Organizace a zpracovani dat

Disks / Memory

- primary - volotile, fast; CPU cache, registers, RAM
- secondary - non volotile, moderate speedů; SSD, HDD, online storage
- tertiary - non volotile, slow; “offline” ~ CD, DVD, floppy

HDD

Head movements/actual search times on a physical HDD - practical !!!

Head movements/actual search times on a physical HDD - practical !!!

- Access time components
- s – seek
  - average seek time from one random track (cylinder) to any other
- r - rotational delay (latency)
  - one revolution equals 2r (r is average latency)
- btt (block transfer time)

• Random access
• set heads → wait for the disk to roll the correct position → data transfer
• s + r + bt

• Seek time
• 3ms – 15ms
• usually between 8 and 12ms

• RPM (Revolutions Per Minute)
• 4,200 – 15,000 • more revolutions → more energetically demanding

• Rotational latency

Parts

- disk pack - magnetic platters on a spindle
- arm assembly - heads on a moving arm

Magnetic platters

- tracks (kruznice) divided into sectors (parts on the track)

- cylinders ~ tracks with the same diameter

SEKTOR = 512B / 4 kB - nejmensi adresovatelna jednotka >> even when just one bit of data is written or read - whole sector is worked on (bylo)
PAGE = 4 kB / 8 kB
BLOCK = ? - filesystem abstraction over sectors

Zoned bit recording?

normally - each track has same number of sectors >> tracks on the ourside waste space - bigger area for the same size of sector
with ZBR - the tracks on the outside have the same width as the ones on the inside whick means they have a larger area which can be devided into more sectors of the same size

RAID

Redundant Arrays of Inexpensive (Independent) Disks

• consists of multiple disk forming a logical unit

• Inexpensive
• original motivation
• utilization of higher number of inexpensive disks
• alternative to high-capacity expensive disks

• Independent
• present-day motivation
• higher reliability – redundancy
• higher bandwidth – parallelism

RAID 0

“stripping” - "multiple disks acting as one"
no redundancy or fault proofing
VS
better I/O preformance as it can write to 4 physical disks as oppose to 1

RAID 1

"mirroring"
using twice as many disks as there is capacity
VS
having a mirror back-up in case of a failure

RAID 5

Block-level striping with distributed parity
slow writes as the parity is being computed
VS
distributed parity

RAID 0+1

"mirrored stripes"
I have a RAID 0 connected by (mirror via) RAID 1

big overhead
VS
fault tolerance

RAID 1+0

"striped mirrors"
mirrored disks in RAID 1 connected via a RAID 0 for wider storage (better performance)

big overhead
VS
very good fault tolerance (1 disk in each of the RAID 1)

Disk Attachment Strategies

- DAS (Direct Attached Storage) • block-level storage
- NAS (Network Attached Storage) • file-level storage
- SAN (Storage Area Network) • block-level storage • accessed as drive (E:) – SAN addresses data by logical block number • multi-server multi-storage network

SSD vs Mechanic Drives (bylo)

Advantages of SSDs

- silent
- lower consumption
- more resistant to shock and vibration
- lower access time (no need to move heads)
- higher transfer rates (up to 500MB/s or even higher in enterprise-level solutions)
- does not require cooling

Disadvantages of SSDs

- lower capacity (up to 2TB, but only hundreds of GBs affordable)
- higher cost
- limited lifetime (writing to the same spot)
    - as not an issue with a typical IO load

SSD

- tranzistor with a floating gate

Uncharged cell = 1
Charged cell = 0

How to charge a cell = high enough voltage and we attract them

How to discharge a cell = high enough voltage (very high) to discharge ~ this is the cause of degradation in size\as it over time destroys the tranzistors

word line (~ kontrolni) set to “on” if we want to read then we read from bit line (NOR)

word line set to “off” then to “on” where i read and to MAX the rest of the bit line (NAND)

- more voltage and heat, but we have less wires >> more capacity

PAGE = 8kB (8192 B) - nejmensi adresovatelna jednotka
BLOCK = 256 kB-4 MB - nejmenjsi smazatelna jednotka - (fyzika - aplikace proudu je plosna)
>>> in place update NEMOZNY (problem pro male zmeny - write amplification - 4-5% increase for SSDs)

Fresh SSD - cells set to 1 (no charge in gates)

PAGE

- free
- live
- dead

INSERT - load to memory, clear block, insert (first insert is fast, then it has to use this)

MLC Multi-level cell (TLC, QLC) link

- separate levels for different combinations of 1s and 0s >> more checks >> slower reads.

