PostgreSQL und memcached

Building a Query Cache

Björn Häuser

imos GmbH

11.11.2011 / PGconf.DE

Outline

1. Einführung
2. Hitting
3. Cache-Maintenance

Szenario

- Webapplikation
- Pro Request viele, größtenteils einfache, Queries

Einteilung von Caches

- Tradeoff zwischen
  - Maintenance
  - Komplexität der Anfragen
  - Dirty-Cache-Hit / Cache-Miss
- Transparenz
  - Beeinflussung des Programmierflusses

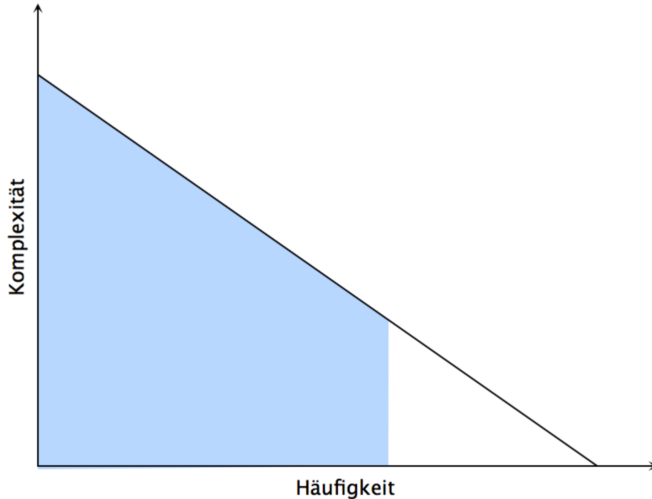
Zielsetzung

- Häufig auftretende Queries speichern
- Selbstständige Maintenance
- Cache-Miss anstatt Dirty-Cache-Hit
- Minimaler Einfluss auf Applikation
- Jederzeit zu- und abschaltbar

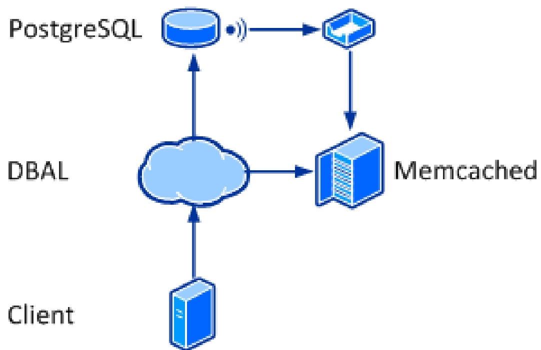
Cache

- In-Memory-Key-Value-Store
- Memcached

Komplexität vs. Häufigkeit



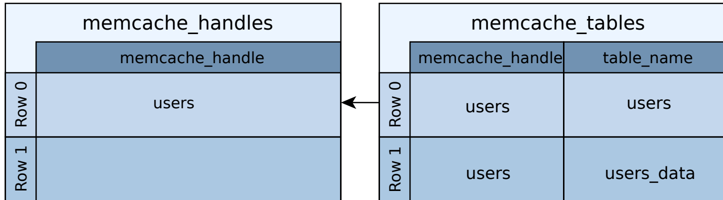
Aufbau



Grundlegende Arbeitsweise

- Tabellen werden in logische Gruppen unterteilt
- Jede Gruppe bekommt einen eindeutigen Bezeichner: *Handle*
  - Beziehung zwischen Tabellen und *Handle* wird gespeichert
- Query wird mindestens einem *Handle* zugeordnet

Schema



Aufbau der Schlüssel

- Ein Handle speichert die aktuelle Revision
  - `$handleRevision = "{$handle}_rev"`
- Key zusammengesetzt aus:
  - Query
  - Parameter
  - Revision des Handles

Beispiel

Query:

```
SELECT user_id, login, firstname, lastname
FROM users
WHERE user_id = ?
```

Handle:

```
users
```

Abfragen der Revision des Handles:

```
$handleRevision = memcache_get("{$handle}_rev");
```

Zusammenbauen des Keys:

```
$key = $handle . "_" . md5($query . implode(",",
  $params)) . "_rev_" . $handleRevision;
```

