Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (2/6)

- indexing
  - one- and multidimensional
  - tree-structured
  - partition-based indexing
  - bulk-loading
  - hash-indexes
  - differential indexing
  - read-optimized indexing
  - write-optimized indexing
  - data warehouse indexing
  - text indexing: inverted files
  - main-memory indexing
  - (flash-indexing)

Indexing.

Indexing.

Introduction.

Motivation: Example Queries

- What is the address of the student having ID 424342?
- Which students live in Saarbrücken?
- What Web-pages contain the keyword “dittrich“?
- What was the average profit of the company last year?
- Where is Saarbrücken? Show it on the map?
- Where is the next Pizzeria?
- Which aircrafts are currently less than 100 miles away from Saarbrücken? How will that change within the next 5 minutes?
- What kind of products should be sold in the Edeka campus store?

How to Compute the Results?

- inspect all data entries:
  - sequential access aka scan or full table scan (FTS)
  - complexity: O(N)
  - very expensive
  - still useful in many situations
- indexing:
  - e.g. organize data entries cleverly such that results may be found quickly: index-based access
  - tree- or hash-structured access
  - complexity: O(log N) or O(1)
  - best option in many situations
  - however: for some situations slower than scan

What does “Indexing“ mean?

- Mapping:
  - key $\longrightarrow$ set of data items
  - key does not have to correspond to a primary key in the data items set

an index materializes this mapping

Requirements for Index Structures

- efficient usage of external storage, memory, and caches
  - low I/O-cost
  - few cache misses
  - low CPU-cost
- low query response times
- low maintenance cost. e.g, low cost for insert, update, and delete
- high throughput of operations
- easy to integrate into existing information systems
- easy to extend to add other functionality

Access Paths

- Very often the same data item can be accessed via different access paths

access path = possible way to retrieve a data item

- access paths have **huge** impact on the efficient computation of a query result

Primary vs. Secondary Access Path

- primary access path:
  - Attended cloakroom: “I would like to have my coat: I have ticket number 42.“
  - SQL Example:
    ```
    SELECT *
     FROM   coats
     WHERE coatID = 42
    ```
- secondary access path:
  - Attended cloakroom: “I would like to have all coats containing big purses.“
  - SQL Example:
    ```
    SELECT *
    FROM   coats
    WHERE purseSize = ‘big’
    ```
  - Text Search Example: show all documents containing keyword “dittrich“

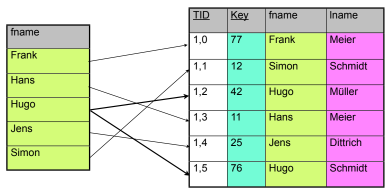
Primary versus Secondary Access Paths

- **Primary** access path:
  Access using an index that is based on the primary key of a relation
- **Secondary** access path:
  Access using an index based on an attribute that is not the primary key
- **Examples**:
  Index of...
  Key $\longrightarrow$ TID = primary access path
  fname $\longrightarrow$ TID = secondary access path
  lname $\longrightarrow$ TID = secondary access path

Key	fname	lname
77	Frank	Meier
12	Simon	Schmidt
42	Hugo	Müller
11	Hans	Meier
25	Jens	Dittrich
76	Hugo	Schmidt

Secondary Access Path and Inversion

- Queries using a secondary access path may return more than one result
- 1:n relationship among keys and results



fname
Frank
Hans
Hugo
Jens
Simon

TID	Key	fname	lname
1,0	77	Frank	Meier
1,1	12	Simon	Schmidt
1,2	42	Hugo	Müller
1,3	11	Hans	Meier
1,4	25	Jens	Dittrich
1,5	76	Hugo	Schmidt

Indexing in Google (without Ranking)

- Example for a secondary access path
- indexing idea:
  - Assign a unique document ID (DocID) to every document (html, pdf, doc, etc.)
  - Create a secondary access path materializing the mapping

$$\text{keyword} \dashrightarrow \{\underbrace{(\text{DocID}, \overbrace{\{\text{pos}\}}^{\text{list of positions of occurrences inside a single document}})}_{\text{occurences of keywords in documents}}\}$$

- This index is named **inverted list** or **inverted file**.

Indexing in Google (without Ranking)

- Example Inverted List:

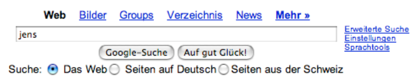
document IDs

occurrences

```
...
jens     -> (42,{3,500,900,1000}),
            (88,{3,300}),
            (4025,{1,20,5000}),
ditttrich -> (12,{2,450,600}),
            (78,{1,4300,7000}),
            (2123,{30}),
uni      -> (15,{2,450,600}),
            (19,{11,100,2000}),
            (77,{16,1200,2000}),
            (345,{17,300,5000}),
            (2123,{30}),
...
```

term “uni“ appears in document 15 at positions 2, 450, and 600

Keyword Search in Google



jens

Suche: [✓ Das Web] Seiten auf Deutsch Seiten aus der Schweiz

- How?
- Lookup the list of entries for keyword “jens“
- Return the first ten entries
- Note: the first ten elements do not have to be the most important ones!
- Next: What happens when we query for multiple keywords?

