

Daily Course Notes for CS634

Database Management

Patrick E. O'Neil
Classes 11-16

Class 11.

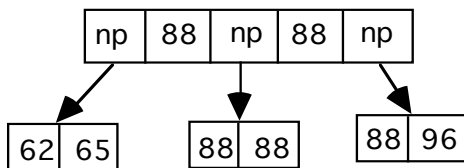
Working on hw 2, Exam soon after due date, so solutions must be posted in advance. After Solutions up, later hw2 submissions won't count!

Any questions from last time?

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



(Work out how get this on board.) Seems 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 duplicate list of 88s. Done in binsearch, remember? Must follow np to left of leftmost sepkey = 88. (No 88 at that leaf level, but there could be one since duplicates are 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 entry there, but then waste time later in retrievals trying to pick up nonexistent rows).

Thus when insert row, must find way to leaf level of ALL indexes and insert entries; when delete row must delete entries. Overhead. (Idea of deferred indexing used in MODEL 204 has never caught on very much.)

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), and carry this even up to separators. But most commercial database B-trees don't do this properly (DB2 keeps RIDs in order for duplicate keyvalues, but lookup doesn't keep track). (Bitmap index, explained below, solves this.)

Said before, there's an algorithm for merging B-tree nodes when delete, opposite of split nodes. But not many products use it: only DB2 UDB seems to have done so recently. (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 in index modifications argues against heavy index use except for columns with few updates (name, socsecno). Inserts and deletes don't occur very often in most business applications (many more credit balance changes than starting or ending an account.)

Most updates of tables occur to increment-decrement fields (account balance), and such fields are rarely indexed, so no index updates needed.

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. Might (but usually don't) hold off most other updates (change address, name) until daily reorganization of indexes, since that would probably be more efficient.

Index RID lists in DB2

DB2 has nice representation for multiple duplicate keyvalues in an index. See Fig. 8.17 on pg 499 for data page layout with duplicate keyvalues.



Each block has keyval once and then a list of RID values, up to s full disk page. Prx contains pointer to prior block (2 bytes), next block (2 bytes) and count of RIDs in this block (2 bytes). [Note: used to be 255 RIDs max block.]

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

Oracle Bitmap Index

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. 501. 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.

A bitmap (or bit vector) can be used 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. 500-501 in text, 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,  
  gender char(1), salarycat smallint, dept char(5));
```

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

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

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

In DB2, RID needs = 4 bytes (In Oracle, ROWID needs 6 bytes). Thus RID-list can represent one row with 32 bits (ROWID list needs 48 bits per row). At density = 1/32 for DB2 (1/48 for Oracle), a bitmap can represent an equal number of rows in the same space.

But if density is a lot *lower* than 1/32, say 1/1000 for 1000 column values, 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 (rather 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.

When updated, compressed bitmaps can cause a slowdown; they may need to be decompressed, and may recompress differently.

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

Class 12.

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, find them all together on shelves!

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

Look at Figure 8.19, pg 504. 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 read in new page from disk so often when read rows in order by clustering key: we will have 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, 20 rows/pg. Total of 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).
(10M rows/20 rows/pg) = 500,000 pages containing 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 selected rows in Boston (200,000 or 1/50 of 10 M).

If each row is a random page, that's close to 200,000 real I/Os (will provide more precision later). Time to read from a disk: 200,000 I/Os/(80 I/Os/sec) = 2500 seconds, about 40 minutes. (Modern disk, 20MB/sec, total 50 sec.)

Even if 2 or 3 rows on a page the rows probably will not be together in the index, and page will drop out of buffer between row accesses.

In 40 minute calculation we assumed only one disk arm moving at a time. Even if more in motion, will answer THIS query more quickly, but might have many users contending for disk arms and this query uses lots of I/O.

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 4 times faster than non-clustered. (in fact, a good deal faster as we will see: no seek time between page I/Os).

Clustered Index in DB2

Now how do we arrange clustered index? In DB2 UDB, have CLUSTER clause in Create Index statement (Look at Fig 8.20, pg 506).

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 self-modifying B-tree clustered 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 requires 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. 507: ORGANIZATION INDEX

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

ORGANIZATION INDEX *does require* a primary key index. The primary key is used automatically. (Can "uniquify" zipcode (say), but unique index is much less efficient for access!! Must repeat NEW keyval for each RID.)

There is also a serious problem for secondary indexes since each secondary index lookup must give unique primary keyvalue instead of a ROWID, and access down secondary index B-tree and then through primary key B-tree.

ORACLE [Table] Clusters: (Not *Clustering in prior sense* -- all rows with identical keyval are stored together but with no ordering)

Fig. 8.21, pg. 509

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

The cluster_clauses are these:

