex>
  </changeSet>
```

Själva indexet som skapas är inte annorlunda än tidigare annat än att en propery används igen för tablespacet och att indexet nu har två kolumner angivna.

Det intressanta här är först och främst `preConditions` taggen. Med den kan vi likt context avgöra vad som skall ske. med preconditions kan man använda en massa färdiga tester som här att en viss tabell måste finnas eller skriva en helt egen kontroll via SQL.

Det är mycket användbart i vissa fall, men det skall förstås inte användas för att veta vad som skall köras, det håller ju Liquibase redan ordning på så överdriven användning kan indikera att man använder preconditions felaktigt.

Skapa ett paket

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Create procedure output" author="Mathias" runAlways="false"
failOnError="true">
    <createProcedure>
      create or replace package output as
        procedure print_line;
      end output;
    </createProcedure>
    <createProcedure>
      create or replace package body output as
       procedure print_line as
    begin
     dbms_output.put_line(q'#It's alive#');
    end print_line;
      end output;
    </createProcedure>
    <rollback>
      <sql>drop package output;</sql>
    </rollback>
  </changeSet>
```

För att skapa ett paket använder vi createProcedure taggen och får då först skapa paketet och sedan implementationen (body). Det görs i ett changeset efter som ett paket kan ses som en atomisk förändring.

För att rulla tillbaka förändringen använder vi egen SQL. Varför inte dropProcedure? Jo, den droppar just en procedur, men inte ett paket.

Installera en APEX applikation i nytt workspace

Det changeset vi lägger till i databaseChangeLog nu ser ut så här:

```
  <changeSet id="Import APEX application into test workspace" author="Mathias"
runAlways="false" failOnError="true">
    <preConditions onFail="MARK_RAN">
      <sqlCheck expectedResult="1">
        select count(*)
          from apex_workspaces
         where workspace = 'TEST';
      </sqlCheck>
    </preConditions>
    <executeCommand executable="sqlplus">
      <arg value="liqdemo/liqdemo@localhost:1521/o12pdb"/>
      <arg value="@single/demo37.sql"/>
    </executeCommand>
    <rollback>
    </rollback>
  </changeSet>
```

Applikationen vi installerar har id 101 och byggdes i ett workspace som heter DEV. Nu skall den installeras i workspace TEST som applikation 200.

För att kunna hantera att alla databaser med den lösning vi bygger kanske inte har APEX så har vi ett villkor i preConditions om att workspace TEST måste finnas. Annars markeras changesetet som kört utan att något görs.

Själva installationen av applikationen görs genom att använda sqlplus och köra ett skript som ställer in några ändringar mot defaulterna från APEX export-fil innnan själva importen körs. Det skriptet ser ut så här.

```
declare
  l_workspace_id number;
begin
  select workspace_id into l_workspace_id
    from apex_workspaces
   where workspace = 'TEST';
  --
  apex_application_install.set_workspace_id( l_workspace_id );
  apex_application_install.set_application_id( 200);
  apex_application_install.generate_offset;
  apex_application_install.set_schema( 'LIQDEMO' );
end;
/

@single/f101.sql
exit
```

Skriptet tar reda på ID för workspace TEST och ställer sen in att det är det workspacet vi vill jobba med, att applikationen är 200 och att schemat applikationen jobbar emot skall vara LIQDEMO. För mer information se Oracles dokumentation för apex_application_install.

Rent generellt så kan många saker ställas in och vad som behövs beror på hur mycket som skiljer från utveckling. Skall applikationen installeras med samma ID i samma workspace i samma databas så behöver man bara köra f101.sql.

Det är allt som behövs för att kunna installera apex via liquibase. Lättare än ryktet säger!

Installera Liquibase

Ladda hem Liquibase från http://www.liquibase.org/download/index.html och Oracle utvidgningarna från https://github.com/liquibase/liquibase-oracle/releases. Båda är rena extraheringar utan egentlig installation. Oracleutvidgningen (liquibase-oracle*.jar skall vara i samma bibliotek som liquibase.jar)

Ställ in din PATH att inkludera basbiblioteket för liquibase.

Därmed skall installationen vara klar.

Kör demonstrationen

Demonstrationen av alla de ändringar som beskrivs i Changelog genomgång kan köras med filerna i den här tarballen.

I biblioteket liqdemo finns en fil liquibase.properties som används för att ställa in diverse konfigurationsiinställningar såsom användare att logga på med och databas att koppla upp sig mot.

Det som behöver ändras i den är classpath som behöver referera ojdbc7.jar från Oracle. Har du inte installerat en Oracle-databas eller komplett Oracle-home på samma maskin kan du hitta den på http://www.oracle.com/technetwork/database/features/jdbc/jdbc-drivers-12c-download-1958347.html.

Dessutom behöver url och referenceurl ändras till din databas. Det är en vanlig jdbc databas-connectionstring.

För att sedan kunna köra demo måste du skapa en användare. Du får det kompletta skriptet med:

```
grep '#' liquibase.properties | cut -f2 -d#
```

För att även kunna testa dbdoc och diff så behövs ett konto att jämföra emot. Vill du skapa även det så kör även det skript som du får med:

```
grep '#' liquibase.properties | cut -f2 -d#|sed
s/liqdemo/liqdemo_ny/g
```

Nu har du användarna skapade, det som nu behövs är bara att köra demonstrationen. Den körs med run.sh. Dvs, den kräver en unix/linux miljö, alternativt bör det gå att köra med putty i Windows.

