Storage hierarchy

Cache	Main memory	Disk	Tape
Very fast	Fast	Slower	Slow
Very small (KB)	Small (MB)	Bigger (GB)	Very big (TB)
Built-in	Expensive	Cheap	Dirt 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**.

Textbook: chapters 11, 12, and 13

Disk I/O

Provide the disk I/O hardware 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 ($1 - 20\ ms$)
2. **rotational delay:** wait for the beginning of the block ($0 - 8\ ms$)
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:

- disk arrays
- data striping.

Physical data organization

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

Files span 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:

Control info	$Field_1$	$\cdots$	$Field_k$

Variable-length records:

Ctrl. info	*Offset of field*$_1$	$\cdots$	*Offset of field*$_k$	$Field_1$	$\cdots$	$Field_k$

Control information: record type, record length...

Block layout

Fixed-length records:

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

Variable-length records:

Ctrl. info	*Offset of* Rec_1	$\cdots$	*Offset of* Rec_n	Free space	Rec_n	$\cdots$	Rec_1

Control information:

- number of records in the block
- USED/UNUSED bits for every record
- DELETED bits for every record
- next block in the file
- ...

Indexing

A **key** (one or more fields) A is selected for the file.

Record location determined by its key value.

Operations:

SCAN : fetch all records in the file.

EQUALITY SEARCH (LOOKUP) : find all records satisfying an equality condition $A = a$.

RANGE SEARCH : find all records satisfying a range search condition $a_1 \leq A \leq a_2$ (where a_1 can be $-\infty$ and a_2 can be $+\infty$).

INSERT : add the record to the file (without considering duplicates)

DELETE : find and remove the record from the file, given the key.

Heap

Features:

- 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:

B blocks in the file.

R records per block (blocking factor).

K records satisfying the range condition.

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

Sorted file

Features:

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

Parameters:

B blocks in the file.

R records per block (blocking factor).

K records satisfying the range condition.

Operation	Number of block accesses
SCAN	B
EQUALITY SEARCH	$\log_2(B)$ (worst case)
RANGE SEARCH	$\log_2(B) + (K/R)$ (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 main file off-line.
- deletion: marking the record as “deleted”, without shifting other records.

Hashed file

Features:

- 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
- hashing can also be used to build an index (*hash index*

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

Linear hashing [Litwin 1980]

Features:

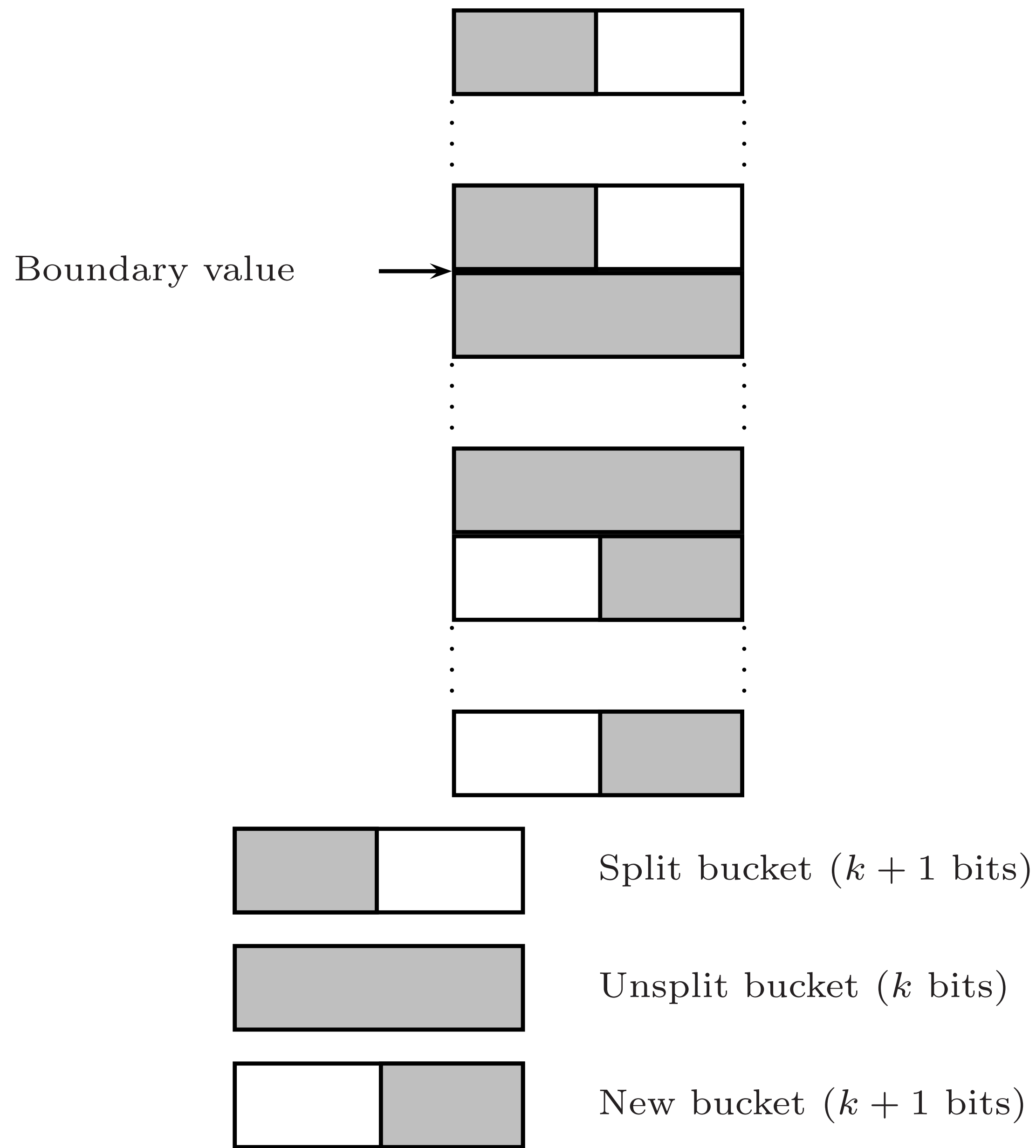
- a hashed file with a dynamically adjusted number of buckets
- used in commercial file systems.

