Performante Datenbankanwendungen

Oliver Glier, Dezember 2017

Latenz/Zugriffszeit Speicherhierarchie

Internet (Europa)	10^{-2} s
Processing-Delay in Switch/Router	10^{-4} s
Festplatte, SSD	10^{-3} s, 10^{-5} s
RAM-Synchronisation (verschiedene Cores)	10^{-7} s
RAM	10^{-8} s
Prozessor-Register	10^{-9} s

Speicherhierarchie und Datenbank

- DB: Dauerhaftes Speichern betrifft langsamste Teile des Systems

  => Applikation muss Transaktionen („Arbeitspakete“) passend dimensionieren

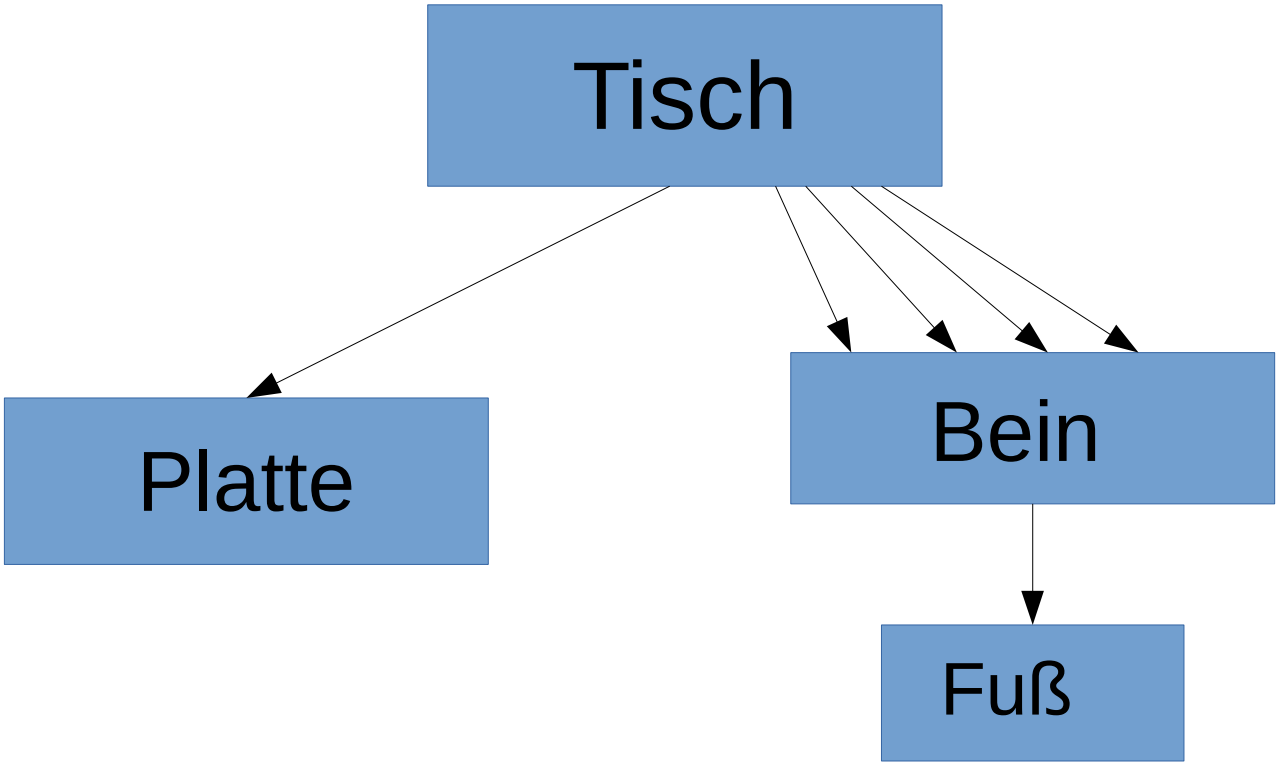
- Zu lang: Konfliktgefahr
- Zu klein: Applikation muss selbst koordinieren
- Kommunikationslatenz DB und Applikation beachten