Google Multiple Keywords (Without Ranking)

- Example:

"jens dittrich" eth

Suche: [✓ Das Web] Seiten auf Deutsch Seiten aus der Schweiz

- Algorithm
  - 3 accesses on secondary access path using keys <eth>, <jens>, <dittrich>
  - Result: 3 sequences T_1', T_2', T_3' of DocIDs (document IDs)
  - Compute intersection
    $T' = T_1' \cap \{ t \mid t \in (T_2' \cap T_3') \wedge \text{pos}(t,T_2') = \text{pos}(t,T_3') - 1 \}$
  - Return first ten elements as result
  - Again note: the first ten elements do not have to be the most important ones!

Indexing in Google (with Ranking)

- Problem: keyword search may return millions of results
- For keyword “jens” Google estimates 39.2 million documents (as of 10/2008).
- Furthermore: The most important documents do not have to be among the top 10 pages.
- Solution: try to order documents based on their (query-independent) rank
- many algorithms, the most important one: **Page Rank**
- However: what is the impact of ranking on indexing?

Indexing in Google (with Ranking)

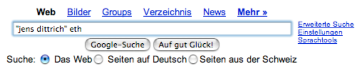
- Core idea:
  1. enumerate documents using docID = rank
     i.e., the document having docID=1 is the most important one
     Note: docIDs still have to be a key! (no duplicates)
  2. Sort each result list by docID (i.e., its rank)
- Impact: sort-based intersection still works
- Example:

Rank= doc ID

```
...
jens     -> (7000,{3,500,900,1000}),
            (8888,{3,300}),
            (40251,{1,20,5000}),
dittrich -> (12,{2,450,600}),
            (78,{1,4300,7000}),
            (2123,{30}),
```

Multiple Keywords (With Ranking)

- Example:


"jens dittrich" eth

- Algorithm (same as algorithm without ranking)
  - 3 accesses on secondary path using keys <eth>, <jens>, <dittrich>
  - Result: 3 sequences T_1', T_2', T_3' of DocIDs (document IDs)
  - Compute intersection

    $T' = T_1' \cap \{ t \mid t \in (T_2' \cap T_3') \wedge pos(t,T_2') = pos(t,T_3')-1 \}$
  - Return first 10 DocIDs of T' as result.
  - Difference to previous approach: the first 10 documents are the most important documents **w.r.t. page rank**

Search Engines: A much bigger Story...

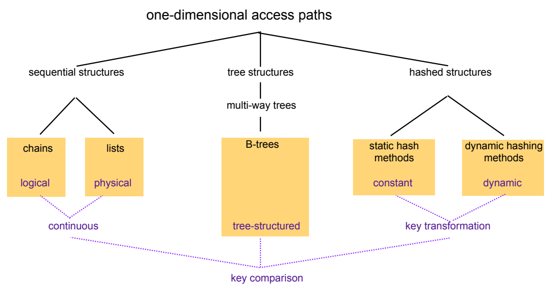
- Note: the previous example was simplified we ignored ranking schemes like tf-idf and Okapi BM25
- we also ignored:
  - stemming:
    - reduce each keyword to its "stem" (usually a prefix)
    - only consider the stem in the index
    - example: "dogs" and "dog" are both indexed as "dog"
  - stop-words:
    - ignore very common words like "the", "and"
    - this reduces the sze of the index considerably
- the details of this other techniques justify a separate lecture, usually named "Information Retrieval"

Search Engines: The Data Managing Aspect

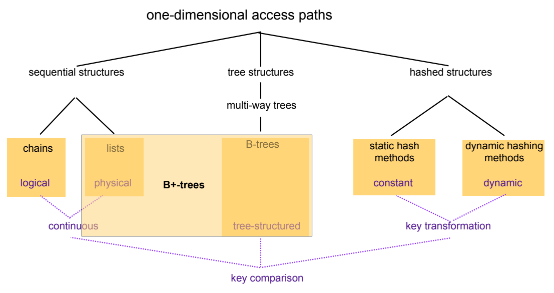
- in this lecture we will however look at the following important aspects to be considered by any search engine
  - given a Terabyte of unindexed data, how to create the index efficiently?
  - how to update the existing index if documents change?
  - how to run the indexing framework on a network of parallel machines?
- these techniques are the basis for data managing in any search engine
- all other search engine-specific techniques are built on top of this
- but let's first look at some indexing basics...

One-dimensional Indexes.

One-dimensional Access Paths



Onde-dimensional Access Paths

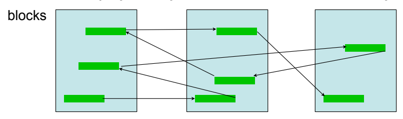


One-dimensional Indexes.

Sequential Structures.

Sequential Access Paths: Chains

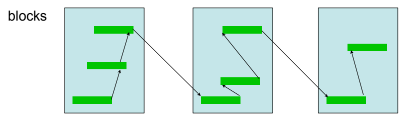
- list of tuples ignoring block order (sequential layout) on external storage
- tuples are **not** physically clustered on blocks/pages
- blocks are **not** physically clustered on disk/memory



- Discussion
  - poor I/O-performance
  - Worst case: 1 random access **for each tuple**
  - should not be used

Sequential Access Paths: Lists

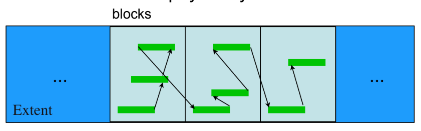
- list of tuples grouped into blocks
- tuples are physically clustered on blocks/pages
- blocks are **not** physically clustered on disk/memory



- Discussion
  - better I/O-performance
  - Worst case: 1 random access **for each page**
  - used in information systems

Sequential Access Paths: Sequence

- Like chains but in addition: blocks sequential on disk/memory
- tuples **and** blocks are both physically clustered



Discussion
- optimal I/O-performance
- Worst case: 1 random access + sequential access
- very important for information systems (especially read-mostly environments)
- Drawback: hard to maintain in the presence of inserts and updates => defrag data layout regularly

One-dimensional Indexes.

