

Datenbanksysteme

DB Tuning

Jan Haase

2025

Abschnitt 12

Themenübersicht

- Wiederholung Teil I
- PL/SQL

DB-Tuning

- Stern-Schema-Modellierung
- Big Data, NoSQL
- Klausurvorbereitung

Themenübersicht

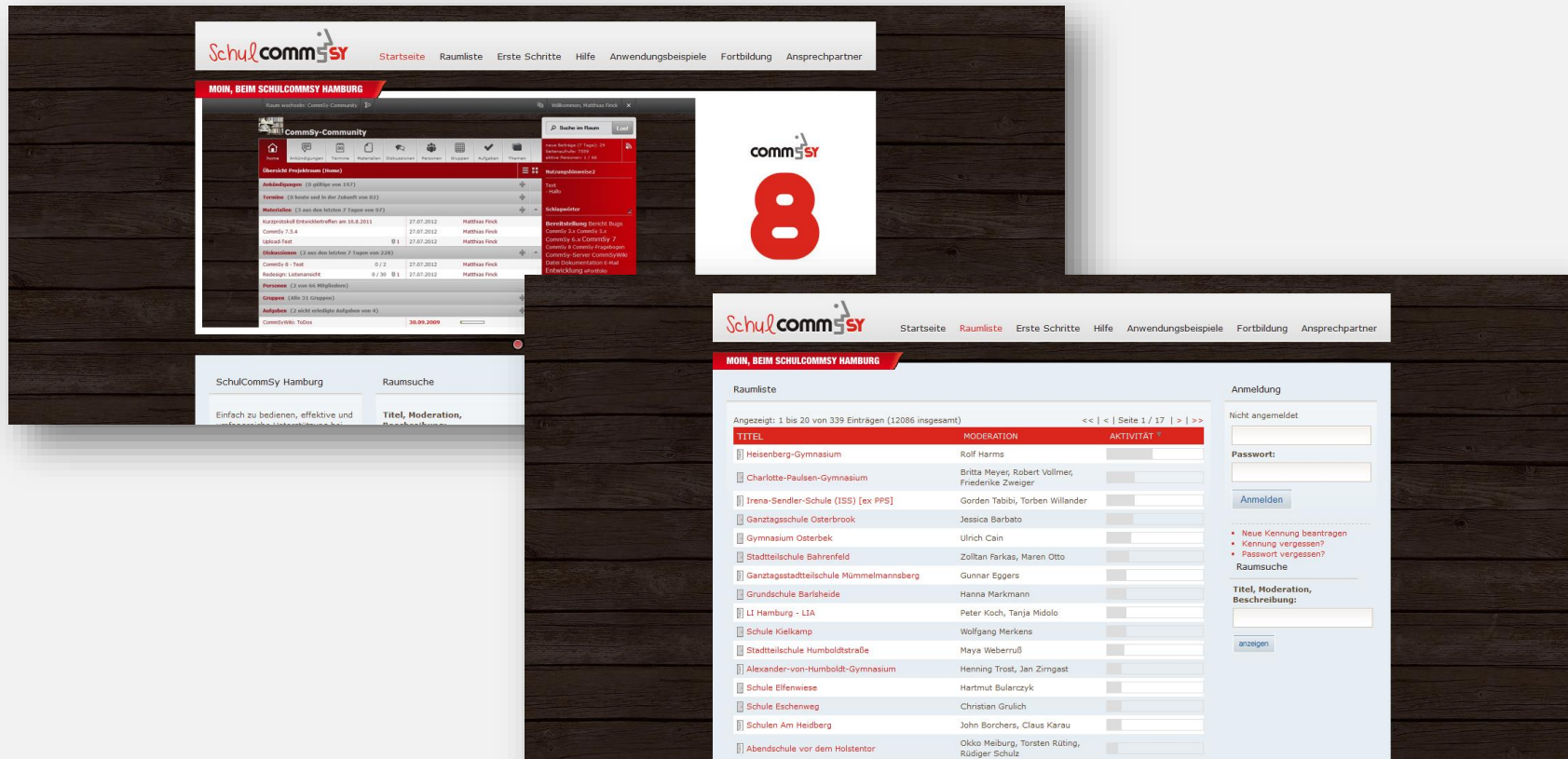
- Wiederholung Teil I
- PL/SQL
- **DB-Tuning**

Einleitendes Fallbeispiel

- Ziele und Ebenen des DB-Tuning
- Manipulation der Relationsschemata
- Verwendung von Indexstrukturen
- Verwendung von MATERIALIZED VIEWS
- EXPLAIN PLAN
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Das SchulCommSy Hamburg

- <http://hamburg.schulcommsy.de>
- Lernplattform für alle staatlichen, allgemeinbildenden Schulen Hamburgs
- ca. 20-30 Aufrufe pro Sekunde zu Stoßzeiten



SchulcommSY Startseite Raumliste Erste Schritte Hilfe Anwendungsbeispiele Fortbildung Ansprechpartner

MOIN, BEIM SCHULCOMMSY HAMBURG

CommSy-Community

Raumliste

Angezeigt: 1 bis 20 von 339 Einträgen (12086 insgesamt) << | < | Seite 1 / 17 | > | >>

TITEL	MODERATION	AKTIVITÄT
Hesenberg-Gymnasium	Rolf Harms	
Charlotte-Paulsen-Gymnasium	Britta Meyer, Robert Vollmer, Friederike Zweiger	
Irena-Sendler-Schule (ISS) [ex PPS]	Gordon Tabibi, Torben Willander	
Ganztagschule Osterbrook	Jessica Barbato	
Gymnasium Osterbek	Ulrich Cain	
Stadtteilschule Bahrenfeld	Zoltan Farkas, Maren Otto	
Ganztagsstadtteilschule Mümmelmannsborg	Gunnar Eggers	
Grundschule Barlshede	Hanna Markmann	
Li Hamburg - LIA	Peter Koch, Tanja Midolo	
Schule Kielkamp	Wolfgang Merkens	
Stadtteilschule Humboldtstraße	Maya Weberruß	
Alexander-von-Humboldt-Gymnasium	Henning Trost, Jan Zirmgast	
Schule Eifenwiese	Hartmut Bularczyk	
Schule Eschenweg	Christian Grulich	
Schulen Am Heidelberg	John Borchers, Claus Karau	
Abendschule vor dem Holstenor	Olko Meiburg, Torsten Rüting, Rüdiger Schulz	

Anmeldung

Nicht angemeldet

Passwort:

Anmelden

• Neue Kennung beantragen
• Kennung vergessen?
• Passwort vergessen?

Raumsuche

Titel, Moderation, Beschreibung:

anzeigen

Die Datenbank im SchulCommSy

- MySQL Datenbank
- ca. 75 DB-Tabellen
- Knapp 20 Mio DB-Einträge

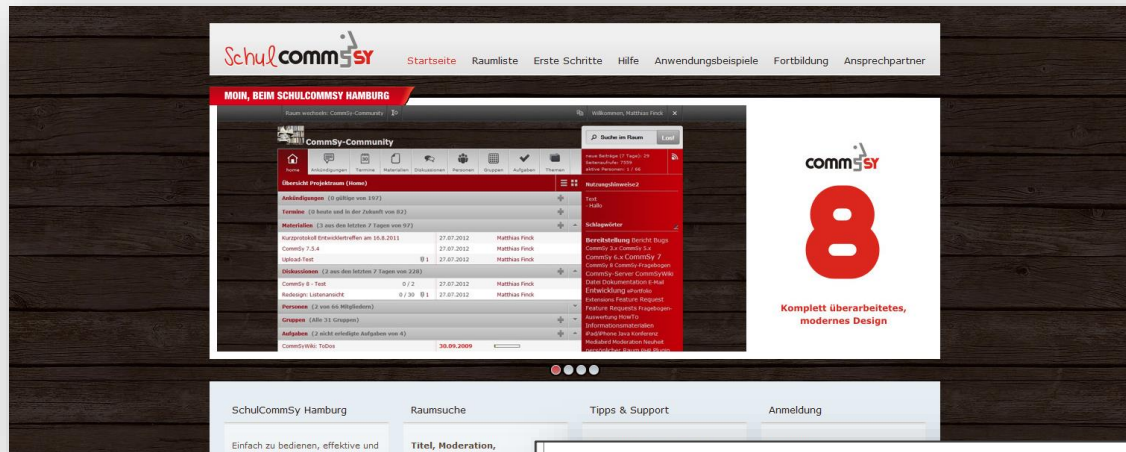
The screenshot shows the phpMyAdmin interface. On the left, the database '_commsy (72)' is selected, and the table 'room' is highlighted in the list. The main area displays the table structure for 'room' in the 'schulhh_commsy' database on 'localhost'.