```
[PCTFREE n] [PCTUSED n]  
[STORAGE clause] -- override tablespace default storage clause  
[SIZE n [KIM]] -- disk space for one keyval: default to 1 disk block  
[TABLESPACE tblspacename]  
[INDEX | HASHKEYS n [HASH is expr]] -- can't also do for table in cluster  
[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. 510-511.

```
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); -- need to create index on the cluster
```

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.
-- e.g., create index enamex on emps(name)

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 clustlename (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 for the indexes.

```
create index deptimepx on cluster deptime;
```

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

Class 13.

8.5 Hash Primary Key Index

OK, now for a complete departure. See pg 512, ORACLE Cluster with use of optional clause

```
HASHKEYS n [HASH is expr]
```

Rows in a table located in a 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 (e.g.: acctid unique). 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 its references.

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

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

Can use a query to determine 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 is normally through hash though (faster).

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

Figure how many slots B on each disk block:

$$B = \text{MAX}(\text{FLOOR}(\text{Number of usable bytes per disk block}/\text{SIZE}), 1)$$

Number of usable bytes in a disk block on UMB database machine is 8000. Assume 2000 in calculations on hws, Exams unless told otherwise.

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

$$D = \text{CEIL}(S/B)$$

(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.

$$\text{sn1} = h(\text{keyval1})$$

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.)

ORACLE handles this by expanding its slots on a page as long as space remains, and after that overflowing 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 $h(\text{keyvalue})$ for 0 to $S-1$. Start with generic function $r(x)$ (random number based on x), so $r(\text{keyvalue})$ is in range (0.0 to 1.0).

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

$$h(\text{keyvalue}) = \text{INTEGER_PART_OF}(S * r(\text{keyvalue})) \text{ -- get 0 to } S-1 \text{ at random}$$

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

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: throw M darts at N slots from a LONG distance. Random hash into slots as in ORACLE, how many slots empty?

ORACLE hashing is one 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. **PICTURE HERE.**

E.g., 1M rows retrieved from 25M rows, 20 rows per disk page: how many disk pages never hit?

$\Pr(\text{slot } s \text{ gets hit by dart } d) = 1/M$ (all slots have equal probability)

$\Pr(\text{slot } s \text{ does not get hit by dart } d) = (1 - 1/M)$

$\Pr(\text{slot } s \text{ does not get hit by } N \text{ darts thrown in succession}) = (1 - 1/M)^N$

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

$E(S) = \text{Expected number of slots hit by } N \text{ darts} = M(1 - (1 - 1/M)^N)$

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

[8.6.4] $E(S) = M(1 - e^{-N/M})$

[8.6.5] $E(\text{Slots not hit}) = M - E(S) = M - (M - M e^{-N/M}) = M e^{-N/M}$

NOTE can also estimate number of pages occupied by number of randomly chosen rows; e.g., 1M rows, 10 rows/pg, total 10,000 pgs. If range search customers in Boston (1/20 of rows, non-clustered), how many pages do these rows occupy? (50,000 darts (rows), 100,000 slots (Blocks).)

