UNIVERSITÄT DES SAARLANDES

Saarbrücken, 23.07.2015
Information Systems Group

Vorlesung „Informationssysteme“

Vertiefung Kapitel 12: Indexstrukturen

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

Foto: M. Strauch

Bevor es losgeht: Datenbankoperationen in MapReduce

- MapReduce verwaltet (key,value)-Paare
  - Map-Phase:
    - speichere jedes Tupel in einem value
    - berechne dazu einen key, der gewünschte DB-Operation erlaubt
  - Reduce-Phase:
    - fasse (key, value)-Paare mit gleichen Keys zusammen und filtere, sodass DB-Ergebnis entsteht

- Beispiel Natural Join: SELECT * FROM A NATURAL JOIN B

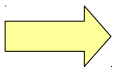
A

a	b
1	2
3	4

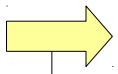
$\bowtie$

B

b	c
2	4
4	5
6	7



Map: erzeuge (b, (a,b,c))
Paare aus den Tupeln
2 → (1, 2, ?)
4 → (3, 4, ?)
2 → (?, 2, 4)
4 → (?, 4, 5)
6 → (?, 6, 7)



MapReduce: gleiche keys landen auf dem selben Rechner

Reduce: Fasse Paare mit gleichlautenden keys zusammen, verwerfe alle Paare mit einzelnen keys

2 → (1, 2, 4)
4 → (3, 4, 5)

Aus den Videos wissen Sie...

- ...wie ein B+ Baum aufgebaut ist
  - Baumstruktur über eine verkettete Liste
- ...dass damit Anfragen beschleunigt werden können
  - Doch wie funktioniert das in der Praxis?

- Vertiefung heute:
  - Indexe in Postgres anlegen
  - Was macht einen guten Index aus?

Motivation

Foto: M. Strauch

Laufendes Beispiel

- Eine einfache Testdatenbank mit Daten unterschiedlicher Verteilung

```
CREATE TABLE test (
    id              INT PRIMARY KEY,
    uniformint      INT,        – Gleichverteilung
    uniformreal     REAL,
    gaussint        INT,        – Normalverteilung
    gaussreal       REAL,
    uniformtext     TEXT);      – 30 Buchstaben A-Za-z mit Gleichvert.
```

- 5.000.000 Datensätze
  - INSERT mehrere Minuten ohne Autocommit aber mit Primärschlüssel

	id [PK] integer	uniformint integer	uniformreal real	gaussint integer	gaussreal real	uniformtext text
1	0	8989	8382.96	3579	4637.04	oepHuKNBiCq
2	1	749	5937.68	5370	3961.99	KGEFQvqLbpb
3	2	250	9602.26	4250	2112.92	HfAWLEeljaq
4	3	8357	6244.43	432	5227.19	svKaNGiDWDC
5	4	8236	7206.15	5004	7264.31	SxyXSlpsYgO

Anfragen im Beispiel (1/3)

Anmerkung: Alle Anfragen liefern höchstens ein paar hundert Ergebnistupel

- Exact Match

  Ausführungszeiten

  - SELECT * FROM test
    WHERE uniformtext = 'yQUyqIIGHJbCFjlsCMADpRehHpjdHb';

    610ms

- Range Query
  - SELECT * FROM test WHERE uniformint BETWEEN 2500 AND 5000
    AND uniformreal BETWEEN 1000 AND 2000
    AND gaussint BETWEEN 8000 AND 10000
    AND gaussreal BETWEEN 5000 AND 6000;

    720ms

- Aggregate
  - SELECT MIN(gaussreal) FROM test;

    620ms
  - SELECT AVG(gaussint) FROM test;

    570ms
  - SELECT * FROM test ORDER BY gaussint LIMIT 1 OFFSET 2500000;

    5510ms

Was sind die teuersten Anfragen auf dieser Seite?

Anfragen im Beispiel (2/3)

Anmerkung: Alle Anfragen liefern höchstens ein paar hundert Ergebnistupel

- Verbund

  Ausführungszeiten

  - SELECT *
    FROM test AS a JOIN test AS b ON (a.gaussint = b.gaussint)
    WHERE a.uniformint = 2500 AND b.gaussreal > 9900.0;

    970ms
  - SELECT *
    FROM test AS a JOIN test AS b ON (a.gaussint > b.gaussint)
    WHERE a.uniformint = 2500 AND b.gaussreal > 9990.0;

    1000ms
- Gruppierung
  - SELECT MIN(gaussint) FROM test GROUP BY gaussint / 1000;

    1200ms
- Sortierung
  - SELECT * FROM test WHERE uniformint = 2500 ORDER BY gaussint;

    449ms

Was sind die teuersten Anfragen auf dieser Seite?