Tree Structures.

Overview on Tree Structures (1/2)

- binary trees
  - perfect in theory
  - however difficult to map tiny nodes to pages
  - **not** suitable for DBMSs
- digital trees
  - only for special applications
  - important for non-relational data (e.g., spatial data)
- b-trees (multiway trees)
  - most important index structure for DBMSs
  - advantage: very versatile, easy to extend
  - invented 30 years ago, still being improved
  - several index structures exist that are based on similar ideas (e.g. R-tree, M-tree)

Overview on Tree Structures (2/2)

- sorted array
  - implicit tree structure
  - in fact a linearization of a tree: array corresponds to inorder traversal of any binary tree
  - perfect sequential layout
  - binary search similar to search in balanced binary tree
  - hower: not the best option in terms of cache misses
- cache optimized trees
  - some based on B-trees (e.g. fpB+-Trees)
  - others based on arrays (e.g. CSS-Trees)

One-dimensional Indexes.

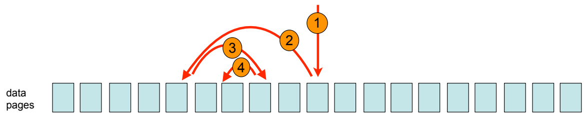
Tree Structures: B-trees.

B-Trees: Agenda

- Basics (repetition)
- ISAM
- Clustered index
- Indirect vs. direct storage
- Primary versus secondary B-tree
- Bulk-loading
- Prefix B+-tree
- Prefix/suffix-compression
- Large index pages
- Cache conscious B+-trees

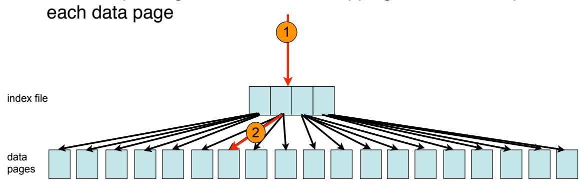
Intuition for Tree Indexes (1/3)

- Consider you have a large dataset with students
- say 10,000 pages
- you want to do a range search:
  “find all students having a gpa (grade point average) 2.0 or higher“
- Effects:
  - binary search may lead to considerable random I/O



Intuition for Tree Indexes (2/3)

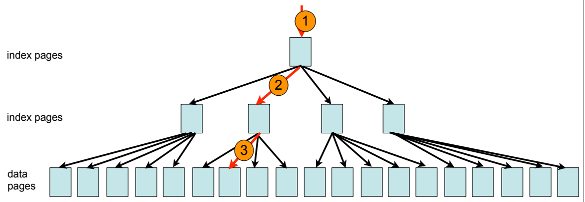
- Idea: keep a large **index file** with mappings to the first key of each data page



- Now:
  - binary search in index file
  - one random I/O to retrieve the actual data page

Intuition for Tree Indexes (3/3)

- Problem: index file potentially large
- Idea: apply idea of index file **recursivley**
- In other words: a bottom-up created recursive index

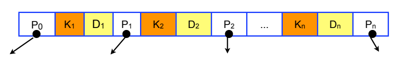


- Now:
  - binary search inside index pages
  - one random I/O for each index and data page

B-Tree

- Pointer P_0 points to subtree with keys **strictly** smaller than K_1
- For P_i (i=1,...,n-1) it holds: $K_i < keys(P_i) < K_{i+1}$
- P_n points to a subtree having keys strictly greater than K_n

node structure:

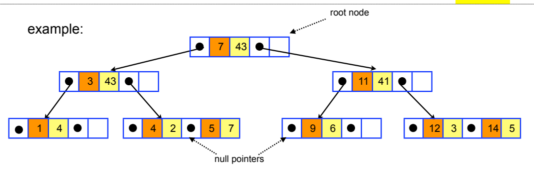


P_i: pointer

K_i: key

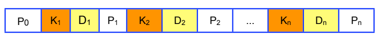
D_i: data

example:



B-Tree: Definition

- a B-tree of type (k, h) has the following invariants:
  - Every path from the root node to any leave has length h.
  - Every node (except root node and leaf level nodes) has at least k +1 children.
  - The root is either a leaf or is a node and has at least two children.
  - Every node has at most 2k+1 children.
- node structure:



P_i: pointer

K_i: key

D_i: data

k <= n <= 2k (if neither root nor leaf level)

- Storage: B-tree nodes are mapped to pages/blocks.
- Literature: R. Bayer, E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1:4. 1972. 290-306.

