PostgreSQL & Performance: Eine Landvermessung

REDUX

Michael Renner

Metalab
22. September 2009

Vorstellung

- Michael Renner
- Sysadmin, DBA, Scrummaster
- Web-Affin
- Open Source
- Kein „gstudierter“
  - Das theoretische ist sehr praktisch

Abgrenzung: Das gibt’s nicht

- Keine Konzepte
  - … horizontale Skalierung
  - … „Cluster“
- Kein Locking
- Kein „Stored Procedures vs. Applikationslogik“

Abgrenzung: Das wird kommen

- Single-Node Performance
  - Hardware
  - Syscalls
- Antworten
  - Was macht das Postgres da?
  - Warum dauert das so lange?
  - Geht das nicht schneller?
- … und ein bisschen Theorie

Der Vortrag

- Performance?!
- … bei RDBMS
- … und PostgreSQL
- Theoretisches
- Praktisches
- Werkzeuge
- Neu in 8.4

Was heisst Performance?

- Leistung
- Arbeit pro Zeit
- Problematisch wenn
  - Einzelne Anfragen „zu lange“ brauchen
    - Ergibt sich aus Problemstellung. Vgl. Webapplikation vs. Datenanalyse
  - Pro Zeit mehr Anfragen ankommen, als abgearbeitet werden können
    - Queueing → Latenz → Alles Schrecklich

Ein paar Beispiele

- Ich weiß nicht wieso, aber die Datenbank ist plötzlich so langsam geworden!
- Seit wir die letzten 5 Besucher auf der Startseite anzeigen ist alles schrecklich
- Die aktuelle Werbeschaltung bringt uns um!

...bei RDBMS

- Komplexes Zusammenspiel von
  - Betriebssystem
    - und seinen (POSIX-)Ressourcen
      - Architektur, CPU, Bus...
      - „Storage“
      - Reifegrad des Betriebssystems
  - Datenbank
    - und seinen Eigenheiten
      - MVCC
  - Applikation
    - Und wie sie die Datenbank verwendet

...bei RDBMS

- Und ein paar hausgemachte Probleme
- ACID
  - Atomizität, Konsistenz, Isolation, Haltbarkeit
    - → Locking Fiasko
    - Für viele Problemstellungen nicht zwingend Notwendig

PostgreSQL kompakt

- Aus sicht des Betriebssystems
  - Shared Memory → shared_buffer → „Heap-Pages“
  - Prozesse
    - CPU → Tuples parsen, Results filtern, Joins, etc.
    - (work_)memory → Sort, hashes, etc.
  - Syscalls
    - Reads
      - Sequential & Random → Heap (Tables, Indizes, etc.)
    - Writes
      - Sequential & Synchron → WAL
      - Random Buffered → Page flushes auf Heap

PostgreSQL kompakt

- Direktes Mapping von Anfragen auf Systemressourcen
- Wenig Magie dazwischen
  - Ist Hardware & OS schnell ist PostgreSQL schnell
- Ein bisschen Magie
  - Synchronized Sequential Scan
  - Autovacuum
  - HOT
  - ...und mit jeder Version mehr

PostgreSQL kompakt

- Das bedeutet im Umkehrschluss:
- Wenns zu langsam ist
  - brauchts schnellere Hardware
  - weniger Fragen
  - oder „bessere“ Fragen!

Fragen an die Datenbank

- Oder
  - „Was passiert eigentlich bei einem Query?“

Select

- Ohne Index

```
root=# EXPLAIN SELECT * from "user" WHERE id='420815';
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on "user"  (cost=0.00..17907.38 rows=1 width=12)
   Filter: (id = 420815)
```

Select

- Mit Index

```
root=# EXPLAIN SELECT * from "user" WHERE id='420815';
                                 QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using user_id_key on "user"  (cost=0.00..8.38 rows=1 width=12)
   Index Cond: (id = 420815)
(2 rows)

root=# EXPLAIN SELECT * FROM "user" ORDER BY id DESC;
                                       QUERY PLAN
-----------------------------------------------------------------------
-----------------
 Index Scan Backward using user_id_key on "user"  (cost=0.00..31389.36
rows=1000000 width=12)
(1 row)
```