Ergebnisse

Memcache-Key:

```
"users_e5c18e041e700d90717a634fb9751207_rev_1"
```

Cache-Maintenance

- Bei Änderungen an Tabellen müssen die zugeordneten *Handles* inkrementiert werden
- Folgen:
    - Alle zwischengespeicherten Queries werden automatisch ungültig
    - Vorher: users_e5c18e041e700d90717a634fb9751207_rev_1
    - Nachher: users_e5c18e041e700d90717a634fb9751207_rev_2
- On-Commit-Event-Problematik

On-Commit-Event-Problematik

- Cache nur invalidieren wenn Transaktion *COMMIT*ed wurde
- Cache und Datenbank synchronisieren
  - Race-Conditions

Maintenance per Trigger

- Reagiert auf Veränderungen der Tabelle (INSERT, UPDATE und DELETE)
- Zugeordnete Handles auslesen und inkrementieren
- Vorteile:
  - Einfach umzusetzen
  - Nur Abhänigkeit auf pg_memcache
- Nachteile:
  - Reset auch bei *ROLLBACK*
  - Race-Conditions

Maintenance per Trigger

```
FOR p_memcached_handle IN
  SELECT mt.memcached_handle
  FROM memcached_tables mt
  WHERE mt.table_name = TG_TABLE_NAME
LOOP
  p_rev_name = p_memcached_handle.
      memcached_handle || '_rev';
  p_rev = memcache_get(p_rev_name);
  IF p_rev IS NOT NULL THEN
    PERFORM memcache_incr(p_rev_name);
  END IF;
END LOOP;
```

Maintenance per Trigger

- Race-Condition 1:
  - Transaktion A resettet *Handles*
  - Transaktion B hinterlegt alte Daten unter neuer Revision
  - Transaktion A *COMMIT*ed seine Daten
  - Folge: Alte Daten liegen im Cache
- Race-Condition 2:
  - Transaktion A resettet *Handles*
  - Transaktion B hinterlegt alte Daten unter neuer Revision
  - Transaktion A zerstört die Transaktion und *ROLLBACK*ed
  - Folge: Richtige Daten im Cache, aber unnötiger Cache-Miss

Maintenance per LISTEN / NOTIFY

- NOTIFY
  - Seit PostgreSQL 9.0 mit Payload
  - Wird erst beim *COMMIT* zugestellt
- Handler übernimmt das Inkrementieren der *Handles*
- Vorteile:
  - Keine Abhängigkeit in der Datenbank
  - Eine Nachricht kann mehreren Destinationen zugestellt werden
  - Gleiche Nachrichten werden zusammengefasst
- Nachteile:
  - Client muss entweder *POLL*en oder *SELECT*en
  - Kein Two-Phase-Commit möglich
- Poor-Mans-Queue?

Maintenance per LISTEN / NOFITY

NOTIFY (innerhalb eines Triggers):

```
EXECUTE pg_notify(
  TG_TABLE_SCHEMA,
  (SELECT string_agg(memcached_handle, ',')
   FROM memcached_tables
  WHERE table_name = TG_TABLE_NAME)
);
```

LISTEN:

http://rhodiumtoad.org.uk/junk/listen-min.pl.txt

Maintenance per LISTEN / NOFITY

- Wartezeiten bis *NOTIFY* die Destination erreicht
- Wenig Kontrolle über die Nachricht

Maintenance per Queue

- Eleganteste Lösung
  - *PgQ*
- Nachricht vs. Event transportieren
- Consumer übernimmt das Inkrementieren der *Handles*

Maintenance per Queue

- Vorteile:
  - Volle Kontrolle über Nachrichten
  - Eine Nachricht kann mehreren Destinationen zugestellt werden
  - *PgQ* übernimmt die Verteilung
- Nachteile:
  - Externer Dienst (Ticker)
- Links:
  - `http://www.pgcon.org/2008/schedule/events/79.en.html`
  - `http://wiki.postgresql.org/wiki/PGQ_Tutorial`

Allgemein

- Probleme
  - Verdecken von langsamen Queries
- Memcached
  - Monitoring
    - `http://code.google.com/p/memcached/wiki/NewProgrammingFAQ`
  - Tuning
    - `http://code.google.com/p/memcached/wiki/NewUserInternals`
  - *Expire* setzen

Summary

- Leicht integrierbare Lösung um Antwortzeiten zu verbessern
- Wenig Einfluss auf Programmierfluss
- Race-Conditions

- Zukunft
  - Race-Conditions minimieren
  - Cache wärmen / vorbefüllen

- Leave Feedback
  - https://www.postgresql.eu/events/feedback/pgconfde2011/