Indexing

Jan Chomicki
University at Buffalo

Storage hierarchy

Cache	Main memory	Disk	Tape
Very fast (nanosec)	Fast (10 nanosec)	Slower (millisec)	Slow (sec)
Very small (MB)	Small (GB)	Bigger (TB)	Very big (PB)
Built-in	Expensive	Cheap	Very cheap

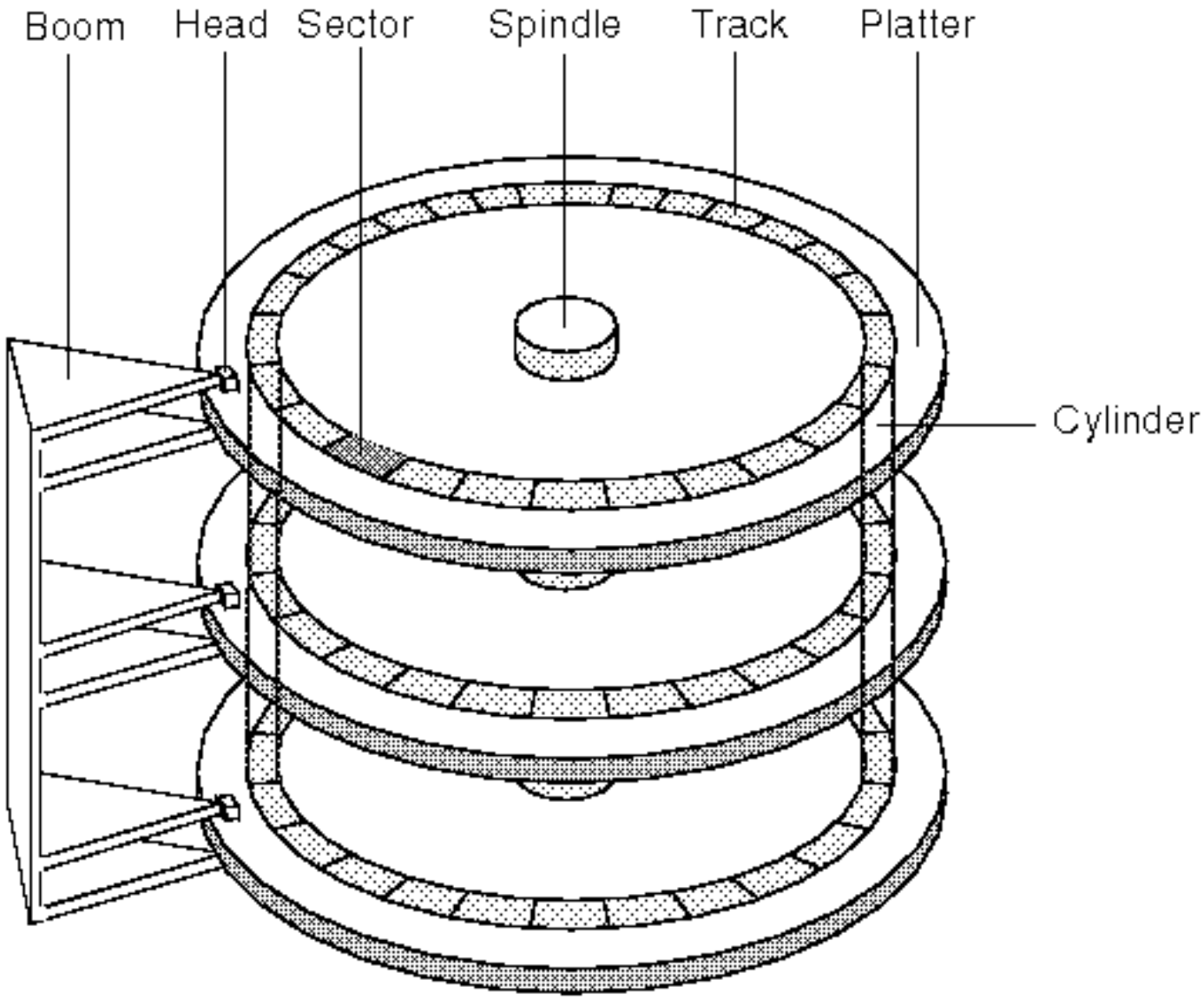
Storage hierarchy

Cache	Main memory	Disk	Tape
Very fast (nanosec)	Fast (10 nanosec)	Slower (millisec)	Slow (sec)
Very small (MB)	Small (GB)	Bigger (TB)	Very big (PB)
Built-in	Expensive	Cheap	Very cheap