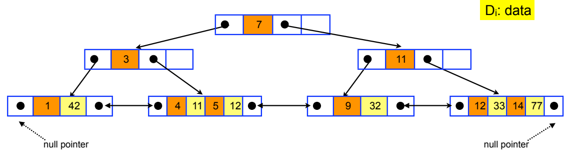
B+-Trees

- Like B-trees, but:
  - data entries are only stored in leaves
  - Impact: higher fan-out of nodes => height of tree decreases
  - leaves are connected to build a double-linked list: **ISAM** (Index Sequential Access Method)

P_i: pointer

K_i: key

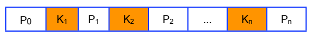
D_i: data



B+-Trees: Definition

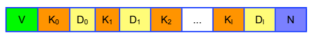
- a B+-tree of type (k, k*, h) has the following invariants:
  - Every path from the root node to any leaf has length h.
  - Every node has at least k+1 children and at most 2k+1 children.
  - Every leaf has at least k* and at most 2k* entries.
  - The root is either a leaf or is a node having at least two children.

node structure:



k <= n <= 2k

leaf structure:



k* <= l <= 2k*

P_i: pointer

K_i: key

D_i: data

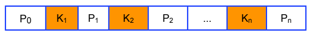
V: left sibling pointer

N: right sibling pointer

B+-Trees

- P_0 points to a subtree having keys <= K_1 (smaller or equal)
- For P_i (i=1,...,n-1) it holds: $K_i < keys(P_i) <= K_{i+1}$
- P_n points to the subtree having keys strictly greater than K_n
- 2 different node types: nodes and leaves

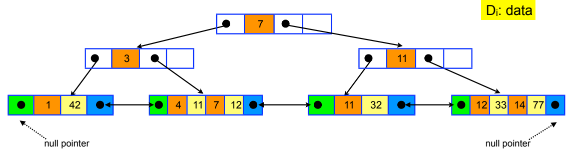
node structure:



P_i: pointer

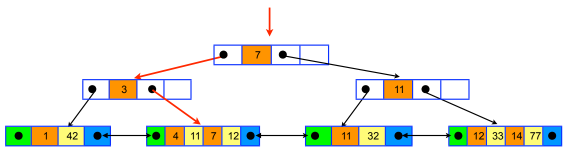
K_i: key

D_i: data



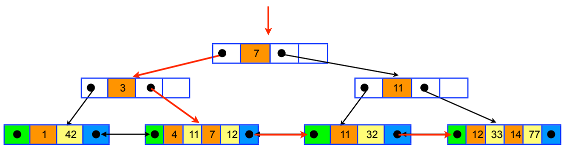
B+-Trees Point Query (find_key)

- Recursive search starting with the root node
- Inside a node or leaf: binary search
- Exactly h-1 nodes and one leaf will be touched
- Example: find_key [4]



B+-Tree Range Query (find_range)

- Example: find_range [4;12]
- Algorithm:
  1. point query with key 4 (i.e., find_key [4])
  2. Read leaves starting at position of key 4 sequentially until keys are strictly greater than 12
     (i.e., ISAM: Index Sequential Access Method)



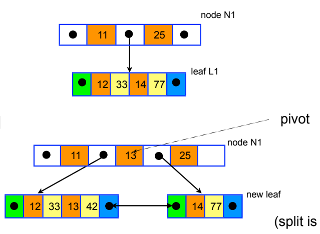
B+-Trees: split-Operation

- If leaf > 2k* entries:
  - Create a new leaf
  - Distribute entries to both leaves such that each leaf has >= k* entries
  - insert pointer and pivot into parent node

Example

Before split:

After insert [13,42] and split:



(split is similar for nodes)

B+-Tree: insert (Procedural Programming)

- insert (42,12):

```
1. node = root
2. while (node!=Leaf)
       node = choose_subtree( node, 42 )
3. node.insert_entry(42,12)
4. If node has more than 2k* entries: (actually a leaf in the first round)
       Split leave into two leaves
       Distribute entries to both leaves
       insert new leaf pointer and pivot into parent node
       If parent node has more than 2k+1 children:
           Split parent node
           etc. (until root is reached or no split necessary anymore)
           ...
```

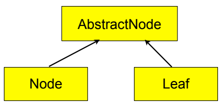
B+-Tree: insert (Object-oriented 1/2)

- Node.insert (42,12):

```
1. (split, left, pivot, right) = choose_subtree( 42 ).insert( 42,12 )
2. If split occurred:
       this.insert_node( pivot, right )
       If this.entries > 2k+1:
           return this.split()
3. return (false, this, NULL, NULL)
```

- Leaf.insert (42,12):

```
1. insert_tuple( 42,12 )
2. If this.entries > 2k*:
       return this.split()
   Else
       return (false, this, NULL, NULL)
```



Call:
root.insert(42, 12)

B+-Tree: insert (Object-oriented 2/2)

- Node.insert (42,12):

```
1. (split, left, pivot, right) = choose_subtree( 42 ).insert( 42,12 )
2. If split occurred:
       this.insert_node( pivot, right )
       If this.entries > 2k+1:
           return this.split()
3. return (false, this, NULL, NULL)
```

usage:
root.insert(42, 12)

Question: what happens if the root node has to be split?

