Saarbrücken, 25.06.2015
Information Systems Group

Vorlesung „Informationssysteme“

Vertiefung Kapitel 8: Transaktionen und wann sie gebraucht werden

Erik Buchmann (buchmann@cs.uni-saarland.de)

Foto: M. Strauch

Aus den Videos wissen Sie...

- ...dass es Transaktionen gibt, und wozu sie gut sind
  - ACID-Prinzip
- …dass man Transaktionen „tunen“ kann
  - Je nach Anwendung Verzicht auf *manche* Features

- Vertiefung heute:
  - Transaktionale Features in Postgres
  - Bulk-Insert: Wie man Transaktionen und Sicherheitsfeatures richtig nutzt

:= Postgres-spezifisch, d.h., kein Standard-SQL

Transaktionen in Postgres

Foto: M. Strauch

Motivation

1) Sie haben eine Datenbank erstellt

2) Sie haben ein ordentliches, normalisiertes Datenbankschema

3) Sie starten den Import von 1 TB Daten mit 500.000.000 Zeilen
→ Nach zwei Tagen brechen Sie ab.

- *Jetzt ist der richtige Zeitpunkt, sich darüber Gedanken zu machen, welche der vielen Features eines DBMS Sie benötigen ;-)*

ACID

- Aus den Video-Vorlesungen
  - **A**: Atomicity
    - Transaktionen werden entweder ganz oder gar nicht durchgeführt
  - **C**: Consistency
    - Constraints sorgen dafür, dass DB nach jeder Transaktion konsistent ist
    - Primary Key, Foreign Keys, Unique, CHECK-Constraints etc.
  - **I**: Isolation
    - Nur Daten aus abgeschlossenen Transaktionen für andere sichtbar
    - Ergebnis paralleler Transaktionen entspricht serieller Ausführung
  - **D**: Durability
    - Logging, Recovery etc. sorgt dafür, dass abgeschlossene Transaktionen „dauerhaft“ auf der Festplatte gespeichert sind
    - dies trotz Festplattencache, Betriebssystemcache, Deferred-Write-Strategien...

Atomicity

- Transaktionen entweder ganz oder gar nicht durchführen
- Standardverhalten der SQL-Konsole psql, pgadmin: Autocommit
  - jedes einzelne SQL-Statement eigene Transaktion
  - nach Abschluss der Ausführung eines Statements ist Transaktion abgeschlossen
  - Mit Autocommit zwei verschiedene Transaktionen

```
INSERT INTO kauft VALUES ('Erik', 'Mausfutter');
INSERT INTO forderungen_aus_lieferungen VALUES ('Erik', '12,78');
```

    - Beide INSERTs können unabhängig voneinander fehlschlagen
      - Offene Forderung an Kunden ohne Einkauf?
      - Einkauf ohne zugehörige Forderung an Kunden?

Wenn Autocommit unerwünscht ist

PostgreSQL

- Entweder abschalten
  - in psql
    \set AUTOCOMMIT off
- Oder Transaktionen explizit beginnen und beenden
  - eine Transaktion:

```
BEGIN TRANSACTION;
INSERT INTO kauft VALUES ('Erik', 'Mausfutter');
INSERT INTO forderungen_aus_lieferungen VALUES ('Erik', '12,78');
COMMIT;
```

  - Transaktionen abschließen
    - **COMMIT**; Transaktion erfolgreich beenden
    - **ROLLBACK**; Transaktion abbrechen, Ursprungszustand wiederherstellen
      → Wenn ein SQL-Statement in einer Transaktion fehlschlägt, erzwingt DBMS ein ROLLBACK für alle SQL-Statements

Consistency (Integritätssicherung)

- DBMS stelt sicher, dass Daten konsistent sind
  - ROLLBACK beim Eintragen ungültiger Daten
- Relationenschema + lokale Integritätsbedingungen
  - „Bestellung“ in *Bestellung* und „Artikel“ in *Artikel* taucht nur einmal auf
  - „Menge“ in *Warenkorb* und „Preis“ in *Artikel* ist >0
- Datenbankschema + globale Integritätsbedingungen
  - zu jeder „Bestellung“ in *Bestellung* ein/mehrere Einträge in *Warenkorb*
  - zu jeder „Bestellung“ in *Warenkorb* existiert exakt ein Eintrag in *Bestellung*

Bestellung

Kunde	Bestellung	Datum
Erik	001	15.03.
Jens	002	03.04.
Jens	003	06.04.
Jens	004	10.04.
Erik	005	20.04.

