Lecture Notes for Database Systems
by Patrick E. O'Neil

Chapter 8. Indexing.

8.1 Overview. Usually, an index is like a card catalog in a library. Each card (entry) has:

(keyvalue, row-pointer) (keyvalue is for lookup, call row-pointer ROWID)
(ROWID is enough to locate row on disk: one I/O)

Entries are placed in Alphabetical order by lookup key in "B-tree" (usually), explained below. Also might be hashed.

An index is a lot like memory resident structures you've seen for lookup: binary tree, 2-3-tree. But index is disk resident. Like the data itself, often won't all fit in memory at once.

X/OPEN Syntax, Fig. 8.1, pg. 467, is extremely basic (not much to standard).

CREATE [UNIQUE] INDEX indexname ON tablename (colname [ASC | DESC]
{,colname [ASC | DESC] . . .});

Index key created is a concatenation of column values. Each index entry looks like (keyvalue, recordpointer) for a row in customers table.

An index entry looks like a small row of a table of its own. If created concatenated index:

create index agnamcit on agents (aname, city);

and had (aname, city) for two rows eual to (SMITH, EATON) and (SMITHE, ATON), the system would be able to tell the difference. Which once comes earlier alphabetically?

After being created, index is sorted and placed on disk. Sort is by column value asc or desc, as in SORT BY description of Select statement.

NOTE: LATER CHANGES to a table are immediately reflected in the index, don't have to create a new index.

Ex 8.1.1. Create an index on the city column of the customers table.

```
create index citiesx on customers (city);
```

Note that a single city value can correspond to many rows of customers table. Therefore term index key is quite different from relational concept of primary key or candidate key. LOTS OF CONFUSION ABOUT THIS.

With no index if you give the query:

```
select * from customers where city = 'Boston'
   and discnt between 12 and 14;
```

Need to look at every row of customers table on disk, check if predicates hold (TABLE SCAN or TABLESPACE SCAN).

Probably not such a bad thing with only a few rows as in our CAP database example, Fig 2.2. (But bad in DB2 from standpoint of concurrency.)

And if we have a million rows, tremendous task to look at all rows. Like in Library. With index on city value, can winnow down to (maybe 10,000) customers in Boston, like searching a small fraction of the volumes.

User doesn't have to say anything about using an Index; just gives Select statement above.

Query Optimizer figures out an index exists to help, creates a Query plan (Access plan) to take advantage of it -- an Access plan is like a PROGRAM that extracts needed data to answer the Select.

Class 7.

Recall X/OPEN form of Create Index:

CREATE [UNIQUE] INDEX indexname ON tablename (colname [ASC | DESC]
{,colname [ASC | DESC] . . .});

The unique clause of create index can be used to guarantee uniqueness of a candidate table key. Example 8.1.2, instead of declaring cid a primary key in the Create Table statement:

create unique index cidx on customers (cid);

OK if multiple nulls, like Create Table with column UNIQUE.

When use Create Table with UNIQUE or PRIMARY KEY clause, this causes a unique index to be created under the covers. Can check this, querying:

USER_INDEXES
USER_IND_COLUMNS

OK, now the X/OPEN standard doesn't say much about what's possible with indexes. Foreshadowings.

Different index types: B-tree, Hashed. Usually, B-tree (really B^+-tree).

Primary index, secondary index, clustered index. Primary index means rows are actually in the index structure in the place of entries pointing to rows.

Clustered index means rows in same order as index entries (volumes on shalves for Dickens all in one area): may be primary index, may not.

Difficult to appreciate the index structures without understanding disk access properties: disk I/O is excruciatingly slow, most of index design is directed at saving disk I/O: even binary search of list inappropriate on disk.

8.2 Disk storage.

Computer memory is very fast but Volatile storage. (Lose data if power interruption.) Disk storage is very slow but non-volatile (like, move data from one computer to another on a diskette) and very cheap.

Model 100 MIPS computer. Takes .00000001 seconds to acess memory and perform an instruction. Getting faster, like car for $9 to take you to the moon on a gallon of gas.

Disk on the other hand is a mechanical device — hasn't kept up.

(Draw picture). rotating platters, multiple surfaces. disk arm with head assemblies that can access any surface.

Disk arm moves in and out like old-style phonograph arm (top view).

When disk arm in one position, cylinder of access (picture?). On one surface is a circle called a track. Angular segment called a sector.

To access data, move disk arm in or out until reach right track (Seek time)

Wait for disk to rotate until right sector under head (rotational latency)

Bring head down on surface and transfer data from a DISK PAGE (2 KB or 4KB: data block, in ORACLE) (Transfer time). Rough idea of time:

Seek time:	.008 seconds
Rotational latency:	.004 seconds (analyze)
Transfer time:	.0005 seconds (few million bytes/sec: ew K bytes)

Total is .0125 seconds = 1/80 second. Huge compared to memory access.

Typically a disk unit addresses ten Gigabytes and costs 1 thousand dollars (new figure) with disk arm attached; not just pack which is much cheaper.

512 bytes per sector, 200 sectors per track, so 100,000 bytes per track. 10 surfaces so 1,000,000 bytes per cylinder. 10,000 cylinders per disk pack.

Total is 10 GB (i.e., gigabytes)

Now takes .0125 seconds to bring in data from disk, .000'000'01 seconds to access (byte, longword) of data in memory. How to picture this?

Analogy of Voltaire's secretary. Copying letters at one word a second. Run into word can't spell. Send letter to St Petersberg, six weeks to get response (1780). Can't do work until response (work on other projects.)

From this see why read in what are called pages, 2K bytes on ORACLE, 4K bytes in DB2 UDB. Want to make sure Voltaire answers all our questions in one letter. Structure Indexes so take advantage of a lot of data together.

Buffer Lookaside

Similarly, idea of buffer lookaside. Read pages into memory buffer so can access them. Once to right place on disk, transfer time is cheap.

Everytime want a page from disk, hash on dkpgaddr, h(dkpgaddr) to entry in Hashlookaside table to see if that page is already in buffer. (Pg. 473)

If so, saved disk I/O. If not, drop some page from buffer to read in requested disk page. Try to fix it so popular pages remain in buffer.