Vill du se vad som kommer gås igenom kan du visa det med:

```
grep Changelog run.sh|cut -f2 -d"'"
```

Varje ändring visas som tre steg:

1. Changelog där den sista är den som kommer köras och de tidigare är redan körda.
2. DDL-skript som kommer köras
3. Körning av ändringen.

I några fall visas innehåll i en fil eller DDL-skript som fås vid olika körningar.

För att gå vidare mellan alla steg så trycker man bara <enter>.

Vill man köra alla ändringar på en gång så man får dem i databasen så tar man reda på vilket nummer den sista filen i single-biblioteket har och kör följande stående i liqdemo biblioteket. Du byter förstås ut NN mot det nummer du letade upp

```
rm demo.xml
ln -s single/demoNN.xml demo.xml
liqdemo update
```

Det förutsätter förstås att objekten i liqdemo schemat antingen tagits bort eller bara skapats via demoskripten så bara de som saknas skapas.

Update kommandon

Kommandon som kan användas för att uppdatera med liquibase. Dvs:
`liquibase <kommando>`

För varje kommando finns det ett med tillägget “Sql” som inte uppdaterar alls utan bara skriver SQL för att göra uppdateringen till stdout. T ex finns `updateSql` om komplement till `update`.

Kommando	Funktion
update	Uppdaterar databasen enligt changelog till senaste version i changeloggen.
updateCount <num>	Kör num ändringar från changeloggen.
updateToTag	Uppdaterar enligt changeloggen tills den specificerade taggen nås.

Rollback kommandon

Kommandon som kan användas för att göra rollback med liquibase. Dvs:
`liquibase <kommando>`

För varje kommando finns det ett med tillägget “Sql” som inte uppdaterar alls utan bara skriver SQL för att göra uppdateringen till stdout. T ex finns `rollbackSql` som komplement till `rollback`.

rollback <tag>	Rullar tillbaka ändringar upp till den angivna taggen
rollbackToDate <date/time>	Rullar tillbaka ändringar till hur databasen var vi den angivna tiden. Tid anges i format yyyy-MM-dd'T'HH:mm:ss.
rollbackCount <num>	Rullar tillbaka num ändringar.
futureRollbackSQL	Skriver ut SQL till stdout för att rulla tillbaka icke körda ändringar till det läge databasen är i nu.
futureRollbackSQL <num>	Skriver ut SQL till stdout för att rulla tillbaka num icke körda ändringar till det läge databasen är i nu.
futureRollbackFromTagSQL <tag>	Skriver ut SQL till stdout för att rulla tillbaka icke körda ändringar till det läge databasen är i nu fram till angiven tag i changeloggen.
updateTestingRollback	Uppdaterar databasen enligt changelog, rullar tillbaka och kör in dem igen. Värdefull för att kunna testa ändringar och dess rollback.

Offline-läge

Om man jobbar i en miljö där man inte tillåts köra direkt mot produktion med sin changelog ens för att skapa de skript som behövs via updateSql så kan man jobba i offline. Det finns två sätt att jobba offline, offline och snapshot.

De beskrivs här mycket översiktligt och utan exempel bara för att påminna om dess funktion ifall man stöter på ett sådant behov.

Offline

Offline innebär att man använder en komma separerad file (CSV) som representerar tabellen databasechange log, på så vis kan Liquibase fatta beslut om vilka förändringar som behövs och vilka som redan är införda.

Snapshot

Snapshot innebär att man har en fil som representerar schemat så att dokumentation och annat kan skapas baserat på vad som finns i databasen just nu. Det är också ett format som man själv kan använda för att skapa rapporter från då det är ett format som låter sig parsas. Det går även att få i formatet json.

Vill man jobba mot ett snapshot för en miljö man inte har åtkomst till så begär man bara en liquibase snapshot from den miljön. Den skapas med något av följande kommandon:

```
liquibase snapshot
liquibase snapshot --snapshotFormat=json
```

Det första formatet ger denna struktur för tabellen betalning:

```
    liquibase.structure.core.Table:
        BETALNING
            columns:
                BELOPP
                    nullable: false
                    remarks: Belopp som kunen betalade
                    type: NUMBER(5, 2)
                BETAL_DATUM
                    nullable: false
                    remarks: Datum då kunden betalde
                    type: DATE(7)
                KUND_ID
                    nullable: false
                    remarks: Unik identifierare för en person
                    type: NUMBER(4, 0)
            indexes:
                BETALNING_UNQ
                    columns:
                        BETAL_DATUM
                        KUND_ID
                    unique: true
            remarks: Betalningar från kunder
```

Det andra (json) formatet ger denna struktur för tabellen betalning:

```
"liquibase.structure.core.Table": [
  {
    "table": {
      "columns": [
        "liquibase.structure.core.Column#f23f150",
        "liquibase.structure.core.Column#f23f151",
        "liquibase.structure.core.Column#f23f152"]
      ,
      "indexes": [
        "liquibase.structure.core.Index#f23f149"]
      ,
      "name": "BETALNING",
      "remarks": "Betalningar från kunder",
      "schema": "liquibase.structure.core.Schema#f23f101",
      "snapshotId": "f23f148"
    }
```

Alla referenserna i json återfinns förstås objekt för på annan plats i json-strukturen. Det första formatet är betydligt smidigare att jobba med om man själv skall läsa filen eller skriva skript mot det i språk som inte har direkt json-stöd.