Warenkorb

Bestellung	Artikel	Menge
001	Trocken	2
001	Naß	2
002	Naß	10
003	Trocken	5
003	Rennmaus	3

Artikel

Artikel	Preis
Trocken	5,76
Naß	6,23
Rennmaus	16,99

Beispiel: Überkreuzbeziehung

Wie Daten einfügen?

```
CREATE TABLE kunden(
    id              int             NOT NULL,
    name            varchar(200)    NOT NULL,
    prim_adr        int             NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE adressen(
    id              int             NOT NULL,
    kunde_id        int             NOT NULL,
    strasse         varchar(200),
    plz             int,
    ort             varchar(200),
    PRIMARY KEY (id),
    FOREIGN KEY (kunde_id)   REFERENCES kunden(id);
);

ALTER TABLE kunden ADD CONSTRAINT fk_adr
    FOREIGN KEY (prim_adr)   REFERENCES adressen (id);
```

Der Zeitpunkt macht den Unterschied

- Standard: Constraints werden bei SQL-Ausführung sofort geprüft
  - Beim Einfügen in kunden:

    BEGIN TRANSACTION;
    INSERT INTO kunden VALUES (1, 'Testkunde', 1);

    ```
    FEHLER:  Einfügen oder Aktualisieren in Tabelle
    „kunden" verletzt Fremdschlüssel-Constraint „fk_adr"
    DETAIL:  Schlüssel (prim_adr)=(1) ist nicht in Tabelle
    „adressen" vorhanden.
    ```

    → *Transaktion wird sofort abgebrochen, ROLLBACK*

- Hier wünschenswertes Verhalten
  - Prüfung der referentiellen Integrität **am Ende** der Transaktion

Zeitpunkt der Prüfung in SQL

- Setzen des Zeitpunkts der Constraint-Prüfung mit
  - CREATE (TABLE | CONSTRAINT | TRIGGER)
    column REFERENCES reftable (refcolumn) [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]
  - ALTER (CONSTRAINT | TRIGGER) constraint_name [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

- Drei Möglichkeiten
  - Prüfung immer sofort, Anwender darf nicht eingreifen:
    NOT DEFERRABLE
  - Prüfung sofort, darf aber pro Transaktion verzögert werden:
    DEFERRABLE INITIALLY IMMEDIATE
  - Prüfung immer am Ende der Transaktion, darf aber vorgezogen werden:
    DEFERRABLE INITIALLY DEFERRED

Beispiele

- NOT DEFERRABLE
  ALTER TABLE kunden ADD CONSTRAINT u_name
  UNIQUE (name) NOT DEFERRABLE;

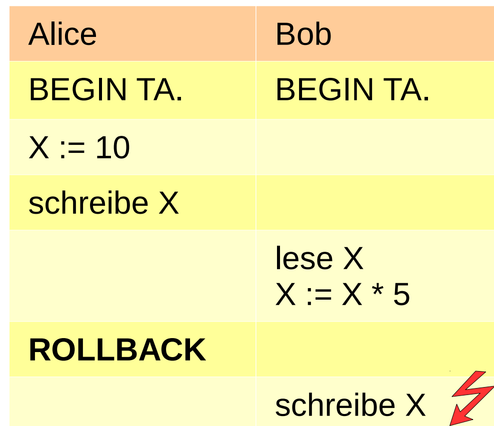
- DEFERRABLE INITIALLY IMMEDIATE
  ALTER TABLE adressen ADD CONSTRAINT fk_knd
  FOREIGN KEY (kunde_id) REFERENCES kunden (id)
  DEFERRABLE INITIALLY IMMEDIATE;
  - verzögern innerhalb einer Transaktion mit
    SET CONSTRAINTS fk_knd DEFERRED;

- DEFERRABLE INITIALLY DEFERRED
  ALTER TABLE kunden ADD CONSTRAINT fk_adr
  FOREIGN KEY (prim_adr) REFERENCES adressen (id)
  DEFERRABLE INITIALLY DEFERRED;
  - sofort prüfen innerhalb einer Transaktion mit
    SET CONSTRAINTS fk_knd IMMEDIATE;

Isolation (1/2)

