Indexing

Jan Chomicki
University at Buffalo
## Storage hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Main memory</th>
<th>Disk</th>
<th>Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very fast (nanosec)</td>
<td>Fast (10 nanosec)</td>
<td>Slower (millisec)</td>
<td>Slow (sec)</td>
</tr>
<tr>
<td>Very small (MB)</td>
<td>Small (GB)</td>
<td>Bigger (TB)</td>
<td>Very big (PB)</td>
</tr>
<tr>
<td>Built-in</td>
<td>Expensive</td>
<td>Cheap</td>
<td>Very cheap</td>
</tr>
</tbody>
</table>
### Storage hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Main memory</th>
<th>Disk</th>
<th>Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very fast</td>
<td>Fast</td>
<td>Slower</td>
<td>Slow</td>
</tr>
<tr>
<td>(nanosec)</td>
<td>(10 nanosec)</td>
<td>(millisec)</td>
<td>(sec)</td>
</tr>
<tr>
<td>Very small</td>
<td>Small</td>
<td>Bigger</td>
<td>Very big</td>
</tr>
<tr>
<td>(MB)</td>
<td>(GB)</td>
<td>(TB)</td>
<td>(PB)</td>
</tr>
<tr>
<td>Built-in</td>
<td>Expensive</td>
<td>Cheap</td>
<td>Very cheap</td>
</tr>
</tbody>
</table>

### Disks:
- Data is stored on concentric circular **tracks**
- Each track is divided into fixed-length **sectors**
- A **block** (page) consists of one or more multiple contiguous hardware sectors
- Between main memory and disk, the data is moved in **blocks**
- Block size: 4K-64K bytes
Hard disk drive
Disk I/O

Provide the disk I/O controller with:

- block disk address (device-dependent)
- buffer main memory address

Basic operations:

- READ: transfer data from disk to buffer
- WRITE: transfer data from buffer to disk

What happens:

1. seek: position the read/write head (1−10 ms)
2. rotational delay: wait for the beginning of the block (0−10 ms)
3. data transfer: (<1 ms)
Disk I/O

Provide the disk I/O controller with:

- block disk address (device-dependent)
- buffer main memory address
Disk I/O

Provide the disk I/O controller with:

- block disk address (device-dependent)
- buffer main memory address

Basic operations:

- READ: transfer data from disk to buffer
- WRITE: transfer data from buffer to disk
Disk I/O

Provide the disk I/O controller with:
- block disk address (device-dependent)
- buffer main memory address

Basic operations:
- **READ**: transfer data from disk to buffer
- **WRITE**: transfer data from buffer to disk

What happens:
1. **seek**: position the read/write head (1 – 10 ms)
2. **rotational delay**: wait for the beginning of the block (0 – 10 ms)
3. **data transfer**: (< 1 ms)
How to do it faster

- Transfer blocks on the same cylinder: no seek
- Transfer contiguous blocks on the same track: no seek, no rotational delay
- Use main memory buffers: no disk access at all, prefetching
- Parallelism: data striping.
How to do it faster

Transfer blocks on the **same cylinder**:
- no seek
How to do it faster

Transfer blocks on the same cylinder:
- no seek

Transfer contiguous blocks on the same track:
- no seek
- no rotational delay
How to do it faster

Transfer blocks on the **same cylinder**:  
- no seek

Transfer contiguous blocks on the **same track**:  
- no seek  
- no rotational delay

Use main memory **buffers**:  
- no disk access at all  
- prefetching
How to do it faster

Transfer blocks on the same cylinder:
- no seek

Transfer contiguous blocks on the same track:
- no seek
- no rotational delay

Use main memory buffers:
- no disk access at all
- prefetching

Parallelism:
- data striping.
Physical data organization

- **Field**: individual data item (integer, real, fixed-length string, variable-length string, disk pointer, ...)
- **Record**: sequence of fields.
- **Record schema (or type)**: sequence of field names and their corresponding data types.
- **File**: collection of records with the same schema (typically), usually spanning a number of blocks.
- **Blocks** contain typically more than one record. If the records are too big, they span more than one block.
- **Blocking** speeds up data access by eliminating some seeks and rotational delays.