SSD Controller

- allows parallelization since there is no head as in HDD
- caching - can hold data before it can be written
- wear leveling - distributes the load between pages so the SSD gets destroyed evenly - also moves the “permanent” pages to nearly destroyed pages and uses the fresh pages for frequent updates etc
- takes cae of GC

TRIM

same as GC in C# - removes pointer to data and marks it as “can be deleted” once the GC runs

without it - GC moves the page around to different places to ensure wear level and it degrades performance

Issues

B+ tree issue

No direct way to update in place - wondering tree - it moves the entire path to the root of the tree because it has to write the whole tree

>>> Minimal update tree

- in memory it looks like a pyramid that represents the tree on different pages
- always start at the root/last updated page
- root split before overflow

BFTL

- buffered FTL (flash translation layer) - is inside file system
- event log to recreate the actual tree
- minimaze insert to disk (leading to deletions and disk degradation)

FlashDB

disk mode + log mode
log mode for frequent inserts
disk mode for more reading

consolidation can be done “offline”

PIO B-tree

SSD parallelism

- disk separated to channels (works in parallel)

RAID like behaviour in multiple levels - channel and package

To use it we have to send it as much relevant data as possible since it will act accordingly

Useful for searching

Update

OPQ - in memory operation queue >> once full - batch update

Indexes - practical !!!

Indexes - practical !!!

(useful bit - building indexes/calculations connected to that)

Sizes in bytes

Primary - straight to data
Secondary - to one of its fields

N ~ Number of records
R ~ Record size
B ~ Block size

RpB Records per block ~ [B/R]

Nb ~ Number of blocks ~ [N/RpB]

Pointers per block [B/(size + pointer (usually 4))]

For secondary indexes

- recalculate how many (not records but) fields per block there are FpB
- recalculate the pointers per block

Then its find out Number of blocks and then the tree

Bitmaps !!!

• Indexing of attributes with small domains
• sex {M, F}, month {1-12}, state {active, inactive}, income level {low, medium, high}

• For each value of the domain a vector of bits is stored telling which objects share given property → array of bits
• size of the bitmap equals the number of records and each record is therefore related to exactly one position in the bit string
• when a record has a given value the corresponding bit in the corresponding bitmap is turned on • querying using bitwise logical operations

---

Hashing

Internal vs. external hashing

Open addressing – collision resolution

- clustering (bylo)

External Hashing

Static !!!

(vlastnosti)

Cormack - practical !!!

Cormack

Access & Insert

h(k) = k mod r
g(k, i , r) = (k >> i) mod r

h(k)	p	i	r

p = max§ + r(last)
r = #cisel ~ block size

Position	Value (last§ + g(k))

INSERT(k)

- calculate h(k)
  - if free then add to table
  - if collision try add i = 0 and r = #cisel
    - if collision increase i (repeat), then increase r (repeat)

Larson & Kalja (bylo) - practical

Larson & Kalja

Access & Insert

h(k, i) = (k + 1) mod 5 (~ cislo stranky)
s(k, i) = (k >> i) mod 7 (signatura) (has to be < separator)

calculate and insert, if overflow > lower separator and reinsert all / move highest s(k, i) to new insert

Hashing schemes

Dynamic !!!

(vlastnosti)

Fagin’s extendible hashing - practical

Fagin’s extendible hashing

powers of 2

calculate k mod 2^global depth >>> global column and add to pointer bucket
local and global depth

- if equal and overflow
  > double global size, add new pointer bucket for overflow, copy path to rest and for new bucket and the one that caused - increase local depth

Find

• Finding a record with a key k

1. Compute$\boldsymbol{k}' = \boldsymbol{H}(\boldsymbol{k})$
2. Compute $\boldsymbol{k}'' = \boldsymbol{hdG}(\boldsymbol{k}')$
3. Access page pointed to by the directory record with key $\boldsymbol{k}''$
4. Scan the accessed page for record with key $\boldsymbol{k}$. If the record is not found it is not present in the file.

