10 Physical schema design

10.1 Introduction
- Motivation
- Disk technology
- RAID

10.2 Index structures in DBS
- Indexing concept
- Primary and Secondary indexes

10.3. ISAM and B⁺-Trees

10.4. SQL and indexes
- Criteria for indexing
- Height of B⁺-Tree

Lit.: Kemper/Eickler: chap 7, O'Neill: chap. 8, Garcia-Molina et al: chap. 13
Kifer et al.: Chap 9.

10.1 Physical Design: Introduction

Physical schema design goal: **PERFORMANCE**

Quality measures

- **Throughput**: how many transactions / sec?
- **Response-time**: time needed for answering an individual query

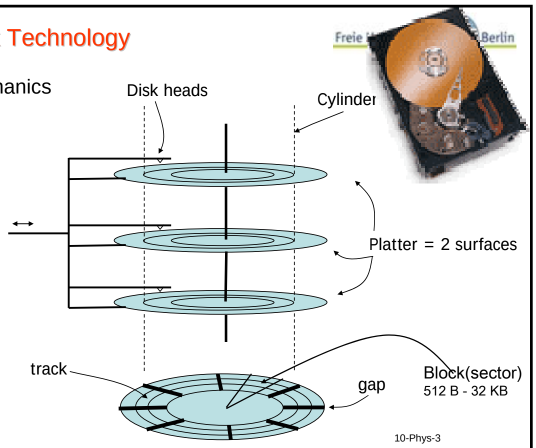
Important factors for defining a "good" physical schema

Application
- size of database
- typical operations
- frequency of operations
- isolation level

System
- storage layout of data
- **access path**, **index structures** ⇐

Disk Technology

Mechanics



Physical Design: I/O cost

Disks are slow!

Data **transfer time** disk - main memory
- Blocks
- Bytes transferred at constant speed
- Transfer rate (tr): * 300MB/s (2010, SATA techn.)

- **Seek time**:
  - Time for positioning the arm over a cylinder/track
  - Move disk heads to a particular cylinder/track: Start (constant), Move (variable), Stop (constant)
  - 0 if arm in position, otherwise long (between 8 to 10 ms)
  - Track-to-track seek time: 0.5ms –2ms

Physical Design: I/O cost

Rotate time (disk latency):

Time until sector to be read positioned under the head
Access to all data within a cylinder within rotate time
12 to 6 ms per rotation / 5000 – 12000 rotations per min
Average: 4,17 rotational latency. (Seagate Baracuda 1TB)
⇨ store **related information in spatial proximity**

Time to read T bytes with transfer rate tr:

Seek time + Rotational time + T/tr

Physical Design: I/O cost

Typical mean access time:

Disk access time =	SeekTime	6 ms
	+ RotateTime	3 ms
	+ TransferTime	1 ms

Seek time dominates !

- Random Disk / RAM:
  $\sim 10 * 10^{-3} / 200 * 10^{-9} = 5*10^{4}$

- Sequential disk read ("scan") may be much faster

Disk parameters

Freie Universität Berlin

Drive capacity and data rate

- Drive capacity increases much faster than transfer rate and access time

year	Capacity GB	rate MB/sec	a. time (msec)	Block size KB	Scan Sequential	Scan Random
1988	0,25	1	20	0.5	4 minutes	16 min
1998	18	10	12	2	30 minutes	3 hrs
2005	250	50	10	2 - 4	1.5 hrs	1.3 days

Disk arm is the limiting resource.

adapted from Jim Gray / D. Bitton 1998

Basic facts summarized

Freie Universität Berlin

RAM / disk **gap will remain**

High increase in storage density
⇨ **Disk space is free** (more or less)

Access time and data rate (seek, rotation) improve much slower
⇨ **reading / writing large quantities of data becomes a crucial problem**

Large capacity disks have one actuator
⇨ throughput bottleneck

RAID storage

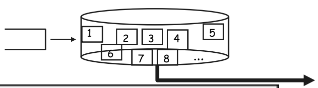
Freie Universität Berlin

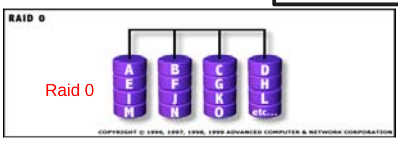
RAID Technology (Redundant Array of Inexpensive Disks)