Feld	Typ	Kollation	Attribute	Null	Standard	Extra	Aktion
<input type="checkbox"/> item_id	int(11)			Nein	0		[Icons]
<input type="checkbox"/> context_id	int(11)			Ja	NULL		[Icons]
<input type="checkbox"/> creator_id	int(11)			Nein	0		[Icons]
<input type="checkbox"/> modifier_id	int(11)			Ja	NULL		[Icons]
<input type="checkbox"/> deleter_id	int(11)			Ja	NULL		[Icons]
<input type="checkbox"/> creation_date	datetime			Nein	0000-00-00 00:00:00		[Icons]
<input type="checkbox"/> modification_date	datetime			Nein	0000-00-00 00:00:00		[Icons]
<input type="checkbox"/> deletion_date	datetime			Ja	NULL		[Icons]
<input type="checkbox"/> title	varchar(255)	utf8_general_ci		Nein	Kein		[Icons]
<input type="checkbox"/> extras	mediumtext	utf8_general_ci		Ja	NULL		[Icons]
<input type="checkbox"/> status	varchar(20)	utf8_general_ci		Nein	Kein		[Icons]
<input type="checkbox"/> activity	int(11)			Nein	0		[Icons]
<input type="checkbox"/> type	varchar(20)	utf8_general_ci		Nein	project		[Icons]
<input type="checkbox"/> public	tinyint(11)			Nein	0		[Icons]
<input type="checkbox"/> is_open_for_guests	tinyint(4)			Nein	0		[Icons]
<input type="checkbox"/> continuous	tinyint(4)			Nein	-1		[Icons]
<input type="checkbox"/> template	tinyint(4)			Nein	-1		[Icons]
<input type="checkbox"/> contact_persons	varchar(255)	utf8_general_ci		Ja	NULL		[Icons]
<input type="checkbox"/> description	text	utf8_general_ci		Ja	NULL		[Icons]
<input type="checkbox"/> room_description	varchar(10000)	utf8_general_ci		Ja	NULL		[Icons]

At the bottom, there are options to 'Druckansicht' (Print view) or 'Tabellenstruktur analysieren' (Analyze table structure). A button '1' is visible, and a dropdown menu shows 'Nach item_id'.

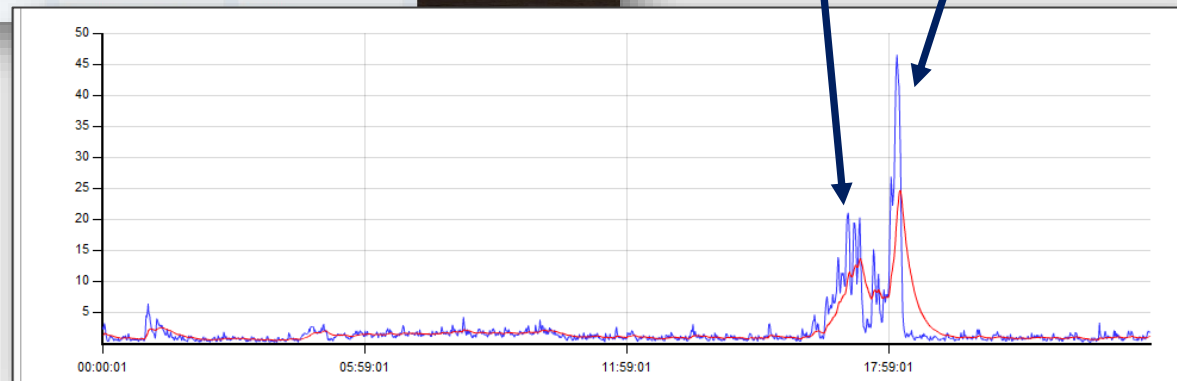
Der Problemfall

- Systemupdate mit Anzeige der Ansprechpartner auf der Raumübersicht
- Raumübersicht ist die zentrale Einstiegsseite
- ca. 10 Aufrufe / Sekunde



Update

Systemabsturz



Analyse des Problems: Ein SQL-Statement

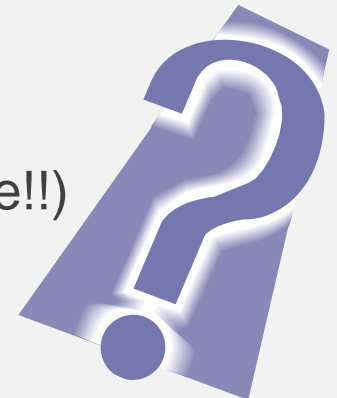
- Tabelle „room“:
 - ca. 15.000 Einträge
 - „item_id“, „title“, „creator_id“, „creation_date“, ...

- Tabelle „user“:
 - ca. 360.000 Einträge
 - „item_id“, „context_id“, „name“, „is_contact“, etc.

- SQL-Statement:

```
SELECT room.title, user.lastname  
FROM room  
LEFTJOIN user ON room.item_id = user.context_id  
WHERE user.is_contact = 1;
```

- Antwortzeit für das eine Statement: 1,38s
- Gesamtseitenaufruf: knapp 4s (bei 10 Aufrufen / Sekunde!!)
- Alle Tests der Systemversion liefen erfolgreich
- Aber auf einer viel kleineren Testdatenbank



Lösung

Lösung 1:

- Verwendung von **Indizes**
- „context_id“ in der „user“-Tabelle indexieren
- Antwortzeit: 0,09s (Faktor 15)

Lösung 2:

- **Redundante Datenhaltung**
- Die Kontaktpersonen als Feld „contacts“ in die „room“-Tabelle
- Keine 1. Normalform mehr, da mehrere Kontaktpersonen pro Raum möglich
- Antwortzeit: 0,002s (Faktor 700)

Fazit des Fallbeispiels

- DB-Tuning als zentrale Aufgabe
 - Kontinuierlich
 - Von zentraler Bedeutung
- Tuning widerspricht teilweise der „reinen Lehre“
 - Im Fallbeispiel findet eine „De-Normalisierung“ statt
 - Vermeidung von JOINS auf Kosten redundanter Daten
- Tuning ist meist relativ
 - Im Fallbeispiel werden die SELECT-Statements optimiert
 - Schreibzugriffe werden komplexer und langsamer wegen der doppelten Datenhaltung

Themenübersicht

- Wiederholung Teil I

- PL/SQL

- **DB-Tuning**

- Einleitendes Fallbeispiel

Ziele und Ebenen des DB-Tuning

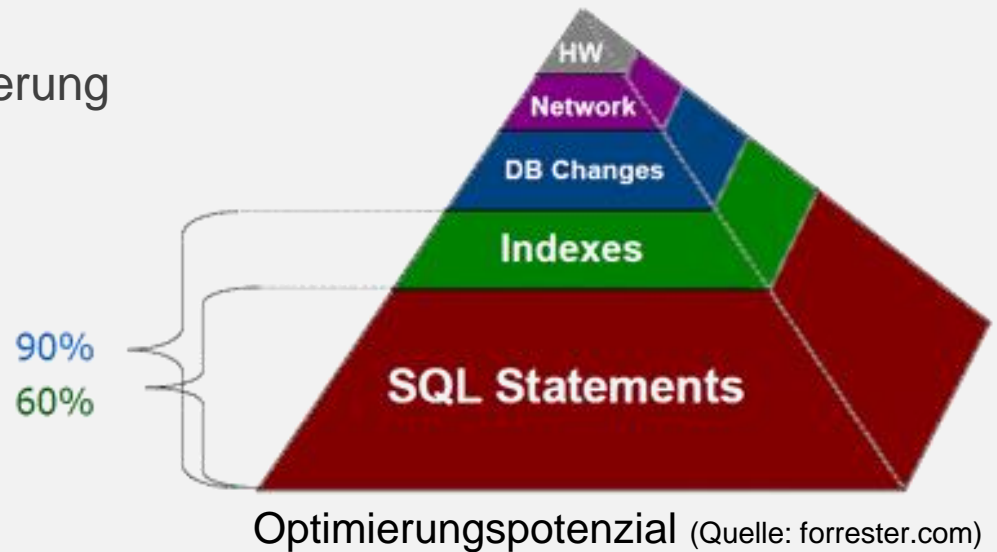
- Manipulation der Relationsschemata
 - Verwendung von Indexstrukturen
 - Verwendung von MATERIALIZED VIEWS
 - EXPLAIN PLAN
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Ziele des DB-Tunings