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

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

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

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

Record layout

Fixed-length records

Header info	$Field_1$	$\cdots$	$Field_k$

Record layout

Fixed-length records

Header info	$Field_1$	$\cdots$	$Field_k$

Variable-length records

Header info	*Offset of* $Field_1$	$\cdots$	*Offset of* $Field_k$	$Field_1$	$\cdots$	$Field_k$

Record layout

Fixed-length records

Header info	$Field_1$	$\cdots$	$Field_k$

Variable-length records

Header info	*Offset of* $Field_1$	$\cdots$	*Offset of* $Field_k$	$Field_1$	$\cdots$	$Field_k$

Header information: record type, record length...

Block layout

Block layout

Fixed-length records

Header info	$Record_1$	$\cdots$	$Record_n$	Free space

Block layout

Fixed-length records

Header info	$Record_1$	$\cdots$	$Record_n$	Free space

Variable-length records

Header info	*Offset of* Rec_1	$\cdots$	*Offset of* Rec_n	Free space	Rec_n	$\cdots$	Rec_1

Block layout

Fixed-length records

Header info	$Record_1$	$\cdots$	$Record_n$	Free space

Variable-length records

Header info	*Offset of* Rec_1	$\cdots$	*Offset of* Rec_n	Free space	Rec_n	$\cdots$	Rec_1

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

Indexing

Search key

- a set of fields A
- record location determined by its (search) 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 ($B = n/R$)

Operation	Number of block accesses
SCAN	B
LOOKUP	$B/2$ (average), B (worst case)
INSERT	2 (read and rewrite)
DELETE	number for LOOKUP +1 (rewrite)

Sorted file

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.

Operation	Number of block accesses
SCAN	B
LOOKUP	$\log_2(B)$ (worst case)
INSERT	number for LOOKUP $+B$ (average)
DELETE	number for LOOKUP $+B$ (average)

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

Sorting

Parameters

B blocks in the file

M main memory buffers

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 $h : Keys \rightarrow \{0, 1, \ldots, N-1\}$
- 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 $h : Keys \rightarrow \{0, 1, \ldots, N-1\}$
- keys are integers or converted to integers
- record located in bucket i iff its key hashes to i
- bucket directory with N entries

Operation	Number of block accesses
SCAN	B
LOOKUP	$B/(2N)$ (average)
	B/N (worst case)
INSERT	2 (read and rewrite)
DELETE	number for LOOKUP +1 (rewrite)

Indexed file

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

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

B-trees

Basic features

- multilevel, dynamic index

B-trees

Basic features

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

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	$\leq n/d$

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	$\leq n/d^2$

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

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	≥ 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	≥ 2

Thus:

$$
\begin{aligned}
n/d^{i-1} &\geq 2 \\
\Rightarrow d^{i-1} &\leq n/2 \\
\Rightarrow i - 1 &\leq \log_d(n/2) \\
\Rightarrow i &\leq 1 + \log_d(n/2)
\end{aligned}
$$

B-trees: lookup

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 ≤ Key and address = Addr;
     address of current block := Addr
until leaf reached;
```

B-trees: insertion

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;
```

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

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

B-trees: analysis

Parameters

$B = n/R$ blocks in the file.

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

an extra block access from the index to the datafile

B-trees: analysis

Parameters

$B = n/R$ blocks in the file.

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

an extra block access from the index to the datafile

Operation	Number of block accesses
SCAN	B
LOOKUP	$\leq 2 + \log_d(n/2)$
INSERT (typically)	number for LOOKUP +2
INSERT (worst case)	$2 *$ number for LOOKUP+2
DELETE	like INSERT

Bitmap index

Bitmap index

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
- *i*th 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

Oracle: indexes

Create an index

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

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`.

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: accessing tables

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)