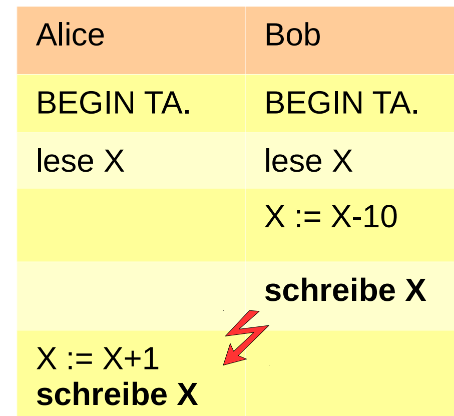
- Synchronisationskomponente stellt sicher, dass sich parallele Transaktionen nicht beeinflussen, sonst:

- **Dirty Reads**
  - Daten nicht abgeschlossener Transaktionen werden von anderen gesehen



Alice	Bob
BEGIN TA.	BEGIN TA.
X := 10	
schreibe X	
	lese X X := X * 5
ROLLBACK	
	schreibe X

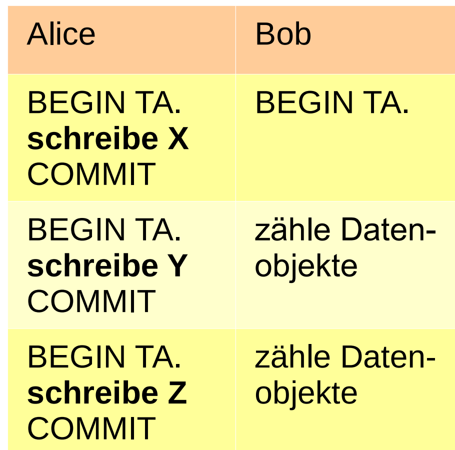
- **Lost Updates**
  - Zwei Transaktionen ändern den selben Datensatz, nur Ergebnisse der letzten Transaktion sichtbar



Alice	Bob
BEGIN TA.	BEGIN TA.
lese X	lese X
	X := X-10
	schreibe X
X := X+1 **schreibe X**	

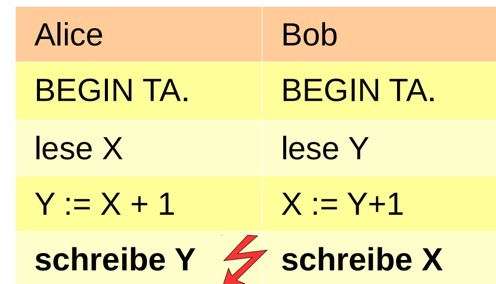
Isolation (2/2)

- **Non-Repeatable Read**
  - Wiederholte Lesevorgänge innerhalb einer Transaktion liefern unterschiedliche Ergebnisse



Alice	Bob
BEGIN TA. **schreibe X** COMMIT	BEGIN TA.
BEGIN TA. **schreibe Y** COMMIT	zähle Daten- objekte
BEGIN TA. **schreibe Z** COMMIT	zähle Daten- objekte

- **Phantom Read**
  - Lese-Schreib-Vorgänge paralleler Transaktionen beeinflussen sich gegenseitig