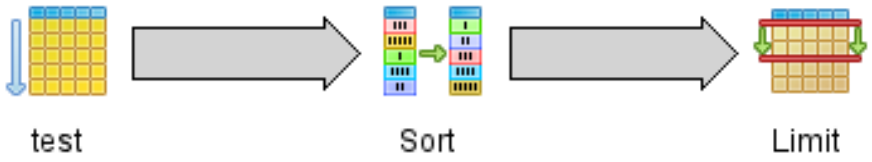
Anfragen im Beispiel (3/3)

Anmerkung: Alle Anfragen liefern höchstens ein paar hundert Ergebnistupel

- Sortierattribut: gaussint
  - SELECT *
    FROM test ORDER BY gaussint LIMIT 100

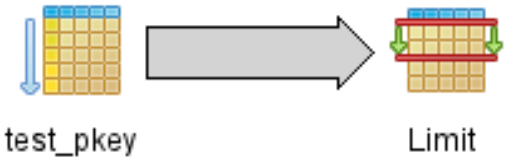
Ausführungszeiten

612ms



- Sortierattribut: id
  - SELECT *
    FROM test ORDER BY id LIMIT 100

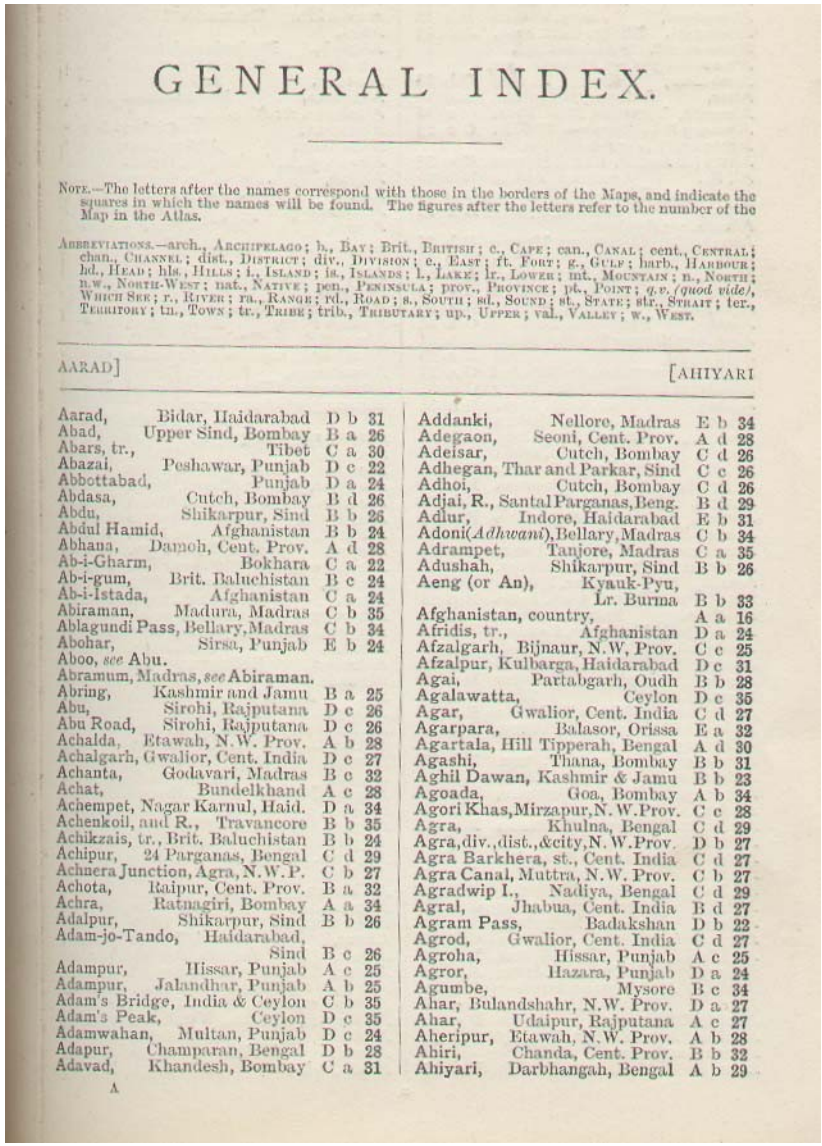
10ms



Welche der beiden Anfragen ist teurer?

Was ist ein Index?

- ein sortiertes Verzeichnis
  - auf welcher Speicherseite findet sich welche Information?
- Verschiedene Organisationformen
  - Primär- und Sekundärindex
  - clustered und non-clustered
  - vollständiger / partieller Index
  - dicht- und dünnbesetzt
  - über ein oder mehrere Attribute
  - verschiedene Speicherstrukturen
    - B+ Baum
    - Hash-Tabelle
- Im Folgenden: dichtbesetzter, nichtgeclusterter, vollständiger Sekundärindex auf ein Attribut



GENERAL INDEX.

NOTE.—The letters after the names correspond with those in the borders of the Maps, and indicate the squares in which the names will be found. The figures after the letters refer to the number of the Map in the Atlas.

ABBREVIATIONS.—arch., ARCHIPELAGO; b., BAY; Brit., BRITISH; c., CAPE; can., CANAL; cent., CENTRAL; chan., CHANNEL; dist., DISTRICT; div., DIVISION; e., EAST; ft. FORT; g., GULF; harb., HARBOUR; hd., HEAD; hls., HILLS; I., ISLAND; is., ISLANDS; L., LAKE; lr., LOWER; mt., MOUNTAIN; n., NORTH; n.w., NORTH-WEST; nat., NATIVE; pen., PENINSULA; prov., PROVINCE; pt., POINT; *q.v. (quod vide)*, WHICH SEE; r., RIVER; ra., RANGE; rd., ROAD; s., SOUTH; sd., SOUND; st., STATE; str., STRAIT; ter., TERRITORY; tn., TOWN; tr., TRIBE; trib., TRIBUTARY; up., UPPER; val., VALLEY; w., WEST.