Another point here is that we want to find something for CPU to do while waiting for I/O. Like having other tasks for Voltaire's secretary.

This is one of the advantages of multi-user timesharing. Can do CPU work for other users while waiting for this disk I/O.

If only have 0.5 ms of CPU work for one user, then wait 12.5 ms for another I/O. Can improve on by having at least 25 disks, trying to keep them all busy, switching to different users when disk access done, ready to run.

Why bother with all this? If memory is faster, why not just use it for database storage? Volatility, but solved. Cost no longer big factor:

Memory storage costs about a $4000 per gigabyte.
Disk storage (with disk arms) costs about 100 dollars a Gigabyte.

So could buy enough memory so bring all data into buffers (no problem about fast access with millions of memory buffers; very efficient access.)
Coming close to this; probably will do it soon enough. Right now, being limited by 4 GBytes of addressing on a 32 bit machine.

Class 8. Create Tablespace

OK, now DBA and disk resource allocation in ORACLE.

We have avoided disk resource considerations up to now because so hard to handle in standard. All the commercial systems are quite different in detail. But a lot of common problems.

A tablespace is built out of OS files (or raw disk partition [TEXT]), and can cross files (disks) See Fig. 8.3, pg. 476. Segment can also cross files.

All the products use something like a tablespace. DB2 uses tablespace. Informix uses dbspaces.

Tablespaces have just the right properties, no matter what OS really living on -- general layer insulates from OS specifics. Typically used to define a table that crosses disks.

See Fig. 8.3 again. When table created, it is given a data segment, Index an index segment. A segment is a unit of allocation from a tablespace.

DBA plan. Construct Tablespace from operating system files (or disk partitions). Can specify tablespace crosses disks, stays on one disk, etc.

Tablespace is the basic resource of disk storage, can grant user RESOURCE privilege in ORACLE to use tablespace in creating a table.

Any ORACLE database has at least one tablespace, named SYSTEM, created with Create Database: holds system tables. Now see Fig 8.4, pg. 476. [**Leave Up**]

```
CREATE TABLESPACE tblspname
    DATAFILE 'filename' [SIZE n [K|M]] [REUSE] [AUTOEXTEND OFF
    | AUTOEXTEND ON [NEXT n [K|M] [MAXSIZE {UNLIMITED | n [K|M]}]
    {, 'filename' . . .}
    -- the following optional clauses can come in any order
    [ONLINE | OFFLINE]
    [DEFAULT STORAGE ([INITIAL n] [NEXT n] [MINEXTENTS n]
        [MAXEXTENTS {n|UNLIMITED}] [PCTINCREASE n])
        (additional DEFAULT STORAGE options not covered)]
    [MINIMUM EXTENT n [K|M]]
   [other optional clauses not covered];
```

Operating systems files named in datafile clause. ORACLE is capable of creating them itself; then DBA loses ability to specify particular disks.

The idea here is that ORACLE (or any database system) CAN create its own "files" for a tablespace.

If SIZE keyword omitted, data files must already exist, ORACLE will use. If SIZE is defined,ORACLE will normally create file. REUSE means use existing files named even when SIZE is defined; then ORACLE will check size is right.

If AUTOEXTEND ON, system can extend size of datafile. The NEXT n [K|M] clause gives size of expansion when new extent is created. MAXSIZE limit.

If tablespace created offline, cannot immediately use for table creation. Can alter offline/online later for recovery purposes, reorganization, etc., without bringing down whole database.

SYSTEM tablespace, created with Create Database, never offline.

When table first created, given an initial disk space allocation. Called an initial *extent*. When load or insert runs out of room, additional allocations are provided, each called a "next extent" and numbered from 1.

Create Tablespace gives DEFAULT values for extent sizes and growth in STORAGE clause; Create Table can override these.

INITIAL n:	size in bytes of initial extent: default 10240
NEXT n:	size in bytes of next extent 1. (same) May grow.
MAXEXTENTS n:	maximum number of extents segment can get
MINEXTENTS n:	start at creation with this number of extents
PCTINCREASE n:	increase from one extent to next. default 50.

Minimum possible extent size is 2048 bytes, max is 4095 Megabytes, all extents are rounded to nearest block (page) size.

The MINIMUM EXTENT clause guarantees that Create Table won't be able to override with too small an extent: extent below this size can't happen.

Next, Create Table in ORACLE, Figure 8.5, pg. 478. (**Leave Up**)

```
CREATE TABLE [schema.]tablename
  ({colname datatype [DEFAULT {constant|NULL}] [col_constr] {, col_constr...}
      | table_constr}
  {, colname datatype etc. . . .}
  [ORGANIZATION HEAP | ORGANIZATION INDEX (with clauses not covered)]
  [TABLESPACE tblspname]
  [STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n]
      [MAXEXTENTS n] [PCTINCREASE n] ) ]
  [PCTFREE n] [PCTUSED n]
  [other disk storage and update tx clauses not covered or deferred]
  [AS subquery]
```

ORGANIZATION HEAP is default: as insert new rows, normally placed left to right. Trace how new pages (data blocks) are allocated, extents. Note if space on old block from delete's, etc., might fill in those.

ORGANIZATION INDEX means place rows in A B-TREE INDEX (will see) in place of entries. ORDER BY primary key!

The STORAGE clause describes how initial and successive allocations of disk space occur to table (data segment). There can be a default storage clause with a tablespace, inherited by table unless overridden.

The PCTFREE n clause determines how much space on each page used for inserts before stop (leave space for varchar expand, Alter table new cols.)

PCTFREE n, n goes from 0 to 99, default 10.

The PCTUSED n clause specifies a condition where if page (block) gets empty enough, inserts will start again! Range n from 1 to 99, default 40.

Require PCTFREE + PCTUSED < 100, or invalid. Problem of hysteresis; don't always want to be switching from inserting to not inserting and back.

E.g., if PCTFREE 10 PCTUSED 90, then stop inserrts when >90% full, start when <90% full. If 20, 90, ill-defined between 80 and 90.

Data Storage Pages and Row Pointers: Fig. 8.6

Typically, after table is created, extents are allocated, rows are placed one after another on successive disk pages (in INGRES, call heap storage). Most products have very little capability to place rows in any other way.