B+-Tree: other insert-Strategies

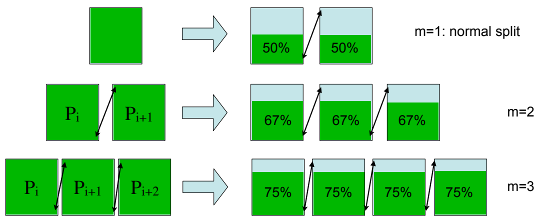
- If node full

```
Try to redistribute entries to d predecessor and/or d successor nodes
If redistribute successful:
    insert without split
Else
    split
```

- Discussion:
  - improves memory usage of the tree
  - Disadvantage: redistribution may be costly (adjust parent nodes?)
  - Requires double-linked list on nodes (similar to ISAM on leaves)

Example: other split-Strategies

- If split occurs:
  - Create new leaf (node)
  - Redistribute entries of m neighboring leaves (nodes) such that every leaf (node) has at least k* entries (k+1 children)

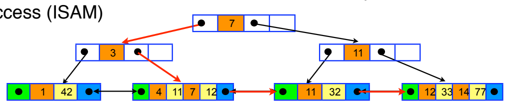


B+-Trees: delete

- discussion analogue to split-operation!
- If node/leaf underflows: **merge**
- **merge** = inverse operation of split
  i.e., put entries of m leaves (nodes) into a single leaf (node), all other leaves (nodes) are deleted
- Other strategies:
  - relax B+-tree invariant and allow nodes/leaves to underflow without performing a merge
  - merge will only be performed if underflow continues for a certain amount of time
  - or: merge will never be performed (tolerate some dead space)
  - improves ISAM-access (less defragmentation of leaves)

ISAM and Block Fragmentation

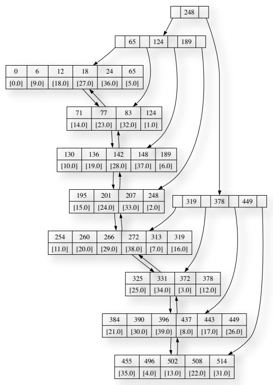
- Double-linked list on leaves allows efficient sequential access (ISAM)



- Problem:
  - inserts and deletes will fragment leaves on disk (leaves not contiguous on disk anymore)
  - many inserts/updates => high fragmentation => increase of random I/O => decrease of ISAM performance
- How to fix:
  - defragment index regularly
  - try to preallocate free blocks in block order on disk in regions where the tree might grow in future, learn from previous patterns

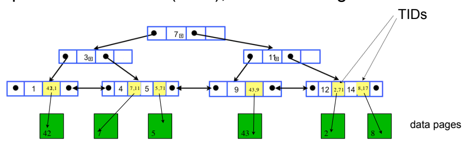
B+-Tree

- Python-demo

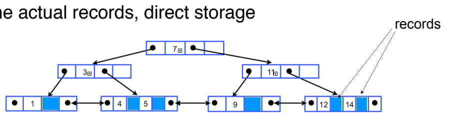


Indirect vs. Direct Storage

- What is stored inside the leaves?
- Either: pointers to records (TIDs), indirect storage

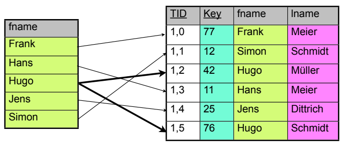


- Or: the actual records, direct storage



B+-trees and Inversion

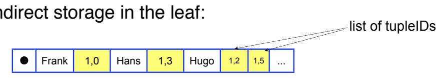
- Recall: Queries using a secondary access path may return more than one result (1:n relationship among keys and results)



fname
Frank
Hans
Hugo
Jens
Simon

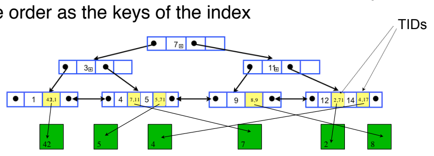
TID	Key	fname	lname
1,0	77	Frank	Meier
1,1	12	Simon	Schmidt
1,2	42	Hugo	Müller
1,3	11	Hans	Meier
1,4	25	Jens	Dittrich
1,5	76	Hugo	Schmidt

- How do we store these lists of results in a B+-tree?
- Example: indirect storage in the leaf:

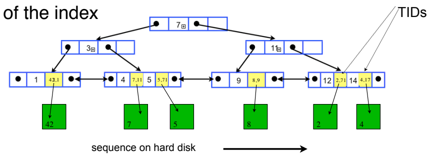


Clustered vs. Non-Clustered Index

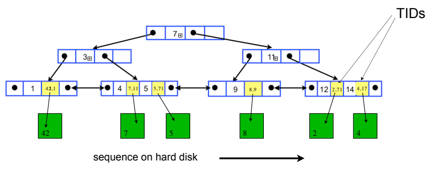
- Non-clustered index: data blocks do not necessarily have the same order as the keys of the index



- Clustered Index: data blocks have the same order as the keys of the index

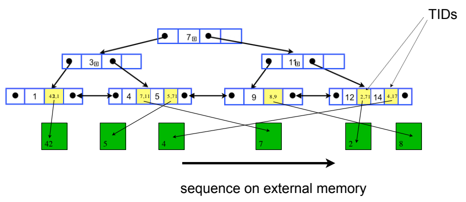


Clustered Index



- at most 1 clustered index possible for each table (usually for the primary key)
- Note:
  Clustering can also be done by forcing direct storage (assuming that the information system will keep leaves in sequential order)

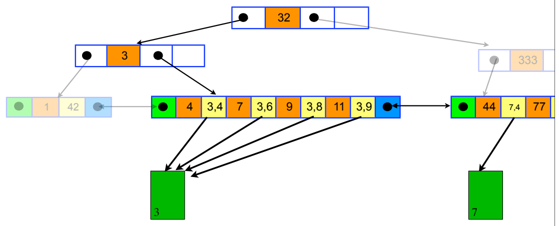
Non-Clustered Index



- multiple non-clustered indexes per table possible
- implies an indirect index
- suitable for
  - selective queries
  - long records