A

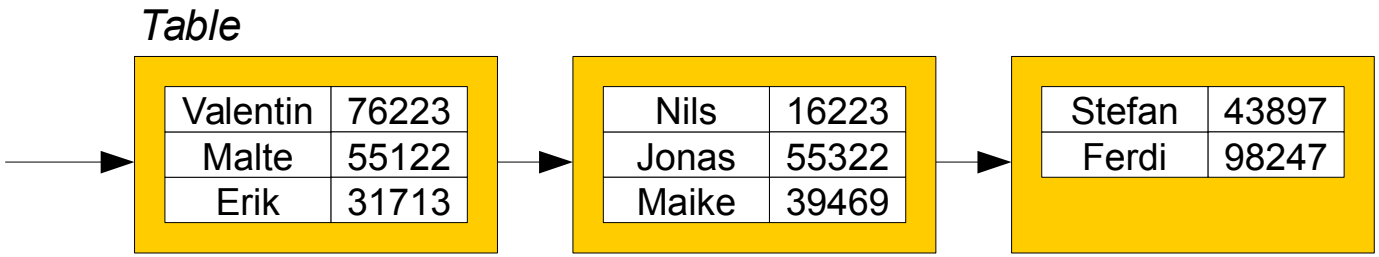
Bild: I. Poyntz

Indexe in DBMS

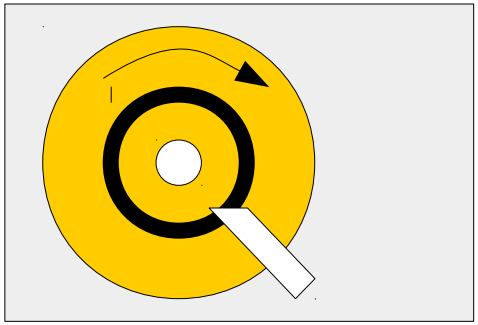
Foto: M. Strauch

Wie können Indexe Anfragen beschleunigen?

- Basismethode zum Datenzugriff: Full Table Scan
  - Sequentiell und unsortiert alle Speicherseiten von Festplatte lesen
    - In Postgres: Speicherseite hat 8192 Bytes

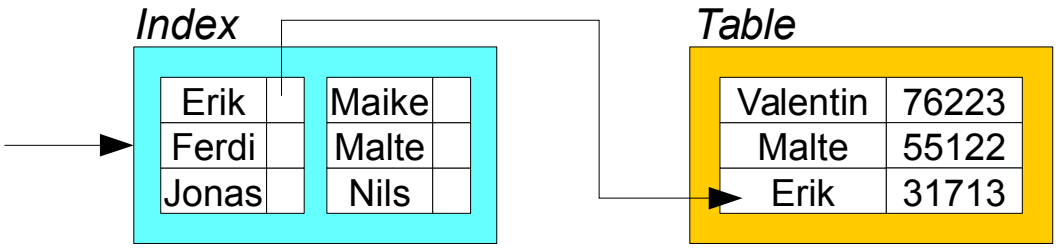


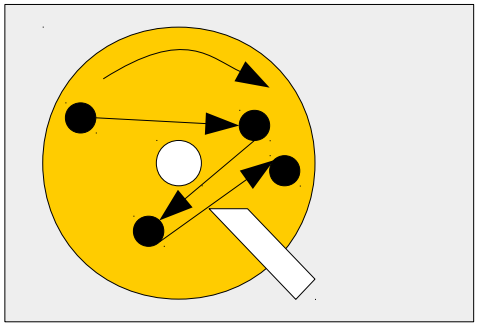
  - Klingt „teuer“, muss es aber nicht sein
    - Festplatten sind gut darin, Daten sequentiell in der Speicherreihenfolge einzulesen
    - Katalog-Informationen im DBMS können Full Table Scan abkürzen
      - Ist das Attribut UNIQUE?
      - Sind die Daten sortiert? (CLUSTER)



Mit Index: Index Lookup

- Index-Zugriff verrät Speicherort der gesuchten Tupel
- Klingt „billig“, muss es aber auch nicht sein
  - Random Access auf Festplatte
    1) Zugriff auf Index (evtl. mehrere Knoten)
    2) Zugriff auf gesuchtes Datentupel

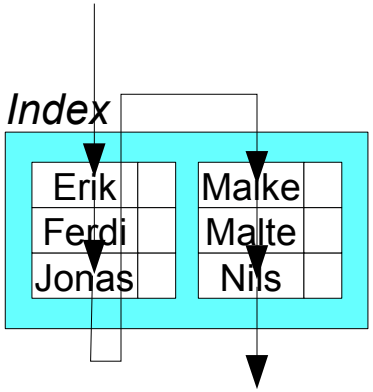


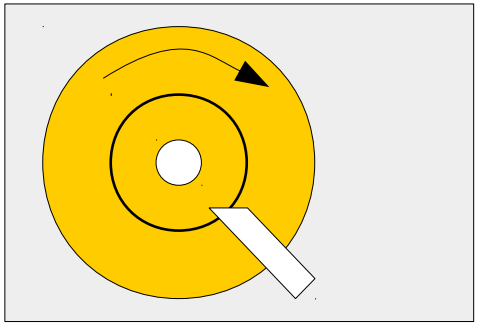


- Viele Abhängigkeiten
  - Selektivität, Datenverteilung, Index-Typ...
  - DBMS braucht „teure“ Statistiken zur Entscheidung