Goals

- **Performance enhancement** by reducing transfer time and queue length
- **Fault tolerance** by "Parity disks"

Large disk ⇒
Long queue,
Long transfer





Principle technique: **striping**

Block striping,
no fault tolerance

(cited from http://www.raid.com)

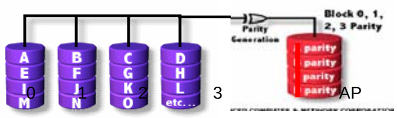
Physical Design: RAID

Freie Universität Berlin

RAID 4 : **reconstruct data by parity disk**

AP [1] = A[0] ⊗ B[1] ⊗ C[2] ⊗ D[3]
D[i] = A[0] ⊗ B[1] ⊗ C[2] ⊗ AP [1] etc

Independent Data disks with block striping and shared Parity disk

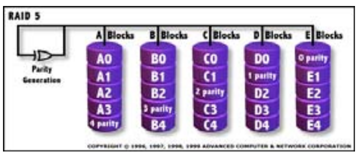


RIAD

Freie Universität Berlin

RAID 5 : **avoid parity disk bottleneck**

Independent Data disks with distributed parity blocks



~ state of the art, many minor modifications

Technological Impact

Freie Universität Berlin

RAID controller provides OS / DBS with standard disk interface

Considerable performance gains for read operations

Writes need recomputation of parity
⇨ Main reason for parity disk bottleneck in RAID-4 architecture

Further info: http://www.raid.com

Solid state disks?

10.2 Indexing in DBS

An **index** is a **data structure** which allows to locate an objects faster than by sequential scan.

- Well known: binary search tree , hash maps. Data: (key, value)-pairs.
- Traversing a search tree is efficient, if node are **in memory**

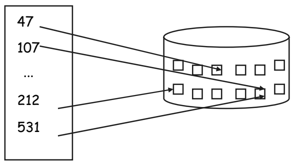
Primary and Secondary indexes

Primary (unique) index

A mapping from **key values** to **records** (tuples)

Typically used for indexing **PRIMARY KEY** or one **UNIQUE column**

Typically assigns a **physical location** to each record.



More than one record (key)on a disk page, one entry for each key ("dense index")

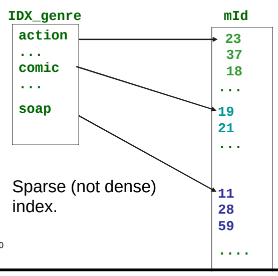
Secondary index

Secondary index on attribute a of table T:

Assigns to each value v of a the set of rows t with t.a=v

Example: Movie database

```
Movie (mId, title, genre, ..., director,...)
```



Logical view:
- Each value **v** of the attribute **a** references a list of tuples **t** with `t.a = v`

Sparse (not dense) index.

Indexing

Goal of **DBS architect** and implementor*:*

Find efficient data structure for indexing arbitrary data (B-tree, R-tree, Hashing, ...?)

Goal of **Database designer**:

Define index for database Schema in order *to increase performance*. Use one of the implementations supplied by DBS and create an index for some or all tables.

Types of indexes and index definition

CREATE INDEX

Most simple case

```
CREATE INDEX movie_idx1 ON Movie (cat );
CREATE INDEX customer_idx1 ON Customer (name, first_name);
```

- **Composite index** is defined on multiple columns
- Different (search tree) indexes on the same columns with different orders sometimes make sense - e.g. abc and bca. Why?

```
CREATE INDEX customer_idx2 ON Customer(first_name,name);
```

Decision which indexes to create is an important task in physical schema design

Defining indexes

Why not index each attribute?

Advantage: fast predicate evaluation

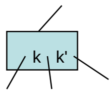
```
Select x from R where y = val
```

Disadvantages: they are not for free
- **Redundancy**
  - Space, can double the space needed for the DB
  - Extrem case: all attributes are indexed: do we need rows at all? ⇒ ... "Column stores"
  - database = set of indexes, no tuples !?
- **Operational cost** in case of updates
  - insertion / deletion / of a row: each attribute effected by the operation has to be updated (delete, insert: all attributes)
  - each index write implies disk I/O – expensive!