Generate ChangeLog

System som inte använt Liquibase från början behöver etablera en start med skript som kan skapa databasen i det format från vilket man börjar använda Liquibase. Det är också möjligt att skapa en baseline för systemet efter varje produktionssättning. Det är inte rekommenderat, men i miljöer där det arbetssättet används kan Liquibase självt skapa det mesta av de changeset man behöver. Det som behöver lite extra arbete är objekt som Liquibase inte stödjer, såsom procedurer, funktioner och paket. Liksom Oracle-specifika saker som databaslänkar. Men får man 95% gratis kan resten skapas ganska lätt.

Det man inte får med här är saker som stored procedures, functions, procedures, triggers och objekt som inte stöds av Liquibase core.

Skapa en changelog motsvarande hur schemat ser ut just nu:

```
liquibase --changeLogFile=nyChangeLog.xml generateChangelog
```

Då får man följande changelog: (Kapat en del av utrymmesskäl)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog-ext
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd
http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.4.xsd">
    <changeSet author="mathias (generated)" id="1453036545067-1">
        <createSequence cacheSize="20" cycle="true" incrementBy="10" maxValue="50000"
minValue="1" ordered="true" sequenceName="KUND_SEQ" startValue="10"/>
    </changeSet>
-- SNIP SNIP ************
    <changeSet author="mathias (generated)" id="1453036545067-11">
        <addForeignKeyConstraint baseColumnNames="KUND_ID" baseTableName="KUND_ORDER_INFO"
constraintName="KUND_ORDER_FK_KUND" deferrable="false" initiallyDeferred="false"
onDelete="RESTRICT" onUpdate="RESTRICT" referencedColumnNames="KUND_ID"
referencedTableName="KUND"/>
    </changeSet>
    <changeSet author="mathias (generated)" id="1453036545067-12">
        <createView fullDefinition="true" viewName="KUND_INFO_VY">CREATE OR REPLACE FORCE VIEW
KUND_INFO_VY (KUND_ID, ADDRESS, KUND_SKAPAD, NASTA_KUNDID, ORDER_ID, KUND_ORDER_SKAPAD) AS
select a.kund_id
            ,a.address
            ,a.skapad       kund_skapad
            ,a.nasta_kundid
            ,b.order_id
            ,b.skapad       kund_order_skapad
       from       kund       a
      inner join kund_order b
         on a.kund_id = b.kund_id</createView>
    </changeSet>
</databaseChangeLog>
```

Diff

Att jämföra två scheman kan ibland vara värdefullt för att se till att man ha changeset för alla införda förändringar. Givetvis skall man jobba med att införa dem bara via changeset. För att testa kan vi köra våra ändringar mot LIQDEMO_NY schemat som då representerar alla förändringar vi gjort och så backa LIQDEMO några ändringar så det finns skillnader att hitta.

Vi installerar allt fram till och med steget före APEX-applikationen i LIQDEMO_NY och backar 10 changeset av de som införts i LIQDEMO. Så nu kan man tänka sig LIQDEMO som produktion och LIQDEMO_NY som en miljö uppdaterad med det senaste för en kommande release.

```
rm demo.xml;ln -s single/demo36.xml demo.xml
liquibase --username=liqdemo_ny --password=liqdemo_ny update
liquibase rollbackCount 10
```

Det första liquibase-kommandot ger nya värden som gäller istället för de som är i liquibase.properties så att schemat LIQDEMO_NY uppdateras i stället för LIQDEMO.

För att nu få fram skillnaden mellan de två miljöerna så använder vi diffkommandot för Liquibase. Diff fungerar utan mer parametrar eftersom referens-schemat mm är definierat redan i liquibase.properties.

```
liqubase diff
```

Vi får nu en rapport som börjar så här:

```
Diff Results:
Reference Database: LIQDEMO_NY @ jdbc:oracle:thin:@localhost:1521/o12pdb (Default Schema:
LIQDEMO_NY)
Comparison Database: LIQDEMO @ jdbc:oracle:thin:@localhost:1521/o12pdb (Default Schema:
LIQDEMO)
Product Name: EQUAL
Product Version: EQUAL
Missing Catalog(s): NONE
Unexpected Catalog(s): NONE
Changed Catalog(s):
     LIQDEMO_NY
          name changed from 'LIQDEMO_NY' to 'LIQDEMO'
Missing Column(s):
     KUND_INFO_VY.ADDRESS
     BETALNING.BELOPP
<snip snip>
     KUND_ORDER_INFO.ORDER_ID
     KUND_ORDER_INFO.SKAPAD
     KUND.SKAPAD_TS
Unexpected Column(s):
     KUNDVY.ADDRESS
<snip snip>
     KUND_ORDER.SKAPAD
Changed Column(s): NONE
Missing Foreign Key(s): NONE
Unexpected Foreign Key(s): NONE
Changed Foreign Key(s):
     KUND_ORDER_FK_KUND(KUND_ORDER_INFO.KUND_ID -> KUND.KUND_ID)
          foreignKeyTable changed from 'KUND_ORDER_INFO' to 'KUND_ORDER'
```

DBDoc

Dokumentation över en databastabeller och även förändringar är ett vanligt krav i alla miljöer. Ofta gör man sådan dokumentation för hand och håller dokumentationen och förändringarna separat. En del sådant är ibland nödvändigt, men om man kunde vore en dokumentation baserat på databas och changeset önskvärt för att slippa manuellt dokumentera ändringar som ändå är implementerade i snyggt taggade filer.

Liquibase har en funktion för att kunna ge just en sådan dokumentation.

Om vi nu kör mot vårt vanliga schema (LIQDEMO) så kan vi få html-dokumentation som visar tabeller och förändringar på dem som är införda och som kommer att införas med den senaste changeloggen.

```
mkdir doc
liquibase dbDoc doc
```

Öppna nu index.html i bibioteket “doc”. Svenska tecken kan visas felaktigt då HTML inte är taggad för att visa att den är utf-8, så det kan man behöva ställa in för hand. (Forefox - meny uppe till höger -> character encoding -> Unicode)

Klicka på pending changes så finns de som inte är körda med där. Klicka på SQL för en ändring så visas SQL-skriptet med den ändringen högst upp på sidan.

Pending SQL Visar hela skriptet som kommer köras.

Change logs visar alla changelogs man har (i det här fallet bara en (demo.xml).

Klicka på current tables och på kund. Här visas vanlig dokumentation för en tabell, men även de ändringar som inte är gjorda på just den här tabellen liksom de som redan är införda av de som finns i changeloggen.

Det här är väldigt smidig dokumentation som visar mycket av det man vill se och som kanske begärs från en driftsavdelning som dokumentation inför en ny release.

Hur jobba med Liquibase - Variant 1 - SQL, SQLFile

För databasspecialister är den vanligaste reaktionen på att kika på Liquibase att det är suveränt man kan automatisera sina DDL-skript. Så man kör den DDL man är van vid.

Eventuellt delart man upp det så det blir en changelog per objekt så man i sitt versionshanteringsverktyg kan se när och hur ändringar infördes.

Det finns även ett SQL-format av changesets som kan användas så att man kan ha flera ändringar i en fil män ändå köra bara de senaste.

Exempel på changelog med flera changesets.

```
--liquibase formatted sql

--changeset mathias:1
create table kund (kund_id number(4)     not null
                  ,address varchar2(255)
                  ,skapad  date          not null)
    tablespace data
    partition by range (skapad)
      interval (numtodsinterval(1,'DAY'))
     (partition minval values less than (timstamp' 2010-01-01 00:00:00');
comment on table kund is 'Aktiva kunder';
comment on column kund.kund_id is 'Unik identifierare för en person';
comment on column kund.address is 'Adress till en person';
comment on column kund.skapad is 'Datum då kunden skapades';
--rollback drop table kund;

--changeset mathias:2
alter table kund add nasta_kundid number(4)  generated always as (kund_id + 1) virtual;
comment on column kund.nasta_kundid IS 'Ett högre än kundid';

--changeset mathias:3
create unique index kund_unq on kund(address) tablespace data;
```

Det är ju fantastiskt. På det viset kan man köra all sin vanliga DDL via liquibase och få ändringarna versionshanterade. Det finns t.o.m. stöd för rollback, valideringar, context mm. Så får man inte det bästa av båda världar här, Liquibase med versionshantering av ändringar och traditionell DDL att underhålla?

Nej, man förlorar väldigt mycket faktiskt. Allt automatiskt stöd för Rollback går förlorat, kunskap om vilka ändringar som påverkat vilken tabell förloras och kan inte rapporteras via dbDoc. Stöd för att köra samma i olika databaser finns inte mer än att man själv får skriva kod för varje sorts databas man vill stödja.

Stöd för olika DDL i olika miljöer blir också mycket svårt att få till.

Möjlighet att påverka DDL med properties blir också betydligt svårare. Att kunna göra en ändring av changeset, men behandla det som samma ändring som tidigare om den var införd förloras (Liquibase kollar checksum av changeset mot vad som registrerats för ändringen i databasen)

Det här gör Liquibase till en produkt som i princip bara kör DDL i en given ordning. Man väljer att inte använda 95% av produktens kapacitet och är det allt man vill så finns det bättre lösningar att använda.

Liquibase-formaterade changesets bör nog ses som en speciallösning och inte som standardmodellen om man vill kunna dra full nytta av Liquibase.

Eftersom alla produktionssättningar vanligen kommer med krav på att kunna rulla tillbaka vid problem så skall inte behovet att skriva och testa rollback i den här modellen underskattas. Det krävs mycket tid att skriva och testa egen rollback. Den finessen är en av de stora mervärdena med en produkt som Liquibase.

Hur jobba med Liquibase - Variant 2 - Fil per struktur

Om man då vill kunna använda alla finesserna med Liquibase alla taggar för olika sorters objekt, men ändå vill ha kontroll per objekt och inte bara ha en enda changeLog? Går det? Ja faktiskt finns det riktigt bra inbyggt stöd för det.

Genom att skapa bibliotek med objekt per typ så kan man få alla de objekten körda i tur och ordning. Det finns en tag - includeAll - med vilken man pekar på ett bibliotek och skripten där körs i bokstavsordning. Varje skript sköter då ett visst objekt. T ex så skapas en tabell och alla kolumnändringar ligger i samma fil i en changelog.

Om alla skript i demonstrationen byggs om till den här strukturen får man följande skript, de ligger i bibilioteket multi i liqdemo.