Block access is a cost unit for file operations.
Physical data organization

**Field:** individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)
Physical data organization

**Field**: individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)

**Record**: sequence of fields.
Physical data organization

**Field**: individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)

**Record**: sequence of fields.

**Record schema (or type)**: sequence of field names and their corresponding data types.
Physical data organization

**Field**: individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)

**Record**: sequence of fields.

**Record schema** (or type): sequence of field names and their corresponding data types.

**File**: collection of records with the same schema (typically), usually spanning a number of blocks.
Physical data organization

Field: individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)

Record: sequence of fields.

Record schema (or type): sequence of field names and their corresponding data types.

File: collection of records with the same schema (typically), usually spanning a number of blocks.

Blocks contain typically more than one record. If the records are too big, they span more than one block.
Physical data organization

**Field**: individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)

**Record**: sequence of fields.

**Record schema (or type)**: sequence of field names and their corresponding data types.

**File**: collection of records with the same schema (typically), usually spanning a number of blocks.

Blocks contain typically more than one record. If the records are too big, they span more than one block.

Blocking speeds up data access by eliminating some seeks and rotational delays.
Physical data organization

**Field**: individual data item (integer, real, fixed-length string, variable-length string, disk pointer,...)

**Record**: sequence of fields.

**Record schema** (or type): sequence of field names and their corresponding data types.

**File**: collection of records with the same schema (typically), usually spanning a number of blocks.

Blocks contain typically more than one record. If the records are too big, they span more than one block.

Blocking speeds up data access by eliminating some seeks and rotational delays.

**Block access is a cost unit for file operations.**
Record layout

Fixed-length records

- Header
- Field 1
- ···
- Field k

Variable-length records

- Header
- Offset 1
- Offset 2
- ···
- Offset
- Field 1
- ···
- Field k

Header information: record type, record length...
Record layout

**Fixed-length records**

<table>
<thead>
<tr>
<th>Header</th>
<th>Field₁</th>
<th>⋯</th>
<th>Fieldₖ</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Record layout

## Fixed-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>Field$_1$</th>
<th>···</th>
<th>Field$_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Variable-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>Offset of Field$_1$</th>
<th>···</th>
<th>Offset of Field$_k$</th>
<th>Field$_1$</th>
<th>···</th>
<th>Field$_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Record layout

### Fixed-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>$Field_1$</th>
<th>⋮</th>
<th>$Field_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Variable-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>Offset of $Field_1$</th>
<th>⋮</th>
<th>Offset of $Field_k$</th>
<th>$Field_1$</th>
<th>⋮</th>
<th>$Field_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Header information: record type, record length...
Block layout
## Block layout

### Fixed-length records

<table>
<thead>
<tr>
<th>Header info</th>
<th>$Record_1$</th>
<th>$\cdots$</th>
<th>$Record_n$</th>
<th>Free space</th>
</tr>
</thead>
</table>
## Block layout

### Fixed-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>$Record_1$</th>
<th>⋯</th>
<th>$Record_n$</th>
<th>Free space</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Variable-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>Offset of $Rec_1$</th>
<th>⋯</th>
<th>Offset of $Rec_n$</th>
<th>Free space</th>
<th>$Rec_n$</th>
<th>⋯</th>
<th>$Rec_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Header:
- number of records in the block
- USED/UNUSED bits for every record slot
- DELETED bits for every record slot
- next block in the file
- timestamp
## Block layout

### Fixed-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>Record₁</th>
<th>⋯</th>
<th>Recordₙ</th>
<th>Free space</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Variable-length records

<table>
<thead>
<tr>
<th>Header</th>
<th>Offset of Rec₁</th>
<th>⋯</th>
<th>Offset of Recₙ</th>
<th>Free space</th>
<th>Recₙ</th>
<th>⋯</th>
<th>Rec₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Header:**
- number of records in the block
- USED/UNUSED bits for every record slot
- DELETED bits for every record slot
- next block in the file
- timestamp
Indexing

A search key is a set of fields used to determine the location of a record within a file. Operations include:

- **SCAN**: Fetch all records in the file.
- **LOOKUP**: Find all records satisfying an equality condition, e.g., $A = a$.
- **INSERT**: Add the record to the file (without considering duplicates).
- **DELETE**: Find and remove the record from the file, given a key value.
Indexing

Search key

- a set of fields $A$
- record location determined by its (search) key value

Operations

- SCAN: fetch all records in the file.
- LOOKUP: find all records satisfying an equality condition $A = a$.
- INSERT: add the record to the file (without considering duplicates).
- DELETE: find and remove the record from the file, given a key value.
Indexing

Search key

- a set of fields $A$
- record location determined by its (search) key value

Operations

- **SCAN**: fetch all records in the file.
- **LOOKUP**: find all records satisfying an equality condition $A = a$.
- **INSERT**: add the record to the file (without considering duplicates)
- **DELETE**: find and remove the record from the file, given a key value.
Heap

- no ordering of records, no special organization of the file
- random access to blocks through a block directory, or sequential access to blocks using a linked list.
Heap

- no ordering of records, no special organization of the file
- random access to blocks through a block directory, or sequential access to blocks using a linked list.

Parameters

- \( n \): records in the file
- \( R \): records per block (blocking factor)
- \( B \): blocks in the file (\( B = n/R \))

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of block accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>( B )</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>( B/2 ) (average), ( B ) (worst case)</td>
</tr>
<tr>
<td>INSERT</td>
<td>2 (read and rewrite)</td>
</tr>
<tr>
<td>DELETE</td>
<td>number for LOOKUP +1 (rewrite)</td>
</tr>
</tbody>
</table>
Sorted file

Parameters

- **B**
  - Blocks in the file.

Operations

- **SCAN**
  - Number of block accesses

- **LOOKUP**
  - \( \log_2(B) \) (worst case)

- **INSERT**
  - Number for LOOKUP + \( B \) (average)

- **DELETE**
  - Number for LOOKUP + \( B \) (average)
Sorted file

- ordering of records (ascending or descending) according to the key
- random access to blocks through a block directory.
Sorted file

- ordering of records (ascending or descending) according to the key
- random access to blocks through a block directory.

Parameters

$B$ blocks in the file.
Sorted file

- ordering of records (ascending or descending) according to the key
- random access to blocks through a block directory.

Parameters

$B$ blocks in the file.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of block accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>$B$</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>$\log_2(B)$ (worst case)</td>
</tr>
<tr>
<td>INSERT</td>
<td>number for LOOKUP + $B$ (average)</td>
</tr>
<tr>
<td>DELETE</td>
<td>number for LOOKUP + $B$ (average)</td>
</tr>
</tbody>
</table>
Optimizations

- insertion: keeping a separate overflow file which is merged with the data file off-line.
- deletion: marking the record as deleted, without shifting other records.
Sorting

Parameters

Two-Phase Multiway Merge-Sort

repeat

fill all the buffers with new tuples from the file;
sort them in main memory;
write the result to disk as a sorted sublist;

until the input is exhausted;

perform an \((M-1)\)-way merge of the sorted sublists;

Analysis

required:

\[ B \leq M (M - 1) \]

otherwise: apply the algorithm recursively
Sorting

Parameters

- $B$ blocks in the file
- $M$ main memory buffers

Two-Phase Multiway Merge-Sort

repeat

fill all the buffers with new tuples from the file;
sort them in main memory;
write the result to disk as a sorted sublist;
until the input is exhausted;
perform an $(M-1)$-way merge of the sorted sublists;

Analysis

required:

$B \leq M (M - 1)$

otherwise: apply the algorithm recursively
Sorting

Parameters

\( B \) blocks in the file
\( M \) main memory buffers

Two-Phase Multiway Merge-Sort

repeat
  fill all the buffers with new tuples from the file;
  sort them in main memory;
  write the result to disk as a sorted sublist;
until the input is exhausted;
perform an \((M-1)\)-way merge of the sorted sublists;
Sorting

Parameters

\( B \) blocks in the file
\( M \) main memory buffers

Two-Phase Multiway Merge-Sort

repeat
  fill all the buffers with new tuples from the file;
  sort them in main memory;
  write the result to disk as a sorted sublist;
until the input is exhausted;
perform an \((M-1)\)-way merge of the sorted sublists;

