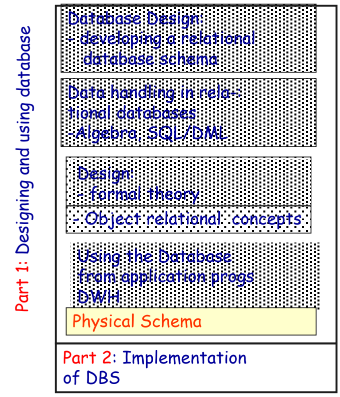
# 13 Physical schema design

- 13.1 Introduction
- 13.2 Technology
  - 13.2.1 Disk technology
  - 13.2.2 RAID
- 13.3 Index structures in DBS
  - 13.3.1 Indexing concept
  - 13.3.2 Primary and Secondary indexes
  - 13.3.3 Types of indexes and index definition in SQL
  - 13.3.4 Implementing indexes: search trees
  - 13.3.5 Criteria for indexing
- 13.4 More index structures
  - 13.4.1 Clustered indexes
  - 13.4.2 Implementation of rows and tables
  - 13.4.3 B+ trees with data leafs
  - 13.4.4 Bitmap indexes
  - 13.4.5 Hash index and inversion
  - 13.4.6 Case study ("Video store")
- 13.5 Multi dimensional indexes

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

## Context






## 13.1 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 quality of physical schema
  - Application
    - size of database
    - typical operations
    - frequency of operations
    - isolation level
  - System
    - storage layout of data
    - access path, index Structures



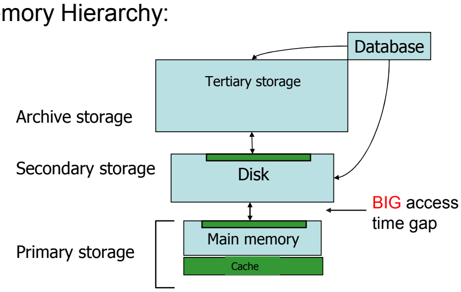
## Physical Design: performance parameters

- System related performance parameters
  - Logging / recovery
  - Blocksize of (DBS-) storage (2 , …, 8KB,…)
  - Size of DB buffers
    i.e. main memory areas (global, user specific)
  - Parallel processing
  - Distribution
  - Query optimizing strategies
  - …. and many more
- Schema related physical parameters
  - e.g. Size of tables (initially),
  - Most important: Indexes



## Physical Design: Storage Devices

- Memory Hierarchy:

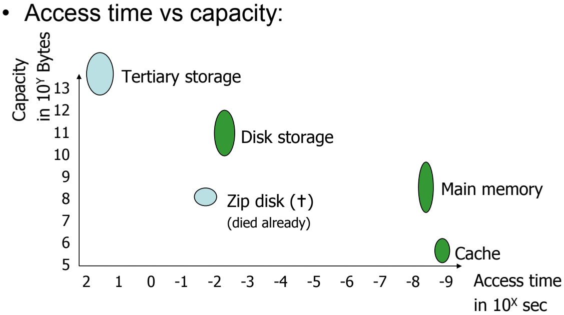



Locality of references ⇨ apply cache principle



## 13.2 Physical Design: Storage Devices

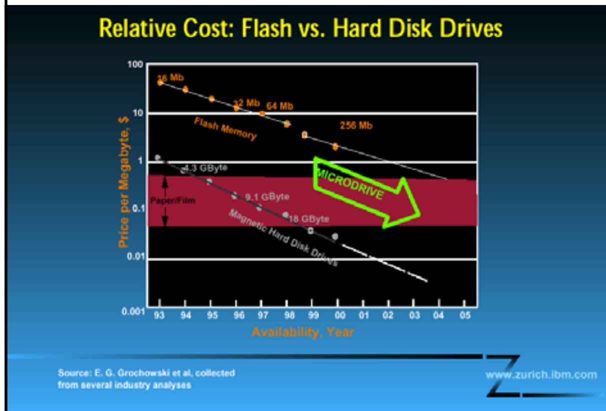
- Access time vs capacity:




Source: Garcia-Molina, Ullman, Widom "Database systems", 2002

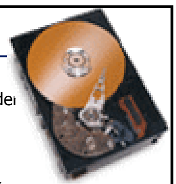

Evangelos Eleftheriou: Millipede - a Nanotechnology Approach to Data Storage

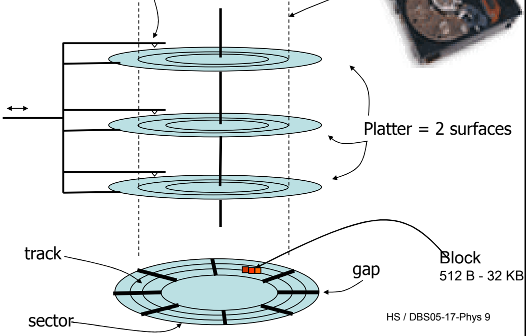





## 13.2.1 Disk Technology

- Mechanics








## Physical Design: I/O cost

- Disks are slow
- Data transfer disk - main memory
  - Blocks
  - Bytes transferred at constant speed
  - Transfer rate (tr): between 120 KB/s and 5 MB/s
  - Seek time:
    - Time for positioning the arm over a cylinder
    - Move disk heads to the right cylinder: 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: 6 to 3 ms rotational latency.
- ⇨ store related information in spatial proximity