- Verkürzung der **Antwortzeit** (Online Analytical Processing -OLAP)
 - Wichtig bei Einbenutzer-Datenbankapplikationen (z.B. Data Mining): der Benutzer möchte möglichst schnell eine Antwort auf seine Anfrage.
- Erhöhung des **Durchsatzes** (Online Transaction Processing - OLTP)
 - Wichtig bei Hochleistungs-Transaktionssystemen (z.B. Flugbuchung): pro Zeiteinheit sollen möglichst viele Anfragen bearbeitet werden.
- **Zielkonflikt**
 - Zwischen der Verkürzung der Antwortzeit und der Erhöhung des Durchsatzes besteht häufig ein Zielkonflikt, da z.B. die Verlagerung von Programmfunktionalität in den DB-Server (gespeicherte Prozeduren) die Antwortzeit im Einbenutzermodus verkürzt, dagegen im Mehrbenutzermodus den Durchsatz hemmt.
- Ziel ist ein „**Moving target**“
 - Eine Regel des SQL-Tunings besagt z.B., dass in der FROM-Klausel die größte Tabelle nach links und die kleinste nach rechts positioniert werden sollen.

Ebenen des Tunings

- Relationenschema
 - Normalisierung, Denormalisierung
- PhysischesDB-Schema
 - Cluster, Index
- Anfrageoptimierung
 - Z.B. Query Rewriting
- Client-Server-Architektur
- Parallelisierung und Verteilung
- Transaktionsdesign
- Betriebssystemparameter und DB-Voreinstellungen:
 - Größe eines DB-Blocks, Größe des DB-Cache



Themenübersicht

- Wiederholung Teil I
- PL/SQL
- **DB-Tuning**
 - Einleitendes Fallbeispiel
 - Ziele und Ebenen des DB-Tuning
- ➡ **Manipulation der Relationsschemata**
 - Verwendung von Indexstrukturen
 - Verwendung von MATERIALIZED VIEWS
 - EXPLAIN PLAN
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Normalisierung

- Wenn auf bestimmte Attribute einer Relation erheblich **häufiger zugegriffen** wird, als auf andere (speicherintensive) Attribute derselben Relation, kann es günstig sein, **die Relation zu zerlegen**.
- Beispiel:
`ERSATZTEILLAGER (TeileNr, Bestand, Teilebeschreibung)`
 Wenn auf Bestand viel häufiger zugegriffen wird als auf Teilebeschreibung, dann ist folgendes Relationenschema günstiger:
`LAGERBESTAND (TeileNr, Bestand)`
`TEILE (TeileNr, Beschreibung)`
- Zerlegung ist z.B. bei Joins mit einer anderen Relation günstiger im Leistungsverhalten, aber auch **platzintensiver**.

Denormalisierung

- **Vermeidung von Joins**

- Beispiel:

```
PRAKTIKUMSTEILNEHMER (PraktikumsNr, MatrikelNr)
STUDENT (MatrikelNr, Name, Adresse, Telefonnummer)
```

Bei den meisten Zugriffen auf die **PRAKTIKUMSTEILNEHMER** wird auch gleichzeitig der Name des Studenten benötigt (Join). Eine **redundante Speicherung** von Name kann daher sinnvoll sein:

```
PRAKTIKUMSTEILNEHMER (PraktikumsNr, MatrikelNr, Name)
STUDENT (MatrikelNr, Name, Adresse, Telefonnummer)
```

- aber: **Gefahr von Update-Anomalien**
 - Formulieren von Integritätsbedingungen
 - Überwachung durch Trigger (aufwändig)

Beispiel

- Mondial Datenbank:

- COUNTRY (Name, CountryCode, Capital, Province, Area, Population)
- CITY (Name, CountryCode, Province, Population, Longitude, Latitude)
- CONTINENT (Name, Area)
- ENCOMPASSES (CountryCode, Continent, Percentage)
- ISMEMBER (CountryCode, Organization, Type)
- PROVINCE (Name, CountryCode, Population, Area, Capital, Capprov)

Ermitteln Sie **drei Beispiele**, bei denen unter bestimmten Bedingungen eine **Denormalisierung** sinnvoll sein könnte.

Bennenen Sie jeweils:

- 1) Welche Änderungen/Denormalisierungen wurden vorgenommen?
- 2) Unter welchen Bedingungen warum schneller?
- 3) Welche Form der Denormalisierung fand statt?
- 4) Wo sind die Gefahren dieser konkreten Denormalisierung?



Themenübersicht

- Wiederholung Teil I
- PL/SQL
- **DB-Tuning**
 - Einleitendes Fallbeispiel
 - Ziele und Ebenen des DB-Tuning
 - Manipulation der Relationsschemata
- ➡ **Verwendung von Indexstrukturen**
 - Verwendung von MATERIALIZED VIEWS
 - EXPLAIN PLAN
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Index in SQL

Ein Index ist eine **von der Datenstruktur getrennte** Indexstruktur in einer Datenbank, die die **Suche und das Sortieren** nach bestimmten Feldern beschleunigt.

Ein Index auf einer Menge von Attributen, die den Primärschlüssel enthalten, wird Primärindex genannt.

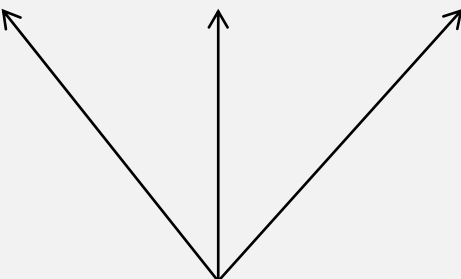
Ein Index, der kein Primärindex ist, wird Sekundärindex genannt.

Index in SQL

- Ein Index kann eine SQL-Abfrage beschleunigen, wenn das Feld in der Where-Klausel indexiert ist: ... WHERE Feld = XYZ.
- Über dem Primärschlüssel liegt i.d.R. immer ein Index.
- Bei einem Primärindex kann die Wertausprägung der Primärschlüsselattribute nicht doppelt vorkommen
- Bei einem Sekundärindex kann die Wertausprägung der Indexattribute doppelt vorkommen, es sei denn man fordert ein UNIQUE bei der Indexdefinition.

Syntax:

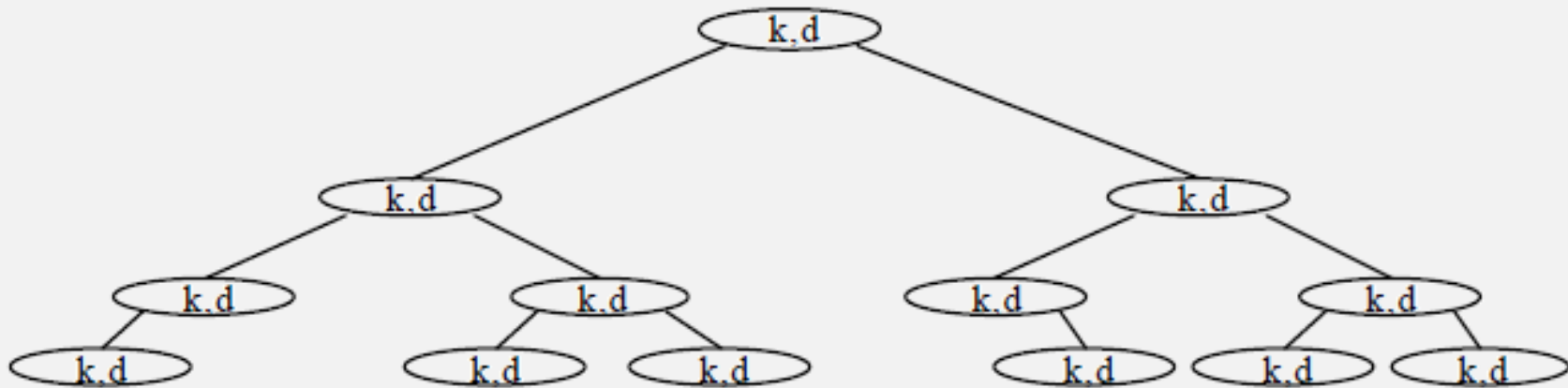
```
CREATE [UNIQUE] INDEX index_name  
ON table_name (column1, column2, column_n)
```



Index kann über mehrere
Felder angelegt werden

Index – Funktionsweise

- K=Schlüssel, nach dem gesucht wird
- d=Zeiger auf Datensatz, der hinter dem Schlüssel liegt.