Analysis

- required: \( B \leq M(M - 1) \)
- otherwise: apply the algorithm recursively
Hashed file

- $N$ buckets, each consisting of one primary and an unlimited number of overflow blocks
- hashing function $h : \text{Keys} \rightarrow \{0, 1, \ldots, N - 1\}$
- keys are integers or converted to integers
- record located in bucket $i$ iff its key hashes to $i$
- bucket directory with $N$ entries
Hashed file

- $N$ buckets, each consisting of one primary and an unlimited number of overflow blocks
- hashing function $h : \text{Keys} \rightarrow \{0, 1, \ldots, N - 1\}$
- keys are integers or converted to integers
- record located in bucket $i$ iff its key hashes to $i$
- bucket directory with $N$ entries

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of block accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>$B$</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>$B/(2N)$ (average)</td>
</tr>
<tr>
<td></td>
<td>$B/N$ (worst case)</td>
</tr>
<tr>
<td>INSERT</td>
<td>2 (read and rewrite)</td>
</tr>
<tr>
<td>DELETE</td>
<td>number for LOOKUP +1 (rewrite)</td>
</tr>
</tbody>
</table>
Indexed file

Indexed files have two kinds of records: data records and index records. A field or a combination of fields is chosen as the index key. Index record: a pair (Value, Address) where Value is a unique value for primary indexes, a nonunique value for secondary indexes, and Address is a block address for sparse indexes, a record id (rid) for dense indexes.

datafile blocks: if file is sorted on the index key (clustered), then index can be sparse
index blocks: always sorted
Indexed file

Indexed files have two kinds of records: data records and index records. A field or a combination of fields is chosen as the index key.
Indexed file

Indexed files have two kinds of records: data records and index records. A field or a combination of fields is chosen as the index key.

**Index record:** a pair 
(Value, Address) where Value is
- a unique value for primary indexes
- a nonunique value for secondary indexes

and Address is
- a block address for sparse indexes
- a record id (rid) for dense indexes.
Indexed file

Indexed files have two kinds of records: data records and index records. A field or a combination of fields is chosen as the index key.

**Index record**: a pair (Value, Address) where Value is
- a unique value for primary indexes
- a nonunique value for secondary indexes
and Address is
- a block address for sparse indexes
- a record id (rid) for dense indexes.

- **datafile blocks**: if file is sorted on the index key (clustered), then index can be sparse
- **index blocks**: always sorted
More index varieties

Multilevel index viewed as a file. Higher levels are sparse.

Static index: the index does not change after it has been built, datafile block overflows are handled by allocating new overflow blocks from time to time.

Dynamic index: the index changes dynamically, no overflow blocks.
More index varieties

Multilevel index

- Index viewed as a file
- Higher levels are sparse.
More index varieties

**Multilevel index**
- index viewed as a file
- higher levels are sparse.

**Static index**
- the index does not change after it has been built
- datafile block overflows are handled by allocating new overflow blocks
- from time to time the index is rebuilt.
More index varieties

**Multilevel index**
- index viewed as a file
- higher levels are sparse.

**Static index**
- the index does not change after it has been built
- datafile block overflows are handled by allocating new overflow blocks
- from time to time the index is rebuilt.

**Dynamic index**
- the index changes dynamically
- no overflow blocks.
B-trees

Basic features
- multilevel, dynamic index
- can be sparse or dense, primary or secondary
- here, dense primary:
  - datafile is not sorted (heap)
  - all index levels are sorted

Balanced trees:
- all leaves at the same level
- updates "ripple up" the tree

Guaranteed minimum fill ratio (at least 50%) in nonroot nodes
Max. number of index records per block = 2d (min. number = d)
B-trees

Basic features

- multilevel, dynamic index
## B-trees

### Basic features
- multilevel, dynamic index
- can be sparse or dense, primary or secondary

Balanced trees:
- all leaves at the same level
- updates "ripple up" the tree

Guaranteed minimum fill ratio (at least 50%) in nonroot nodes

Max. number of index records per block = 2
(min. number = 1)
B-trees

Basic features

- multilevel, dynamic index
- can be sparse or dense, primary or secondary
- here **dense primary**:
  - datafile is not sorted (heap)
  - all index levels are sorted
B-trees