Transfer time tr (read time):

- Depends on # bytes to be transferred

Total time to transfer T bytes:

Seek time + Rotational time + T/tr



## Physical Design: I/O cost

- Typical access time:

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

Seek time dominates !
Compare: RAM 3-10 nsec

- Random Disk / RAM:
  - ~10 * $10^{-3}$ / 10 * $10^{-9}$ = $10^{6}$
- Sequential disk read ("scan") may be much faster

## Technological Impact    Disks

- Disk characteristics (J. Gray)

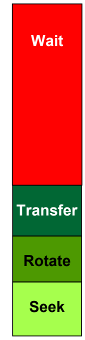
| year | Capacity GB | $/GB | Scan Sequential | Scan Random |
|---|---|---|---|---|
| 1988 | 0.25 | 20,000 | 2 minutes | 20 minutes |
| 1998 | 18 | 50 | 20 minutes | 5 hrs |
| 2003 | 200 | 5 | 2 hrs | 1.2 days |

- Consequence: Random access (and indexing!) only pays off, if a small percentage of the data is accessed frequently
  rule of thumb: less than 15 % on a large table
- Cost of indexing?



## Technological Impact    Disks

- Disk characteristics (2) (J. Gray)
- The Myth: seek time dominates
- The Reality: (1) Queuing dominates
  (2) Transfer dominates BLOB
  (3) Disk seeks often short
- Implication: many cheap servers better than one fast expensive server
  - shorter queues
  - parallel transfer
  - lower cost/access and cost/byte
- Gives rise to table and index partitioning





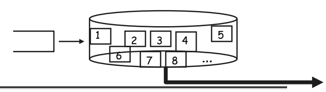

## Technology impact: I/O cost

- Accelerate secondary storage access
  - Strategies
    - Place blocks that are accessed together on same cylinder (avoids seek time)
    - Divide data between smaller disks (independent heads increase # block accesses)
    - Replicate data: simultaneous access to several blocks
    - Disk-scheduling algorithm: selects order of block access
    - Prefetch blocks in main memory
  - Disk architectures can enhance disk access considerably

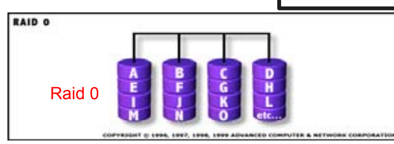


## 13.2.2 RAID storage

- 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

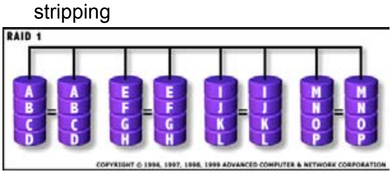
Block striping,
no fault tolerance

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

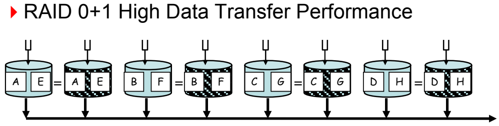


## Technology: RAID

- RAID 1 Mirroring and Duplexing: mirror without stripping



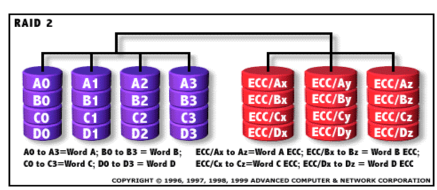

- RAID 0+1 High Data Transfer Performance






## Technology: RAID

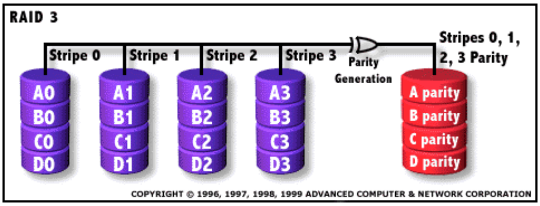
- RAID 2 Byte (Bit) level striping + error correcting disks




Each bit of data word is written to a data disk drive (4 in this example: 0 to 3). Each data word has its Hamming Code ECC word recorded on the ECC disks. On Read, the ECC code verifies correct data or corrects single disk errors.

## Physical Design: RAID

- RAID 3 Bit (Byte) level striping with parity



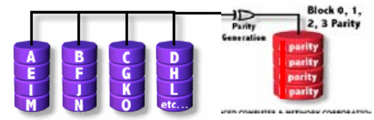

AP [1] = A[0] ⊗ A[1] ⊗ A[2] ⊗ A[3]

Data online reconstructable, when ONE disk fails

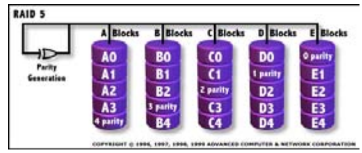


## Physical Design: RAID

RAID 4 Independent Data disks with block striping and shared Parity disk




RAID 5 Independent Data disks with distributed parity blocks






## Technological Impact Disks

- 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



## 13.3.1 Indexing in DBS

### Index

Important

- Optional data structure for fast access to data items ….in the DB
- Index $I_a$ assigns to each value v of a the set of data objects