Disk based Search Trees

Search trees well known,
e.g. binary Search Tree, (2,3) – trees, Red Black trees

Big issue of ST for Databases:



(1) Trees may **degenerate** ⇒ (2,3) trees, balanced!
i.e. same height h of all leaves

(2) nodes in RAM vs **nodes on disk** ⇒ traversing a disk based tree is time consuming.

⇒ cost measured in Number of disk access, not number of constant time operations !

Implementing indexes

Index sequential (ISAM)

tree like index with a **fixed number of levels** (2-4)

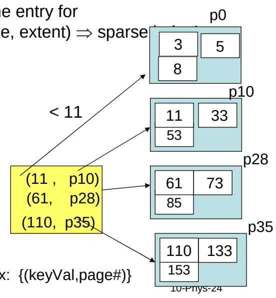
records stored **sequentially**

index on lowest level contains one entry for each record storage area (page, extent) ⇒ sparse

Example: one-level index

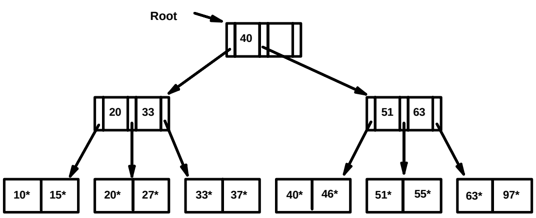
Simple idea, efficient,...

.. but what happens in case of insertion or update ?



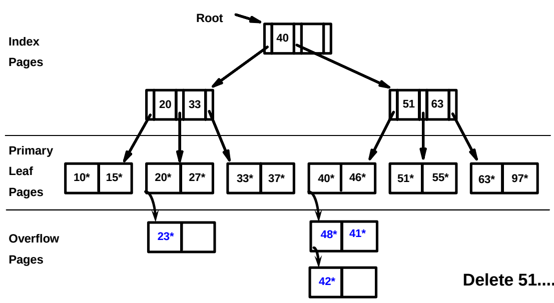
Index: {(keyVal,page#)}

ISAM example, 2 index levels



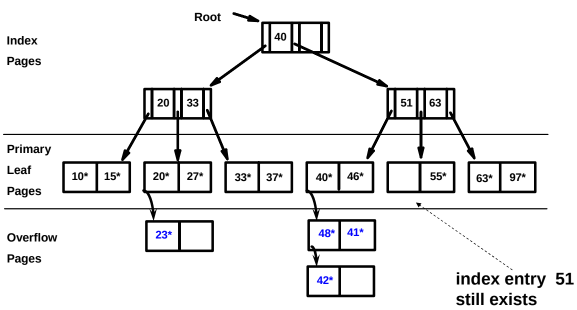
Insert 23*, 48*, 41*, 42* ...

ISAM overflow



Delete 51....

ISAM deletion



ISAM

Index **"Sequential"** since records may be **read in key sequence**

Operations

lookup of **key k**: straight forward

delete:

lookup; set delete bit or remove (in leaf, not inner nodes)

insert:

lookup;

if sufficient space insert else insert into **overflow bucket**

ISAM

Insertion / deletion only affects leaf pages

Main **disadvantage of ISAM** organization:
no dynamic adaptation to growing and shrinking files, periodical reorganization needed.

Index setup algorithm?

B^+-Tree

Base requirement:

- node size = disk page (as before)
- no performance degradation: **balanced search tree**
- **Rebalancing** in case of inserts should be "easy"

Additional characteristics of B^+ **trees:**

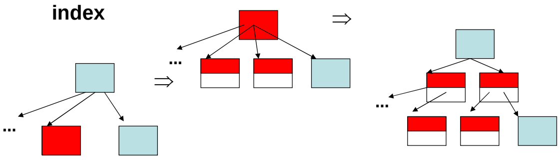
- no data in inner nodes but only keys and pointers like ISAM
- Data (records) only in leaf pages

⇒ Sequential key sequence access enabled if leafs are chained and search tree property

B^+-Tree

Basic idea of B- and B^+-trees:

Dynamically growing and shrinking tree-structured index



temporary state

Very popular, implemented in most DB systems

Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica Vol 1,173-189 , 1972

B+ - index trees

inner node: **35 40 50 53**

k=2, # keys ≤ 4
3 ≤ child# ≤5

Characteristics

- inner node (*except root*) has k ≤ t ≤ 2k keys and t+1 child nodes, **degree k** B+-tree.
- **Search tree invariant**: Subtree "between" keys s_i and s_{i+1} stores all data with key s: $s_i \le s < s_{i+1}$
- **All leaf nodes have depth h** ⇒ height of the tree
- B^+-property: **(key, value) pairs in leafs**, not in inner nodes

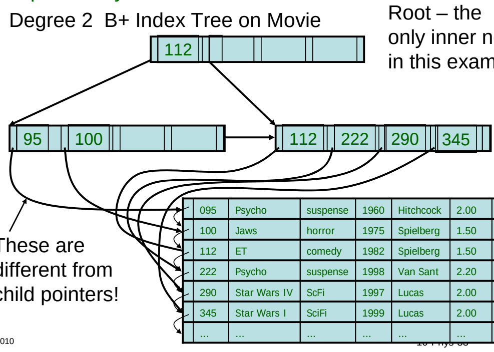
tuples(records) in DB context

B+ -Trees

Example: a very small B+-Tree:

Degree 2 B+ Index Tree on Movie

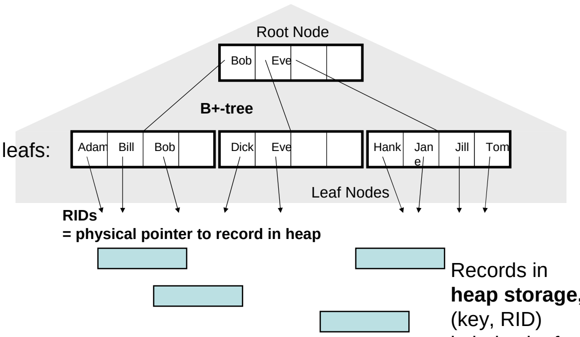
Root – the only inner node in this example (!)



095	Psycho	suspense	1960	Hitchcock	2.00	...
100	Jaws	horror	1975	Spielberg	1.50	...
112	ET	comedy	1982	Spielberg	1.50	...
222	Psycho	suspense	1998	Van Sant	2.20	...
290	Star Wars IV	ScFi	1997	Lucas	2.00	...
345	Star Wars I	SciFi	1999	Lucas	2.00	...
...	...	...	...	...	...	...

These are different from child pointers!

B^+-Tree: example

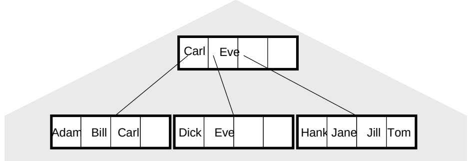


Records in **heap storage,** (key, RID) in index leafs

Following examples by Weikum/Vossen

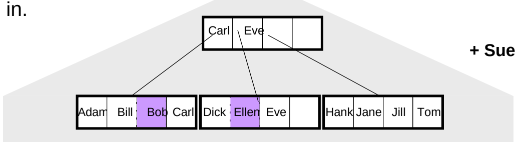
Simple Insertion into B+-Tree Index

Freie Universität Berlin



Space left for keys in the leaf, a key should be in.