Basic features

- multilevel, dynamic index
- can be sparse or dense, primary or secondary
- here **dense primary**:
  - datafile is not sorted (heap)
  - all index levels are sorted
- balanced trees:
  - all leaves at the same level
  - updates “ripple up” the tree
B-trees

Basic features

- multilevel, dynamic index
- can be sparse or dense, primary or secondary
- here dense primary:
  - datafile is not sorted (heap)
  - all index levels are sorted
- balanced trees:
  - all leaves at the same level
  - updates “ripple up” the tree
- guaranteed minimum fill ratio (at least 50%) in nonroot nodes
B-trees

Basic features

- multilevel, dynamic index
- can be sparse or dense, primary or secondary
- here dense primary:
  - datafile is not sorted (heap)
  - all index levels are sorted
- balanced trees:
  - all leaves at the same level
  - updates “ripple up” the tree
- guaranteed minimum fill ratio (at least 50%) in nonroot nodes
- max. number of index records per block = 2d (min. number = d)
B-trees are very low
B-trees are very low

Number of nodes on any path from the root to any leaf = \( i \).

Number of records in the file \( n \)
B-trees are very low

Number of nodes on any path from the root to any leaf = $i$.

Number of records in the file $n$

Number of leaf index blocks $\leq n/d$

Thus:

\[
\frac{n}{d^{i-1}} \geq 2 \implies d^{i-1} \leq \frac{n}{2} \implies i-1 \leq \log_d \left( \frac{n}{2} \right) \implies i \leq 1 + \log_d \left( \frac{n}{2} \right)
\]
B-trees are very low

Number of nodes on any path from the root to any leaf = \( i \).

Number of records in the file \( n \)

Number of leaf index blocks \( \leq \frac{n}{d} \)

Number of parents of leaf index blocks \( \leq \frac{n}{d^2} \)
B-trees are very low

Number of nodes on any path from the root to any leaf = \( i \).

- Number of records in the file = \( n \)
- Number of leaf index blocks \( \leq n/d \)
- Number of parents of leaf index blocks \( \leq n/d^2 \)
  ...
- Number of index blocks immediately below the root \( \leq n/d^{i-1} \)
B-trees are very low

Number of nodes on any path from the root to any leaf $= i$.

- Number of records in the file $n$
- Number of leaf index blocks $\leq n/d$
- Number of parents of leaf index blocks $\leq n/d^2$

... $\leq n/d^{i-1}$

Number of index blocks immediately below the root $\geq 2$
B-trees are very low

Number of nodes on any path from the root to any leaf = \( i \).

Number of records in the file \( n \)

Number of leaf index blocks \( \leq n/d \)

Number of parents of leaf index blocks \( \leq n/d^2 \)

... 

Number of index blocks immediately \( \leq n/d^{i-1} \)

below the root \( \geq 2 \)

Thus:

\[
\frac{n}{d^{i-1}} \geq 2 \\
\Rightarrow d^{i-1} \leq n/2 \\
\Rightarrow i - 1 \leq \log_d(n/2) \\
\Rightarrow i \leq 1 + \log_d(n/2)
\]
B-trees: lookup

Lookup algorithm

current block := root;
repeat
  if current block is a leaf
    then retrieve index record with key = Key;
    retrieve the data record with rid= Record Id;
  else find the last index record in the current block
    with key \leq Key and address = Addr;
    address of current block := Addr
  until leaf reached;
B-trees: lookup

Lookup algorithm

```plaintext
current block := root;
repeat
    if current block is a leaf
        then retrieve index record with key = Key;
           retrieve the data record with rid= Record Id;
    else find the last index record in the current block
        with key ≤ Key and address = Addr;
        address of current block := Addr
until leaf reached;
```
B-trees: insertion