> $I_a$:: $Val_a$ -> POWERSET(D)
> $Val_a$ = set of values of attribute a
> D = {d1, ... dn)} set of data objects

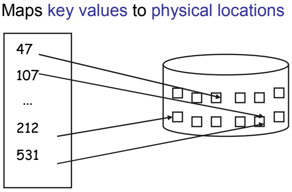
- Locates the rows of a table having v as value of attribute a in an efficient way
- May be extended to attribute / value sequences:
  $I_{ab\ldots c}$::$Val_{a,b,\ldots,c}$ -> POWERSET(D)
- Disk based data structure



## 13.3.2 Primary and Secondary indexes

### Primary (unique) index

- For each v ∈ $Val_a$, there is at most one row r with r.a=v
  i.e. | I(v) | ≤ 1
- Typically used for indexing PRIMARY KEY or one UNIQUE column
- Important: Maps key values to physical locations




More than one key in a disk block (page)

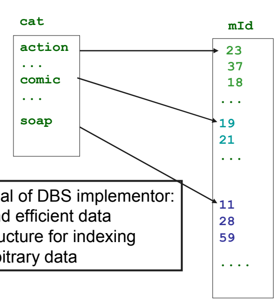
- Indexes on other attribute (sequences) are called secondary keys, even if unique



## Secondary index

- In most cases not unique
- Example: Movie database

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




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

Goal of DBS implementor: Find efficient data structure for indexing arbitrary data

Goal of DB designer: Define index for database Schema in order to increase performance. Use one of the implementations supplied by DBS

## 13.3.3 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 needed, can double the space needed for the DB
    - Extrem case: all attributes are indexed: do we need rows at all?
    - 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!



## 13 Physical schema design



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

## Types of indexes

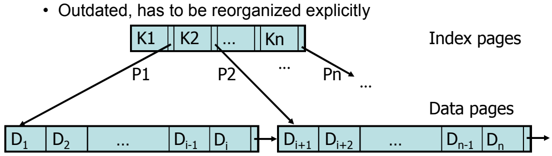
- Hash Index
  - Same as well known hash functions
    h :: Val -> {0,…….n} ("map values to disk block numbers" )??
    - Useful only for unique values (hash collisions!)
    - No key sequential access to rows
    - Reorganisation needed when size of table increases considerably
- Bitmap Index
  - Stores for each value v of field a and each row i a bit b(v,i) -- true, if i has value v in field a
- Cluster Index
  - Store "logically related data" in physical neighborhood
- Search Trees



## 13.3.4 Implementing indexes: search trees

Hierachical index trees (search trees)

- ISAM (Index sequential Access method)
  - Index blocks for physical areas (cylinder, track, sector) keep (lowVal – highVal) pairs for each cylinder ("cylinder index"), track ("track index") etc.
  - "sequential" since rows may be read in key sequence
  - Outdated, has to be reorganized explicitly



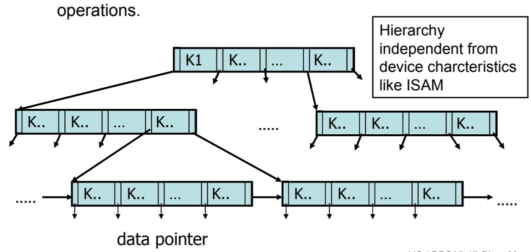

Keys Ki, Data tuple Di, Pi pointer to data $D_j$: $K_{i-1} < D_j.key \le K_i$



## Index implentation: B-Tree

$B^+$-Trees: the standard for most DBS *) Important

- like B-Trees, but inner nodes contain only keys and pointers
- Sequential key sequence access is possible
- "self-reorganizing" because of implementation of update operations.

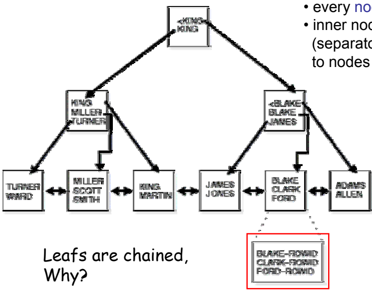





*) Sometimes called B* -trees (Bayer- | Boeing-tree ?)

## Index implementation

B⁺-tree



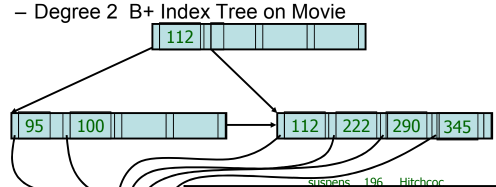

- all leaves on the same level
- every node is a disk page
- inner nodes contain n-1 (separator) keys and n pointers to nodes
- the search tree invariant holds for all (prt, key, ptr) – tripels in inner node and root
- all nodes below the root are at least 50% filled
- leaf nodes contain (keyval, rowid) pairs

Leafs are chained, Why?