Dense Index

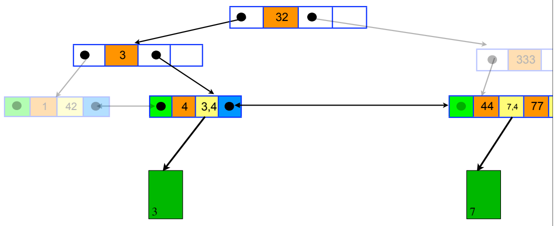
- as shown before:
  - for **each data item** in the data there is a mapping in the index



- **possibly multiple** mappings from the index to the same data page

Sparse Index

- keeps less information than a dense index:
  - for **each data block** there is only one mapping in the index



- **single** mapping from the index to the same data page

Sparse Index

- Advantages
  - less mappings in the index
  - => less space required in the leaves
  - => probability that tree-height decreases
  - => overall: index much smaller
  - => searching the index becomes more efficient
  - Example:
    - assume a data page holds 100 data items
    - instead of keeping 100 mappings, we keep only one...
- Disadvantages:
  - we have to search through the data page to find the data item
  - index-lookup alone cannot determine whether a data item with a given key exists anyway
  - in contrast: if **dense index** says there is a mapping
    => there is a data item with that key

An Even Sparser Index

- We may relax the sparse index to the following rule
  - for **a certain unit of data** there is only one mapping in the index
  - dropped “block“ and replaced it by “certain unit of data“
- certain unit of data may be:
  - a block
  - an extent
  - any external container, e.g., an external queue
- then we should rather be naming this index a **partitioning scheme**...

Partitioning Scheme

- Given
  - a set of data items M
  - a partitioning function P mapping a data item m of M to a partition ID 0,...,p
- If we apply P to M and group all elements having the same partition ID we receive p+1 **partitions** $P_0,...,P_{p+1}$
- Example: given a set of people we may group them into ranges according to their age 0-10, 11-20,21-30,...
- This is the core idea of all **divide-and-conquer** approaches including fundamental examples such as QuickSort.
- In other words: a top-down created recursive index.
- This idea is also used in hashing.

Trees and Partitioning Schemes

- So what is the relationship among trees and partitioning schemes?
- tree indexes:
  - basically a recursive partitioning scheme
  - multi-level partitioning
  - different partitiong functions at diffferent levels
  - for example: more fine granular partitioning at the next level
  - more a bottom-up approach (however: bulk-loading)
- partitioning:
  - typically only one level without recursion
  - recursive partitioning in use but typically not named trees
  - recursive partitioning tends to throw away partitioning tree
  - applied in top-down fashion
- transition among both approaches possible

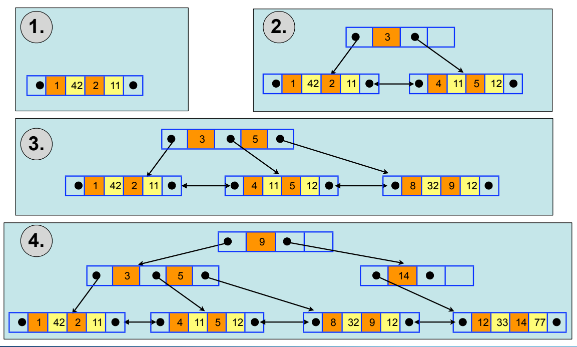
Bulk-Loading

- Problem: How to create an index for an existing table?
- Simple algorithm (Bottom-up index creation):
  - sort records into sequence S‘ (sort key: key to use for the new index)
  - while S‘ not empty
    - take the first F·2k* records of S‘ and put them into a new leaf (0.5 < F <=1, desired filling rate of the leaf)
    - put a pointer to this leaf in the parent node (recursively create if not exists)
- Discussion
  - creates B+-tree from the left to the right and from bottom to top
  - no node/leaf split will ever occur during the bulk-loading
  - easy to implement (assuming external sorting operator is available)

How to choose F here?

Bulk-Loading Example

k=k*=1



Bulk-Loading: Discussion

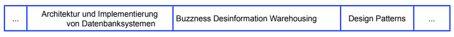
- Cost
  - Cost for (external) sorting
  - Linear cost in the number of leaves for tree creation
- Advantage: Algorithm creates contiguous output of leaves as a side-effect (fully defragmented tree)
- Many other algorithms exist (e.g. Bulk-loading of an index already having entries)
- Literature
  - Lars Arge: The Buffer Tree: A New Technique for Optimal I/O-Algorithms. WADS 1995: 334-345 .
  - Jochen Van den Bercken, Bernhard Seeger: An Evaluation of Generic Bulk Loading Techniques. VLDB 2001: 461-470.

Performance Optimizations of B+-Trees

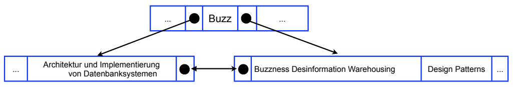
1. Problem: height of the tree has a huge impact on performance
   - Goal: maximize fan-out (number of children)
   - Approaches:
     - Prefix B+-trees
     - Prefix/suffix-compression
     - Large pages
2. Problem: B+-trees not optimized for main memory and CPU-caches
   - Goal: optimize cache usage of trees
   - Approaches:
     - Cache conscious B+-trees

Prefix-B+-Trees

- What happens if we want to use very long keys in the B+-tree?