Litwin (Linear hashing) - practical

Litwin (Linear hashing)

|Page size| = 3
|Split after N inserts| = 2

2^d ~ # stranek

2 pointers - moving one and stage delimiter

overflow allowed

always split page under pointer, insert under correcct ending (binary version of the number and look at the ending)

Addressing

```
ADDR GetAddres(KEY k, int cnt_pages)
{
  d = floor(log(cnt_pages, 2));
  s = exp(2, d);
  p = cnt_pages % s;

  addr = h(k) % s;
  if (addr < p)
    addr = h(k) % exp(2, d + 1);

  return addr;
}
```

Deferred splitting

Sharing overflow pages
• space utilization can be increased by sharing overflow pages
• more pages share one overflow page
• similar to having smaller overflow pages

Buddy pages
• logical pairing of pages
• if a page overflows, the overflown records are inserted into the buddy page
• if the buddy page needs its space or too many overflows occur the original page overflows

Bloom filter ~ approximate membership tester (bylo)

Originally proposed by Bloom, 1970 – Bloom filter
• Bit string of length $\boldsymbol{b} \bullet \boldsymbol{k}$ hash functions hi : U → 1,...,b
• $\boldsymbol{x} \in \boldsymbol{X}$: $\boldsymbol{set\ the\ bits\ corresponding\ to\ h1\ x}$,$\boldsymbol{h2\ x}$,...,$\boldsymbol{hk\ x\ to\ 1}$

• In order for $\boldsymbol{y}$ to belong to $\boldsymbol{X}$ it has to be true that $\boldsymbol{h1\ y} = \boldsymbol{h2\ y} = \cdots = \boldsymbol{hk\ y} = \boldsymbol{1}$
• can return false positives
• under the uniform hashing assumption ($hi\ x$ are independent and uniformly distributed), $X = n$, b = (log2 e)kn bits → the error rate is upper bounded by 2−k (Bloom (1970), Carter et al. (1978) and Mullin (1983))
• Deletion not available since that might modify other inserted values

Kukackove hashovani ?

File organization levels !!!

Logical schema
• algorithms
• defined to secure optimal manipulation with the data for given task
• minimization of the number of operations while manipulating the file

• logical blocks (pages)
• logical blocks structure
• logical blocks relations
• logical blocks manipulation
• fill factor

• logical files
• how are logical pages related to each other
• primary file • data
• auxiliary files • efficient access to the data (indexes, …)

Physical schema
• mapping between logical schema and physical pages
• physical files • One logical file can span multiple physical files and the other way around

Implementation schema
• implementation of the physical files
• shielded from the logical level by OS

File organization types

Heap file (halda)
• variable-length records
• a record placed always at the end of the file

Sequential file (sekvenční soubor)
• unsorted • as heap file but contains fixed-length records
• sorted • stores records in sequential order, based on the value of the search key of each record

Indexed sequential file (index-sekvenční soubor)
• records stored based on the order of the search key on which index is built

Index file (indexový soubor)
• resembles indexed sequential file but multiple indexes can be present

Hashed file (hashovaný/hešovaný soubor)
• a hash function is computed on a chosen attribute of a record; the resulting structure specifies in which block of the primary file given record should be placed/found

---

Stromy

Binary

Insert

Delete

B tree - practical (bylo)

B tree !!!

(vlastnosti)

• B-trees are balanced m-ary trees fulfilling the following conditions:

1. The root has at least two children unless it is a leaf.
2. Every inner node except of the root has at least [m/2] and at most $\boldsymbol{m}$ children.
3. Every node contains at least $[\boldsymbol{m}\ /\ \boldsymbol{2}] - \boldsymbol{1}$ and at most $\boldsymbol{m} - \boldsymbol{1}$ (pointers to) data records.
4. Each branch has the same length.
5.
    5. The nodes have following organization: $\boldsymbol{p0}$, $\boldsymbol{k1}$,$\boldsymbol{p1}$,$\boldsymbol{d1}$, $\boldsymbol{k2}$,$\boldsymbol{p2}$,$\boldsymbol{d2}$,..., $\boldsymbol{kn}$,$\boldsymbol{pn}$,$\boldsymbol{dn}$,$\boldsymbol{u}$
    • where $\boldsymbol{p0}$,$\boldsymbol{p1}$,...,$\boldsymbol{pn}$ are pointers to the child nodes, $\boldsymbol{k1}$,$\boldsymbol{k2}$,...,$\boldsymbol{kn}$ are keys, $\boldsymbol{d1}$,$\boldsymbol{d2}$,...,$\boldsymbol{dn}$ are associated data, $\boldsymbol{u}$ is unused space and the records $\boldsymbol{ki}$,$\boldsymbol{pi}$,$\boldsymbol{di}$ are sorted in increasing order with respect to the keys and $\boldsymbol{m}\ \boldsymbol{2}\ \boldsymbol{-1} \leq \boldsymbol{n} \leq \boldsymbol{m}\boldsymbol{-1}$
6. If a subtree $\boldsymbol{U}\ \boldsymbol{pi}$ corresponds to the pointer pi then: i. $\forall \boldsymbol{k} \in \boldsymbol{U}\ \boldsymbol{pi}\boldsymbol{-1} : \boldsymbol{k} < \boldsymbol{ki}$ ii. $\forall \boldsymbol{k} \in \boldsymbol{U}\ \boldsymbol{pi} : \boldsymbol{k} > \boldsymbol{ki}$

B-Tree implementations
• Nodes vs. pages/blocks • usually one page/block contains one node

Insert

Vkládám ala binary tree, dokud nezaplním
if overflow
sort and
first half left
mid goes up a level (recursion)
second half right

Delete

• Find a node $\boldsymbol{N}$ containing the key k.
• Remove $\boldsymbol{r}$ from N.
• If number of keys in $\boldsymbol{N} \geq \boldsymbol{m}\ \boldsymbol{2}\ \boldsymbol{-1}$, return.
• Else, if possible, merge $\boldsymbol{N}$ with either right or left sibling (pick max Lt or min Rt) (includes update of the parent node accompanied by the decrease of the number of keys in the parent node).

• Else reorganize records among N and its sibling and the parent node.
• If needed, reorganize the parent node in the same way (steps 3 – 5).

Redundant variant

• Redundant vs. non-redundant B-trees
• the presented definition introduced the non-redundant B-tree where each key value occurred just once in the whole tree
• redundant B-trees store the data values in the leaves and thus have to allow repeating of keys in the inner nodes
• restriction 6(i) is modified so that
• $\forall k \in U\ pi-1 : k \leq ki$
• moreover, the inner nodesdo not contain pointers to the data records

B+ tree - practical !!!

B+ tree !!!

(vlastnosti)

• Redundant B-tree • records are pointed to from the leaf nodes only
• The leaf level is chained • the leaf nodes do not have to be physically next to each other since they are connected using pointers
• Preferred in existing DBMS
• In some DBMS (e.g., Microsoft SQL Server), also the non-leaf levels are chained

Insert

Split items to two halves and take the index splitter from the bigger one and push above
Above - as a B-tree

Delete

Delet only on leaf level + check underflow > then redestribute

Differences

B vs. B+ Tree
B-tree Advantages
• Non-redundant
• can be redundant as well
• A record can be identified faster (if it resides in an inner level)

B+ tree Advantages
• Smaller inner nodes (no pointers to the data) → nodes can accommodate more records → lower tree height → faster search
• Insert and delete operations are simpler
• Simple implementation • especially of range queries

Page balancing

Page Balancing
• Modification of B-tree where an overflow does not have to lead to a page split
• When a page overflows
• sibling pages are checked
• the content of the overflowed page is joined into a set X with the left or right neighbors
• The record to be inserted is added into $\boldsymbol{X}$ and the content is equally distributed into the two nodes
• the changes are projected into the parent node where the keys have to be modified (but no new key is inserted → no split cascade)
• For high m this change leads to about 75% utilization in the worst case

B* tree - practical !!!

B* tree !!!