## B+ -Trees

- Example:
  - Degree 2 B+ Index Tree on Movie

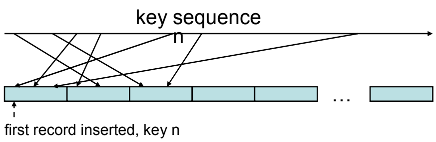



| | | | | | | |
|---|---|---|---|---|---|---|
| 095 | Psycho | suspense | 1960 | Hitchcock | 2.00 | ... |
| 100 | Jaws | horror | 5 | g | 1.50 | ... |
| 112 | ET | comedy | 2 | g | 1.50 | ... |
| 222 | Psycho | e | 8 | Sant | 2.20 | ... |
| 290 | IV | ScFi | 7 | Lucas | 2.00 | ... |
| 345 | I | SciFi | 9 | Lucas | 2.00 | ... |
| ... | ... | ... | ... | ... | ... | ... |

## Data storage

- Heap storage

  Storage area into which records stored in "time sequence" not key sequence




ISAM: Sequential storage of data
B+ tree with row pointers: random placement of data



## 13.3.5 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

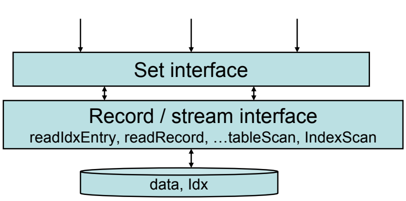
  Depends on
  - The way, rows are stored in pages
  - how pointers to rows ("tuple ids") are implemented
  - how index pages are organized
- Typical load: which query / update types (the hardest part!)
- Kind of Index
  - B+ tree variants as a standard
  - Clustering: storing related data in physical neighborhood
- Physical I/Os, if number of page access is the most important cost measure



## Physical schema design

### Random versus sequential access: case study

- Typical task: read n random rows of table
- Index based access: for each record: read block which contains one or more tuples
- Table scan: read all blocks sequentially and extract the records ("on the fly")




Query processing architecture



## Sequential versus direct access

Example:
table: 10000 pages (blocks) of 1 KB
task: read 200 records, each in a different block

- Read 200 records = read 200 pages = 200 * 12 msec **~ 2 sec**
- Table scan = sequential read of 10000*1KB = 10 MB **~ 10 MB / 5 MB/sec = 2 sec**

  Block access time: 12 msec, Data transfer rate = 5 MB/sec
- read 600 records ⇨ factor 3 in favour of scan

Sequential access more cost effective (in this case….)!

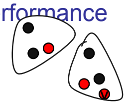
Question: how many blocks have to be read when reading n tuples?

## 13.4.1 Clustered indexes

Clustering – another way to increase performance

Cluster principle

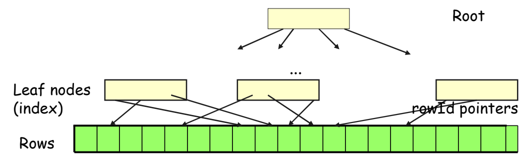
- put related data into a group (a cluster)
  - Clustering : a statistical technique to group data with similar features together.
  - No statistics available during DB design. Goal: efficient access to related ("clustered") data.
  - Reasonable application pattern: Rows of a table may be primarily accessed in value (key) sequence of one attribute





## Storage of Data Clustering

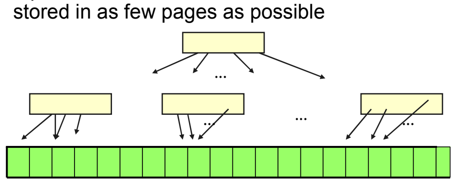
- Clustered Index
  - The sequence of row-Ids in a leaf page is normally different from the physical sequence of rows
    ⇒ Sequential index scan means random access to rows
- Heap Storage, Index without clustering






## Storage of Data Clustering

- Clustered index
  - Controls physical placement of rows
  - Obvious: only one cluster per table
  - tuples which have value v in cluster attribute a are stored in as few pages as possible




Not necessarily stored in cluster attribute sequence



## Storage of Data Clustering

- Example

  Big company with 1 Mill customers in 20 cities,

  Frequent access to all customer records (100 B) in a particular city:

```
SELECT name, location, street, no FROM customer
where location = :loc
```

  VERY Rough estimate:

  a) 50000 random access ~ $10*10^{-3}*5*10^{4}$ ~ 10 min

  b) 25000 /(rows/4K-block) sequential reads
  ~ 25000/40 * $10*10^{-3}$ = 6250 msec ~ 6 sec

  Warning: queuing and buffering neglected, gives only a rough impression of the sequential / random ratio



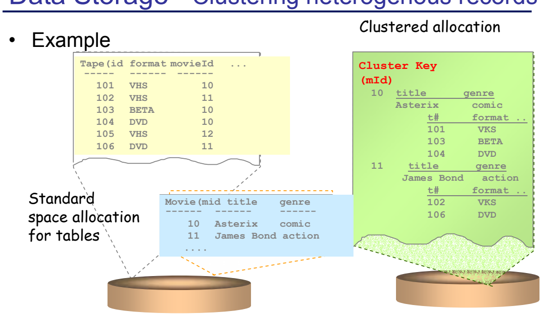
## Data Storage Clustering heterogenous records