Initially:

- use the last k_0 bits of the hash value as the bucket number.

Bucket split:

- assuming the bucket number has k bits, split the bucket into two buckets with numbers having $k+1$ bits (using the last $k+1$ bits of key hash values).


Boundary value
Split bucket ($k + 1$ bits)
Unsplit bucket (k bits)
New bucket ($k + 1$ bits)

Lookup

File header contains:

- boundary value
- current number of bucket number bits (k).

Lookup algorithm:

```
calculate hash function h(Key);
if last k bits of h(Key)
are less than boundary value
then i := last k+1 bits of h(Key)
else i := last k bits of h(Key);
search bucket i
```

Insertion

```
find bucket i using h(Key);
if last block of bucket i full
then allocate a new overflow block for i;
     add the record there;

if TIME TO SPLIT
then create a new bucket
     distribute records
     from the first unsplit bucket
     between this bucket and the new one;
     increment boundary value by 1;

if ALL BUCKETS SPLIT
then boundary value := 0;
     mark all buckets as unsplit;
     k := k + 1
```

Analysis

When is it `TIME TO SPLIT`?

- after a fixed number of insertions after last split
- number of insertions = desired average number of records in a block.

Deletion:

- works in the opposite direction: **contraction**.

Cost:

- cost formulas difficult to derive
- experimentally:

Operation	Number of block accesses
LOOKUP	close to 1 (average)
INSERT	$2 - 3$

Indexed file

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** (clustered) indexes
- a record id (rid) for **dense** (unclustered) indexes.

Indexed files have two kinds of blocks:

- main file blocks
- index blocks.

Sorted (clustered) blocks:

- index blocks are always sorted
- if main file sorted on the index key, sparse index can be used.

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
- block overflows are handled by allocating new overflow blocks
- from time to time the index may be 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
  - sparse primary: main file and all index levels are sorted
- balanced trees:
  - all leaves at the same level
  - updates "ripple up" the tree
- guaranteed fill ratio (50%) in nonroot nodes.

Other features:

- main file records in the leaves only (B+ trees)
- key value $-\infty$ omitted
- parameters:
  - max. number of index records per block = $2d$ (min. number = d)
  - max. number of main file records per block = $2e$ (min. number = e).