A row on most architectures is a contiguous sequence of bytes. See Figure 8.6 (pg. 480): N rows placed on one page (called a *Block* in ORACLE).

Header info names the kind of page (data segment, index segment), and the page number (in ORACLE the OS file and page number). Rows added right to left from right end on block.

Row Directory entries left to right on left after header. Give offsets of corresponding row beginnings. Provide number of row slot on page.

When new row added, tell immediately if we have free space for new directory entry and new row. Conceptually, all space in middle, implies when delete row and reclaim space must shift to fill in gap. (Can defer that.)

Also might have "Table Directory" in block when have CLUSTER. (Later.)

There's overhead for column values within a row (offsets), not shown here. Must move rows on pg if updates changes char varying field to larger size.

Class 9. Review Fig. 8.6 above (put on board, discuss)

Disk Pointer. RID (DB2), ROWID (ORACLE), TID (INGRES). A row in a table can be uniquely specified with the page number (P) and slot number (S). In INGRES, have TID (Tuple ID), given by:

TID = 512*P + S

Pages in INGRES are numbered successively within the table allocation, from zero to 2^{23} - 1: all 1's in 23 bit positions. Slots are 0 to 511. all 1's in 9 bit positions: total container is 32 bits, unsigned int, 4 bytes.

So if rows are 500 bytes long (small variation), 2K byte pages in UNIX will contain only 4 rows, and TIDs go: 0, 1, 2, 3, 512, 513, 514, 515, 1024, 1025, . . .

Note value of information hiding. Give row ptr in terms of Page and Slot number; if gave byte offset instead of slot number would have to change RID/ROWID/TID when reorganized page

Example 8.2.1, pg 481, (variant). INGRES DBA (or any user) can check space calculations are right (200 byte rows 10 to 2 KB disk page, make sure knows what's going on) by selecting TID from table for successive rows:

select tid from employees where tid <= 1024;

Note tid is a Virtual column associated with every table, so can select it. Will appear in order (tablespace scan goes to first pg, first row, then . . .)

A DB2 record pointer is called a RID, also 4 bytes, encoding page number within Tablespace and slot number, but DB2 RID structure is not public, and we can't Select a DB2 RID from a tableas a virtual column.

In ORACLE, row pointer is called ROWID. and is normally **6 bytes long!!** Restricted ROWID display representation is made up of Block number (page) within OS file, Slot number in block, & file no. (why?):

BBBBBBBB.RRRR.FFFF (each hexadecimal, total of 8 bytes for ROWID)

(Block number, Row (Slot) number, File number)

The ROWID value for each row can be retrieved as a virtual column by an SQL Select statement on any table, as with the following query:

```
select cname, rowid from customers where city = 'Dallas';
```

which might return the following row information (if the ROWID retrieved is in restricted form):

```
CNAME          CROWID
-------------- ------------------
Basics         00000EF3.0000.0001
Allied         00000EF3.0001.0001
```

The alternative “extended ROWID” form is displayed as a string of four components having the following layout, with letters in each component representing a base-64 encoding:

```
OOOOOOFFFBBBBBRRR
```

Here OOOOOO is the data object number, and represents the database segment (e.g., a table). The components FFF, BBBBB, and RRR represent the file number, block number, and row number (slot number).

Here is the “base-64” encoding, comparable to hexadecimal representation except that we have 64 digits. The digits are printable characters:

DIGITS	CHARACTERS
0 to 25	A to Z (Capital letters)
26 to 51	a to z (lower case letters)
52 to 61	0 to 9 (decimal digits)
62 and 63	+ and / respectively

For example, AAAAm5AABAAAEtMAAB represents object AAAAm5 = 38*64 + 5, file AAb = 1, block AAEtM = 4*642 + 44*64 + 13, and slot 1. The query:

```
select cname, rowid from customers where city = 'Dallas';
```

might return the row information (different values than restricted ROWID):

```
CNAME          ROWID
-------------- ------------------
Basics         AAAAm5AABAAAEtMAAB
Allied         AAAAm5AABAAAEtMAAC
```

Since an index is for a specific table, don't need extended ROWID with object number of table segment. But Select statement commonly displays the extended form. Functions exist to convert (need special library).

We use ROWID nomenclature generically if database indeterminate.

Example 8.2.2. See Embedded SQL program on pg. 483. Idea is that after find a row (through an index or slower means) can retrieve it a second time through ROWID in ORACLE (not in DB2). This saves time.

Danger to use ROWID if someone might come and delete a row with a given ROWID, then reclaim space and new row gets inserted in that slot. If out of date ROWID could refer to wrong row.

But certainly safe to remember ROWID for length of a transaction that has a referenced row locked.

Rows Extending Over Several Pages: Product Variations

In ORACLE, allow rows that are larger than any single page (2 KB).

If a row gets too large to fit into it's home block, it is split to a second block. Second part of row has new ROWID, not available to any external method, only to chainer.

Means that random access requires two or more page reads (Voltaire problem). DB2 on other hand limits size of row to disk page (4005 bytes).

Still may need to move row when it grows, leave forwarding pointer at original RID position (forwarding pointer called "overflow record").

Why do you think DB2 does this?

But only ever need ONE redirection, update forwarding pointer in original position if row moves again, no chain of forwarding pointers.

8.3 B-tree Index

Most common index is B-tree form. Assumed by X/OPEN Create Index. We will see other types — particularly hashed, but B-tree is most flexible.

The B-tree is like the (2-3) tree in memory, except that nodes of the tree take up a full disk page and have a lot of *fanout*. (Picture on board.)

ORACLE Form of Create Index Figure 8.7, pg. 485 (leave on board):

```
CREATE [UNIQUE | BITMAP] INDEX [schema.]indexname ON tablename
   (colname [ASC | DESC] {,colname [ASC | DESC]...})
   [TABLESPACE tblespace]
   [STORAGE . . . ] (see pg. 478, par 4 ff, override Tablespace default)
   [PCTFREE n]
   [other disk storage and update tx clauses not covered or deferred]
   [NOSORT]
```

We will discuss the concept of a BITMAP Index later. Non-Bitmap for now.