Mit Index: Index Scan

- Index speichert alle nötigen Informationen für eine (Teil-)Anfrage
  - „Full Table Scan“ auf einem Index, d.h., Sequentieller Zugriff auf Festplatte
  - Index-Struktur spielt keine Rolle

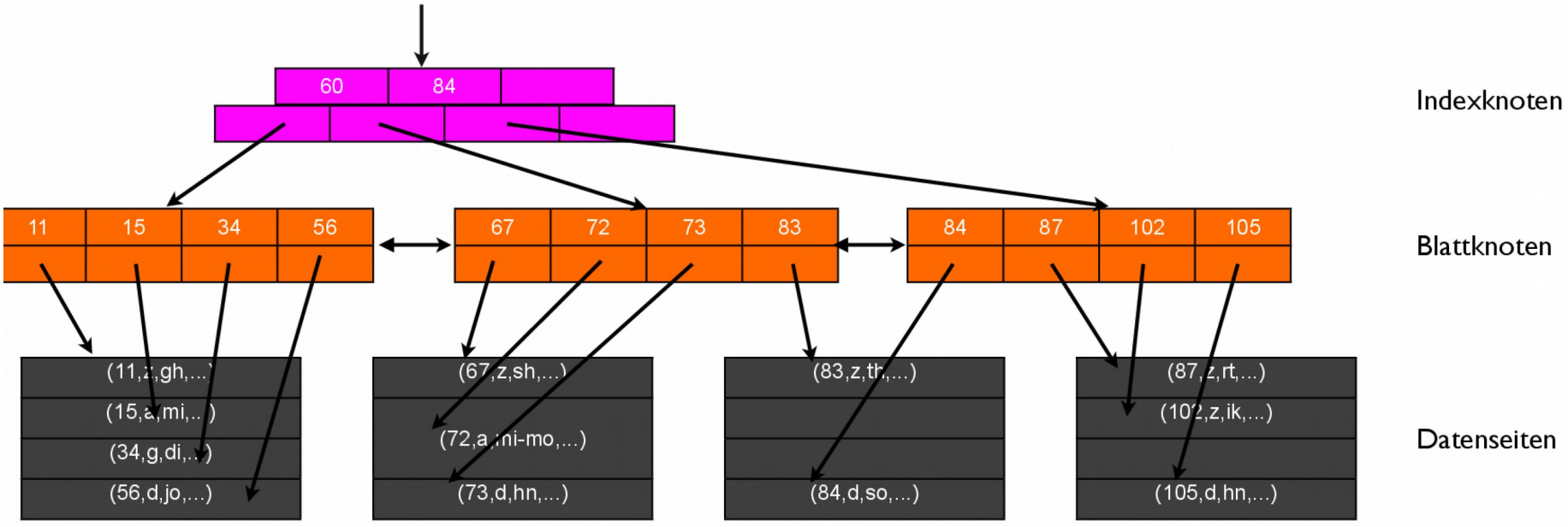




- z.B. bei Anfragen, die sich nur auf Index-Attribut beziehen
  Relation [table] = {[name, schuhgröße, alter, gewicht]} mit Index auf name
  SELECT name FROM table;
  - Größe des Indexes im Vergleich zur Größe der Table?