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 leaves (blocks)	$\leq n/e$
Number of parents of leaves (blocks)	$\leq n/de$
...	
Number of blocks immediately	$\leq n/d^{i-2}e$
below the root	≥ 2

Thus:

$$n/d^{i-2}e \geq 2$$

$$\Rightarrow d^{i-2} \leq n/2e$$

$$\Rightarrow i - 2 \leq \log_d(n/2e)$$

$$\Rightarrow i \leq 2 + \log_d(n/2e)$$

B-trees: lookup

Lookup algorithm:

```
current block := root;
repeat
if current block is a leaf
then retrieve record with key = Key
else find the last index record
     with key ≤ Key and address = Addr;
     address of current block := Addr
until leaf reached
```

B-trees: insertion

Insertion algorithm:

```
lookup the block to insert;
if enough room
then store the record
else /*OVERFLOW*/
     allocate a new 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 fill ratio.

Updating the parent index block may lead to an index block OVERFLOW: handled using the same algorithm.

Updates may ripple up to the root.

B-trees: deletion

Deletion algorithm:

```
find the record to be deleted;
remove the record;
if less than e records remain
then /*UNDERFLOW*/
     consider left (or right) sibling block;
     if sibling contains e records
     then combine it with underfull block;
          release one block to the file system;
          update the parent index block;
     else/*sibling contains more than e records*/
          redistribute records
          between the two blocks;
          update the parent index block;
```

Record redistribution should preserve ordering and B-tree fill ratio.

B-trees: UNDERFLOW

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: analysis

Parameters:

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

R records per block (blocking factor):

- $R = 2e$ for main file blocks
- $R = 2d$ for index blocks

K records satisfying the range condition.

Operation	Number of block accesses
SCAN	B
EQUALITY SEARCH	$\leq 2 + \log_d(n/2e)$
RANGE SEARCH	number for EQUALITY SEARCH $+(K/R)$
INSERT (typically)	number for LOOKUP $+1$
INSERT (worst case)	$2 *$ number for LOOKUP
DELETE	like INSERT

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

Bitvectors:

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

Bitmap indexes used where there are *few* domain values.

Evaluating $\sigma_E(R)$

Pick the option of least estimated cost.

For any E: linear scan.

E is $A = c$:

- if file sorted on A, use binary search.
- if file has an index on A, use the index.
- if file hashed on A, use hashing.

E is $A\theta c$ (where $\theta \in \{<, \leq, >, \geq\}$):

- if file sorted on A, then:
  - use binary search.
  - if there is an index on A, use the index.

Complex selection conditions

E is $E_1 \wedge E_2$:

- use E_1 or E_2 to narrow down the search
- use composite index (if available)
- use indexes for E_1 and E_2 to obtain sets of rids, then intersect those sets.
- use bitmap indexes for E_1 and E_2 to obtain bitvectors, then use boolean AND.
- use a combination of bitmap and other kinds of indexes.

E is $E_1 \vee E_2$:

- E_1 or E_2 cannot be used separately to narrow down the search
- use indexes for E_1 and E_2 to obtain sets of rids, then take the union of those sets.
- use bitmap indexes for E_1 and E_2 to obtain bitvectors, then use boolean OR.
- use a combination of bitmap and other kinds of indexes.

Evaluating $R \underset{A=B}{\bowtie} S$

Nested loops:

```
foreach t ∈ R do
  foreach s ∈ S do
    if t[A] = s[B] then output (t, s)
```

Index join (S has an index on B):

```
foreach t ∈ R do
  lookup S using the index
  and the value t[A]
```

Sort-merge join (R sorted on A, S sorted on B):

```
scan files in sorted order
matching A-values in R
with B-values in S
```

Hash-join:

```
create new hashed files HR and HS;
foreach t ∈ R do
  i := h(t[A]);
  store t in bucket r_i of HR;
foreach t ∈ S do
  i := h(t[A]);
  store t in bucket s_i of HS;
for each bucket r_i of HR
  read r_i and build an in-memory hash index
  for each tuple t in the bucket s_i of S
    probe the hash index to find and output
    the tuples w such that w[A] = t[A]
```

- the hash function used to build the hash index and to probe it has to be different than h
- each buckets of HR has to fit in main memory

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

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: selection

Equality condition $A = c$:

- unique index on A: `INDEX UNIQUE SCAN`
- nonunique index on A: `INDEX RANGE SCAN`

Range condition on A:

- any index on A: `INDEX RANGE SCAN`

Conjunction:

- both indexes: `AND-EQUAL` (intersection of RowId sets)

Oracle: join

`SORT JOIN/MERGE JOIN` (sort-merge).

`NESTED LOOPS` (nested loop or index join)

`HASH JOIN`