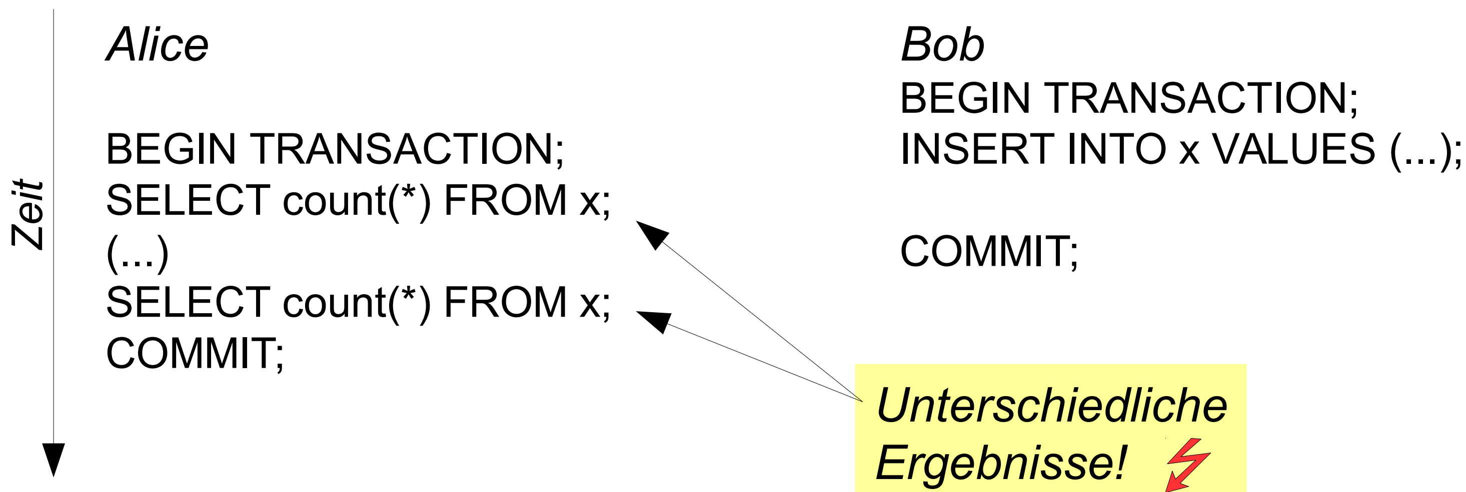


Alice	Bob
BEGIN TA.	BEGIN TA.
lese X	lese Y
Y := X + 1	X := Y+1
schreibe Y	**schreibe X**

Die vier Isolation Levels von SQL (1/3)

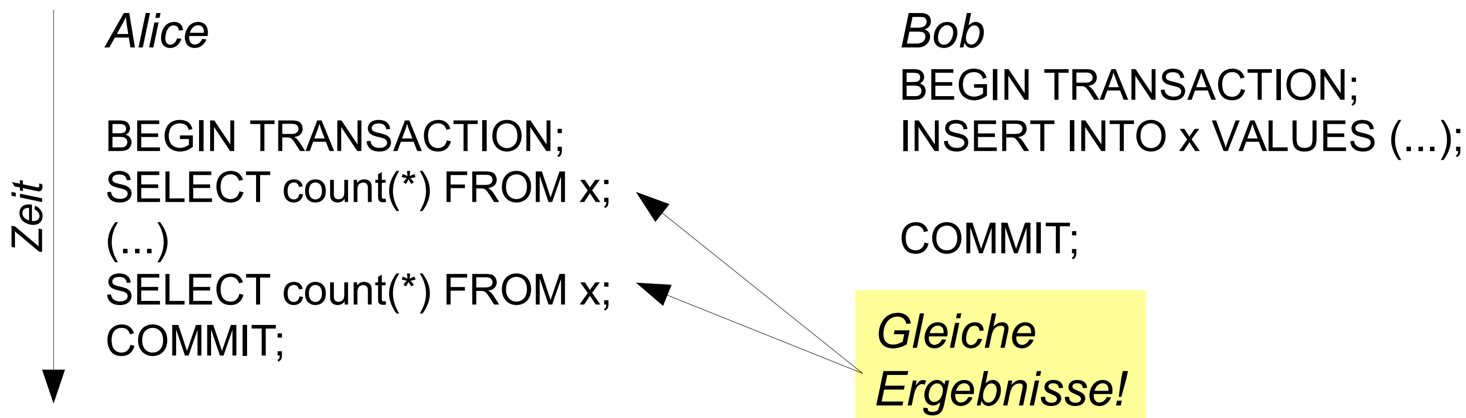
- Read Uncommitted
  - keinerlei Isolation, Anfragen sehen alles was derzeit in der DB passiert z.B. für: Single-User-Szenarien, Read-Only Anwendungen
  - nicht von Postgres unterstützt

- Read Committed (Default-Einstellung in Postgres)
  - Anfragen sehen alles, was andere Transaktionen committed haben



Die vier Isolation Levels von SQL (2/3)

- Repeatable Read
  - Anfragen sehen nur Daten, die zu Beginn der Transaktion gültig waren



  - Updates aus untersch. Transaktionen dürfen sich nicht überschreiben
    → Transaktionen können wegen anderer Transaktionen fehlschlagen
    `ERROR:  could not serialize access due to concurrent update`
    - Anwendung muss abgebrochene Transaktionen wiederholen können
    - Neues BEGIN TRANSACTION, neuer Datenbankzustand

Die vier Isolation Levels von SQL (3/3)

- Serializable
  - Anfrageausführung entspricht einer strikt seriellen Abarbeitung
  - Auch Anfragen mit nachfolgenden Schreiboperationen berücksichtigt