- Clustering heterogenous objects (rows)
  - Rows of different tables may be accessed frequently together
  - Estimate the "access correlation" between different rows or tables.
    What is the probability that row y in table A is accessed, after row x in table A' has been accessed?
- Example: Video-movie DB
  Access to a Movie record is often followed by an access to a tape containing this movie.
  Tape- and movie records with the same mId - value should be placed in one block (page)
- Heterogeneous cluster: set of blocks which may contain rows of more than one table
- More general notion for "cluster"
  Be careful with different notions



## Data Storage Clustering heterogenous records

- Example




- Clustered are defined by a common cluster key ck, not necessarily primary key, but frequently ck is primary key in one table, foreign key in another

## Data Storage   Clustering heterogenous records

- Defining a cluster
  - First create a cluster

```
Create Cluster videoDB.movieTape_clu
 (mId NUMBER (6)) ;
Create Index idx on cluster videoDB.movieTape_clu;
```

  - Create a cluster index: clusters are accessed primarily through the cluster key
    -> fast access by using an index
    - B*-tree index
    - Hash cluster (Oracle allows hash-index only for clusters )
  - Finally create the tables in the cluster

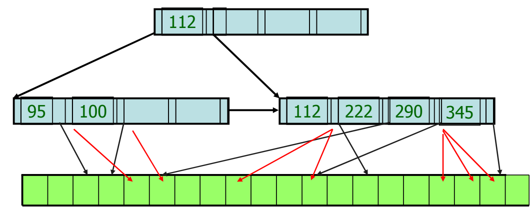
```
CREATE TABLE Movie (....) CLUSTER
  movieTape_clu(mId)
CREATE TABLE Tape (....) CLUSTER
  movieTape_clu(movieId)
```



## 13.4.2 Implementation of rows and tables

Remember…

- Unique (primary) index



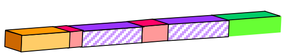

- Nonunique (secondary) index :
  more than one row for a key value



## Index implementation

### Index entry

Example shows concatenated index (two columns)



- Index entry header: number of columns, locks
- Key column length
- Key column value
- rowId (tupleId)

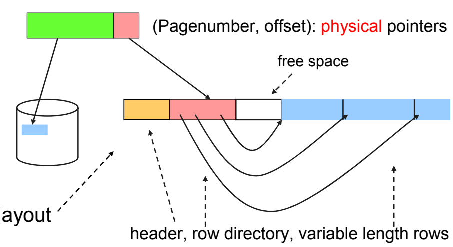
- Non-unique index – different implementations:
  - (key length, key val) pair repeated for each rowId with this key val (Oracle implementation)
  - (key length, key val) [list-of rowid] entries (DB2 )
  - (key length, key val) [list of primary keys]



## Storage of data   Rows and pages

What's in a row ID (tuple ID, TID)?

- Simplified view:

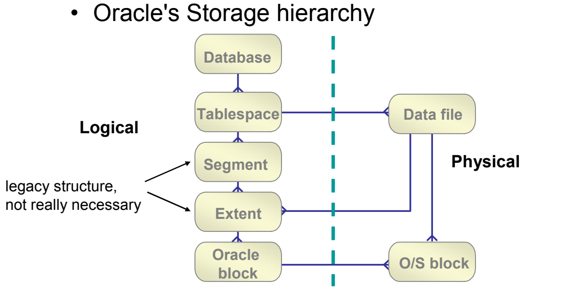



Disadvantage: uniform page address space -> large pagenumber



## Storage of data   Descreasing pointer size

- Oracle's Storage hierarchy



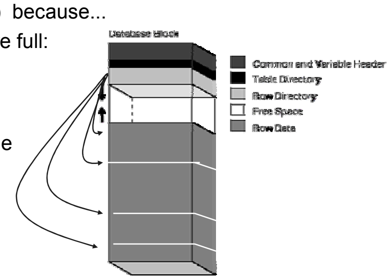

RowId: OOOOOO FFF BBBBBB RRR:

Segment No, data file No (in table space) Block in file, row in block



## Storage of Data   Anticipating growth

1. Initial loading and indexing
   - May reserve freespace (PCTFREE) and used space (PCTUSED) because...
   - if pages were full:
     small number of insertion would result in many page splits

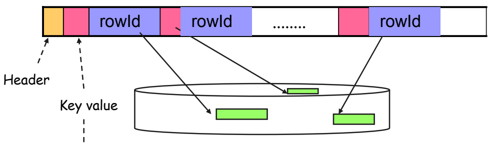


## 13.4.3 Index tree with data leafs

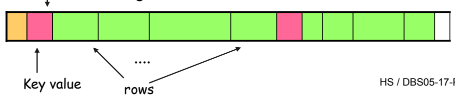
### B+-Tree index with data leaves

("Index organized tables" Oracle)