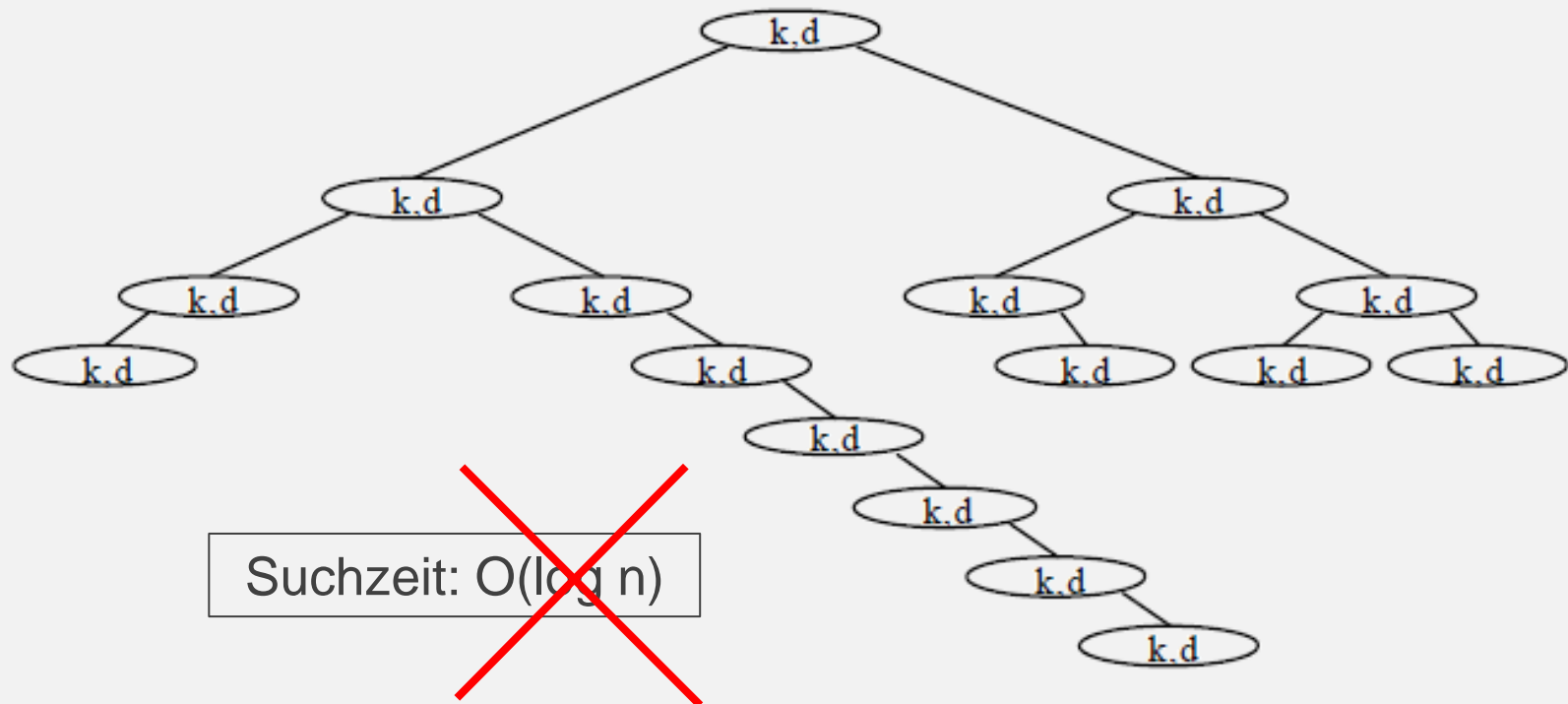


Suchzeit: $O(\log n)$

(Wenn der Baum gut ausbalanciert ist!)

Index – Funktionsweise

- K=Schlüssel, nach dem gesucht wird
- d=Zeiger auf Datensatz, der hinter dem Schlüssel liegt.

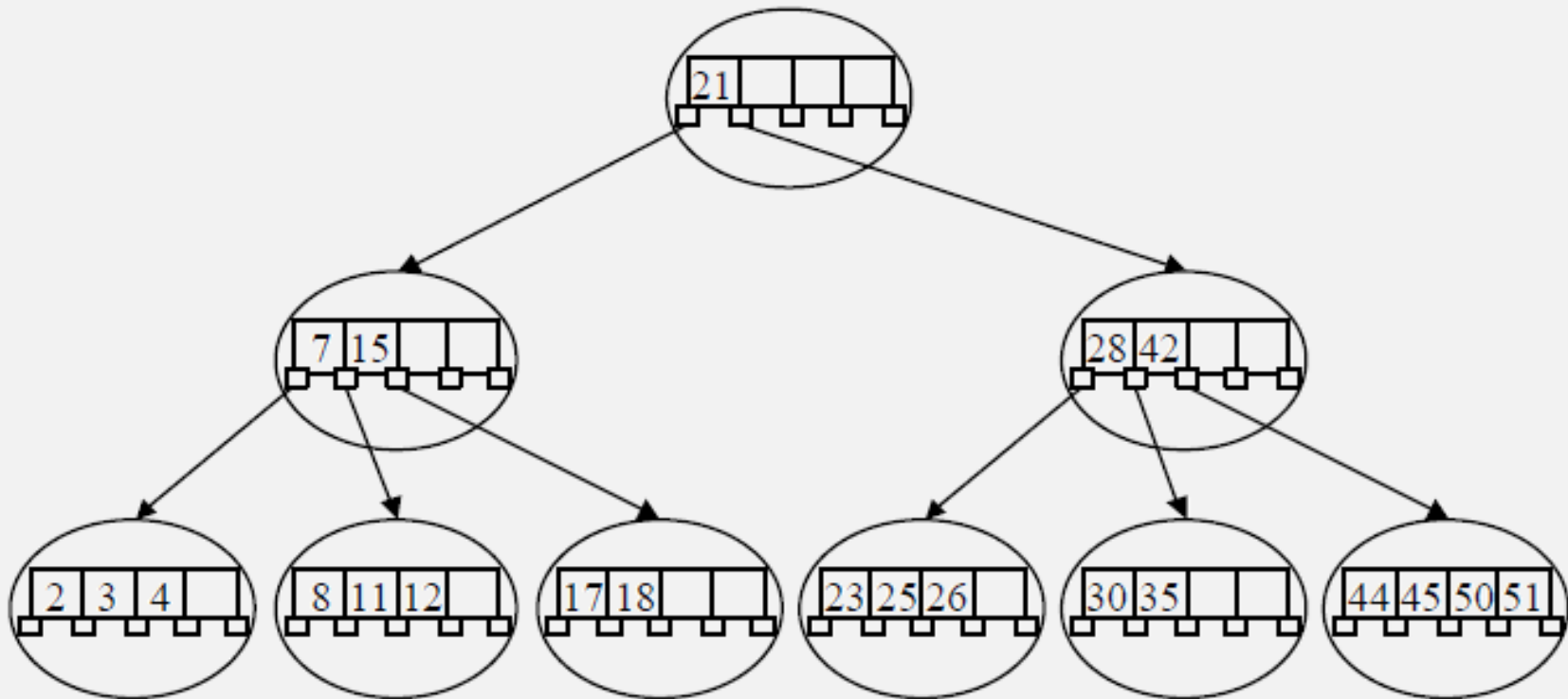


Index – Funktionsweise: B-Bäume

1. Jeder Knoten hat mindestens k und maximal $2k$ Einträge.
Ausnahme: die Wurzel kann zwischen 1 und $2k$ Einträge haben.
2. Wenn ein Knoten n Einträge hat, hat er $n+1$ Verweise auf seine Söhne.
Ausnahme: Blätter haben keine bzw. undefinierte Verweise.
3. Ein Blatt muss folgendermaßen organisiert sein:
 1. Wenn k_1 bis k_n Schlüssel sind, sind z_1 bis z_{n+1} Zeiger auf Söhne.
 2. z_1 zeigt auf Teilbaum mit Schlüsseln kleiner k_1
 3. z_i ($i=1\dots n$) zeigt auf Teilbaum mit Schlüsseln zwischen k_i und k_{i+1}