```
data
dbl
fk
ix
mv
mvl
pk
pkg
prv
seq
syn
tb
uk
vw
```

Alla namn/förkortningar är nog självförklarande utom eventuellt dessa:

- data -> Data som laddas till tabeller
- dbl -> Databaslänkar
- mv(l) -> Materialiserad vy (log)
- prv -> Privileges (grants)

De filer vi då får är dessa:

```
multi
|-- data
|   |-- demo26.csv
|   |-- demo27.csv
|   `-- kund_data.xml
|-- dbl
|   `-- dbl_dbl.xml
|-- fk
|   `-- kund_order_fk_kund_fk.xml
|-- ix
|   |-- betalning_unq_ix.xml
|   |-- kund_skapad_ix.xml
|   `-- kund_unq_ix.xml
|-- mv
|   `-- kundmv_mv.xml
|-- mvl
|   `-- kund_mvl.xml
|-- pk
|   |-- kund_order_pk.xml
|   `-- kund_pk.xml
|-- pkg
|   `-- output_pkg.xml
|-- prv
|   `-- multi_grant_prv.xml
|-- seq
|   `-- kund_seq.xml
|-- syn
|   `-- kundsyn_syn.xml
|-- tb
|   |-- betalning_tb.xml
|   |-- kund_order_info_tb.xml
|   `-- kund_tb.xml
|-- uk
|   `-- kund_order_uc_kund_id_uk.xml
`-- vw
    `-- kund_info_vy_vw.xml
```

Som listan ovan visar så finns tabellens skript i en fil för det i bibliotek tb. Men samma tabell har referenser i många andra bibliotek såsom ix, syn, fk, och uk.

Huvud-XML filen innehåller bara referenser till biblioteken i den ordning man vill köra dem. Se multi.xml i liquidemo. Det här gör att det väldigt lätt redan i versionshanteringssystemet att se när en ändring av ett visst objekt infördes.

Låt oss testa att lägga in det här i stället för den kronologiska fil vi körde i demonstrationen.

```
liquibase dropAll
sqlplus
   <log in>
   drop database link "DBL";
   drop materialized view "LIQDEMO"."KUNDMV";
   drop synonym "LIQDEMO"."KUNDSYN";
   exit
liquibase --changeLogFile=multi.xml update
```

Sådärja, det fungerar lika bra att införa exakt samma ändringar med skripten skapade på det viset.

Nackdelar då? Finns det inga? Jo, allt är en fråga om kompromiss och behov. Om vi inför en ny ändring i en tabell där en kolumn byter namn så har vi en omedelbar påverkan på alla skript i senare kataloger som refererar den kolumnen. Inget som är svårt eller omöjligt att hantera. Faktum är att det nog är så att man vill tvingas hantera sådant.

Varför händer det då, körs inte skripten i kronologisk ordning. Nej, Liquibase hanterar inte ordningen de är skapta i. Det som görs är att köra skripten i den ordning respektive fil har dem skrivna i. Det betyder att alla skript i `tb` körs före något skript i t ex `ix` körs. Så index skapade mot tabellen med kolumnen refererad som det ursprungliga namnet fungerar inte när ändringen av namn inträffat innan det skriptet körs.

I mitt tycke är det hanterbart för hur ofta görs ändringar som skapar problem? Mycket sällan. Men vill man hitta en variant som garanterar ordningen så kan man ha ett huvudskript som inkluderar en fil i taget och bara låta varje fil inkludera en enda ändring.

För att uppnå den effekten skulle man i bibliotek `tb` ha ett underbibliotek för varje tabell. Sen från masterfilen (motsvarande multi.xml) lägger man sist en referens per ny fil. Det gör att allt i `tb` inte körs före något körs i `ix`. Allt körs i den ordning det införs, men det är fortfarande ordning på var filerna är och hur man kan se när ändringar införs. Det här är kanske lite extremt för de flesta, men vill man ha ordning både på körordning och gruppera ändringar per objekt är det modellen.

Slutsats

Det finns många sätt att jobba på med liquibase och det är upp till var och en att hitta det som fungerar för den mängd komplexitet man behöver hantera. Min rekommendation är att ta steget bortom sql/sqlfile och anpassa sig till att tagga upp ändringarna med de färdiga funktioner som produkten stödjer. Utan det drar man inte full nytta av Liquibase.

Än viktigare är att sätta upp en testmiljö. Som de säger - lärande är inte en publiksport. För att förstå hur man själv kan använda det behöver man testa och utvärdera varje finess. T ex kan Liquibase byggas ut med metoder som hanterar något specifikt man behöver som inte ingår in vare sig Liquibase eller dess extensions. Det gör man genom att skriva lite Java-kod, eller be närmast vänligt sinnade java-utvecklare om lite hjälp.

Så kör och lek med Liquibase! Hör av er med häftiga finesser ni hittar som jag inte nämnt.

Appendix A - Changelog in JSON