Leaf of a standard index: additional I/O to access row




Leaf of "index organized table" : row in leaf node instead of rowId

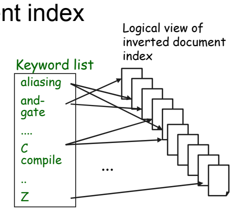





## Data Storage Index tree with data leafs

- Case study: inverted document index

```
CREATE TABLE Docindex
   ( keyword   CHAR(20),
     doc_id    NUMBER,
     frequency NUMBER,
     CONSTRAINT Pk_docindex
     PRIMARY KEY(keyword, doc_id)
   )
ORGANIZATION INDEX TABLESPACE
Ind_tbs;
```




  - Note: index organization must be specified when table is created – as opposed to standard table organization
  - Standard index on `keyword` would need more than twice as much space … and would be inefficient



## Data Storage Index tree with data leafs

- Case study (cont.)

```
SELECT doc_id FROM docindex
WHERE keyword LIKE 'compile%' OR keyword LIKE
  'parse%'
AND k_frequency LT 3 ;
```

  - Processing
    - Suppose 10 million entries, keywords 'compile' and 'parse' occur in 10000 documents each
    - Standard index organization: 2 x 10000 row (random!) page accesses ⇨ 100 sec
    - Read 10 Mill entries sequentially: 16 K pages, 40 B per entry ⇨ 400 / page ⇨ 2,5* $10^4$ pages to read sequentially



## Index tree with data leafs

Compared to sequential read of leaf pages of the $B^+$ tree:

(2 x 10000)/ rows per page ~ 300 pages (assuming 4K pages, 75% filled, 40 B rows)

Secondary index on table may reduce processing time for AND queries:

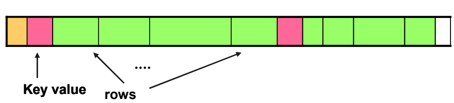
```
… keyword LIKE 'compile%' AND keyword LIKE
'parse%' ...
CREATE INDEX doc_id_idx ON docindex (doc_id,
keyword);
```



## Data Storage Index tree with data leafs

### Characteristics of index organized tables

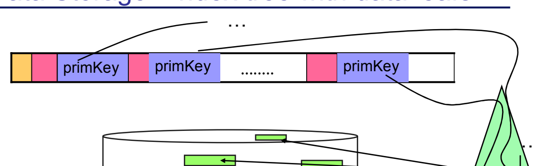
- Only primary key index




- Secondary indexes
  - No rowIds: Location of records may chance after split
  - Use primary key as "pointer"



## Data Storage Index tree with data leafs



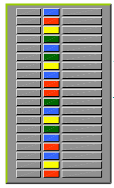

- Needs two index traversals (secondary and primary) to locate the rows
- Possible optimization in case of few updates: use current physical location as "rowId-guess".
- Space reduction, key value is not repeated in row data, no pointer (rowID) in leaf pages
- Very good performance properties if key is long (e.g. several attributes) and row is short to medium, otherwise frequent splits

## 13.4.4 More on indexes

### 13.4.4 Bitmap Index

– Less space for rowids, if few different values in a large table

Table

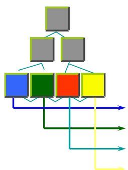


File 3
Block 10

Block 11

Block 12

Index



ORACLE

```
           start     end
key        ROWID     ROWID      bitmap
<Blue,     10.0.3,   12.8.3,    1000100100010010100>
<Green,    10.0.3,   12.8.3,    0001010000100100000>
<Red,      10.0.3,   12.8.3,    0100000011000001001>
<Yellow,   10.0.3,   12.8.3,    0010001000001000010>
```

Segment relative block, row, file



## Physical Schema More on indexes

- Operations on Bitmap indexes
  - Efficient implementation of set operations
  - Example:

```
SELECT x,y,z FROM people
WHERE (color = 'Blue' OR color = 'Red' )
AND       sex = 'm'
```

```
( <Blue, 10.0.3, 12.8.3, 1000100100010010100>
                                                   OR
  <Red, 10.0.3, 12.8.3,  0100000011000001001> )
                                                   AND
  <male 10.0.3, 12.8.3,  1010101001001001010>

  <RESULT                1000100001000001000>
```



## More on indexes

- Bitmap versus regular indexes
  - Advantage
    - If few values and many rows e.g. sex, marital status,..
    - Compression of bit lists saves space compared to standard idx
    - Efficient processing of OR / AND queries
  - Disadvantage
    - Updates expensive.... Why?
      - bitmaps must be locked during update (why?)
      - all blocks (and all rows) in a segment have to be locked
    - In comparison: one row is locked during update in a standard $B^+$-tree

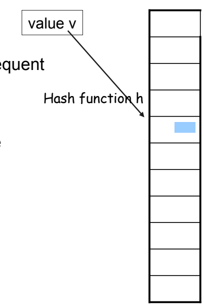
```
CREATE BITMAP INDEX customer_bidx1 ON Customer
  (sex)
    TABLESPACE myTBS PCTFREE 10;
```