(vlastnosti)

Generalization of page balancing
• the root node has at least 2 children
• all the branches are of equal length
• every node different from the root has at least $[(\boldsymbol{2m - 1})/\boldsymbol{3}]$ children

Tree modification
• two full pages are split into 3 pages (one new page)
• if a node is full but none of its neighbors is full, page balancing takes place
• if the insert occurs in a full page which has full left or right neighbor
• their content is joined into a set X
• the new record is added into X
• a new page P is allocated
• the records from X are equally distributed into the 3 pages (the 2 existing and P)
• a new key is added into the parent node and the keys are adjusted
• the delete operation is handled similarly

Insert

until 2 pages are full we do not split

Delete

get some from siblings and if underflow then merge

Deferred Splitting

• In the original B-trees, certain sequences of inserts can lead to only half utilization → deferred splitting
• let us keep an overflow page for each node
• when a page overflows, the overflown record is inserted into the respective overflow page

• when both the original and the overflow page are full, the original page is split and the overflow page is emptie

---

Indexes

Primary vs secondary Index

Primary index (primární index) • index over the attribute based on which the records in the primary file are sorted • if the value of the primary attribute is modified the file needs to be reorganized → should be relatively invariable • well-suited for range queries • there does not have to be a primary index in the IF

Secondary index (sekundární index) • IF can have multiple secondary indexes • range queries for long ranges can be very expensive (an extreme example is a sequential scan based on a secondary index which can lead to an extremely deteriorated performance)

Direct vs. indirect indexing/addressing

Direct indexing (přímé indexování) • index is bound directly to the record • primary file reorganization leads to modification of all the indexing structures

Indirect indexing (nepřímé indexování) • secondary indexes contain keys of the primary index and not pointers to the primary file • accessing a record needs one more accesses to the primary index • if the primary file is reorganized, the secondary indexes stay intact

Clustered X non-clustered

• Clustered index
• logical order of the key values determines the physical order of the corresponding data records, i.e., the order of the data in the data file follows the order of data in the index file
• a file can be clustered over at most one search key → on clustered index
• basically corresponds to the idea of index-sequential file organization

• Nonclustered index
• order of data in the index and the primary file is not related
• multiple nonclustered indexes can exist

• Motivation
• when range range querying over a nonclustered index, every record might reside in a different data page
• clustered index should be defined over an attribute over which range scan often happens

Support

~	**Oracle 11g**	**Microsoft SQL Server 2012**	**PostgreSQL 9.2**	**MySQL 5.5**
Standard index	B±tree	B±tree	B±tree	B±tree
Bitmap index	Yes	No	No	No
Hash index	Yes (clustering)	Yes (clustering)	Yes	Yes

~	Oracle 11g	Microsoft SQL Server 2012	PostgreSQL 9.2	MySQL 5.5
Spatial index	R-tree	B±tree (mapping 2D to 1D)	R-tree	R-tree

Less Traditional Approaches to Indexing

Update-optimized structures
• Buffered repository tree (BRT)
• Streaming B-tree

Cache-oblivious structures
• Cache-oblivious B-tree
• Streaming B-tree

Frakralove stromy

van Emde Boasovo usporadani?

//R stromy probrane, ale neprocvicene, Hilbert R-Tree …
(aspon rozdily proti B…)

Others

• Spatial DBMS

Spatial join !!!

Prostorové spojení

Given two sets of spatial objects, spatial join pairs the sets' objects based on a given spatial predicate

• intersection
• identify pairs of objects from the two sets which intersect
• "Find all pairs of rivers and cities that intersect."

• distance
• identify pairs of objects from the two sets which are within given distance

• … any relation including a pair of spatial objects

Spatial Objects Representation !!!

When dealing with spatial objects in terms of storage and manipulation we can either

• project (serialize) them into 1D space and employ existing singledimensional methods
• utilize the full spatial information with specialized techniques for spatial management

One-Dimensional Embedding of Spatial Objects (2) !!!