```
{
  "databaseChangeLog":[{
      "changeSet": {
          "id": "Createx"
        ,"author": "Mathias"
        ,"changes": [{
         "createTable": {
              "tableName": "person"
            ,"tablespace": "DATA"
            ,"remarks": "Innehåller personer"
            ,"columns": [{
                "column": {
                     "name": "person_id"
                    ,"type": "number(4)"
                    ,"remarks": "Unik identifierare för en person"
                }}
                ,{"column": {
                      "name": "address",
                      "type": "varchar(255)",
                      "remarks": "Adress till en person"
                }}
              ]
           }
         }]
       ,"rollback":[{
            "dropTable": {
                "cascadeConstraints": true
              ,"tableName": "person"
            }
         }]
       }
     }
    ,{
      "changeSet": {
         "id": "Tag version 1",
         "author": "Mathias",
         "changes": [{
             "tagDatabase": {
                 "tag": "version_1.3"
             }
         }]
      }
    }
    ,{
      "changeSet": {
         "id": "Add kolumn kommentar",
         "author": "Mathias",
         "changes": [{
             "addColumn": {
                 "tableName": "person"
               ,"columns": [{
                    "column": {"name": "kommentar","type": "varchar(255)","remarks":
"Kommentar för personen"}
                }]
             }
         }]
      }
    }
    ,{
      "changeSet": {
         "id": "Drop table person",
```