B+ Baum-Index

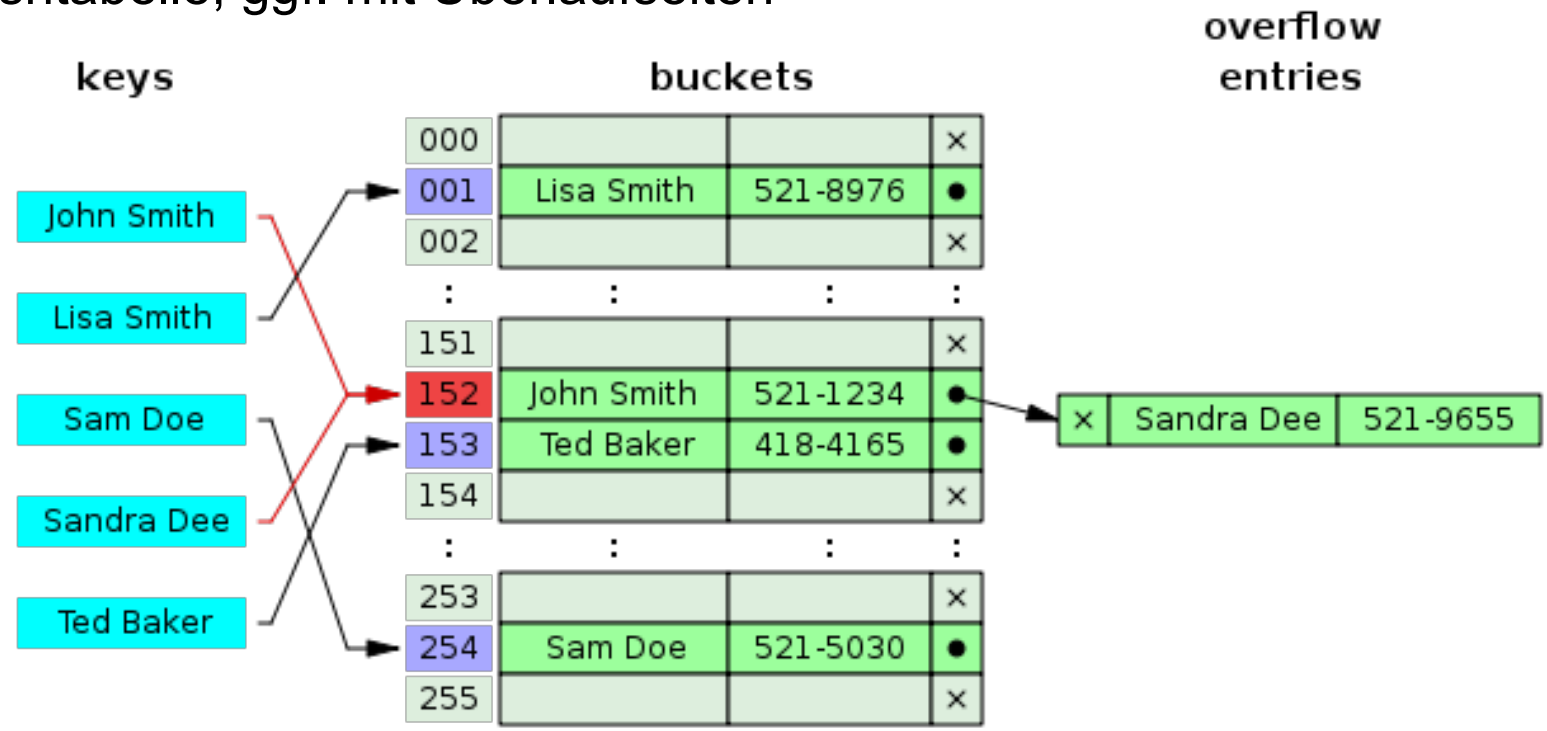
- Aus der Video-Vorlesung bekannt



- O(log N) Zugriffe von der Wurzel bis zum Blatt, dann Datenseite holen
- Unterstützt >, =, <, MIN, MAX, BETWEEN,IN,EXISTS, LIKE 'ABC%' *(Nur wenn Wildcard % nicht am Anfang der Bedingung!)*, ORDER BY, GROUP BY

Hash-Index

- Hashtabelle, ggf. mit Überlaufseiten



- Hash(key), dann O(1) Zugriffe auf Bucket, dann Datenseite holen
- Unterstützt =, IN, EXISTS

Bild: Wikimedia

Indexe anlegen und löschen

- CREATE [UNIQUE] INDEX [name] ON table_name [USING (btree|hash)] (column_1, …, column_n) [WHERE predicate]
  - UNIQUE: neue Constraint!
    - Index erzwingt uniqueness
  - btree, hash: Index-Methode, default ist btree
  - predicate: partieller Index nur über einen Teil der Werte

- DROP INDEX index_name
  - falls UNIQUE-Index: uniqueness ist ebenfalls weg

- Beispiele

```
CREATE INDEX my_index ON katzenfutter (wasseranteil, trockenmasse);

CREATE INDEX ON katzen USING hash (rasse)
        WHERE rasse = 'Kurzhaar' AND gewicht > 5kg;
```

Automatische Indexe

- Die meisten DBMS legen automatisch Indexe an für
  - PRIMARY KEY-Constraints
  - UNIQUE-Constraints
    - 

verrät darüber nichts, ist aber sichtbar im Ausführungsplan
- Constraint-Prüfung läuft über den Index
  - Zugriff auf Hash-Index mit O(1) oder B+ Tree Index mit O(log N) statt Full Table Scan mit O(N)

Gute Indexe, schlechte Indexe?

- Indexe kosten
  - Speicherplatz
  - Rechenleistung bei INSERT, UPDATE, DELETE
    - DMBS aktualisiert einmal angelegte Indexe automatisch
  - Zugriffszeit (Random Access vs. Sequential Access auf Festplatte)

- Indexe bringen keinen Leistungszuwachs wenn
  - Nach Indexzugriff sowieso (fast) alle Speicherseiten geladen werden müssen
  - Index genauso groß ist wie die Ursprungsrealation, und Sortierung spielt keine Rolle
  - Index-Architektur ungeeignet, z.B. MIN() auf Hash-Index

    → dann gleich sequenziellen Datenzugriff

- **Optimierer des DBMS entscheidet über Indexeinsatz**

Wann wählt Optimierer Zugriff über Index?