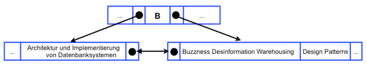

...
Architektur und Implementierung von Datenbanksystemen
Buzzness Desinformation Warehousing
Design Patterns
...

- Considerable space required for keys => less entries per node/ leaf => low fan-out => high tree!
- similar problems as in dense indexes
- Note: any string key in-between “Archi...” < key <= “Design..” may serve as a pivot in the node.
- Solution: modify split-operation such that a shorter pivot is generated in the node.


...
Buzz
...
...
Architektur und Implementierung von Datenbanksystemen
Buzzness Desinformation Warehousing
Design Patterns
...

Prefix-Compression

- Previous slide: any string key in-between “Archi...” < key <= “Design..” may serve as a pivot in the node.
- Improvement: choose key such that key is minimal, i.e., key is a prefix for keys K_i und K_{i+1}, if:
  - K_i < key <= K_{i+1}
  - no other key‘ exists such that K_i < key‘ <= K_{i+1} and len(key‘) < len(key)


...
B
...
...
Architektur und Implementierung von Datenbanksystemen
Buzzness Desinformation Warehousing
Design Patterns
...

- Impact:
  - subtrees have a common prefix
  - for any node N it holds: the children of N do not have to store the prefix anymore
  - But: for ISAM we require the entire key on the leaf level!
  - Prefix B+-trees are in fact a generalization of digital trees

Suffix-Compression (aka front coding)

- Terminology:
  - **Prefix**-compression: the **suffix** will be omitted
  - **Suffix**-compression: the **prefix** will be omitted
- Idea:
  For each key K_i only store the difference to its predecessor key K_{i-1}
- Example:
  - Hugo, Hummel, Hummer, Hund, Hupen
  - (0, Hugo), (2, mmel), (5, r), (2, nd), (2, pen)

suffix

F = length of the common prefix

requires appropriate physical representation, e.g., byte (but **not** integer)

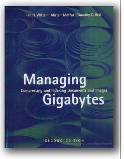
Suffix-Compression: Discussion

- Actual low-level storage layout should be based on statistical information on the data (if not we may not gain anything but rather inflate the data)
- Disadvantage: binary search on data does not work anymore
- Improvement: Partial front coding

Partial Suffix-Compression (aka partial front coding)

- Idea:
  same as front coding, but: store every k-th entry uncompressed
- Example: (k=4)
  - Hugo, Hummel, Hummer, Hund, Hupen, Husky, Husten, ...
  - (0, Hugo), (2, mmel), (5, r), (2, nd), (0, Hupen) , (2, sky), (3, ten), ...

binary search can use these entries, other entries may be reached by scanning from these “hub”-entries



- Literature: Witten, Moffat, Bell: Managing Gigabytes

Performance Optimizations of B+-Trees

1. Problem: height of the tree has a huge impact on performance
   - Goal: maximize fan-out (number of children)
   - Approaches:
     - Prefix B+-trees
     - Prefix/suffix-compression
     - Large pages
2. Problem: B+-trees not optimized for main memory and CPU-caches
   - Goal: optimize cache usage of trees
   - Approaches:
     - Cache conscious B+-trees

Large Pages

Scenario: double page size => double node/leaf size

- Advantages:
  - Fan-out increases => probability that height of the tree decreases
  - I/O-cost do not increase much
- Disadvantages:
  - more clipping due to large pages
  - more dead data in the DB-buffer
  - time to perform binary search inside a node/leaf increases (slightly)
- How to fix:
  - "logically large pages/blocks": but have to be placed adjacent on disk

Large Index Pages

- Other idea:
  - use pages/blocks of different size
  - large index pages
  - small leaf pages
- Disadvantages:
  - DB-buffer typically only supprts a fixed page size

Performance Optimizations of B+-Trees

1. Problem: height of the tree has a huge impact on performance
   - Goal: maximize fan-out (number of children)
   - Approaches:
     - Prefix B+-trees
     - Prefix/suffix-compression
     - Large pages
2. Problem: B+-trees not optimized for main memory and CPU-caches
   - Goal: optimize cache usage of trees
   - Approaches:
     - Cache conscious B+-trees

Cache-Conscious B+-Trees

- Problem: B+-trees were designed for external memory (hard disks)
- How to optimize B+-trees such that they perform well w.r.t. the different caches?
- two major approaches:
  - cache-oblivious B+-trees
    - no knowledge (obliviousness) on the precise characteristics of the memory hierarchy
    - generic approach
    - not taylored towards a specific architecture

    (the topic 'cache oblivious-Algos.' could actually fill a separate lecture.)
  - cache-conscious B+-trees
    - precise knowledge (consciousness) on the precise characteristics of the memory hierarchy
    - requires more fine-tuning
    - taylored towards a specific architecture

Overview on Techniques

- Literature: Goetz Graefe, Per-Åke Larson: B-Tree Indexes and CPU Caches. ICDE 2001:349-358.

B-tree Indexes and CPU Caches

Goetz Graefe and Per-Åke Larson
Microsoft
{goetzg, palarson}@microsoft.com

Abstract

Since many existing techniques for exploiting CPU caches in the implementation of B-tree indexes have not been discussed in the literature, most of them are surveyed here. Rather than providing a detailed performance evaluation for one or two of them on some specific contemporary hardware, the purpose here is to survey and to make widely available this heretofore-folkloric knowledge in order to enable, structure, and hopefully stimulate future research.