Insertion algorithm

insert the record into a datafile block;
lookup the leaf index block to insert;
if enough room then store the record
else /*OVERFLOW*/
allocate a new leaf index block;
redistribute the records between the old and the new blocks;
update the parent index block
to reflect the new distribution of records;
Record redistribution should preserve ordering and B-tree minimum fill ratio.
Updating the parent index block may lead to another index block
OVERFLOW: handled using the same algorithm.
Updates may ripple up to the root.
B-trees: insertion

Insertion algorithm

insert the record into a datafile block;
lookup the leaf index block to insert;
if enough room then store the record
else /*OVERFLOW*/
    allocate a new leaf index block;
    redistribute the records between the old and the new blocks;
    update the parent index block
to reflect the new distribution of records;

Record redistribution should preserve ordering and B-tree minimum fill ratio.
Updating the parent index block may lead to another index block
OVERFLOW: handled using the same algorithm.
Updates may ripple up to the root.
B-trees: insertion

Insertion algorithm

insert the record into a datafile block;
lookup the leaf index block to insert;
if enough room then store the record
else /*OVERFLOW*/
    allocate a new leaf index block;
    redistribute the records between the old and the new blocks;
    update the parent index block
to reflect the new distribution of records;

- Record redistribution should preserve ordering and B-tree minimum fill ratio.
- Updating the parent index block may lead to another index block
OVERFLOW: handled using the same algorithm.
- Updates may ripple up to the root.
B-trees: deletion

Deletion algorithm

- lookup the index record given the key;
- delete the data record;
- delete the index record;
- if less than $d$ records remain in the block
  then /*UNDERFLOW*/
    - consider left (or right) sibling block;
    - if sibling contains $d$ records
      then combine it with underfull block;
        - release one block to the file system;
        - update the parent index block;
    - else/*sibling contains more than $d$ records*/
      redistribute records between the two blocks;
      update the parent index block;
B-trees: deletion

Deletion algorithm

lookup the index record given the key;
delete the data record;
delete the index record;
if less than $d$ records remain in the block
then /*UNDERFLOW*/
    consider left (or right) sibling block;
    if sibling contains $d$ records
    then combine it with underfull block;
    release one block to the file system;
    update the parent index block;
else /*sibling contains more than $d$ records*/
    redistribute records between the two blocks;
    update the parent index block;

Record redistribution should preserve ordering and B-tree minimum fill ratio.
B-trees: UNDERFLOW

Propagation

Updating the parent index block during deletion may lead to an index block UNDERFLOW: handled using the same algorithm.

Special treatment of the root

if root has at least two records remaining then no UNDERFLOW else new root := child of old root; release the old root block

Some B-tree implementations underflows ignored index rebuilt from time to time
B-trees: UNDERFLOW

Propagation

Updating the parent index block during deletion may lead to an index block UNDERFLOW: handled using the same algorithm.
B-trees: UNDERFLOW

Propagation

Updating the parent index block during deletion may lead to an index block UNDERFLOW: handled using the same algorithm.

Special treatment of the root

if root has at least two records remaining
then no UNDERFLOW
else new root := child of old root;
    release the old root block
B-trees: UNDERFLOW

Propagation

Updating the parent index block during deletion may lead to an index block UNDERFLOW: handled using the same algorithm.

Special treatment of the root

if root has at least two records remaining
then no UNDERFLOW
else new root := child of old root;
    release the old root block

Some B-tree implementations

- underflows ignored
- index rebuilt from time to time
B-trees: analysis

Parameters

$B = \frac{n}{R}$

blocks in the file.

$2d$ index records per block (blocking factor): $2d > R$

an extra block access from the index to the datafile

Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of block accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>$B$</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>$\leq 2 + \log d \left(\frac{n}{2}\right)$</td>
</tr>
<tr>
<td>INSERT (typically)</td>
<td>$\text{number for LOOKUP} + 2$</td>
</tr>
<tr>
<td>INSERT (worst case)</td>
<td>$2 \times \text{number for LOOKUP} + 2$</td>
</tr>
<tr>
<td>DELETE</td>
<td>like INSERT</td>
</tr>
</tbody>
</table>
B-trees: analysis