- Wenn geschätzte Kosten für Index-Scan < Full Table Scan
  - Beispiel: [table] = {[a,b,c,d]}, alles Int-Werte, Index auf a

1) Selektivität ist hoch
   - Beispiel: SELECT MIN(a) FROM table;
     - Nur Index-Seiten + eine Seite von 'table' einlesen

2) Bereichsanfragen, die WENIGE Tupel zurückliefern
   - Beispiel: SELECT * FROM table WHERE a BETWEEN 100 AND 200;
     - Nur Index-Seiten + alle Seiten von 'table' einlesen, die im Bereich liegen
     - Wenn Tupel über zuviele Seiten verstreut, lohnt es sich nicht mehr

3) Indexzugriff genügt
   - Beispiel SELECT a FROM table;
     - Anfrage lässt sich vollständig aus dem Index beantworten

So (ähnlich) rechnet der Optimierer

- Beispielanfrage auf [table] = {[a,b,c,d]}
  SELECT * FROM table WHERE a > 100;
- Statistiken des DBMS
  - Seitengröße von Postgres ist 8192 Bytes
  - table hat 100.000 Tupel, ca. 1000 Tupel pro Seite, 100 Seiten insges.
  - Index auf table.a auf 25 Seiten mit einer Tiefe von 3 Seiten
  - Histogramm über a, Verteilung der Daten für Selektivitätsabschätzung
- Selektivität von a>100 ist 50%, d.h., die Hälfte aller Tupel im Ergebnis
  - Wahrscheinlichkeit, dass eine Speicherseite von table geholt werden muss, ist 1 - 0.5^100 ≈ 100% → **mache einen Full Table Scan**
- Selektivität von a>100 ist 1%, d.h., 1% aller Tupel im Ergebnis
  - Wahrscheinlichkeit, dass eine Speicherseite von table geholt werden muss, ist 1 - 0.99^100 ≈ 63% → **mache einen Index Lookup**

Kann man das DBMS zwingen?

- Manche DBMS erlauben sog. „Hints“
  - dem Optimizer mitteilen, dass er bestimmte Indexe nutzen muss
  - sinnvoll, wenn Operator mehr weiß als die DBMS-Statistiken

- Beispiel Oracle
  SELECT /*+ INDEX(katzen rasse_index) */ name, gewicht
  FROM katzen WHERE rasse = 'Kurzhaar';

- Nicht implementiert in PostgreSQL weil
  - Widerspricht dem deklarativen Paradigma
  - Datenverteilung in der Zukunft?
  - Größe der Relation in der Zukunft?
  - Algorithmen in der nächsten Version vom DBMS?

Mit diesem Wissen...

Exact-Match Anfrage von vorhin

Ausführungszeiten

SELECT * FROM test
WHERE uniformtext = 'yQUyqIIGHJbCFjlsCMADpRehHpjdHb';

610ms

Welcher Index ist schneller aufgebaut? Welche Anfrage ist schneller?

Mit Hash-Index 7900ms

CREATE INDEX mtch_idx ON test USING hash (uniformtext);

SELECT * FROM test
WHERE uniformtext = 'yQUyqIIGHJbCFjlsCMADpRehHpjdHb';

6ms

Mit B+ Tree-Index 37100ms

CREATE INDEX mtch_idx ON test USING btree (uniformtext);

SELECT * FROM test
WHERE uniformtext = 'yQUyqIIGHJbCFjlsCMADpRehHpjdHb';

11ms

Aggregatanfragen

Aggregate von vorhin	Ausführungszeiten
SELECT MIN(gaussreal) FROM test;	620ms
SELECT AVG(gaussint) FROM test;	570ms
SELECT * FROM test ORDER BY gaussint LIMIT 1 OFFSET 2500000;	5510ms

Mit folgenden Indexen

```
CREATE INDEX agg_1_idx ON test (gaussint);
CREATE INDEX agg_2_idx ON test (gaussreal);
```

ergibt

SELECT MIN(gaussreal) FROM test;	12ms
SELECT AVG(gaussint) FROM test;	550ms
SELECT * FROM test ORDER BY gaussint LIMIT 1 OFFSET 2500000;	5306ms

Warum nutzt die letzten beiden Anfrage keinen Index? → Optimierer patzt

Einzelne Indexe auf allen Attributen

Ausführungszeiten

Verbund

```
SELECT *
FROM test AS a JOIN test AS b ON (a.gaussint = b.gaussint)
WHERE a.uniformint = 2500 AND b.gaussreal > 9900.0;
```

970ms
7ms

```
SELECT *
FROM test AS a JOIN test AS b ON (a.gaussint > b.gaussint)
WHERE a.uniformint = 2500 AND b.gaussreal > 9990.0;
```

1000ms
160ms

Gruppierung

```
SELECT MIN(gaussint) FROM test GROUP BY gaussint / 1000;
```

1200ms
1205ms

Index nicht benutzt, vermutlich gleiches Problem wie eben

Sortierung

```
SELECT * FROM test WHERE uniformint = 2500 ORDER BY gaussint;
```