Join

- Nested Loop, Hash, Merge

```
root=# EXPLAIN SELECT u.id,u.name,p.product FROM "user" u, product p WHERE
u.id = p.userid;
                                QUERY PLAN
-------------------------------------------------------------------------
---
 Hash Join  (cost=31289.00..81660.00 rows=910000 width=26)
   Hash Cond: (u.id = p.userid)
   ->  Seq Scan on "user" u  (cost=0.00..20811.00 rows=1000000 width=14)
   ->  Hash  (cost=15470.00..15470.00 rows=910000 width=16)
         ->  Seq Scan on product p  (cost=0.00..15470.00 rows=910000
width=16)
(5 rows)
```

Änderungen

- Änderungen bedürfen besonderer Sorgfalt
- Bei PostrgreSQL
- MVCC
  - minimum Visible TxID → „ab wann sichtbar?“
    - insert/update
  - maximum Visible TxID → „bis wann sichtbar?“
    - update/delete
- Write Ahead Log

Insert

- Page mit ausreichend Platz im Heap finden
- Tuple schreiben
- Indizes aktualisieren → $n Page-Modifikationen
- WAL Einträge schreiben
- Bonuspunkte:
  - Foreign Keys
  - Sonstige Constraints
  - Triggers

Update

- Regulär
  - UPDATE == SELECT, INSERT, DELETE
    - Natürlich optimiert, keine einzelnen Queries
- Heap-only Tuples (HOT)
  - Wenn Platz in aktueller Page
  - Und Column nicht indiziert
  - Wird der neue Wert an bestehendes Tuple „dazugeheftet“

Aus dem Leben

- Was bedeutet das für die Praxis?
- Wie eingangs erwähnt:
- Performance in Datenbanksystemen ist ein sehr komplexes Thema
  - Man kann oft nur symptomatisch vorgehen
  - Je mehr Anhaltspunkte, desto besser
  - Die Lücken in der Beweislage durch Wissen, Raten und Messen füllen

Eine Annahme

- Wir haben ein Problem
- Dafür gibt's Schubladen
  - Zu viel CPU-Last
  - Zu hohe Query-Latenz
  - Zu viel Disk I/O

Effizienter Rechnen

- CPU-Bound
  - Komplexe Operationen raus aus der Datenbank
  - Zuviel Queries?
  - Sequential Scans auf große Tables im Page Cache
- Memory-Bound
  - Mit PostgreSQL nur schwer zu erreichen?
    - Auf „richtiger“ Hardware zumindest ;)
    - Verarbeiten von Tuples dauert immer länger als das Laden der Daten

Schneller Verbinden

- Latency-Bound
  - Wie lange brauchen Pakete zwischen Applikation und Datenbank?
  - Wie teuer ist ein Connectionaufbau?
    - Backend-Fork
    - SSL Handshake
- → Connection Pooling

I/O-Bound: Besser Lesen

- Grundsätzlich
  - Disk I/O ist der Untergang von Datenbanken
    - Page Fetch von RAM: Im µs Bereich
    - Page Fetch von SSD: 200 - 500 µs
    - Page Fetch von Festplatte: 4-15 ms
    - Quiz: Wieviel Pages braucht dein Query?
  - Kompaktere Daten → Datenmodell → Greg Stark „How long is a string?“
  - Indizes kontrollieren:
    - Mehr/Weniger/Kompakter
    - Index Slack → Reindizieren

I/O-Bound: Besser Schreiben

- Reads
  - Mehr RAM
    - Working Set sollte abgedeckt sein → Keine Reads im Normalbetrieb
- Writes
  - Write Cache für RAID-Controller
    - Und bitte Battery Backed!

