Migration von Oracle zu PostgreSQL

… aus Sicht eines PostgreSQL-DBA

Das Projekt

- ELSTER, beim Bayerischen Landesamt für Steuern
  - **El**ektronische **St**eu**er**erklärung
  - Im Prinzip: der elektronische Briefkasten der Finanzämter
- Datenbanken bisher: Oracle; Umstellung auf PostgreSQL 2014/2015
- Aufgaben:
  - Planung, Installation, Konfiguration und Betrieb der DB-Infrastruktur
  - Beratung der Entwickler, Durchführung der Migration

PostgreSQL-Setup

* Mehrere Umgebungen, laut Planung bestehend aus:
    * PostgreSQL Master und mehrere Slaves via Streaming Replication (synchron!)
    * pgpool-II für Load-Balancing
    * redundante Switches, zwei Rechenzentren, dicke Hardware …
* Anwendungen (Java) in anderem Netz, connect via pgpool-II
    * bringen eigenen Pool mit

Richtige Tool-Wahl

* Oder: die drei besten Tipps für Tools zur Migration von Oracle nach Postgres:
* 1. Nehme **Ora2Pg:** http://ora2pg.darold.net/
* 2. Schreibe kein eigenes Tool, nehme **Ora2Pg**
* 3. Nehme nicht die CLI-Version Deiner Lieblings-GUI, nehme **Ora2Pg!**

* Wirklich!

Selbstgebastelte Tools

* An Sicherheit grenzender Wahrscheinlichkeit immer
  * schlechter
  * langsamer
  * weniger flexibel
  * machen mehr Ärger!

* Ausnahmen bestätigen die Regel

Andere vorhandene Tools …

* … wie die SQL Workbench
  * sind nicht primär zur Migration gemacht
  * GUI, relativ schlecht Skriptbar
  * Java auf dem DB-Server? Lieber nicht!
  * Bloated & langsam, eingeschränkt, …
* … mögen auf den ersten Blick funktionieren, machen aber meist viel mehr Ärger

Viele Ora2Pg Features

* Nicht perfekt, aber oft: Migration ohne Nachbearbeitung
* Kann sehr viel, bei uns in keinem Fall Nachbearbeitung o.ä. nötig
* Inkl. (B)LOBs
* Hohe Performance
  * Parallel lesen, schreiben, konfigurierbar
* Tipp: Wenn Oracle langsamer, in PG ausnahmsweise die Indexe VORHER anlegen

Trete die Entwickler!

- Bei ELSTER: verschiedene Teams entwickeln verschiedene Software
- Größtenteils extern => kaum Berührung mit Betrieb
- Aber: Betrieb sollte Migration durchführen
- Daher war sinnvoll:
  - Wünsche, Vorschläge, Bitten, Vorgaben machen
  - Java-Entwicklern Hilfe und Unterstützung bei PostgreSQL anbieten

Bitte alles am Stück!

* Entwickler liefern gerne 10 Skripte, nacheinander auszuführen
  * Und am besten mit 3 verschiedenen Config-Files
  * … die in jeder neuen Version überschrieben werden.
* Kleinigkeiten sind wichtig: Nutze *relative* Pfade
* Fehler nicht ignorieren, sondern Skript abbrechen!
* Logging: Alles sauber protokollieren

Mache Lasttest!

* Echte Lasttests mit passenden Datenmengen
* In Produktion(sähnlicher Umgebung)
* Entwickler testen gerne nur mit ein paar Dutzend Daten

* … ein Beispiel:

Der 3-Sekunden-Query

* Nach Migration aufgefallen: UUUPS, 150000 mal pro Tag läuft ein Query mit Dauer ~3 Sekunden.
* Total simpler Query:

```
UPDATE table1
SET
    date        = ?,
    somesthing = ?,
    some_other = ?,
    [... lange Liste ...]
WHERE
    id          = ?;
```

WTF?

Simpler Query mit WHERE auf Primary Key ist lahm?

Index ist da: PK

```
> \d table1

Table "public.table1"
         Column           |             Type              |        Modifiers
--------------------------+-------------------------------+--------------------------------
 id                       | bigint                        | not null

 [...]

Indexes:
    „id_pkey" PRIMARY KEY, btree (id)
```

Query Plan? Ist OK!

```
> EXPLAIN (ANALYSE, BUFFERS) UPDATE table1 SET date = now() where id = 1;
                       QUERY PLAN
----------------------------------------------------------
 Update on table1  (cost=0.43..1.65 rows=1 width=635)
                       (actual time=0.016..0.016 rows=0 loops=1)
   Buffers: shared hit=3
   ->  Index Scan using id_pkey on table1
          (cost=0.43..1.65 rows=1 width=635) (actual time=0.015..0.015 rows=0 loops=1)
         Index Cond: (id = 1)
         Buffers: shared hit=3
 Total runtime: 0.099 ms
(6 rows)
```

WTF?!?

Aber wir sehen in den Logs und Live: Query Lahm!