Note ASC | DESC is not really used. ORACLE B-tree is usable in both directions (leaf sibling pointers left and right).

What Create Index does: reads through all rows on disk (assume N), pulls out (keyvalue, rowid) pairs for each row. Get following list put out on disk.

(keyval1, rowid1) (keyval2, rowid2) . . . (keyvalN, rowidN)

Now sort these on disk, so in order by kevalues. Explain NOSORT clause (telling ORACLE rows are in right order, so don't have to sort, but error returned if ORACLE notices this is false).

Now idea of binary search, Example 8.3.1, pg. 486 ff. Assume N = 7, searching array of structs: arr[K].keyval, arr[K].rowid (list of pairs named above). Ordered by keyval K = 0 to 6.

```
/* binsearch:  return K so that arr[K].keyval == x, or -1 if no match;      */
/* arr is assumed to be external, size of 7 is wired in                     */
  int binsearch(int x)
  {
     int probe = 3,          /* first probe position at subscript K =  3  */
         diff = 2;

     while (diff > 0) {      /* loop until K to return                    */
       if (probe <= 6 && x > arr[probe].keyval)
         probe = probe + diff;
       else
```

```
        probe = probe - diff;
      diff = diff/2;
    }                              /* we have reached final K          */
    if (probe <= 6 && x == arr[probe].keyval) /* have we found it?      */
      return probe;
    else if (probe+1 <= 6 && x == arr[probe+1].keyval) /* maybe next    */
      return probe + 1;
    else return -1;                /* otherwise, return failure        */
}
```

Figure 8.8 Function binsearch, with 7 entries wired in

Consider the sequence of keyval values {1, 7, 7, 8, 9, 9, 10} at subscript positions 0-6.

Work through if x = 1, binsearch will probe successive subscripts 3, 1, and 0 and will return 0. If x = 7, the successive subscript probes will be 3, 1, 0. (undershoot — see why?) return 1. If x = 5, return -1. More?

Given duplicate values in the array, binsearch will always return the *smallest* subscript K such that x == arr[K].keyval (exercise).

The binsearch function generalizes easily: given N entries rather than 7, choose m so that $2^{m-1} < N \leq 2^m$; then initialize probe to 2^{m-1}-1 and diff to 2^{m-2}. Tests if probe <= 6 or probe +1 <= 6 become <= N-1.

Optimal number of comparisons (if all different). Log_2 N. But not optimal in terms of disk accesses.

Example 8.3.2. Assume 1 million entries. First probe is to entry 524,287 (2^{19}-1). Second prove is 2^{18} = 262,144 entries away. Look at list pg. 488.

But with 2K byte pages, assume keyvalue is 4 bytes (simple int), ROWID is 4 bytes (will be either 6 or 7 in ORACLE), then 8 bytes for whole entry; 2000/8 is about 250 entries per page. (1M/250 = 4000 pgs.) Therefore successive probes are always to different pages until probe 14 (maybe).

That's 13 I/Os!! A year of letters to Voltaire. We can make it 9 weeks!

Class 10.

Last time, talked about binary search on 1M entries of 8 bytes each assuming search list is on disk, Example 8.3.2, pg. 487. Required 13 I/Os.

OK, now we can improve on that. Layout 1M entries on 4000 pgs. These are called leaf nodes of the B-tree.

Each pg has a range of keyvalues, and want to create optimal directory to get to proper leaf node. Node ptrs np and separators.

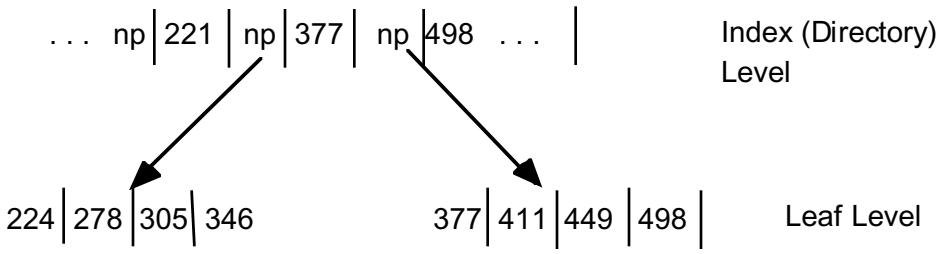


Figure 8.10 Directory structure to leaf level nodes

Search for entry keyvalue 305. Bin search in index (directory) for largest key smaller than x. Then following separator points to proper node.

OK, now put directory on disk pages. Each entry is: (sepkeyval, np). About same size as B-tree entry, 8 bytes. Number of pages is 4000/250 = 16.

Say have gotten to right index directory node; do binary search on index node for np to follow, then follow it to proper leaf page.

Now repeat trick with HIGHER level directory to index nodes just created. Same types of entries, only 16 pointers needed, all on one pg.

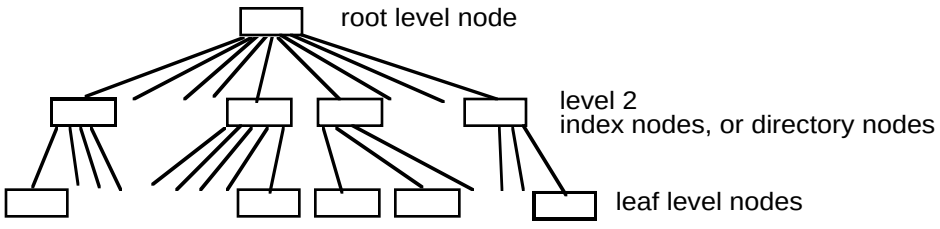


Figure 8.11 Schematic Picture of a three level B-tree

Above *leaf nodes*, have *index nodes* or *directory nodes*. Depth of 3.

Search algorithm is to start at root, find separators surrounding, follow np down, until leaf. Then search for entry with keyval == x if exists.

Only 3 I/Os. Get the most out of all information on (index node) page to locate appropriate subordinate page. 250 way fanout. Say this is a *bushy tree* rather than *sparse tree* of binary search, and therefore flat. Fanout f:

depth = $\log_f(N)$ — e.g., $\log_2(1{,}000{,}000) = 20$, $\log_f(1{,}000{,}000) = 3$ if f >100

Actually, will have commonly accessed index pages resident in buffer. 1 + 16 in f = 250 case fits, but not 4000 leaf nodes: only ONE I/O.