Applikationsentwurf: Experimentelle Methode

1. Modell erstellen
2. Hypothesen aufstellen
3. Messen und validieren

- *Regelmäßig wiederholen!*

Beispiel: Komponentenbibliothek für 3D-Druck-Modelle



Fragestellung

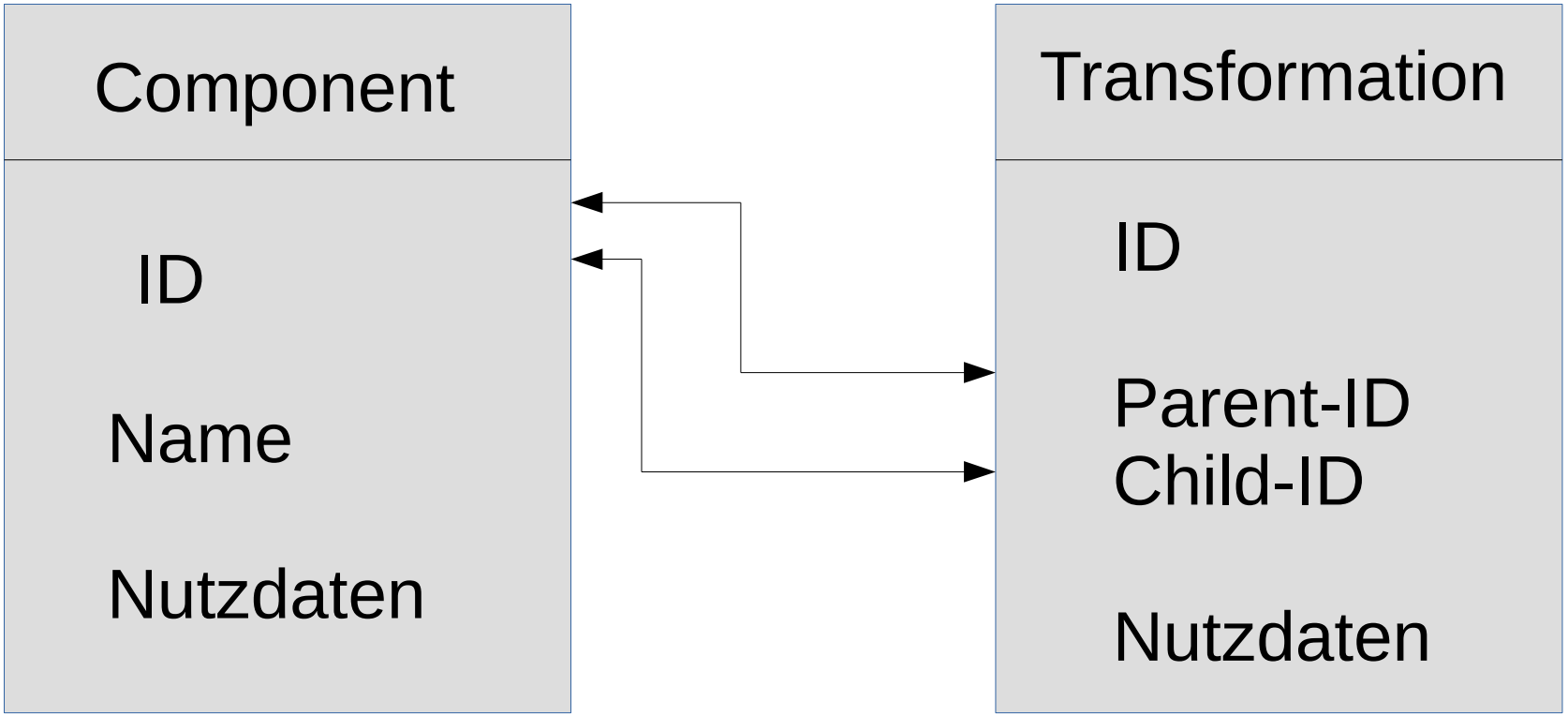
Gegeben: Komponente (Tisch)

Wie lesen wir möglichst schnell alle erreichbaren Unterkomponenten (Platte, Bein, Fuß) aus der DB?

Datenstruktur

- Azyklischer, gerichteter Graph
- Komponenten als Knoten
- Kante (Pfeil) von Eltern- zu Kindkomponente
- Mehrfach-Kanten erlaubt (warum?)
- Kantenbeschriftung: Transformation (Rotation, Translation) der Kindkomponente

Entity-Model



Primary-Key-Indizes: ID (Component, Transformation)
Index auf Parent-ID (Transformation)

Erreichbarkeit

Eine Komponente y ist von x erreichbar, wenn es eine Folge von Transformationen t1,…,t(n) gibt mit

- t1.parent = x,
- t(n).child = y und
- t(i).child = t(i+1).parent für i=1..n-1.