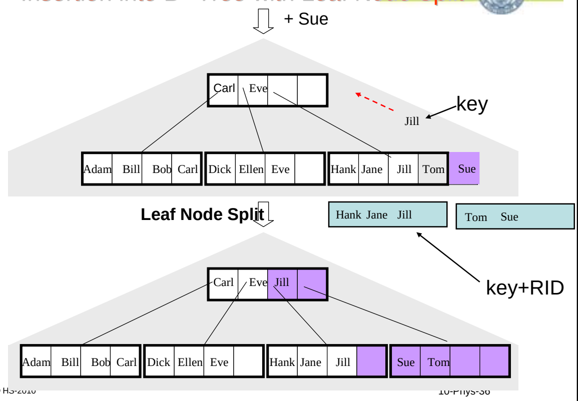
+ Ellen, + Bob



+ Sue

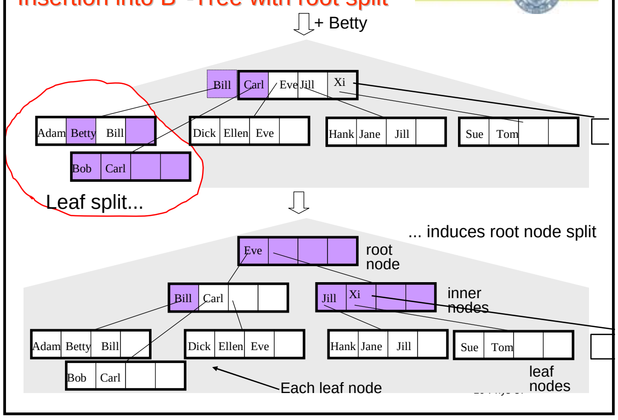
Insertion into B+-Tree with Leaf Node Split

Freie Universität Berlin



Insertion into B+-Tree with root split

Freie Universität Berlin



B+ tree insertion

Freie Universität Berlin

```
boolean insert(key, recPtr, nodePtr) {
 if (! leaf(nodePtr)) // always insert in leaf
    insert (key, recPtr, findChild(key)) //recursive traversal
else // we have reached a leaf
 {if (space_enough) insertInLeaf (key,recPtr, nodePtr);
   else {   //split
      splitkey = splitNode(left, right); // allocate
                               //a new page and distribute keys
     if( key<=splitkey) insertInLeaf(key, recPtr,left);
     else insertInLeaf (key, recPtr,right);
     insertSplitKey(parent.nodePtr,splitkey,leftPtr,rightPtr);
  }
 }
```

insertSplitKey inserts splitkey and pointer to allocated page into parent node – if space available. Else split the inner node, insert splitkey and apply **insertSpitKey** recursively.

B+-Tree: real world

Freie Universität Berlin

Deletion

- may cause underflow (< k keys in node)
- "join" two neighbor pages – inverse operation to page spit.
- avoid unstable behaviour (delete-insert-delete-...): postpone join until only k-*delta* keys in node

B+ trees: real world

Freie Universität Berlin

Page occupancy

Keys often have variable length (strings!)
⇒ replace k ≤ # keys ≤ 2*k by:

Node (= disk block) should be at **least 50%** full.

Fanout:
number of childs – the more the better
Compress keys in order to increase fanout.

... c..

(abel abend aber) (dart data database)

10.3 Criteria for physical schema design

Design parameters for physical schema

Data volume:

- how many records and pages in a relation?
- how many leaves in the tree, how many inner node

Depends on

- The way, rows are stored in pages
- how pointers to rows ("tuple ids") are implemented
- how index pages are organized

Typical load:

which query / update types (the hardest part!)

Which attributes to index? Which type of index?

Physical Design: criteria

Which kind of Index?

- B+ tree and variants as a standard index type
- Clustering: storing related data in physical neighborhood

Physical I/Os

Number of page accesses is the most important cost measure

Depends on height of the tree...
and buffering, e.g. root of an index is always in RAM

How to calculate the height?

Performance

How many disk accesses to fetch a record?

Assumptions:

- n = number of records: 1000000
- r = average record size: 80 B
- b = effective page size without header: 4000 B
- ptr = Pointer size: 4 B, tid = TID / (RID) size: 6 B
- k = average key size: 10 B
- a = average node fill degree (both inner and leaf) 0.8

$eLeaf = \lfloor (b / (k+tid)) * a \rfloor$ # entries (max) per leaf,

$Ln = \lceil n / eLeaf \rceil$ = # leaf pages

Inner nodes: $i = \lfloor (b/ (k+ptr)) * a \rfloor$ # (key, ptr)-entries

Performance

Height (including leafs):

$$1 + \lceil \log_i Ln \rceil = 1 + \lceil \log_i \lceil (n / eLeaf) \rceil \rceil$$

Example: $1 + \lceil 1.56 \rceil = 3$

Root in memory ⇒ effectively $\lceil \log_i L(n) \rceil$ accesses

How to reduce disk accesses?

increase fan-out: larger blocksize, compression

store **records in leaf-pages** (instead of tids)

Summary

Data stored on disk

Access time crucial in query processing

I/Os is THE cost measure

Access Time: Seek time + Rotational time + Transfer time

Indexes accelerate access to secondary storage

B+ tree is standard in most DBs

Great differences in physical organization in DBS

Indexing (SQL interface) not standardized
(except `CREATE INDEX...`)