Get same savings in binary search. First several levels have1 + 2 + 4 + 8 +16 nodes, but to get down to one I/O would have to have 2000 pages in buffer. There's a limit on buffer, more likely save 6-7 I/Os out of 13.

(Of course 2000 pages fits in only 4 MB of buffer, not much nowadays. But in a commercial database we may have HUNDREDS of indexes for DOZENS of tables! Thus buffer space must be shared and becomes more limited.)

Dynamic changes in the B-tree

OK, now saw how to create an index when know all in advance. Sort entries on leaf nodes and created directory, and directory to directory.

But a B-tree also grows in an even, balanced way. Consider a sequence of inserts, **SEE Figure 8.12, pg. 491.**

(2-3)-tree. All nodes below root contain 2 or 3 entries, split if go to 4. This is a 2-3 tree, B-tree is probably more like 100-200.

(Follow along in Text). Insert new entries at leaf level. Start with simple tree, root = leaf (don't show rowid values). Insert 7, 96, 41. Keep ordered. Insert 39, split. Etc. Correct bottom tree, first separator is 39.

Note separator key doesn't HAVE to be equal to some keyvalue below, so don't correct it if later delete entry with keyvalue 39.

Insert 88, 65, 55, 62 (double split).

Note: stays balanced because only way depth can increase is when root node splits. All leaf nodes stay same distance down from root.

All nodes are between half full (right after split) and full (just before split) in growing tree. Average is .707 full: SQRT(2)/2.

Now explain what happens when an entry is deleted from a B-tree (because the corresponding row is deleted).

In a (2-3)-tree if number of entries falls below 2 (to 1), either BORROW from sibling entry or MERGE with sibling entry. Can REDUCE depth.

This doesn't actually get implemented in most B-tree products: nodes might have very small number of entries. Note B-tree needs malloc/free for disk pages, malloc new page if node splits, return free page when empty.

In cases where lots of pages become ALMOST empty, DBA will simply reorganize index.

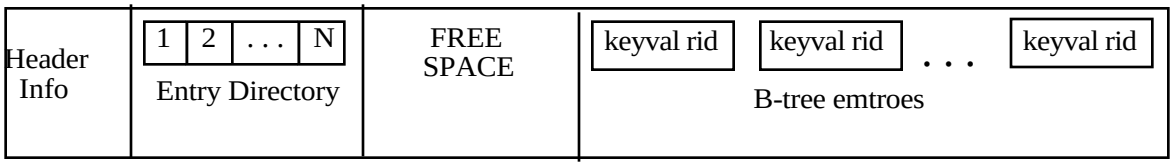
Properties of the B-tree, pg. 499 ff, Talk through. (Actually, B+tree.)

- Every node is disk-page sized and resides in a well-defined location on the disk.
- Nodes above the leaf level contain directory entries, with n – 1 separator keys and n disk pointers to lower-level B-tree nodes.
- Nodes at the leaf level contain entries with (keyval, ROWID) pairs pointing to individual rows indexed.
- All nodes below the root are at least half full with entry information.
- The root node contains at least two entries (except when only one row is indexed and the root is a leaf node).

Index Node Layout and Free Space on a page

See Figure 8.13, pg. 494. Always know how much free space we have on a node. Usually use free space (not entry count) to decide when to split.

This is because we are assuming that keyval can be variable length, so entry is variable length; often ignored in research papers.



Note Figure looks like rows in block; Entry directory slots for entries give opportunity to perform binary search although entries are var. length.

In Load, leave some free space when create index so don't start with string of splits on inserts. See Figure 8.7, pg. 485, PCTFREE n.

Idea is, when load, stop when pctfree space left. Split to new node. Free space available for later inserts.

Example 8.3.4. Corrected B-tree structure for a million index entries.

Index entry of 8 bytes, page (node) of 2048 bytes, assume 48 bytes for header info (pg. 500), and node is 70% full.

Thus 1400 bytes of entries per node, 1400/8 = 175 entries. With 1,000,000 entries on leaf, this requires 1,000,000/175 = 5715 leaf node pages. Round up in division, CEIL(1,000,000/175), say why.

Next level up entries are almost the same in form, (sepkeyvalue, nptr), say 8 bytes, and 5715 entries, so 5715/175 = 33 index node pages.

On next level up, will fit 33 entries on one node (root) page. Thus have B-tree of depth 3.

ROUGH CALCULATIONS LIKE THIS ARE FINE. We will be using such calculations a lot! (It turns out, they're quite precise.)

Generally most efficient way to create indexes for tables, to first load the tables, then create the indexes (and REORG in DB2).

This is because it is efficient to sort and build left-to-right, since multiple entries are extracted from each page or rows in the table.

Create Index Statements in ORACLE and DB2. See Figures 8.14, and 8.15., pg. 496 & 497.

ORACLE makes an effort to be compatible with DB2. Have mentioned PCTFREE before, leaves free space on a page for future expansion.

DB2 adds 2 clauses, INCLUDE columname-list to carry extra info in keyval and CLUSTER to speed up range retrieval when rows must be accessed. See how this works below.

Class 11.

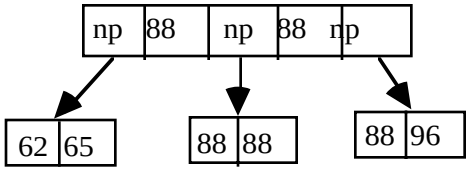
Missed a class. Exam 1 is Wed, March 22. (After Vac.) Hw 2 due by Monday, March 20, solutions put up on Tuesday to allow you to study.

If you are late, your hw 2 doesn't count. **NO EXCEPTIONS. TURN IN WHAT YOU HAVE READY!**

Duplicate Keyvalues in an Index.

What happens if have duplicate keyvalues in an index? Theoretically OK, but differs from one product to another.

Look at Fig 8.12, pg 498, right-hand, and assume add we a bunch of new rows with duplicate keyvalues 88. Get e.g. Fig 8.16 following on pg 504.