```
        "author": "Mathias",
        "changes": [{
            "dropTable": {
                "tableName": "person"
               ,"cascadeConstraints": true
            }
        }]
      }
    }
  ]
}
```

Appendix B - Changelog in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
   xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
   xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
 http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.2.xsd
 http://www.liquibase.org/xml/ns/dbchangelog-ext
 http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.xsd">

  <property name="ts.betalning" value="data"/>

  <changeSet id="Create tabell kund" author="Mathias" runAlways="false" failOnError="true">
    <createTable tableName="kund"
                 tablespace="data"
                 remarks="Aktiva kunder">
      <column name="kund_id" type="number(4)"    remarks="Unik identifierare för en person">
<constraints nullable="false"/></column>
      <column name="address" type="varchar(255)" remarks="Adress till en person"           >
<constraints nullable="true" /></column>
      <column name="skapad"  type="date"         remarks="Datum då kunden skapades"        >
<constraints nullable="false"/></column>
    </createTable>
    <modifySql context="prod">
      <replace replace="data"
               with="data partition by range (skapad) interval (numtodsinterval(1,'DAY'))
                       (PARTITION MINVAL  VALUES LESS THAN (TIMESTAMP' 2010-01-01 00:00:00')
                         SEGMENT CREATION DEFERRED TABLESPACE data)
                       ENABLE ROW MOVEMENT" />
    </modifySql>
  </changeSet>
  <changeSet id="Add virt col nasta_kundid (kundid + 1)" author="Mathias" runAlways="false"
failOnError="true">
    <addColumn tableName="kund">
      <column name="nasta_kundid" type="number(4)" defaultValueComputed="generated always as
(kund_id + 1) virtual" remarks="Ett högre än kundid"/>
    </addColumn>
    <modifySql context="prod">
      <replace replace="DEFAULT"
               with=""/>
    </modifySql>
  </changeSet>
  <changeSet id="add unique ix kund_unq" author="Mathias" runAlways="false"
failOnError="true">
    <createIndex tableName="kund"
                 indexName="kund_unq"
                 unique="true"
                 tablespace="data">
      <column name="address"/>
    </createIndex>
  </changeSet>
  <changeSet id="add partitioned ix kund_skapad" author="Mathias" runAlways="false"
failOnError="true">
    <createIndex tableName="kund"
                 indexName="kund_skapad"
                 unique="true"
                 tablespace="data">
      <column name="skapad"/>
    </createIndex>
    <modifySql context="prod">
      <replace replace="TABLESPACE data"
```

```
              with="tablespace data local"/>
    </modifySql>
  </changeSet>
  <changeSet id="Create sequence kund_seq" author="Mathias" runAlways="false"
failOnError="true">
    <createSequence sequenceName="kund_seq"
                    startValue="1"
                   incrementBy="1"
                    minValue="1"
                    maxValue="10000"
                    ordered="true"
                    cycle="true"/>
  </changeSet>
  <changeSet id="Alter sequence kund_seq" author="Mathias" runAlways="false"
failOnError="true">
    <alterSequence sequenceName="kund_seq"
                  incrementBy="10"
                   maxValue="50000"
                   ordered="false"/>
  </changeSet>
  <changeSet id="Create depending table kund_order" author="Mathias" runAlways="false"
failOnError="true">
    <createTable tableName="kund_order"
                 tablespace="data"
                 remarks="Orders from kunder">
      <column name="kund_id"  type="number(4)" remarks="Unik identifierare för en person">
<constraints nullable="false"/></column>
      <column name="order_id" type="number(4)" remarks="Unik identifierare för en order" >
<constraints nullable="false"/></column>
      <column name="skapad"   type="date"      remarks="Datum då kunden skapades"        >
<constraints nullable="false"/></column>
    </createTable>
  </changeSet>
  <changeSet id="Add primary key for kund" author="Mathias" runAlways="false"
failOnError="true">
    <addPrimaryKey tableName="kund"
                   columnNames="kund_id"
                   constraintName="kund_pk"
                   tablespace="data"/>
  </changeSet>
  <changeSet id="Add primary key for kundorder" author="Mathias" runAlways="false"
failOnError="true">
    <addPrimaryKey tableName="kund_order"
                   columnNames="order_id"
                   constraintName="kund_order_pk"
                   tablespace="data"/>
  </changeSet>
  <changeSet id="Add foreign key for kundorder to kund" author="Mathias" runAlways="false"
failOnError="true">
    <addForeignKeyConstraint baseTableName="kund_order"
                             baseColumnNames="kund_id"
                             constraintName="kund_order_fk_kund"
                             deferrable="false"
                             onDelete="RESTRICT"
                             onUpdate="RESTRICT"
                             referencedTableName="kund"
                             referencedColumnNames="kund_id"/>
  </changeSet>
  <changeSet id="Add unique key for kundorder on kund_id" author="Mathias" runAlways="false"
failOnError="true">
    <addUniqueConstraint tableName="kund_order"
                         columnNames="kund_id"
                         constraintName="kund_order_uc_kund_id"
                         deferrable="false"
                         disabled="false"
                         tablespace="data"/>
```

```
  </changeSet>
  <changeSet id="Add nott null constraint on kund.skapad" author="Mathias" runAlways="false"
failOnError="true">
    <addNotNullConstraint tableName="kund"
                          columnName="address"
                          defaultNullValue="Ingen Adress"/>
  </changeSet>
  <changeSet id="Add view kundvy" author="Mathias" runAlways="false" failOnError="true">
    <createView viewName="kundvy"
                replaceIfExists="false">
      select a.kund_id
            ,a.address
            ,a.skapad       kund_skapad
            ,a.nasta_kundid
            ,b.order_id
            ,b.skapad       kund_order_skapad
       from       kund       a
      inner join kund_order b
         on a.kund_id = b.kund_id
    </createView>
  </changeSet>
  <changeSet id="Add synonym kundsyn" author="Mathias" runAlways="false" failOnError="true">
    <sql endDelimiter=";"
         splitStatements="true"
         stripComments="true">
         <comment>Ingen standard-tag för synonymer än</comment>
      create synonym kundsyn for kund;
    </sql>
    <rollback>
      drop synonym kundsyn;
    </rollback>
  </changeSet>
  <changeSet id="Add materialilized view log on kund" author="Mathias" runAlways="false"
failOnError="true">
    <sql endDelimiter=";"
         splitStatements="true"
         stripComments="true">
         <comment>Ingen standard-tag för loggar som stödjer materialiserade vyer</comment>
      create materialized view log on kund tablespace data with rowid including new values;
    </sql>
    <rollback>
      drop materialized view log on kund;
    </rollback>
  </changeSet>
  <changeSet id="Add materialilized view kundmv" author="Mathias" runAlways="false"
failOnError="true">