One-Dimensional Embedding of Spatial Objects (2) • Space filling curve • a curve visiting cells of the grid representing the space; each cell is visited exactly once • the points on the line are ordered thus giving the points in the space (grid) linear ordering
• Typical approaches for space filling curves • Naïve curve, spiral curve, Z-curve, Hilbert curve, …
• With space filling curves one can implement file operations similarly to standard ordered files

• Naïve curve
• Spiral curve
• Z-curve (bylo)
• Hilbert curve

Hilbert Curve

• Recursive representation of the space
• space is divided into four parts and their ordering is given by the "cup"-like curve
• every square is divided into another four parts using another cup-like curve which needs to be rotated so that neighboring squares in higher level ordering are connected

Indexes - Others

Buffered repository tree (bylo)
• Spatial DBMS

2D → 1D mapping ~ Spatial indexing !!!

• Methods for efficient search in multidimensional data; i.e., spatial queries should access as few pages as possible
• B±trees are usable but they are basically single-domain indexes, although they can support multi-dimensional data/queries (e.g. using space-filling curves)
• To efficiently index multi-dimensional data we need multi-dimensional indexes
• Quadtree, k-d-tree
• R-tree, R±tree, R*-tree
• Hilbert R-tree, X-tree, …
• UB-tree, …

Single Dimension-Based Indexing (1)

• B+Trees are capable of storing multi-dimensional information in form of an ordered tuple – compound (chained) search keys (složený klíč)
• the tuples are ordered first based on the first element, then on the second and so on (lexicographical order)
• the standard ordering of tuples in a B±tree resambles naïve space-filling curve
• the way in which we define ordering on the tuples defines the type of space-filling curve

Multidimensional Indexing !!!

• Multidimensional indexes focus on storing spatial objects in such a way that objects close to each other in the space are also close in the structure and on the disk, i.e., maintain locality
• As in single dimensional indexing we are interested in tree or hash structures to avoid sequential scan of every record in the database

Grid-Based Indexing !!!

• N-dimensional grid covers the space and is not dependent on the data distribution in any way, i.e. the grid is formed in advance
• every point object can be addressed by the grid address
• basically corresponds to hashing where a grid cell corresponds to a bucket
• Objects distribution in the grid does not have to be uniform → retrieval times for different grid cells may differ substantially for different parts of the space

Trees

k-d-Tree

• [Bentley; 1975]
• k-dimensional tree
• Binary tree where inner nodes consist of a point, an axis identification (hyperplane in nD) and two pointers
• inner nodes correspond to hyper planes splitting space into two parts where the location of the hyper plane is defined by the point
• points in one part are pointed to by one pointer and the other part by the second one

K-D-B-Tree

• [Robinson; 1981]
• Combination of K-D-Tree and BTree
• k-d-tree is designed for main memory
• In case when the dataset does not fit in main memory it is not clear how to group nodes into pages on the disk
• multiway balanced tree
• each node stored as a page but unlike B-trees 50% utilization can not be guaranteed
• each inner node contains multiple split axis to fill the node's capacity
• leaf nodes contain indexed records (points)
• splitting and merging happens analogously to B-trees

Quad-tree (bylo)

Quad-Tree • [Finkel, Bentley; 1974] • Tree structure representing recursive splitting of a space into quadrants • each node has from zero to four children • typically the regions are squares (although any arbitrary shape is possible)

R-tree

insert, split

• [Guttman, 1984]
• Quad-trees and k-d-trees are not suitable for large collections since they do not take paging of secondary memory into account
• K-D-B-tree is capable of storing point objects only and might not be ballanced
• R-Tree can be viewed as a direct multidimensional extension of the B±tree
• Leaf records contain pointers to the spatial objects
• Inner records contain MBRs of the underlying MBRs (or objects)

• an MBR corresponding to a node $\boldsymbol{N}$ covers all the objects (MBRs and spatial objects) in all the descendants of $\boldsymbol{N}$

R-Tree Definition (1)
• Height-balanced tree
• Nodes correspond to disk pages

• Each node contains a set of entries $\boldsymbol{E}$ consisting of
• $\boldsymbol{E.p}$ – pointer to the child node (inner node) or spatial object identifier (leafs)
• $\boldsymbol{E.I}$ – $\boldsymbol{n}$-dimensional bounding box I = ($I0$,$I1$,...,$In-1$), where Ij corresponds to the extent of the object I along j-th dimension