Parameters

\[ B = \frac{n}{R} \text{ blocks in the file.} \]

\[ 2d \text{ index records per block (blocking factor): } 2d > R \]

an extra block access from the index to the datafile
B-trees: analysis

Parameters

\[ B = \frac{n}{R} \text{ blocks in the file.} \]

\[ 2d \text{ index records per block (blocking factor): } 2d > R \]

an extra block access from the index to the datafile

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of block accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCAN</td>
<td>( B )</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>( \leq 2 + \log_d(\frac{n}{2}) )</td>
</tr>
<tr>
<td>INSERT (typically)</td>
<td>number for LOOKUP + 2</td>
</tr>
<tr>
<td>INSERT (worst case)</td>
<td>( 2 \times \text{number for LOOKUP} + 2 )</td>
</tr>
<tr>
<td>DELETE</td>
<td>like INSERT</td>
</tr>
</tbody>
</table>
Bitmap index

Indexes associated with individual columns.

Index record (Value, BitVector):

BitVector has one bit for every record in the file. The i-th bit of BitVector is set iff record i has Value in the given column.

Bitvectors typically compressed. Converted to sets of rids during query evaluation.

Bitmap indexes used where there are few domain values.
Indexes associated with individual columns.
Bitmap index

Indexes associated with individual columns.

Index record (Value, BitVector):

- BitVector has one bit for every record in the file
- ith bit of BitVector is set iff record i has Value in the given column
Bitmap index

Indexes associated with individual columns.

Index record (Value, BitVector):
- BitVector has one bit for every record in the file
- \( i \)th bit of BitVector is set iff record \( i \) has Value in the given column

Bitvectors
- typically compressed.
- converted to sets of rids during query evaluation.
Bitmap index

Indexes associated with individual columns.

Index record (Value, BitVector):
- BitVector has one bit for every record in the file
- $i$th bit of BitVector is set iff record $i$ has Value in the given column

Bitvectors
- typically compressed.
- converted to sets of rids during query evaluation.

Bitmap indexes used where there are few domain values.
Oracle: indexes

Create an index

```
CREATE INDEX Index-name ON Rel (Attr1, ... , Attrn)
```

Drop an index

```
DROP INDEX Rel.Index-name.
```

Indexes are automatically created on attributes defined as `PRIMARY KEY` or `UNIQUE`.

Bitmap indexes:

```
CREATE BITMAP INDEX
```
Oracle: indexes

Create an index

CREATE INDEX  \textit{Index-name}  
ON  \textit{Rel}(Attr1,\ldots,Attrn)

Indexes are automatically created on attributes defined as PRIMARY KEY or UNIQUE.

Bitmap indexes:

CREATE BITMAP INDEX
Oracle: indexes

Create an index

CREATE INDEX  Index-name  
  ON  Rel(Attr1,...,Attrn)

Drop an index

DROP INDEX  Rel.Index-name
Oracle: indexes

Create an index

CREATE INDEX  *Index-name*
    ON  *Rel*(*Attr1*,...,*Attrn*)

Drop an index

DROP INDEX  *Rel*.  *Index-name*

Indexes are automatically created on attributes defined as PRIMARY KEY or UNIQUE.
Create an index

CREATE INDEX Index-name
    ON Rel(Attr1,...,Attrn)

Drop an index

DROP INDEX Rel.Index-name

Indexes are automatically created on attributes defined as PRIMARY KEY or UNIQUE.

Bitmap indexes: CREATE BITMAP INDEX
Oracle: accessing tables

Each row is identified by the pseudo-column RowID consisting of file number, block number, and row number.

Different ways of accessing a table:
- TABLE ACCESS FULL (linear scan)
- TABLE ACCESS BY ROWID (single row)
- INDEX UNIQUE SCAN (single row through an index)
- INDEX RANGE SCAN (a range of rows through an index)
Oracle: accessing tables

Each row is identified by the pseudo-column RowID consisting of file number, block number, and row number.
Oracle: accessing tables

Each row is identified by the pseudo-column RowID consisting of file number, block number, and row number.

Different ways of accessing a table

**TABLE ACCESS FULL** (linear scan)

**TABLE ACCESS BY ROWID** (single row)

**INDEX UNIQUE SCAN** (single row through an index)

**INDEX RANGE SCAN** (a range of rows through an index)