Index – Funktionsweise: B-Bäume

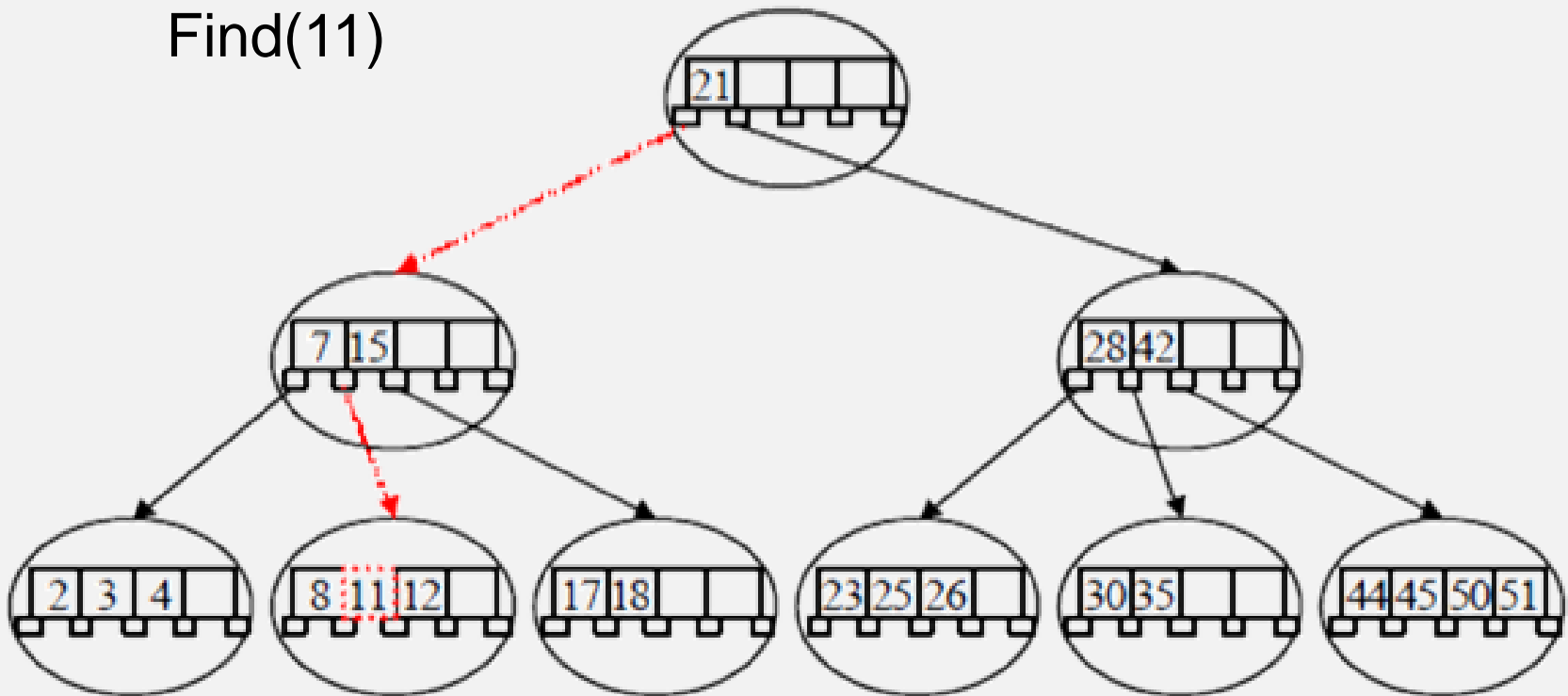
Hat mehrere Schlüssel in einem Blatt und damit auch mehrere Abzweigungen



Index – Funktionsweise: B-Bäume

- Suchen in B-Bäumen

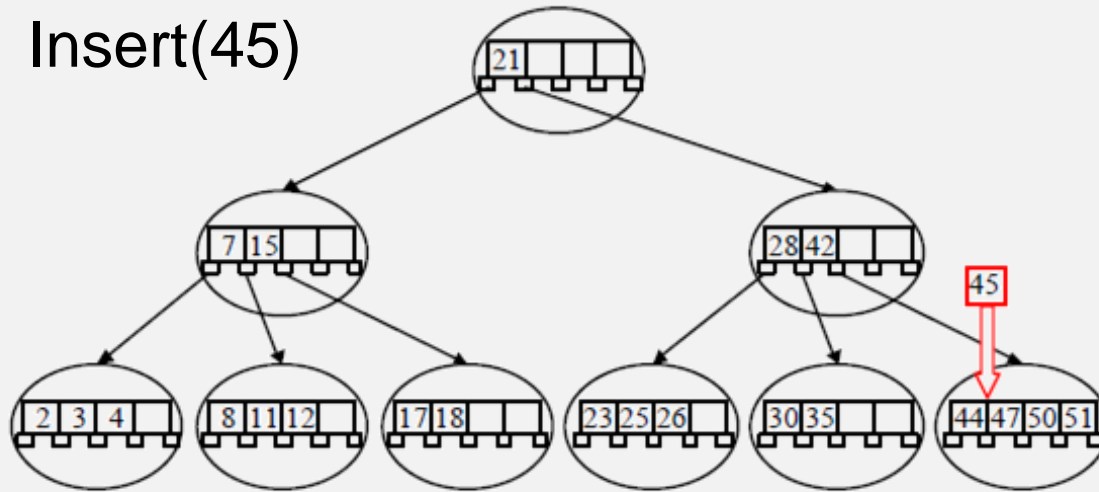
Find(11)



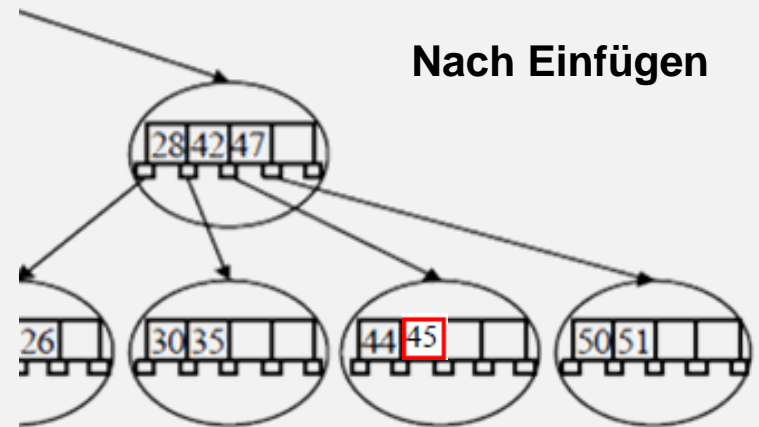
Index – Funktionsweise: B-Bäume

- **Einfügen** in B-Bäumen

Insert(45)