• Let $\boldsymbol{M}$ be the maximum number of entries in a node and let $\boldsymbol{m \leq M / 2}$

• Given the labeling from previous slide, R-Tree is an M-ary tree fulfilling the following conditions

• Every leaf contains between $\boldsymbol{m}$and $\boldsymbol{M}$index records.
• Every non-leaf node other than root contains between $\boldsymbol{m}$and $\boldsymbol{M}$entries.
• The root has at least 2 children unless it is a leaf.
• For each record $\boldsymbol{E}$, $\boldsymbol{E.I}$is the minimum bounding rectangle.
• All leaves appear on the same level.

• It follows that height of an R-tree with n index records ≤ log$m$ n

Searching in R-trees (1)
• Result of a search is a set of objects intersecting the query object
• Search key is represented by the bounding box of a query object

• Unlike in B-trees, the search procedure can follow multiple paths → worst-case performance cannot be guaranteed
• the more the MBRs intersect the worse the performance

• Update algorithms force the bounding rectangles to be as much separated as possible allowing efficient filtering while searching

Inserting into R-trees (1)

```
Insert_R(T,E)
Input: R-tree with a root T, index record E
Output: updated R-tree

ChooseLeaf(T,L,E); { chooses leaf L for E}
IF E fits in L THEN
  Insert(L,E); LL ← NIL;
ELSE
  SplitNode(L,LL,E)
AdjustTree(L,LL,T); {propagates changes upwards}
IF T was split THEN
  install a new root;


ChooseLeaf(T,L,E)
Input: R-tree with a root T, index record E
```

```
Output: leaf L
N ← T;
WHILE N ≠leaf DO
  chose such entry F from N whose F.I needs least enlargement to include E.I in cas
  N ← F.p;
L ← N;


AdjustTree(L,LL,T)
Input: R-tree with a root T, leafs L and LL
Output: updated R-tree
N ← L; NN ← LL;
WHILE N ≠ T DO
  P ← Parent(N); PP ← NIL;
  modify EN.I in P so that it contains all rectangles in N;
  IF NN ! = NIL THEN
    create ENN, where ENN.p = NN and ENN.I covers all rectangles in NN;
    IF ENN fits in P THEN
      Insert(P, ENN); PP ← NIL
    ELSE
      SplitNode(P,PP, ENN)
  N ← P;
  NN ← PP
LL ← NN;


SplitNode(P,PP,E)
Input: node P, new node PP, m original entries, new entry E
Output: modified P, PP
PickSeeds(); {chooses first Ei and Ej for P and PP}
WHILE not assigned entry exists DO
  IF remaining entries need to be assigned to P or PP in order to have the minimum
    assign them;
  ELSE
    Ei ← PickNext() {choose where to assign next entry}
    Add Ei into group that will have to be enlarged least to accommodate it. Resolv

PickSeeds
FOREACH Ei, Ej (i ≠ j) DO
  dij ← area(J) – area(Ei.I) – area(Ej.I) (J is the MBR covering Ei and Ej);
pick Ei and Ej with maximal dij;

PickNext
FOREACH remaining Ei DO
  d1 ← area increase required for MBR of P and Ei.I;
  d2 ← area increase required for MBR of PP and Ei.I;
pick Ei with maximal |d1 – d2|;
```

R±Tree

- [Sellis et al.; 1987]
- MBRs of R±tree have zero overlap while allowing underfilled nodes and duplication of MBRs in the nodes
- achieved by splitting an object and placing it into multiple leaves if necessary

• Takes into account not only coverage (total area of a covering rectangle) but also overlap (area existing in one or more rectangles)

Pros
• fewer paths are explored when searching
• point queries go along one path only

Cons
• Overlapping rectangles need to be split → more frequent splitting → higher tree → slower queries

R-Tree (Green) (1)
• [Green; 1989]
• Modification of the split algorithm of the original R-tree
• Splitting is based on a hyperplane which defines in which node the objects will fall
SplitNode_G(P,PP,E) ChooseAxis() Distribute()

ChooseAxis
PickSeads; {from Guttman's version – returns seeds Ei and Ej} For every axis compute the distance between MBRs Ei, Ej; Normalize the distances by the respective edge length of the bounding rectangle of the original node. Pick the axis with greatest normalized separation;

Distribute
Sort Eis in the chosen axis j based on the j-th coordinate. Add first ⌈(M+1)/2⌉ records into P and rest of them into PP.