(Work out slowly how get it). Seems kind of strange to call 88 a separator value, since 88 on both nodes below. Now consider Select statement:

```
select * from tbl where keyval = 88;
```

Have to find way down to list of 88s. Done in binsearch, remember? Leftmost one GE 88. Must follow np to left of sepkey = 88. (No 88 at that leaf level, but there could be one since dups allowed)

Then will go right on leaf level while keyval <= 88, following sibling pointers, access ROWIDs and read in rows. Point of this is that everything works fine with multiple duplicates: you should think about how.

Consider the process of deleting a row: must delete its index entries as well (or index entries point to non-existent row; could just keep slot empty, but then waste time later in retrievals).

Thus when insert row, must find way to leaf level of ALL indexes and insert entries; when delete row must do the same. Overhead.

Treat update of column value involved in index key as delete then insert.

Point is that when there are multiple duplicate values in index, find it HARD to delete specific entry matching a particular row. Look up entry with given value, search among ALL duplicates for right RID.

Ought to be easy: keep entries ordered by keyvalue||RID (unique). But most commercial database B-trees don't do that. (Bitmap index solves this.)

Said before, there exists an algorithm for efficient self-modifying B-tree when delete, opposite of split nodes, merge nodes. But not many products use it: only DB2 UDB seems to. (Picture B-tree shrinking from 2 levels to one in Figure 8.12! After put in 39, take out 41, e.g., return to root only.)

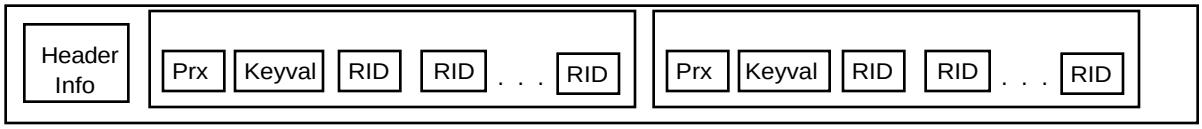
Lot of I/O: argues against index use except for table with no updates. But in fact most updates of tables occur to increment-decrement fields.

Example. Credit card rows. Indexed by location (state || city || staddress), credit-card number, socsecno, name (lname || fname || midinit), and possibly many others.

But not by balance. Why would we want to look up all customers who have a balance due of exactly 1007.51? Might want all customers who have attempted to overdraw their balance but this would use an (indexed) flag.

Anyway, most common change to record is balance. Can hold off most other updates (change address, name) until overnight processing, and reorganize indexes. Probably more efficient.

DB2 has nice compression for multiple duplicate keyvalues in an index. See Fig. 8.17 on pg 505 for data page layout with duplicate keyvalues.



Each block has keyval once and then a list of RID values, up to 255. Prx contains pointer to prior block (2 bytes), next block (2 bytes) and count of RIDs in this block (1 bytes) plus 1 byte unused. [TEXT: now 1 block/page?]

Since Keyval can be quite long (lname || fname || midint, say 30 chars or bytes), get great space saving (if multiple duplicates). With 255 RIDs for each keyval, down nearly to 4 bytes/entry instead of 34 bytes/entry.

Oracle Bitmap Index

A bitmap (or bit vector) can be used in an ADT to represent a set, and this leads to very fast union, intersection, and member operations.

A bitmap index uses ONE bitmap for each distinct keyval, like DB2 with RID list. A bitmap *takes the place of a RID list*, specifying a set of rows.

See example, pg. 505-506, emps table with various columns. Low card columns are good candidates for bitmap indexes. Easy to build them.

```
create table emps (eid char(5) not null primary key,
   ename varchar(16), mgrid char(5) references emps,(ERROR IN TXT)
   gender char(1), salarycat smallint, dept char(5));

create bitmap index genderx on usemps(gender); (2 values, 'M' &'F')
create bitmap index salx on usemps(salrycat);  (10 values, 1-10)
create bitmap index deptx on usemps(dept);  (12 vals, 5 char: 'ACCNT')
```

These columns are said to have *low cardinality* (cardinality is number of values in a column).

For structure of a bitmap index, think of assigning ordinal numbers to N rows, 0, 1, 2, 3, . . . Keep bits in load order for rows (order as they lie on disk?). Require a function to go from ordinal number to ROWID and back.

Then bitmap for a property such as gender = 'F' is a sequence of N bits (8 bits/byte) so bit in position j is 1 iff row j has property, 0 otherwise.

OK, have bitmaps for each column *value*. Put one for each keyvalue in B-tree to make an index?

Bitmap index, Fig. 8.18, pg. 506. Note that 100,000 rows gives 12,500 bytes of bitmap; Bitmap broken into *Segments* on successive leaf pages.

Same idea as Blocks of RID-lists in DB2 UDB.

Idea of bitmap density: number of 1-bits divided by N (total # bits)

In DB2, RID needs = 4 bytes. Thus RID-list can represent one row with 32 bits. At density = 1/32 , bitmap can represent in same space.

But if density is a lot lower than 1/32, say 1/1000 (out of 100,000 rows) need 1000 N-bit bitmaps. Total RID-list stays the same length with 1000 column values, but (Verbatim) bitmap index requires a lot more space.

ORACLE uses compression for low-density bitmaps, so don't waste space. Call bitmap "verbatim" if not compressed (means moderately high density).

Fast AND and OR of verbatim bitmaps speeds queries. Idea is: overlay unsigned int array on bitmap, loop through two arrays ANDing array (& in C), and producing result of AND of predicates. Parallelism speeds things.

But for updates, bitmaps can cause a slowdown when the bitmaps are compressed (need to be decompressed, may recompress differently).

Don't use bitmap indexes if have frequent updates (OLTP situation).

8.4 Clustered and Non-Clustered Indexes

The idea of a clustered index is that the rows of the table are in the same order as the index entries — by keyvalue.

In library, order fiction on shelves by Author (lname || fname || midinit || title). If look up novels by Dickens, go to shelves, they're all together.

In most database products, default placemen of rows on data pages on disk is in order by load or by insertion (heap). If we did that with books in a library, would be hard to collect up all Dickens.