Des Rätsels Lösung ist Java …

```
final int[] parameterTypes = {
                // date1=?, something1=?, otherk=?, date2=?, something2=?, something3=?,
                Types.TIMESTAMP,
                Types.VARCHAR,
                Types.DECIMAL,
[...]
                // date2=?, date3=?, date4=?, something4=?, something5=?
                Types.TIMESTAMP,
                Types.TIMESTAMP,
                Types.TIMESTAMP,
                Types.BIGINT,
                Types.VARCHAR,

                // where ID=?
                Types.DECIMAL,
};
```

Oder in SQL:

```
> EXPLAIN (ANALYSE, BUFFERS) UPDATE table1
                               SET date = now()
                              WHERE ids = 1::NUMERIC;
                                          QUERY PLAN
-----------------------------------------------------------------------------------
 Update on teble1  (cost=0.00..219705.25 rows=29579 width=635) (actual
time=2789.282..2789.282 rows=0 loops=1)
   Buffers: shared hit=261639
   ->  Seq Scan on table1  (cost=0.00..219705.25 rows=29579 width=635)
                               (actual time=2789.277..2789.277 rows=0 loops=1)
         Filter: ((id)::numeric = 1::numeric)
         Rows Removed by Filter: 5902411
         Buffers: shared hit=261639
 Total runtime: 2789.378 ms
(7 rows)
```

Workaround

```
CREATE INDEX temp_performance__id_numeric_idx
          ON table1 ( CAST(id AS NUMERIC) );
```

- Anwendung kann nicht ohne weiteres schnell geändert und getauscht werden
- Also ein Workaround bis zum nächsten Release

Workaround Klappt!

```
> EXPLAIN (ANALYSE, BUFFERS) UPDATE table1 SET date = now()
                          WHERE puid_pk = 1::NUMERIC;
                                       QUERY PLAN
---------------------------------------------------------------------------
 Update on table1  (cost=297.55..16737.11 rows=29512 width=635)
                       (actual time=0.051..0.051 rows=0 loops=1)
   Buffers: shared hit=3
             ->  Bitmap Heap Scan on table1  (cost=297.55..16737.11 rows=29512 width=635)
                                        (actual time=0.047..0.047 rows=0 loops=1)
         Recheck Cond: ((id)::numeric = 1::numeric)
         Buffers: shared hit=3
         ->  Bitmap Index Scan on temp_performance__puid_pk_numeric_idx
                    (cost=0.00..290.18 rows=29512 width=0)
                    (actual time=0.039..0.039 rows=0 loops=1)
               Index Cond: ((puid_pk)::numeric = 1::numeric)
               Buffers: shared hit=3
 Total runtime: 0.185 ms
(9 rows)
```

Mache Performance-Tests mit Echtdaten

* Echtdaten oder anonymisierte Echtdaten
    * Datenmenge
    * Verteilung der Daten
    * Spezielle Häufungen von Queries und Daten

Migration verifizieren!

* Es kann immer etwas schief gehen
* Also: Verifizieren, dass die Daten korrekt sind
* Verschiedene Möglichkeiten
  * 1:1 Vergleich
  * Prüfsummen über alle Daten
  * manuelle Sichtprüfung
  * …

Viele Testdurchläufe machen

* Die Migration mit Echtdaten testen
  * Laufzeit, Probleme, …
  * Wissen und Übung für den Ablauf

* Dann geht auch bei der endgültigen Migration nichts schief …

Langsame Queries, langsame Anwendungen

* Viele Ursachen
* Oft das allgemein Übliche:
  * schlechte Indexe; zu viele (teilweise ungenutzt), zu wenige, die falschen
  * Gammel-Code, z.B. zu viele kleine Queries. ORM! Whuaaa!
  * Falsches Datenmodell
  * Schlecht verteilte Daten

„Auf meinem Notebook ist's aber schneller!!!"

* Typisch: Entwickler wundert sich: Anwendung in Test / Produktion lahm
* WTF!?!
* SQL-Full-Query-Log hilft:
  * tausende kleine Queries:
    * => Latenz ohne Ende über Netzwerk und mit pgpool!

Problem: pgpool und Load-Balancing

- Mit pgpool, Streaming Replication und Load-Balancing kann eine spätere Transaktion/Verbindung alte Daten sehen
- Auch bei synchroner Repliaktion
  - Anwendung schreibt Daten; COMMIT => synchron *received* auf Slave
  - Aber: nicht synchron *replayed!*
  - Nächster Prozess sieht u.U. *alte* Daten!
- Automatische Tests schlagen fehl => Load Balancing deaktiviert

Fazit: insgesamt problemlos

* Erste Anwendungen seit über einem Jahr, kompletter ELSTER (Betrieb Bayern) seit halbem Jahr migriert: alles stabil.
* Insgesamt lief die Umstellung weitgehend problemlos
* Hohe Performance
* Hohe Stabilität
* In unserem Fall: Dicke Hardware fackelt Query-Probleme ab …
* Hohe Lizenzkosten gespart.

Danke, noch Fragen?

* Alvar C.H. Freude

  http://alvar.a-blast.org/

  http://blog.alvar-freude.de/

  http://www.perl-blog.de/

  @alvar_f