R*-Tree (1)
• [Beckmann et al.; 1990]
• R*-tree tries to minimize coverage(area) and overlap by adding another criterion - margin
• Overlap defined as $overlap\ Ek = i{=}1,i{\neq}k\ n$
$area(Ek.I \cap Ei.I)$

R*-Tree (2)
ChooseLeaf_RS(T,L,E) Input: R-tree with a root T, index record E Output: leaf L
N ← T; WHILE N ≠ leaf DO IF following level contains leaves THEN choose F from N minimizing $\mathbf{overlap}(\mathbf{F} \cup \mathbf{E})$ and solve ties by picking F whose F.I needs minimal extension or having minimal volume; ELSE choose F from N where F.I needs minimal extension to I' while E.I ⊂ F.I' and area(F.I') is minimal N:=F.p L:= N

R*-Tree Splitting (1)
• Exhaustive algorithm where entries are sorted first based on $\boldsymbol{x1}$ axis and second on $\boldsymbol{x2}$
• For each axis, $\boldsymbol{M-2m+2}$ distributions of $\boldsymbol{M+1}$ entries into 2 groups are determined
• in k-th distribution, the first group contains $(m-1)+k$ entries and the second group the rest, $k = 1,\ldots,M-2m+2$

R*-tree Splitting (2)
• For each distribution following so-called goodness values are computed (Gi denotes i-th group)
• area-value (h-objem) • $area\ MBR\ G1 + area\ MBR\ G2$
• margin-value (h-okraj) • $margin\ MBR\ G1 + margin\ MBR\ G2$
• overlap-value (h-překrytí) • $area\ MBR\ G1 \cap MBR\ G2$

R*-tree Splitting (3)
Split_RS ChooseSplitAxis(); {Determines the axis perpendicular to which the split is performed}

ChooseSplitIndex(); {Determines the distribution} Distribute the entries into two groups;
ChooseSplitAxis FOREACH axis DO Sort the entries along given axis; S ← sum of all margin-values of the different distributions; Choose the axis with the minimum S as split axis;
ChooseSplitIndex Along the split axis, choose the distribution with minimum overlapvalue. Resolve ties by choosing the distribution with minimum area-value

Hilbert R-Tree

• [Kamel&Faloutsos; 1994]
• Idea • facilitates deferred splitting in R-tree • ordering is defined on the R-tree nodes which enables to define sibling of a node in given order • Hilbert space filling curves • when a split is needed, the overflown entries can be moved to their neighboring nodes thus deferring the split • search procedure identical to ordinary R-tree (i.e. the idea of a MBR covering its descendants' MBRs still holds)
Hilbert R-Tree Definition
• Hilbert value of a rectangle – Hilbert value of its center
• Hilbert R-tree • behaves exactly the same as R-tree on search • on insertion supports deferred splitting using the Hilbert values
• leaf nodes contain pairs (R, obj_id) • R - MBR of the indexed object; obj_id – indexed object's id (pointer) • non-leaf nodes contain triplets (R, ptr, LHV) • R - MBR of the region corresponding to the entry; ptr – pointer to subtree; LHV – largest Hilbert value among the data rectangles enclosed by R

Non-spatial join !!!

There exist algorithms for standard relational join
• only equi-joins (the join predicate is equality) considered here
• nested loop join (hnízděné cykly)
• sort-merge join (setřídění-slévání)
• hash join (hashované spojení)
• most of the standard relational join algorithms are not suitable for spatial data because the join condition involves multidimensional spatial attribute

• Nested loop join
• Index nested loop join
• Plane Sweep
• Z-order

EXTRAS + unsorted

SQL / DBs

• Database server structure
• databases
• database files

- memory
- pages

- Data access
- heaps
- clustered indexes
- nonclustered indexes

Query lifecycle

MSSQL Server hierarchy

Data row stracture

(last lecture) Different types of memory and file systems