Komponentenabfrage

- Gegeben: Komponenten-ID x
- Aufgabe: Lies

  a) alle erreichbaren Komponenten y.

  b) alle zugehörigen Transformationen.

  Vorschläge?

Strategie 1: Erreichbarkeit in „Subpart“-Tabelle pflegen

- Bei neuer Komponente y:

  Paar (x,y) in Subpart-Tabelle speichern für alle x, welche y erreichen

  Paar (y,z) in Subpart-Tabelle speichern für alle von x erreichbaren z

Strategie 2: Traversierung in Applikation
(Java/Pseudocode)

```
void loadRecursive(Session session, long componentId,
    HashMap<Long, Component> componentsById,
    HashSet<Transformation> transformations) {

    if (componentsById.containsKey(componentId))
        Return

    Component c = session.load(Component.class, componentId)

    componentsById.put(componentId, c)

    forall (Transformation t with t.parent_id = c.id) {
         transformations.add(t)
         loadRecursive(session, t.getChildComponentId(),
        ComponentsById, transformations);
      }
}
```

Strategie 3: SQL-Rekursion

Lies Komponente mit ID=X und erreichbare Unterkomponenten:

```
WITH RECURSIVE parts(id) AS (

    SELECT id FROM component where id = X
    UNION
      SELECT t.child_id
      FROM transformation t, parts p WHERE
        p.id = t.parent_id
)
SELECT * FROM component c WHERE EXISTS (
  SELECT * FROM parts p WHERE p.id = c.id)
```

(für Transformationen analog)

Hypothesen

- Erwartete Laufzeit (bspw. relativ zueinander)?

Laufzeit

- Asymptotisch sind alle 3 Strategien gleich:

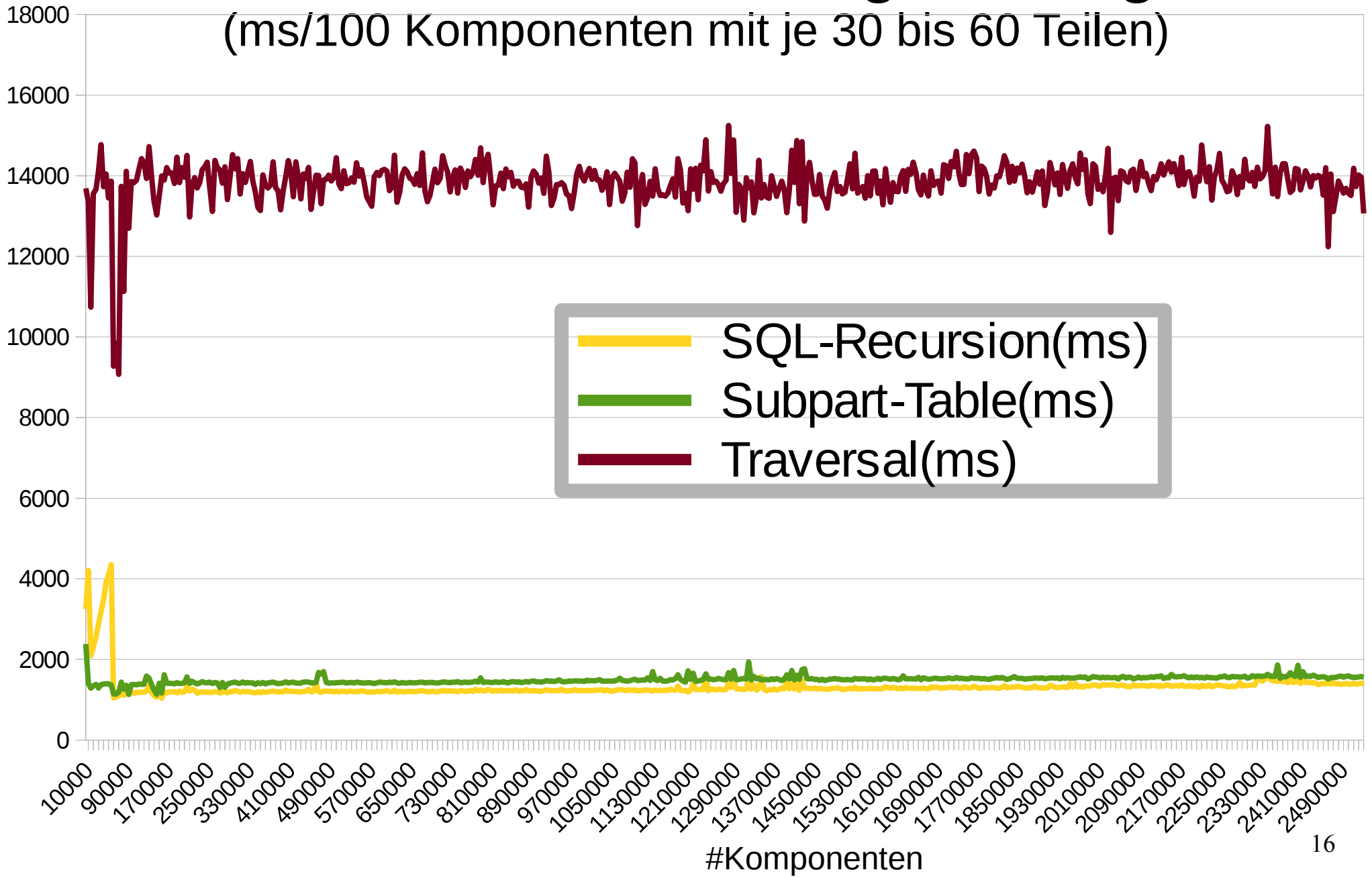
  O(n log m)