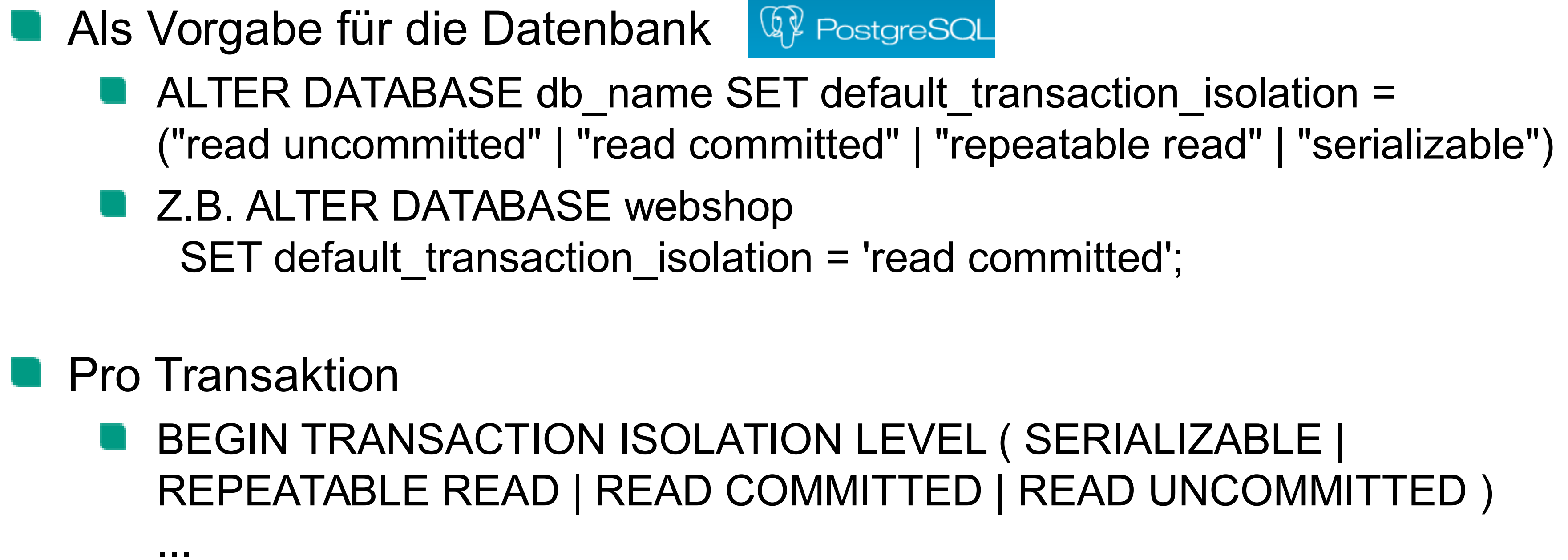
Alice	*Bob*
BEGIN TRANSACTION;	BEGIN TRANSACTION;
SELECT sum(i) FROM x WHERE t<5;	SELECT sum(i) FROM x WHERE t>5;
…	…
*berechne q = sum(i)*5*	*berechne q = sum(i)-50*
…	…
INSERT INTO x(t, i) VALUES (10,q);	INSERT INTO x(t, i) VALUES (1,q);
COMMIT;	COMMIT;

Zeit

```
ERROR:  could not serialize access due to read/write dependencies
among transactions
```

Keine gültige serielle Ausführung bei gleichzeitigem Start beider Transaktionen. ROLLBACK für eine TA

Setzen des Isolation Levels



- Als Vorgabe für die Datenbank PostgreSQL
  - ALTER DATABASE db_name SET default_transaction_isolation = ("read uncommitted" | "read committed" | "repeatable read" | "serializable")
  - Z.B. ALTER DATABASE webshop
    SET default_transaction_isolation = 'read committed';
- Pro Transaktion
  - BEGIN TRANSACTION ISOLATION LEVEL (SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED)
    ...

	Lost Updates	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	möglich	möglich	möglich	möglich
Read Committed	nein	nein	möglich	möglich
Repeatable Read	nein	nein	nein	möglich
Serializable	nein	nein	nein	nein

Hinweis: Postgres wechselt bei Read Uncommitted immer zu Read Committed!

Bulk-Insert

Foto: M. Strauch

Worum geht es?

- **Effizientes** Einfügen großer Datenmengen
  - z.B. aus einer anderen DB oder einem Backup

```
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
…
INSERT INTO x VALUES (…);
```

- Sie haben detailliertes Wissen über diese Daten
  1) Alle Daten haben identisches Schema
  2) Referentielle Integrität gegeben (Fremdschlüsselbeziehungen)
  3) Konsistenz gegeben (Unique-Attribute, Primärschlüssel, CHECK...)