449ms
30ms

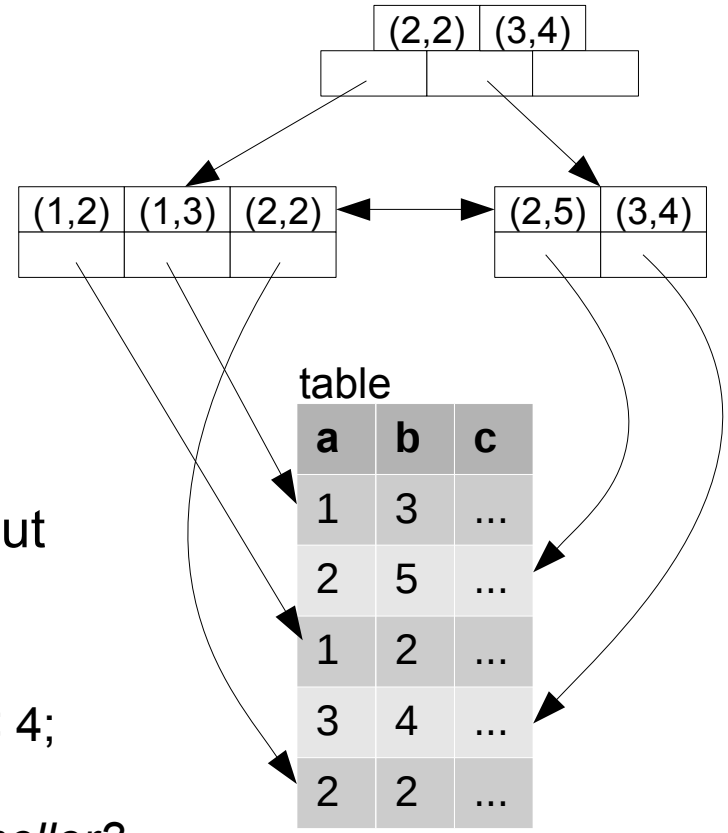
Weitere Optionen für Indexe

Foto: M. Strauch

Indexe mit mehreren Attributen

- CREATE INDEX ON table (a, b)
  auf Relation [table] = {[a, b, c, …]}
  - Mehrere Attribute im gleichen Index
    - NICHT: mehrere unabhängige Indexe!
- Sortierreihenfolge im Index
  gemäß CREATE INDEX
  - hier: erst nach a, dann nach b sortiert
- Beschleunigt Anfragen auf das erste Attribut
  SELECT * FROM table WHERE a > 1;
- Beschleunigt Anfragen auf beide Attribute
  SELECT * FROM table WHERE a > 1 AND b < 4;

  Unter welchen Umständen wird folgendes schneller?
  *SELECT * FROM table WHERE b < 4;*



In unserem Beispiel

Range Query von vorhin — Ausführungszeiten

```
SELECT * FROM test WHERE uniformint BETWEEN 2500 AND 5000          720ms
                   AND uniformreal BETWEEN 1000 AND 2000
                   AND gaussint BETWEEN 8000 AND 10000
                   AND gaussreal BETWEEN 5000 AND 6000;
```

Welche Indexe sind besser?

Folgende Indexe

```
CREATE INDEX rng_idx ON test USING hash (uniformint);                                  729ms
CREATE INDEX rng_idx ON test (uniformint);                                             802ms
CREATE INDEX rng_idx ON test (uniformreal);                                            761ms
CREATE INDEX rng_idx ON test (uniformint, uniformreal);                                750ms
CREATE INDEX rng_idx ON test (uniformint, uniformreal, gaussint, gaussreal);           190ms
CREATE INDEX rng_idx ON test (gaussint, gaussreal, uniformint, uniformreal);           126ms
CREATE INDEX rng_idx ON test (gaussreal, gaussint, uniformint, uniformreal);           156ms
```

Multiattribut-Index für eindimensionale Anfragen?

Ausführungszeiten

SELECT MIN(gaussreal) FROM test; 808ms

SELECT MIN(gaussint) FROM test; 695ms

SELECT MIN(uniformint) FROM test; 670ms

SELECT * FROM test AS a JOIN test AS b ON (a.gaussreal = b.gaussreal) 1300ms
WHERE a.uniformint = 2500 AND b.gaussreal > 9000.0;

Mit Index von eben:
CREATE INDEX mltp_idx ON test (gaussreal, gaussint, uniformint, uniformreal);

ergibt:

SELECT MIN(gaussreal) FROM test; 11ms

SELECT MIN(gaussint) FROM test; 685ms

SELECT MIN(uniformint) FROM test; 663ms

SELECT * FROM test AS a JOIN test AS b ON (a.gaussreal = b.gaussreal) 609ms
WHERE a.uniformint = 2500 AND b.gaussreal > 9000.0;

→ *nur nützlich wenn Attribut an erster Stelle im Index*

Partielle Indexe

- Indexe, die nur einen Teil der Daten indexieren
  - In PostgreSQL über WHERE-Condition angegeben

- Beispiel: häufigste Anfragen auf [table] = {[a,b,c,d]}
  SELECT * FROM table WHERE a > 100;
  SELECT * FROM table WHERE a > 1000;
  SELECT * FROM table WHERE a = 200;

  CREATE INDEX table_idx ON table (a) WHERE a > 100;

- Was ist der Vorteil?
  - B+ Baum-Index wird flacher
    - weniger Speicherseiten von Wurzel bis Blatt laden
  - uninteressante oder häufig geänderte Datensätze ausschließen

Was bringt ein partieller Index?