Look at Figure 8.19, pg 510. Advantage of clustering in range search is that rows are not on random pages but in appropriate order to read in successive rows (don't need to perform new page: multiple rows/page).

Example 8.4.1. Advantage of Clustering. Large department store with hundreds of branch stores, has records for 10 million customers.

Assume each row is 100 bytes. 1 Gbyte of data. Probably only small fraction of data is in memory buffers (most systems have less than 200 MBytes of buffer available for a single table).

Boston has 1/50 of the 10M customers, or 200,000. Do a mailing,

```
select name, staddress, city, state, zip from customers
   where city = 'Boston' and age between 18 and 50 and hobby in
      ('racket sports', 'jogging', 'hiking', 'bodybuilding', 'outdoor sports');
```

OK, so each row is on a random page in non-clustered case, not found in buffer. Say read all rows in Boston (200,000 or 1/50 of 10 M).

If each row is a random page, that's 200,000 real I/Os. But 200,000 I/Os divided by 80 (I/Os/(I/Os/sec)) = secs) = 2500 seconds, about 40 minutes.

In this calculation we assumed only one disk arm moving at a time. Even if more in motion, will answer THIS query more quickly, but important point is resource use: might have multiple users contending for disk arms.

In clustered case, fit 100 byte rows twenty to a page. 20 rows/page (2KBytes/page). Since clustered, right next to one another, place 200,000 rows on 200,000/20 = 10,000 pages. (pages/(rows/page) = pages).

Now reading all rows in Boston takes (10,000/80) = 125 secs. Two minutes. 20 times faster.

That's reading all the rows for Boston. If we have indexes on age and hobby we don't need to read all these rows. Suppose these predicates eliminate 80% of the rows, leaving 1/5 of them to read, or 40,000 rows.

In the non-clustered case, 40,000 reads take 40,000/80 = 500 secs.

In the clustered case, we might still have to read the 10,000 pages: depends if hobby or age cluster city = 'Boston' further; probably not.

So still need 125 seconds (assume every page still contains at least one row, when 1/5 are hit). But still 5 times faster than non-clustered.

Clustered Index in DB2

Now how do we arrange clustered index? In DB2 UDB, have CLUSTER clause in Create Index statement (Fig 8.20, pg 512).

At most one cluster index. Create index with empty table, then LOAD table with sorted rows (or REORG after load with unsorted rows).

This is NOT a primary key index: DB2 UDB places all rows in data pages rather than on leaf of B-tree. No guarantee that newly inserted rows will be placed in clustered order!

DB2 UDB leaves space on data pages using PCTFREE spec, so newly inserted row can be *guided* to a slot in the right order. When run out of extra space get inserted rows far away on disk from their cluster ordering position.

DB2 uses multipage reads to speed things up: "prefetch I/O"; more on this in next chapter.

ORACLE Index-Organized Tables

This can provide a *primary key* clustered index, with table rows on leaf level of a B-tree, kept in order by B-tree splitting even when there are lots of changes to table. See pg. 513-514: ORGANIZATION INDEX

create table schema.tablename ({columnname datatype . . .
[ORGANIZATION HEAP | ORGANIZATION INDEX (with clauses not covered)]

(Back before ORACLE 8.i, used not to be able to have secondary indexes at the same time. But that's no longer a problem.)

ORGANIZATION INDEX *does require* a primary key index. The primary key is used automatically.

Class 12.

ORACLE [Table] Clusters: (Not *Clustering*), Fig. 8.21, pg. 514

CREATE CLUSTER [schema.]clustername -- this is BEFORE any table!
(colname datatype {, . . .}
[cluster_clause {. . .}];

The cluster_clauses are chosen from the following:

[PCTFREE n] [PCTUSED n]
[STORAGE ([INITIAL n [K|M]] [NEXT n [K|M]] [MINEXTENTS n] [MAXEXTENTS n]
[PCTINCREASE n])]
[SIZE n [K|M]] -- disk space for one keyval: default to 1 disk block
[TABLESPACE tblspacename]
[INDEX | HASHKEYS n [HASH is expr]]
[other clauses not covered];

DROP CLUSTER statement:

DROP CLUSTER [schema.]clustername
[INCLUDING TABLES [CASCADE CONSTRAINTS]];

Consider example of employees and departments, where the two tables are very commonly joined by deptno. Example 8.4.2, pg. 516.

```
create cluster deptemp
   (deptno int)
   size 2000;

create table emps
(  empno int primary key,
   ename varchar(10) not null,
   mgr int references emps,
   deptno int not null)
   cluster deptemp (deptno);

create table depts
(  deptno int primary key,
   dname varchar(9),
   address varchar(20))
```

cluster deptemp (deptno);

When we use deptno as a cluster key for these two tables, then all data for each deptno, including the departments row and all employee rows, will be clustered together on disk.

Steps:
1. create a cluster (an index cluster, the default type)
2. create the tables in the cluster
3. create an index on the cluster
4. load the data, treating the tables normally.
5. Possibly, add more (secondary) indexes.

Step 1. includes all the physical-level specifications (STORAGE clauses), so these are not done in step 2.

CREATE TABLE [schema.]tablename
(column definitions as in Basic SQL, see Figure 8.5)
CLUSTER clustername (columnname {, columnname})
-- table columns listed here that map to cluster key
[AS subquery];

The cluster owns the table data. It does not own the index data, so step 3 can specify tablespace or STORAGE specs.

create index deptempx on cluster deptemp;

Note this does NOT give us the clustered index features mentioned earlier. ROWS ARE NOT PUT IN ORDER BY KEYVALUE WHEN LOADED IN CLUSTER.

All rows with single keyvalue are stored together, but there is no ordering attempted; index on keyvalyue is separate idea from cluster.

8.5 Hash Primary Key Index

OK, now for a complete departure. See pg 517, ORACLE Cluster with use of optional clause

HASHKEYS n [HASH is expr]

Rows in a table located in hash cluster are placed in pseudo-random data page slots using a hash function, and looked up the same way, often with only one I/O. THERE IS NO DIRECTORY ABOVE ROW LEVEL.

Also, no order by keyvalue. Successive keyvals are not close together, probably on entirely different pages (depends on hash function).