Vor Einfügen

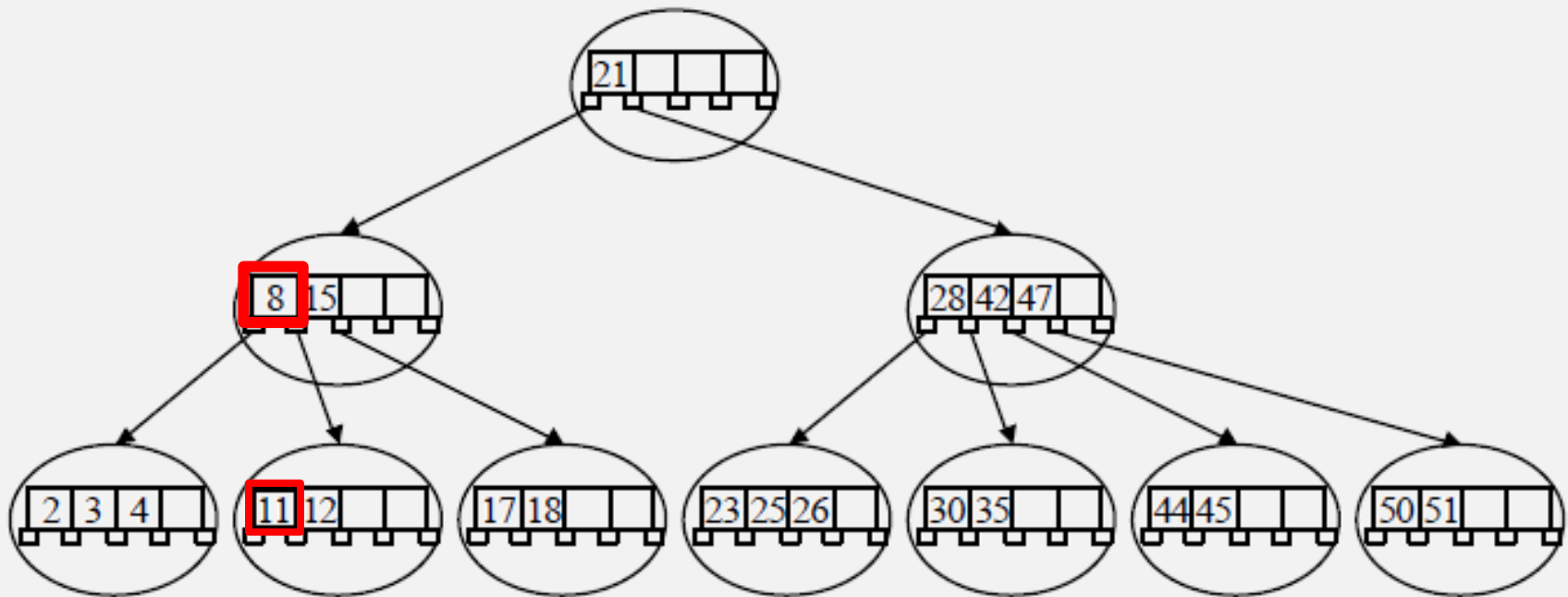


Nach Einfügen

Index – Funktionsweise: B-Bäume

- **Löschen** in B-Bäumen

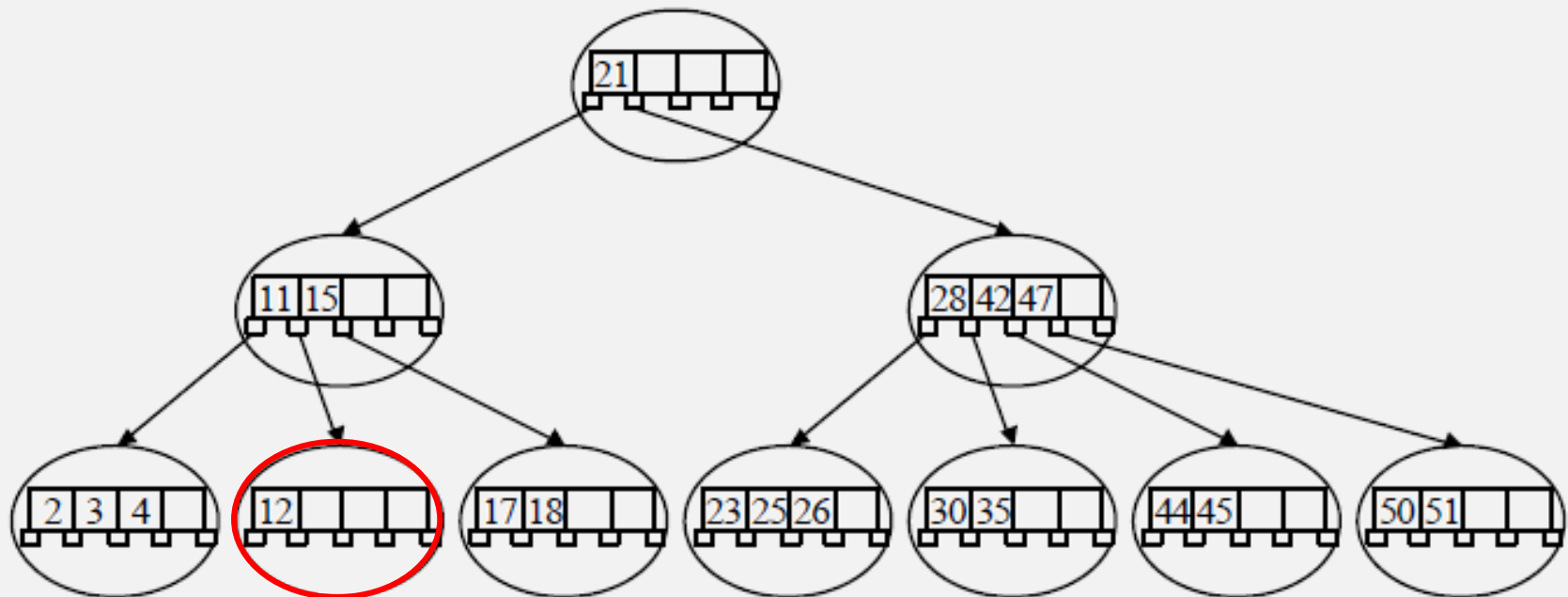
DELETE(8)



Index – Funktionsweise: B-Bäume

- **Löschen** in B-Bäumen

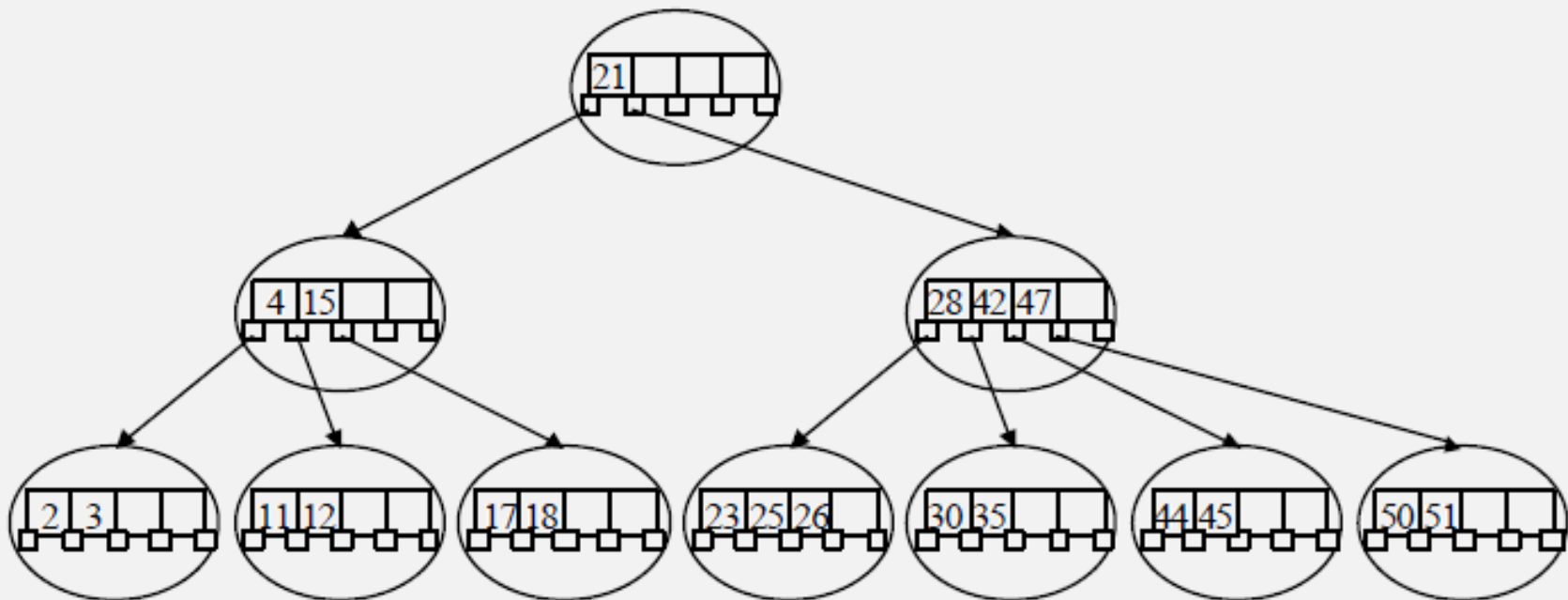
DELETE(8)



- Nachdem der leere „slot“ durch den 11er-Schlüssel aufgefüllt wurde, verletzt das 2. Blatt von links die B-Baum-Definition

Index – Funktionsweise: B-Bäume

- **Löschen** in B-Bäumen
- B-Baum nach der „**Re-Distribution**“



Index – Funktionsweise: Hashing

- Das Hashverfahren ist ein **Algorithmus** zum Suchen von Datenobjekten in großen Datenmengen.
- Eine Hashfunktion bildet eine beliebig große Quellmenge auf eine kleinere Zielmenge (mit konstanter Größe) ab.
- Die Umkehrung der Hashfunktion ist nicht zweifelsfrei möglich.