Fallbeispiele soup.io

- „Microblogging-Plattform“
  - user
    - friends
    - stalkers
    - friends of friends
  - blogs
    - posts
      - attributes

Das Problem mit Indizes

- Nahezu überall
  - Zu viel
    - Besonders bei multicolumn indexes
  - Zu wenig
    - Merkt man (durch Statement-Log-Analyse) sehr schnell
  - Zu groß
    - Index Bloat? Besonders nach „Tablearbeiten“
  - Zu kleines statistics_target
    - „Ein sequential Scan ist da sicher schneller!“
    - Siehe „Zu wenig“

Speichergesättigte Architektur

- Working Set?!
  - Wenig Analysemöglichkeiten
    - → 8.5! (zumindest für Queries)
  - PostgreSQL kann nur shared_buffers überwachen
  - Page Cache → Betriebssystem
- Mehr RAM → Weniger IO
- „If it reads, it has too few RAM“

Putzneurosen

- Ein Table mit einigen hundert Rows wird relativ oft UPDATEd
- Live Tuples: 300. Dead Tuples: 5000. Relation Size: ein paar hundert MB
  - Autovacuum sagt **PANIK!**
  - ...und Vacuumed
    - Und das sehr oft, damit nix sein kann.
- HOT & Heap Fill-Factor to the rescue

Der Table. Unendliche Weiten.

```
soup_production=# EXPLAIN SELECT * FROM posts WHERE posts.blog_id = 16665
ORDER BY posts.created_at DESC, posts.id DESC LIMIT 40;

------------------------------------------------------------------------
 Limit  (cost=0.00..2906.70 rows=40 width=16)
   ->  Index Scan Backward using posts_for_one_soup on posts
(cost=0.00..176436.71 rows=2428 width=16)
         Index Cond: (blog_id = 16665)
 Total runtime: 83447.289 ms
```

Ein Nachtrag zum Thema Disk I/O

- Analyse von Storage Performance (unter Linux) ist ein einziges Tal der Finsternis
  - Latenz
  - Durchsatz
  - Parallelität
  - Queueing & Reordering
- ...und es gibt keine sinnvollen Werkzeuge
- ...und noch weniger Praxiserfahrung
- Aber SSDs werden das alles richten...

Werkzeuge

- Schätzungen und Prognosen bedürfen Vergleichswerten
- Je mehr desto besser
- Je länger desto besser
- Aber: Sie müssen korrelierbar sein

Betriebssystem

- Munin, Cacti, Nagios Perfdata, Zabbix, etc.
  - CPU
  - Memory
  - Interrupts
  - Disk Stats
    - IOs
    - Latency
    - Utilization
  - → Greg Smith „Database Hardware Benchmarking“

Datenbank

- Query Stats → pgfouine
- PostgreSQL statistics
  - pg_stat(io)_user_(indexes|tables)
  - pg_stat_bgwriter
  - pg_stat_...
- Keine hübschen Werkzeuge
  - („Can I haz MySQL Enterprise Monitor plz?“)

Applikation(en)

- Je mehr Messdaten desto besser
- Wenn Kapazitätsplanung ein Thema ist, überall Probes einbauen
- Bei wachsenden Applikationen wird jedes Feature zum Problem, die Frage ist nur „Wann?“
  - Ansteigende Anfragen
  - Ansteigendes Working Set
  - Bugs
  - „Features“

Neu in 8.4

- Bei jedem Major Release gibt es immer ein paar Performance-relevante Änderungen
  - Meistens in Richtung „schneller“

fadvise & AIO

- Prefetching bei Index-Scans
- Asynchrone Kernel-Threads übernehmen Sammeln der Daten
- Vorher:
  - read, *wait*, read, *wait*, read, *wait,* done
- Jetzt:
  - fadvise, **Kernel startet background worker,**
  - read, *wait*, read, read, done

Free-Space & Visibility Map

- Free Space Map (FSM)
  - Früher: Statisch konfigurierter Wert
    - Zu klein → Datenbank vergisst „freien Platz“ → Heap wächst
    - Zu groß → Verschwendeter RAM
  - Jetzt: Relation Fork
- Visibility Map
  - (Auto-)Vacuum kann gezielt Pages besuchen die verwertbaren Platz beinhalten → Deutlich weniger „überflüssiger“ I/O

Parallel Restore

- Beschleunigt Restore durch Verwendung von mehreren Cores
- Parallelisiert Befüllen von Tables und Erzeugen der Indizes

Danke!

Fragen?

michael.renner@amd.co.at
#postgresql(-de), Freenode, „Robe“