## Physical Schema More on indexes

### 13.4.5 Hash index

- Advantage
  - Efficient access, if inserts infrequent
- Disadvantages
  - No sequential scan
  - No dynamic increase of space but reorganization (position is a function of initial size of hash table)
  - Range queries inefficient ('22 < val <= 1000')
  - Non unique index: retrieval has to scan the whole rehash chain – can be very long




⇒ Most DBS don't use hash as an alternative to B* trees



## 13.4.6 Physical Schema Case study

- The E-Videoshop

```
CREATE TABLE Rents (
 tapeId          INTEGER,
 cuNo        INTEGER NOT NULL,
 since      DATE NOT NULL,
 back       DATE,
 PRIMARY KEY (tapeId,since),
 ….);
```

5000000 Rents

```
CREATE TABLE Movie (
  mId          INTEGER PRIMARY
KEY;
  title        VARCHAR(60) NOT
NULL,
  category     CHAR(10),
  pricePDay    DECIMAL(4,2),
  director     VARCHAR(30),
  year         DATE,
```

1 Mio Movies

```
CREATE TABLE Tape (
 id      INTEGER PRIMARY KEY,
acDate  DATE,
format  CHAR(5) NOT NULL,
movieId INTEGER NOT NULL UNIQUE
 );
```

3 Mio Tapes

Find a suitable physical schema



## Physical Schema Case study

Data volume

- Rents: ~ 20 B / row, ~100 MB -> 2,5 * $10^4$ pages à 4KB
  + PCFREE = 30% -> 3,3 $10^4$ pages
  High update frequency, high growth rate
- Tape: ~ 20 B / row, ~ 60 MB
  -> 1,5 * $10^4$ + 30% = 2*$10^4$ 4 KB pages
  Low update frequency, high read load, medium growth
- Movie: ~ 100B / row (average), ~100 MB
  -> 2,5 * $10^4$ Pages + 30% = 3,3 * $10^4$ pages
  low update frequency, high read load, medium growth
- Extremely simplified: customer and other relations not considered

## Physical Schema    Case study

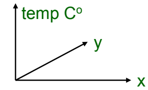
- Typical operations
  - Rent a tape: access customer (by name or id), access tape (tape-id – is printed on the tape), access Movie (mId) to get the price? Insert into Rents table
    High frequency (10 / minute ?)
  - Browse the movie table (category | director | year)
    Very high frequency
  - Query a specific title
    Very high frequency
  - Return a tape: access Rents table, access Movie table to calculate the price, update Rents
    High frequency
  - Insert new rows into Movie and Tape table
    low frequency (20 / day?)

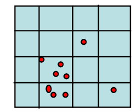


## 13.5 Multidimensional indexing in a nutshell

- Interpretation of attributes as coordinates of n-dimensional space

Example:



tuple = point in n-dim space

Basic issues:
- preserve topology – neighbors in data space -> neighbors in storage (index)
- density of objects in data space very different

Why not 1-dimensional indexes?



## Query types

Query types:

exact match query: (point query) $Q \equiv D1=a \wedge D2 = v \wedge ..$
-- all dimensions specified

partial match query: $Q \equiv D1=a \wedge D2 = v \wedge ..$
-- k < n dimensions specified

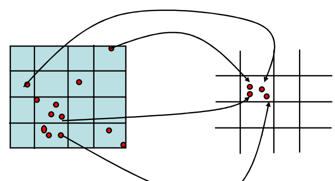
range query: $Q \equiv a1 <= D1 <= a2 \wedge v1 <= D <= v2 \wedge ..$
-- find all records in a particular range

Nearest neighbor: Q(p) = { r | distance (p,r) = min}
-- find the record(s) with minimal distance from $p=(a_1,a_2,...,a_n)$



## Independent Hash functions

$h(a1, a2,..an) = h_1(a1') \mid h_2(a2') \mid .. h_n(an')$



n = 2

Efficient for exact match queries, but…

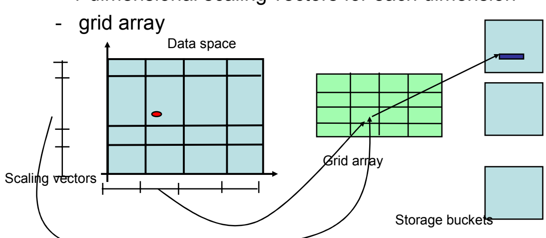
-not topology preserving
- partial match: inefficient
- range query, nearest neighbor: impossible



## Grid File

Organize data space

- partion data space into non-overlapping n-dimensional hyper cubes
- 1-dimensional scaling vectors for each dimension
- grid array



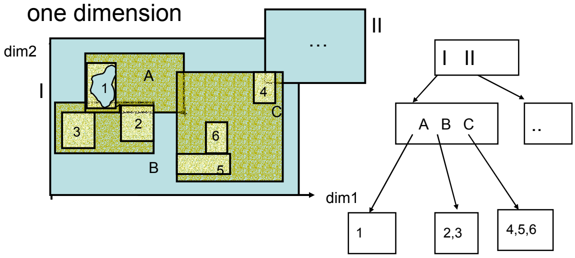



## Grid File: Search and Insertion

- Search
  determine each dimension of query in scale arrays
  ⇨ grid array entry (entries)
  ⇨ buckets with records
- Insert
  locate bucket of record to be inserted
  - if no more space
    - either overflow bucket
    - or refine partition by splitting blocks