Eine einfache Hash-Funktion

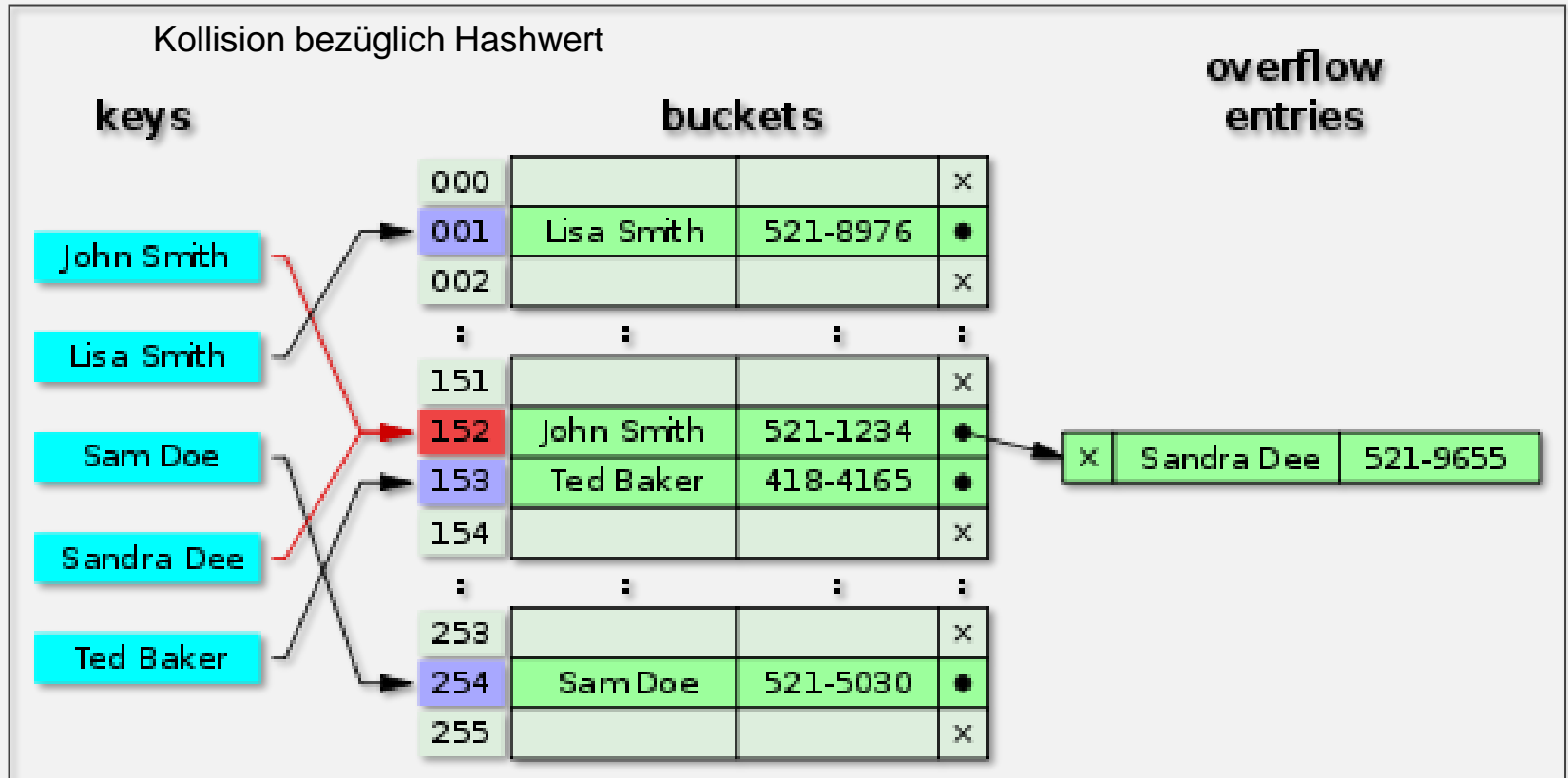
- Divisions-Rest-Methode (Modulo)

$$h(k) = k \bmod m$$

- m Größe der Hashtabelle
- k Ausgangswert, der „gehasht“ werden soll (=Schlüssel)
- $h(k)$ Hashwert von k

Index – Funktionsweise: Hashing

- Hashtable aufbauen und Kollisionen



Themenübersicht

- Wiederholung Teil I
- PL/SQL
- **DB-Tuning**
 - Einleitendes Fallbeispiel
 - Ziele und Ebenen des DB-Tuning
 - Manipulation der Relationsschemata
 - Verwendung von Indexstrukturen
- ➡ **Verwendung von MATERIALIZED VIEWS**
 - EXPLAIN PLAN
- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

Materialized Views

Was sind **Materialized Views**?

- Wie normale VIEW, jedoch handelt es sich um **Datenkopien** der Originaldaten in der Materialized VIEW.
- Normale VIEWS werden bei Aufruf berechnet, Materialized VIEWS werden einmal berechnet und dann **dauerhaft gespeichert**.
- Vorteil: Wenn man eine VIEW öfter nutzt und die Berechnung der VIEW aufwändig ist, dann ist die **Performance** erheblich besser.
- Nachteil: Wenn sich Daten ändern, auf denen die materialized VIEW beruht, dann muss die materialized VIEW neu berechnet werden (und anders herum).

Materialized Views

Syntax:

```
CREATE MATERIALIZED VIEW sichtname
    [ (<Spaltendefinition> [,
<Spaltendefinition>]...)      [<build_clause>]
    <refresh_clause>
    AS <Unterabfrage>;

<refresh_clause> ::= NEVER REFRESH
                  | { REFRESH [ FAST | COMPLETE | FORCE ]
                      [ ON DEMAND | ON COMMIT ]
                      [ START WITH date ] [ NEXT date
] }

<build_clause> ::= BUILD { IMMEDIATE | DEFFERED }
```

Materialized Views

- Beispiel:

```
CREATE MATERIALIZED VIEW Angestellte_Abteilung_mv  
(Mitarbeitername, Privates_Telefon, Abteilung)  
REFRESH FAST ON COMMIT AS  
    SELECT  Nachname || ', ' || Vorname,  
            Telefon_priv,  
            Abteilungen.Name  
FROM    Angestellte, Abteilungen  
WHERE   Angestellte.Abt_nr = Abteilungen.Abt_nr  
        AND    Gehalt > 3000;
```

Materialized Views

Build-Arten in Oracle:

Festlegung, wann das Abfrageergebnis in der Sicht materialisiert wird.

- **BUILD IMMEDIATE:** erzwingt das sofortige Füllen
- **BUILD DEFERRED:** lässt das Füllen zu einem späteren Zeitpunkt zu.

Materialized Views

Refresh-Arten in Oracle:

- **NEVER REFRESH:** Es wird **nie** eine Aktualisierung der materialisierten Daten durchgeführt.
- **COMPLETE:** Der SELECT-Ausdruck wird **vollständig neu** ausgewertet und alle Datensätze aus den Tabellen unabhängig von irgendwelchen Änderungen erneut ermittelt,
- **FAST:** In Abhängigkeit von den Änderungen in den zugrunde liegenden Tabellen, die in LOG-Dateien protokolliert werden, werden nur die von diesen Änderungen betroffenen Datensätze der materialisierten Sicht aktualisiert. Diese inkrementelle Aktualisierung spart erheblich viel Zeit gegenüber der vollständigen Rematerialisierung im COMPLETE-Modus.
- **FORCE:** Ist ein FAST-Refresh möglich, so wird er durchgeführt, sonst ein COMPLETE-Refresh. Das DBMS trifft diese Entscheidung.