1. Introduction

Today, most DBMS installations are "compute bound", at least after sufficient disk drives and indexes have been added. However, the bottlenecks are actually not CPU cycles for instruction execution but cache faults and pipeline stalls. At the present time, many product developers believe that the traditional goal of reducing instruction path length is much less important than techniques that improve cache faults and branch prediction.

This brief paper is a summary of knowledge, much of it folkloric, about how to adapt database indexing software, in particular B-tree code, to memory hierarchies with multiple levels of caches. Given the current trends in hardware design that require increasing sophistication from system software and compilers, for example Intel's 64-bit processors or Transmeta's innovative processors that employ on-the-fly translation among different machine codes [18], this survey might be useful to both database researchers and implementers of commercial products.

There are just a few principal sources of ideas behind these techniques. Unfortunately, they often conflict, and careful engineering tradeoffs are required, if possible supported by rigorous research and measurements on the specific target architecture if it is known. First, anything that reduces code complexity and code size is good since it reduces cache faults for instructions. In particular, reducing the need for "if", "case", and "while" statements as well as virtual functions reduces erroneous predictions of conditional and computed branches. Second, anything that helps that compresses or shortens B-tree entries, in leaves or in interior nodes. In fact, compaction of database structure is a continuing goal, unaffected by the continually decreasing prices for disk space and memory, because the higher levels of the memory hierarchy continue to be small and expensive (on-chip and on-board caches) and because the bandwidth between all levels of the memory hierarchy seems eternally insufficient. Third, many of the techniques to deal with the delay (access and transfer time) between memory and disk can be re-applied to CPU caches. Finally, since B-trees and (comparison-based) sorting are related in many interesting ways, many of the techniques that make sorting more cache-efficient can be re-applied to B-tree indexes.

1.1 Scope of this Survey

Most of this paper focuses on exact match index searches as typical in OLTP applications and in index nested loops joins. Inasmuch as materialized views satisfy their purpose, even decision support (DSS) and OLAP applications behave like read-only OLTP applications, i.e., lots of index searches rather than large sort and hash operations, but no issues with respect to concurrency control and recovery. Therefore, materialized views and their effect on OLAP and DSS applications increase the importance of very efficient implementation of B-tree indexes. Range queries typically are combinations of exact match searches for the lower or upper end of the range, combined with sequential scans. This also applies to searches required for existence checks or semi-joins. Considerations specific to insert, update, and delete operations as well as index creation or page compaction are mentioned where warranted. Concurrency control and other transactional issues are generally ignored here since they are largely orthogonal to the issue of cache efficiency.

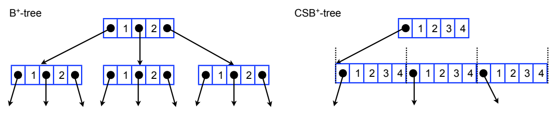
We only outline the key ideas for each technique, without attempting to list explicitly every thinkable variant and combination. In particular, we do not consider how to tune data structures and algorithms to multiple levels of caches and multiple sizes of cache lines, extended-data-out memory and other burst transfer and read-ahead techniques, etc. Moreover, we do not consider in detail which techniques complement or inhibit each other, or how specifically these techniques should be realized in data structures and code. Without doubt, there is plenty of room for future experimental research in this area. The purpose of this paper is to stimulate and to provide a starting point for such advanced research, not to preempt it.

349

1063-6382/01 $10.00 © 2001 IEEE

CSB+-Trees

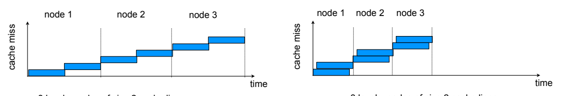
- =Cache conscious B+-Trees
- Idea:
  - level-wise layout of nodes
  - omit all node pointers except the first one
  - Impact: doubles fan-out of nodes



- search becomes up to 35% faster. But: updates become 30% slower!
- Literature: Jun Rao, Kenneth A. Ross: Making B+-Trees Cache Conscious in Main Memory. SIGMOD Conference 2000: 475-486

Prefetching: pB+-Trees

- Idea:
  - node size is a multiple of a cache line
  - prefetch all cache lines before starting the binary search inside a node
  - Layout: first store all keys than all pointers



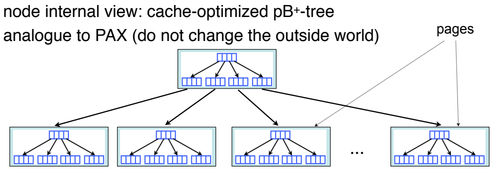
3 levels: nodes of size 2 cache lines

3 levels: nodes of size 2 cache lines using prefetching

- >2 performance improvement for both search and update operations
- orthogonal to CSB+-trees
- Literature: Shimin Chen, Phillip B. Gibbons, Todd C. Mowry: Improving Index Performance through Prefetching. SIGMOD Conference 2001

Fractal Prefetching: fpB+-Trees

- Idea:
  - tree of trees
  - node external view: disk-optimized B+-tree
  - node internal view: cache-optimized pB+-tree
  - analogue to PAX (do not change the outside world)



- Literature: Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, Gary Valentin: Fractal prefetching B+-Trees: optimizing both cache and disk performance. SIGMOD Conference 2002: 157-168

Next Topic: One-dimensional Indexes (continued).

Hashing.

Tree Structures in Main Memory: Sorted Arrays and CSS-trees.