## R-Tree: Index structure for spatial objects

- Region trees: extension of B-trees to more than one dimension




1,..6: leaf node entries, each "spatial" object is contained in minimal bounding n-dim rectangle



## R-Tree

Leaf nodes

contained in minimal bounding rectangle for the object

entry: ((x1,y1),( x2,y2), OID) -- 2-dim case

Directory nodes:

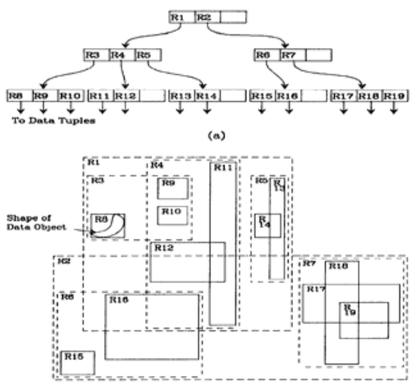
- m <= Number of entries <= M

entry ((x1,y1), (x2,y2),child-ptr)

all entries in subtree "child-prt" are contained in rectangle (x1,y1), (x2,y2)

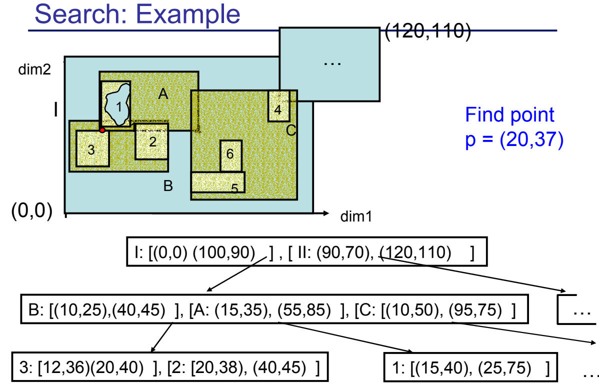
- All leaves have the same depth






See paper by Guttmann (1987) -> "Unterlagen")

## Search: Example

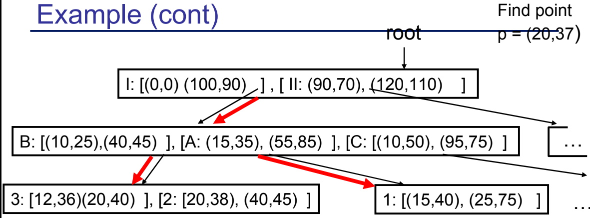



Find point
p = (20,37)



## Example (cont)

Find point
p = (20,37)




Check each rectangle r in each leaf node which may contain p if p ∈ r: 1, 2, 3 , 3 contains the point



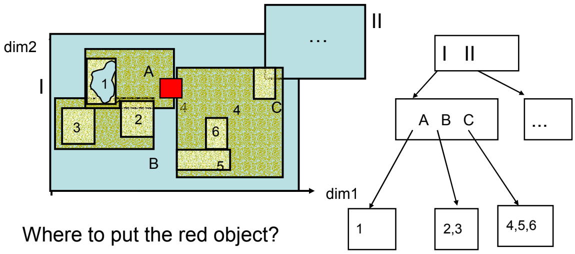
## R-Tree: Search algorithm

Point query: given p, find the leafs p *could* be in

```
Let entry = (dirRect,childPtr)
LeafSet RTreeTrav (pageId nodeID; point p){
  LeafSet res = new LeafSet();
  page n = READ(nodeID);
  if (isLeaf(n)) res.union(n); //all obj.into res
  while (n.hasNext()) {    -- traverse entries
    entry e = n.next();    -- of the node
    if (contains(e.dirRect, p)
        res.add(RTreeTrav (e.childPtr));
  } return res;
}
```

How can directory entries overlap??

## RTree: insertion

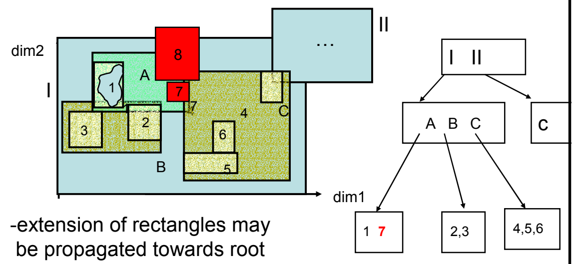



Where to put the red object?

Choose candidate with largest overlap and extend it.



## RTree: insertion




-extension of rectangles may be propagated towards root (see 8)

- if leaf is full: split similar to B-tree



## Multidimensional search

- Several refinements of basic RTree mechanism
  - essential: controlling overlap
  - shapes different from rectangles - e.g. general polygons – could make sense
- Many more index structures for multidimensional data
- Scalability problem: methods do not scale with increasing dimensions

  e.g. image retrieval: feature vector with >= 50 features ?



## 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
  - Clustering: related data in physical neighborhood

- Great differences in physical organization in DBS
- Indexing not standardized