Can't retrieve, "next row up in keyvalue," need to know exact value. Serious limitation, but 1 I/O lookup is important for certain applications.

Idea of Primary Index. Determines placement of rows of a table. Cluster index does this, but implies that rows close in keyvalue are close on disk. More correct to say Primary Index when hash organization.

In hash cluster, HASHKEYS n is the proposed number of slots S for rows to go to (like darts thrown at random slots, but repeatable when search).

ORACLE equates the value n with "HASHKEYS" in it's references.

It will actually choose the number of slots S to be the first prime number >= HASHKEYS, called PRIMEROOF(HASHKEYS). Example 8.5.1 has this:

```
create cluster acctclust (acctid int)
   size 80
   hashkeys 100000;
```

Can figure out how many slots actually allocated:

```
select hashkeys from user_clusters
   where cluster_name = 'ACCTCLUST';
```

Find out it is 100003. Note we can create our own hash function (not normally advisable, add to Create Cluster above):

```
create cluster acctclust (acctid int)
   size 80
   hashkeys 100000 hash is mod(acctid, 100003);
```

Now in Example, create single table in this cluster with unique hashkey:

```
create table accounts
(  acctid integer primary key,
```

```
    . . .)  -- list of other columns here
  cluster  acctclust  (acctid);
```

Note that in accounts table, B-tree secondary index created on acctid because it's a primary key. Access normally through hash though.

HASHKEYS and SIZE determine how many disk blocks are initially allocated to cluster. First determine how many slots S = PRIMEROOF(HASHKEYS).

Figure how many slots B will fit on a disk block: [TEXT] [WAS ERROR]

B = MAX(FLOOR(Number of usable bytes per disk block/SIZE), 1)

The number of usable byte in a disk block on UMB machine db is 1962. Will change. Can use 2000 for calculations on hw, Exam.

Next, calculate how many Disk blocks needed for S slots when B to a block.

D = CEIL(S/B)

(So SIZE and HASHKEYS determine number of disk blocks in hash cluster; can't change HASHKEYS later; SIZE only advisory.)

Assign one or more tables to hash cluster with (common) hashkey (i.e., clusterkey for hash cluster). I will tend to speak of one table, unique key.

Example given in Ex. 8.5.1 is single accounts table with accountid hashkey.

Key values will be hashed to some slot and row sent to that slot.

sn1 = h(keyval1)

Could have two distinct key values hashed to same slot (collision).

(When multiple tables involved, could clearly have multiple rows with SAME keyvalue go to same slot, not a collision but must be handled.)

(Even if unique, if number of distinct keys grows beyond number of slots, must have a lot of collisions. Will have collisions before that.)

The way ORACLE handles this is to expand its slots on a page as long as space remains, and after that overflow to new pages. See Fig. 8.23, pg 516.

(Draw a picture of Root pages originally allocated and overflow pages.)

In hashing, try not to overflow. If lot of rows would hash to same slot, not appropriate for hashing. E.g., depts-emps when lot of emps per deptno.

In accounts case when lot of collisions, while most rows on root block, Query plan uses hashed access; after a lot of overflow, tries to use secondary index on acctid.

Usually try to create enough slots of sufficient size so collision relatively rare, disk block pages only half full.

In Unique hashkey case, make enough slots so on average only half-full.

No Incremental Changes in Size of Hash Table

How create function h(keyvalue) for 0 to S-1. Have more generic function r(x) (random number based on x), so r(keyvalue) is in range (0.0 to 1.0).

(Easy — just generate random mantissa, characteristic is 2^0). Now set:

h(keyvalue) = INTEGER_PART_OF(S*r(keyvalue))

But note that we can't incrementally increase size of hash table based on this. If have different number of slots S', new h' won't give same slots for old values. If r(keyvalue) = 0.49, INT(4*0.49) = 1, int(5*0.49) = 2.

Therefore can't enlarge number of slots incrementally to reduce number of collisions. (There is a known algorithm allowing dynamic hash table growth, used for example in Sybase IQ).

8.6 Throwing Darts at Random Slots

Conceptual experiment of throwing darts at slots from a LONG way away. Hash into slots as in ORACLE (random numbers), how many slots empty?

The case of ORACLE is the experiment: Unlimited Slot Occupancy, question of how many slots are occupied. Number of darts N, number of slots M.

Question is: how many slots occupied. Not just M - N because some slots will be occupied twice, some three times.

Pr(slot s gets hit by dart d) = 1/M

Pr(slot s does not get hit by dart d) = (1 – 1/M)

Pr(slot s does not get hit by N darts thrown in succession) = $(1 – 1/M)^N$

Pr(slot s does get hit by one or more of N darts) = 1 - $(1 – 1/M)^N$

E(S) = Expected number of slots hit by N darts = M(1 - $(1 – 1/M)^N$)

Show that (approximately) **[8.6.3]** $e^{-1} = (1 – 1/M)^M$

[8.6.4] E(S) = M $(1 – e^{–N/M})$

[8.6.5] E(Slots not hit) = M $e^{–N/M}$

Then solve question when slot occupancy of 1: how many retries to get last dart in slot (this is question of Retry Chain, will use later).

Easy to figure out that as number of darts gets closer and closer to number of slots, number of retries becomes longer and longer.

If (N-1)/M close to 1, Length of Retry Chain E(L) to place Nth dart approaches infinity. If N-1 = M, can't do it. Call (N-1)/M the FULLNESS, F.

Then **[8.6.11]** E(L) = 1/(1 - F)

Finally: When Do Hash Pages Fill Up. Say disk page can contain 20 rows, gets average of 10 rows hashed to it.

When will it get more than it can hold (hash overflow, both cases above)?

Say we have 1,000,000 rows to hash. Hash them to 100,000 pages. This is a binary distribution, approximated by Normal Distribution.

Each row has 1/100,000 probability P of hitting a particular page.

Expected number of rows that hit page is P*1,000,000 = 10.

Standard Deviation is: SQRT(P*(1-P)*1,000,000) = SQRT(10) = 3.162.

Probility of 20 or more rows going to one page is PHI(10/3.162) = .000831, but that's not unlikely with 100,000 pages: 83 of them.