Range Query von vorhin — Ausführungszeiten

```
SELECT * FROM test WHERE uniformint BETWEEN 2500 AND 5000          720ms
                    AND uniformreal BETWEEN 1000 AND 2000
                    AND gaussint BETWEEN 8000 AND 10000
                    AND gaussreal BETWEEN 5000 AND 6000;
```

der beste Index eben

```
CREATE INDEX rng_idx ON test (gaussint, gaussreal, uniformint, uniformreal);126ms
```

als partieller Index

```
CREATE INDEX rng_idx ON test (gaussint, gaussreal, uniformint, uniformreal) 106ms
      WHERE uniformint BETWEEN 2500 AND 5000
                    AND uniformreal BETWEEN 1000 AND 2000
                    AND gaussint BETWEEN 8000 AND 10000
                    AND gaussreal BETWEEN 5000 AND 6000;
```

Clustered Indexe

- Normalerweise: Speicherung erfolgt unsortiert
  - schnelles Einfügen
  - Reihenfolge im Speicher für die meisten Operationen irrelevant
- CLUSTER table_name [USING index_name]
  - Speicherfolge der Relation gemäß Index
    - Wenn kein Index angegeben, wird nach Primary Key sortiert
  - CLUSTER ist einmalige Operation!
    - bei INSERT, UPDATE, DELETE Sortierreihenfolge nicht beibehalten
    - fast dasselbe wie
      CREATE TABLE neu AS (SELECT * FROM alt ORDER BY attribut);
- Sinnvoll bei Anfrage auf indexierten Wertebereich
  - aufeinanderfolgende Werte hintereinander gespeichert,
    Sequential Access statt Random Access auf Festplatte

Was bringt das Sortieren?

Ausführungszeiten

Bereichsanfrage:

SELECT avg(gaussreal) FROM test
WHERE gaussint BETWEEN 2500 AND 5000;

806ms

Anfrage mit folgendem Index:

CREATE INDEX rng_idx ON test USING btree (gaussint);

941ms

→ *Optimierer nutzt Index, schlechte Wahl!*

Über den Index clustern:

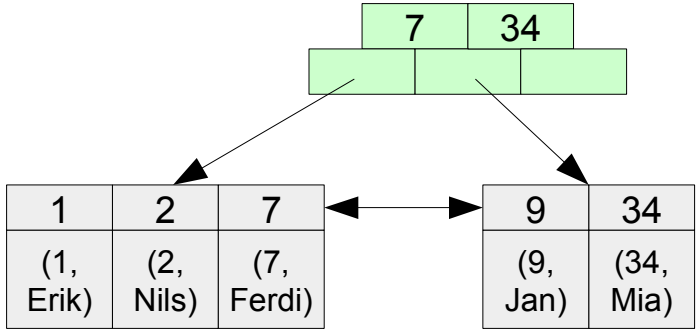
CLUSTER test USING rng_idx;

472ms

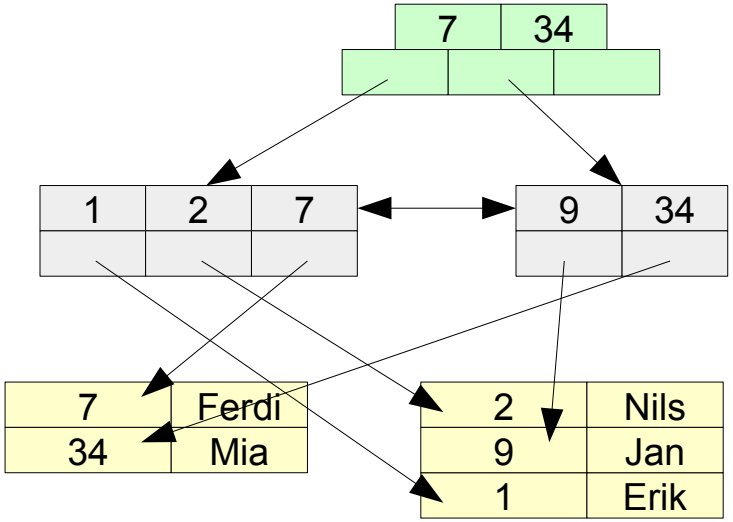
→ *Sortierreihenfolge hat etwas gebracht!*

Primary und Secondary Indexes

- Primary Index, auch: Integrated Index
  - Table direkt in den Blättern des B+ Baumes gespeichert
  - Primary Index ist automatisch clustered
  - ein Primary Index pro Table möglich



- Secondary Index
  - Table ist separat vom Index gespeichert
  - beliebig viele Secondary Indexes pro Table
  - PostgreSQL kennt nur Secondary Indexes!



Welche Anfragen profitieren nicht von Indexen?

- LIKE mit Wildcard '%' am Anfang
  SELECT * FROM table
  WHERE name LIKE '%Buchmann';
  - Indexe sortieren nach dem Anfang

- Anfragen mit ungleich (<>)
  SELECT * FROM table
  WHERE plz <> 76119;
  - Index kann nichts finden was ungleich ist

manchmal Anfragen aber durch Index Scan schneller

- Anfragen mit NOT
  SELECT * FROM table
  WHERE NOT name = 'Erik Buchmann';
  - Index kann nicht nach etwas suchen, das nicht existiert

Zum Abschluss

Foto: M. Strauch

Wie geht es weiter?

- Dienstag, 28.07., GHH 12-14 Uhr: Tutoriumstermin
  - kurze Besprechung von Aufgabenblatt 12
- Donnerstag, 30.07. GHH 10-12 Uhr:
  - Große Fragerunde zur Klausur
- Donnerstag, 06.08. ab 13 Uhr:
  - ***Endklausur, 120 Minuten***
  - Parallel im GHH und im HS 1 in E2.5,
    Aufteilung wird noch per Email über Moodle bekanntgegeben
- 10.09. ab 10 Uhr
  - Nachklausur, 120 Minuten
  - Parallel im GHH und im HS 1 in E2.5