  (n=Größe des gelesenen Komponentengraphen, m=Größe der DB)

- „Tatsächliche“ Laufzeit: siehe Messung

  (Java, Hibernate und Postgres)

Performance der Abfragestrategien



Beobachtungen (Beispiele)

- Der Zeitbedarf der SQL-Rekursion sinkt stark nach wenigen Durchläufen.
- Der Anstieg (O(log m)) ist kaum wahrnehmbar.

Validierung

- SQL-Rekursion und Extra-Subparttabelle sind am schnellsten.
- Extra-Subparttabelle ist sehr fehleranfällig. (warum?)

Ende des Vortrags

Seien Sie experimentierfreudig,

vielen Dank!

Repository und Folien:

git clone https://github.com/halori/Parts.git

freie Lizent (MIT-Lizenz)

Readme mit kurzer Anleitung

Übungsvorschläge:

- Messen Sie die zusätzliche Zeit, welche das Pflegen der Subpart-Tabelle (Erreichbarkeitsrelation) bei Updates benötigt, sowie den relativen Speicherzuwachs.
- Entfernen Sie bei der Subpart-Tabelle die ID-Spalte (Sie können ggf. die beiden verbliebenen Komponenten-ID-Spalten als Composite-ID/Primary-Key verwenden). Warum geht das hier und bei der Transformations-Tabelle nicht?
- Ergänzen Sie Nutzdaten und messen Sie das Laufzeitverhalten erneut. Bspw. könnten Sie jeder Komponente beliebig viele geometrische Grundformen zuordnen.
- Welchen Effekt hat Hibernate-Caching (für die Messungen wurde es deaktiviert)?

Viel Spaß!

Ergänzung: Weitere Ideen

- Sie erwarten mehrere gleichzeitige Benutzer. Welche weiteren Experimente würden Sie durchführen?
- Welche zusätzlichen Probleme entstehen?
- Sie möchten den Komponentengraphen aller Komponenten im Applikations-Hauptspeicher halten. Wie verändert sich die Aufgabenteilung von DB und Applikation?

Ergänzung: Typische Tradeoffs

- Lese- vs. Update-Geschwindigkeit
- Applikationscode vs. Datenbank
- Isolation (DB vs. Applikation vs. tolerierbare Anomalien)
- Skalierbarkeit
- Verfügbarkeit
- Fehlertoleranz

Rahmen für Tradeoffs

- Anforderungen
- Verfügbare Technologie
- Beherrschbare Technologie, Wartbarkeit
- Physikalische Schranken (bspw. Lichtgeschwindigkeit)
- Logische Schranken, Berechenbarkeit
- Ressourcenbedarf (Zeit, Speicher, Kommunikation)
- Aktueller Forschungsstand

Beispiel: Anforderung

	Latenz/Dauer	Serielle Isolation erforderlich
Komponentenupdate	< 3s	ja
Komponente für Katalogansicht auslesen	< 1s	nein
Komponente für Druck auslesen	< 5s	ja
Abgleich der Statistik-Datenbank, nur lesend	Minuten bis Stunden	eingeschränkt