    <ext:createMaterializedView viewName="kundmv"
                                tablespace="data"
                                queryRewrite="true"
                                subquery="select count(*) from kund"/>
  </changeSet>
  <changeSet id="Add database link dbl" author="Mathias" runAlways="false" failOnError="true">
    <sql endDelimiter=";"
         splitStatements="true"
         stripComments="true">
         <comment>Ingen standard-tag för databaslänkar</comment>
      create database link dbl connect to dbuser identified by iqwaszx using 'otherdb';
    </sql>
    <rollback>
      drop database link dbl;
    </rollback>
  </changeSet>
  <changeSet id="Insert row into kund" author="Mathias" runAlways="false" failOnError="true">
    <insert tableName="kund">
      <column name="kund_id" valueNumeric="128"             />
      <column name="address" value="St Eriksgatan 117"      />
```

```
      <column name="skapad"  valueDate="2015-12-13 13:14:15"/>
    </insert>
    <rollback>
      <delete tableName="kund">
        <where>kund_id = 128</where>
      </delete>
    </rollback>
  </changeSet>
  <changeSet id="Load data into kund" author="Mathias" runAlways="false" failOnError="true">
    <loadData tableName="kund"
              encoding="UTF-8"
              file="single/demo26.csv"
              quotchar="'"
              separator=",">
      <column name="kund_id" type="numeric"/>
      <column name="address" type="string"/>
      <column name="skapad"  type="date"/>
    </loadData>
    <rollback>
      <delete tableName="kund">
        <where>kund_id between 200 and 202</where>
      </delete>
    </rollback>
  </changeSet>
  <changeSet id="Load and replace data into kund" author="Mathias" runAlways="false"
failOnError="true">
    <loadUpdateData tableName="kund"
                    primaryKey="kund_id"
                    encoding="UTF-8"
                    file="single/demo27.csv"
                    quotchar="'"
                    separator=",">
      <column name="kund_id" type="numeric"/>
      <column name="address" type="string"/>
      <column name="skapad"  type="date"/>
    </loadUpdateData>
    <rollback>
      <delete tableName="kund">
        <where>kund_id between 203 and 204</where>
      </delete>
      <loadUpdateData tableName="kund"
                      primaryKey="kund_id"
                      encoding="UTF-8"
                      file="single/demo26.csv"
                      quotchar="'"
                      separator=",">
        <column name="kund_id" type="numeric"/>
        <column name="address" type="string"/>
        <column name="skapad"  type="date"/>
      </loadUpdateData>
    </rollback>
  </changeSet>
  <changeSet id="Change datatype on skapad to timestamp" author="Mathias" runAlways="false"
failOnError="true">
    <modifyDataType tableName="kund_order"
                    columnName="skapad"
                    newDataType="timestamp(6)"/>
    <rollback>
      <modifyDataType tableName="kund_order"
                      columnName="skapad"
                      newDataType="date"/>
    </rollback>
  </changeSet>
  <changeSet id="Rename column kund.skapad" author="Mathias" runAlways="false"
failOnError="true">
    <renameColumn tableName="kund"
```

```
                oldColumnName="skapad"
                newColumnName="skapad_ts"
                remarks="Timestamp då kunden skapades"/>
  </changeSet>
  <changeSet id="Rename table kund_order" author="Mathias" runAlways="false"
failOnError="true">
    <renameTable oldTableName="kund_order"
                 newTableName="kund_order_info"/>
  </changeSet>
  <changeSet id="Rename view kundmv" author="Mathias" runAlways="false" failOnError="true">
    <renameView oldViewName="kundvy"
                newViewName="kund_info_vy"/>
  </changeSet>
  <changeSet id="Run multiple grants" author="Mathias" runAlways="false" failOnError="true">
    <sqlFile encoding="utf8"
             endDelimiter=";"
             path="single/demo32.sql"
             relativeToChangelogFile="true"
             splitStatements="true"
             stripComments="false"/>
    <rollback>
      <sqlFile encoding="utf8"
               endDelimiter=";"
               path="single/demo32.rbk"
               relativeToChangelogFile="true"
               splitStatements="true"
               stripComments="false"/>
    </rollback>
  </changeSet>
  <changeSet id="Stop running through the changelog" author="Mathias" runAlways="false"
failOnError="true">
    <update tableName="kund">
      <column name="address" value="St Eriksgatan 17" />
      <where>kund_id = 128</where>
    </update>
    <rollback>
      <update tableName="kund">
        <column name="address" value="St Eriksgatan 117" />
        <where>kund_id = 128</where>
      </update>
    </rollback>
  </changeSet>
  <changeSet id="Create table betalning with property for tablespace" author="Mathias"
runAlways="false" failOnError="true">
    <createTable tableName="betalning"
                 tablespace="${ts.betalning}"
                 remarks="Betalningar från kunder">
      <column name="kund_id"     type="number(4)"    remarks="Unik identifierare för en
person"> <constraints nullable="false"/></column>
      <column name="betal_datum" type="date"         remarks="Datum då kunden betalde"
> <constraints nullable="false"/></column>
      <column name="belopp"      type="number(5,2)"  remarks="Belopp som kunen betalade"
> <constraints nullable="false"/></column>
    </createTable>
  </changeSet>
  <changeSet id="Create index betalning.betalning_unq bara om tabell finns" author="Mathias"
runAlways="false" failOnError="true">
    <preConditions>
      <tableExists tableName="betalning"/>
    </preConditions>
    <comment>Index skapas bara om tabellen finns definierad.
             Dessutom är den här kommentaren med för att visa kommentering av change set.
             NOTERA: updateSQL gör inget med precondition, de kollas bara vid live exekvering.
    </comment>
    <createIndex tableName="betalning"
                 indexName="betalning_unq"
```

```
                unique="true"
                tablespace="${ts.betalning}">
      <column name="kund_id"/>
      <column name="betal_datum"/>
    </createIndex>
  </changeSet>
  <changeSet id="Create procedure output" author="Mathias" runAlways="false"
failOnError="true">
    <createProcedure>
      create or replace package output as
        procedure print_line;
      end output;
    </createProcedure>
    <createProcedure>
      create or replace package body output as
        procedure print_line as
        begin
          dbms_output.put_line(q'#It's alive#');
        end print_line;
      end output;
    </createProcedure>
    <rollback>
      <sql>drop package output;</sql>
    </rollback>
  </changeSet>
  </changeSet>
  <changeSet id="Import APEX application into test workspace" author="Mathias"
runAlways="false" failOnError="true">
    <preConditions onFail="MARK_RAN">
      <sqlCheck expectedResult="1">
        select count(*)
          from apex_workspaces
         where workspace = 'TEST';
      </sqlCheck>
    </preConditions>
    <executeCommand executable="sqlplus">
      <arg value="liqdemo/liqdemo@localhost:1521/o12pdb"/>
      <arg value="@single/demo37.sql"/>
    </executeCommand>
    <rollback>
    </rollback>
  </changeSet>
</databaseChangeLog>
```