Materialized Views

Refresh-Ereignisse (Aktualisierungszeitpunkte) in Oracle:

- **ON COMMIT:** Die Aktualisierung wird automatisch zum Transaktionsende (COMMIT-Anweisung) durchgeführt, wenn eine Transaktion die Daten ändert, auf denen die VIEW aufgebaut ist.
- **ON DEMAND:** Die Aktualisierung wird automatisch durchgeführt, wenn spezielle Prozeduren des Package DBMS_MVIEW zur Aktualisierung von Sichten aufgerufen werden.
- **NEXT Date:** Es wird ein Intervall für die nächste und die weiteren Aktualisierung spezifiziert.
- Mit der **START WITH-Option** kann ein Starttermin für die Aktualisierungen definiert werden.

Themenübersicht

- Wiederholung Teil I
- PL/SQL
- **DB-Tuning**
 - Einleitendes Fallbeispiel
 - Ziele und Ebenen des DB-Tuning
 - Manipulation der Relationsschemata
 - Verwendung von Indexstrukturen
 - Verwendung von MATERIALIZED VIEWS

EXPLAIN PLAN

- DB-Zugriff aus Programmiersprachen
- Big Data, NoSQL
- Business Intelligence, Data Warehouses
- Klausurvorbereitung

EXPLAIN PLAN

- Ermöglicht den **Ausführungsplan** eines SQL Statements durch das Datenbankmanagementsystem (DBMS) einzusehen.
- Zeigt auf, welche Schritte das DBMS durchführt, um ein Statement abzuarbeiten.

Erstellung:

EXPLAIN PLAN

SET statement_id = 'MY_STATE100'

FOR

SELECT * FROM TABLE WHERE FIELD='XYZ'

- Oder im SQL-Developer

Explain Plan Option



EXPLAIN PLAN

Ausgabe:

- Es wird eine Systemtabelle ausgelesen (muss man sich nicht merken)

```
SELECT cardinality "Rows",  
       lpad(' ',level-1)||operation||' '||options||'  
                                              '||object_name  
"Plan"  
FROM PLAN_TABLE  
CONNECT BY prior id = parent_id  
           AND prior statement_id = statement_id  
START WITH id = 0  
           AND statement_id = 'MY_STATE100'  
ORDER BY id;
```

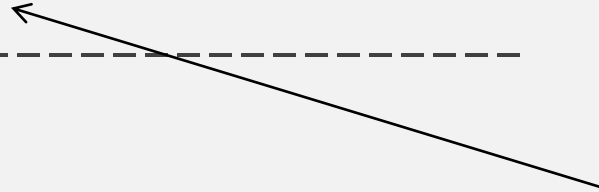
EXPLAIN PLAN

Ergebnis:

```
SELECT phone_number
FROM employees
WHERE phone_number LIKE '650%';
```

Id		Operation	Name

0		SELECT STATEMENT	
1		TABLE ACCESS FULL	EMPLOYEES



Der ganze Table
Employees wird
vollständig
durchsucht =
ACCESS FULL

EXPLAIN PLAN

Ergebnis:

```
SELECT lastname
FROM employees
WHERE phone_number LIKE 'Pe';
```

Id		Operation	Name

0		SELECT STATEMENT	
1		INDEX RANGE SCAN	EMP_NAME_IX

Der ganze Table
Employees wird
anhand des Indexes
untersucht. Es ist
kein FULL TABLE
SCAN notwendig

EXPLAIN PLAN

Ergebnis:


```
SELECT *
FROM auftrag_pos ap, auftrag a
WHERE a.auftrag_nr = ap.auftrag_nr;
```

QUERY_PLAN

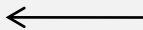
NESTED LOOPS

```
TABLE ACCESS FULL AUFTRAG
TABLE ACCESS BY INDEX ROWID AUFTRAG_POS
INDEX RANGE SCAN PK_AUFTRAG_POS
```

Alle Einträge in „Auftrag werden durchlaufen“ in der „nested loop“



Für jeden Auftrag wird die Tabelle AUFTRAG_POS durchsucht. Allerdings kann hier der Index für direkten Zugriff verwendet werden und die Tabelle AUFTRAG_POS muss nicht vollständig jeweils durchsucht werden



EXPLAIN PLAN

- **Full Table Scan (FTS)**

- Alle Datensätze einer Tabelle werden gelesen

- **Index Lookup**

- Es müssen nicht alle Datensätze gelesen werden, weil ein Index einen gezielteren Zugriff erlaubt

- **Unique index scan**

- Ein einzelner Datensatz wird über einen Index ermittelt

- **Index range scan**

- Der Index wird verwendet, um Datensätze in einem Wertebereich zu ermitteln

- **Nested Loop Join**

- Es werden nacheinander alle Tupel aus der Relation R ausgewählt und mit jedem Tupel aus der Relation S verglichen.

EXPLAIN PLAN

● Sort-Merge Join

- Beide Relationen werden nach ihren Join-Attributen sortiert. Das Ergebnis kann dann mit einmaligem Scan durch beide sortierte Relationen berechnet werden.

● Hash Join

- Berechnet für jeden Datensatz in der kleineren Tabelle einen Hash-Wert. Liest dann sequenziell die andere Tabelle und berechnet für diese Datensätze mit der selben Hash-Funktion die Hash-Werte. Die Entsprechungen der Hash-Werte (=JOIN) können dann anschließend unmittelbar ermittelt werden, ohne jeden Datensatz der ersten Tabelle mit jedem Datensatz der zweiten Tabelle vergleichen zu müssen (wie beim Nested Loop Join).

● Cartesian Product

- Ist die Gegenüberstellung aller Datensätze der Tabelle 1 mit allen Datensätzen der Tabelle 2 (jede Kombination), weil kein JOIN im SQL-Statement definiert wurde.

EXPLAIN PLAN

● Cache-Effekt

- Die Ausführung von SQL-Abfragen führt eventuell zur Ablage der Daten im Hauptspeicher, so dass aufeinanderfolgende, gleiche Abfragen nicht nochmal geladen werden. Dadurch kann die zweite Abfrage schneller laufen als die erste und die Performance-Messung verfälschen.
- Die Caches können ebenfalls optimiert werden, indem sie auf die Nutzung spezialisiert werden: High Level DB-Tuning

- **Buffer Cache:** speichert Tabellen, Indexes und andere relevante Daten. Leeren mit: **ALTER SYSTEM FLUSH BUFFER_CACHE**

- **Shared Pool:** speichert eine Reihe von Informationen (z.B. SQL- und PL/SQL-Statements).
Leeren mit: **ALTER SYSTEM FLUSH SHARED_POOL**

Die Ausführung in einer Produktivumgebung kann schwere Folgen haben!
Performance-Tuning findet deshalb immer auf einem **TESTSYSTEM** statt.

Übungsaufgabe

- Explain Plan
 - Abfrage: Geben Sie die Namen aller Länder aus, die jeweils mindestens einen Berg, einen Fluss und einen See haben und an ein Meer grenzen.
 - Formulieren Sie die Abfrage auf unterschiedliche Arten und erstellen Sie jeweils einen Explain Plan. Erstellen Sie dabei auch komplizierte und ineffiziente Varianten.
 - Vergleichen und Diskutieren Sie die Ergebnisse.

Zusammenfassung (1/2)

- Ziele und Ebenen des DB-Tuning
 - Verkürzung von Antwortzeiten
 - Erhöhung des Durchsatzes
 - Größtes Potential: Optimierung SQL-Statements und Indexes
- Manipulation der Relationsschemata
 - Normalisierung / Denormalisierung
- Verwendung von Indexstrukturen
 - Index, Primärindex, Sekundärindex
 - B-Bäume, Hashing

Zusammenfassung (2/2)

- Verwendung von MATERIALIZED VIEWS
 - Dauerhaft gespeicherte Datenkopie
 - Performance abhängig von Lese- / Schreibzugriffen
 - Gezielte Steuerung der Aktualisierung
- EXPLAIN PLAN
 - Mittel zur Auswertung der SQL-Statements
 - Aufruf
 - Interpretation der Ergebnisse
 - Caches