- Sie haben detailliertes Wissen über den Vorgang
  1) Sie sind beim Bulk-Insert der einzige Nutzer der DB
  2) Sie können bei einem Systemabsturz problemlos von vorn anfangen

→ Sie benötigen die allermeisten DBMS-Features nicht

1.: Einzeltransaktionen vermeiden

- Autocommit abschalten
- Oder: Transaktion explizit starten

```
BEGIN TRANSACTION;

INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
INSERT INTO x VALUES (…);
…
INSERT INTO x VALUES (…);

COMMIT;
```

Warum sind Einzeltransaktionen langsamer?

2. Keine Constraint-Prüfungen beim Einfügen

- Alle Constraints entfernen, danach wieder anlegen
  - wirklich alle, also PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE...

```
ALTER TABLE x DROP CONSTRAINT c1, c2, … cn;
INSERT INTO x VALUES (…);
…
ALTER TABLE x ADD CONSTRAINT (...) PRIMARY KEY (…)
ALTER TABLE x ADD CONSTRAINT (...) FOREIGN KEY (…) NOT VALID;
ALTER TABLE x ADD CONSTRAINT (...) CHECK (…) NOT VALID;
...
```

    - NOT VALID sorgt dafür, dass keine Constraint-Prüfung der eben angelegten Daten stattfindet
    - NOT VALID derzeit nur für CHECK und FOREIGN KEY-Constraints

Warum sind gesammelte Constraint-Prüfungen am Ende schneller?

3. Entferne alles, was zusätzlichen Overhead verursacht

- Indexe entfernen, später wieder anlegen
  DROP INDEX i1, i2,...;

  CREATE INDEX i1...;

- Trigger auf ON INSERT deaktivieren, später wieder aktivieren
  ALTER TABLE x DISABLE TRIGGER ALL;
  ...
  ALTER TABLE x ENABLE TRIGGER ALL;

- Rules auf ON INSERT deaktivieren, später wieder aktivieren
  ALTER TABLE x DISABLE RULE r1, r2, r3...;
  ...
  ALTER TABLE x ENABLE RULE r1, r2, r3...;

4. Nutze SQL-Erweiterungen und Systemtools

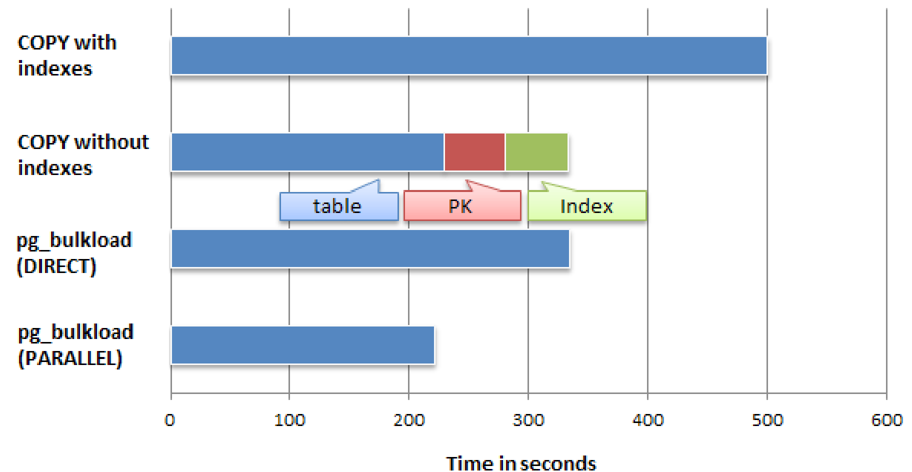
- Nutze COPY anstelle von INSERT INTO
  - verkürztes Datenformat, Postgres-eigene SQL-Erweiterung

```
INSERT INTO x VALUES (4, 'Erik');
INSERT INTO x VALUES (2, 'Erik');
INSERT INTO x VALUES (6, 'Jens');
```

```
COPY x (value, name) FROM stdin;
4           Erik
2           Erik
6           Jens
\.
```

- Spezielle Bulkload-Tools
  - pg_bulkload

    Diagramm: Einfügen von 4 GB Daten auf Maschine mit 2 CPUs und 24 Cores



Zum Abschluss

Foto: M. Strauch

Wie geht es weiter?

- bis Montag, 29.06., 12 Uhr
  - Quiz: Trigger, JDBC, etc.
- Dienstag, 30.06., GHH 12-14 Uhr: Tutoriumstermin
  - kurze Besprechung von Aufgabenblatt 8
  - nächstes Aufgabenblatt: Schemadefinitionen in SQL
- Donnerstag, 02.07.: Präsenztermin
  - Relationale DBMS als Objektspeicher