$E(S) = 100,000(1 - e^{-N/M})$; $N = 50,000$, $M = 100,000$, so

$E(S) = 100,000(1 - e^{-1/2}) = (\text{calculator}) = 100,000(1 - .6487213)$

$= 35128$ Blocks occupied by rows. **GUARANTEED EXAM QUESTION!!**

Next, 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).

As number of darts gets closer and closer to number of slots, number of retries becomes longer and longer. (Approximate hyperbola curve.)

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: $\text{SQRT}(P*(1-P)*1,000,000) = \text{SQRT}(10) = 3.162$. NOTE: Approach **Birthday Surprise** when fewer rows fit!

Probability of 20 or more rows going to one page is $\text{PHI}(10/3.162) = .000831255$, but that's not unlikely with 100,000 pages: 83 of them.

$\Phi(x)$ handout on next page.

$\Phi(x)$, the area under the standard Gaussian density function to the left of x

x	000.00	000.01	000.02	000.03	000.04	000.05	000.06	000.07	000.08	000.09
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
1.0	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9131	0.9147	0.9162	0.9177
1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
2.0	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817
2.1	0.9821	0.9826	0.9830	0.9834	0.9838	0.9842	0.9846	0.9850	0.9854	0.9857
2.2	0.9861	0.9864	0.9868	0.9871	0.9875	0.9878	0.9881	0.9884	0.9887	0.9890
2.3	0.9893	0.9896	0.9898	0.9901	0.9904	0.9906	0.9909	0.9911	0.9913	0.9916
2.4	0.9918	0.9920	0.9922	0.9925	0.9927	0.9929	0.9931	0.9932	0.9934	0.9936
2.5	0.9938	0.9940	0.9941	0.9943	0.9945	0.9946	0.9948	0.9949	0.9951	0.9952
2.6	0.9953	0.9955	0.9956	0.9957	0.9959	0.9960	0.9961	0.9962	0.9963	0.9964
2.7	0.9965	0.9966	0.9967	0.9968	0.9969	0.9970	0.9971	0.9972	0.9973	0.9974
2.8	0.9974	0.9975	0.9976	0.9977	0.9977	0.9978	0.9979	0.9979	0.9980	0.9981
2.9	0.9981	0.9982	0.9982	0.9983	0.9984	0.9984	0.9985	0.9985	0.9986	0.9986
3.0	0.9987	0.9987	0.9987	0.9988	0.9988	0.9989	0.9989	0.9989	0.9990	0.9990
3.1	0.9990	0.9991	0.9991	0.9991	0.9992	0.9992	0.9992	0.9992	0.9993	0.9993
3.2	0.9993	0.9993	0.9994	0.9994	0.9994	0.9994	0.9994	0.9995	0.9995	0.9995
3.3	0.9995	0.9995	0.9995	0.9996	0.9996	0.9996	0.9996	0.9996	0.9996	0.9997
3.4	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9997	0.9998

Class 14. Hand out PRACTICE EXAM. Any questions?

Chapter 9. Query Optimization

Query optimizer creates a procedural access plan for a query. After submit a Select statement, three steps.

```
select * from customers where city = 'Boston'
and discnt between 10 and 12;
```

(1) Parse (as in BNF), look up tables, columns, views (query modification), check privileges, verify integrity constraints.

(2) Query Optimization. Look up statistics in System tables. Figure out what to do. Basically consider competing plans, choose the cheapest. (What do we mean by cheapest? What are we minimizing? See shortly.)

(3) Now Create the "program" (code generation). Then Query Execution.

We will be talking about WHY a particular plan is chosen, not HOW all the plan search space is considered. Question of how compiler chooses best plan is beyond scope of course. Assume QOPT follows human reasoning.

For now (& in most of theory), considering only queries, not updates.

Section 9.1. pg. 535. Now what do we mean by cheapest alternative plan? What resources are we trying to minimize? CPU and I/O. Given a PLAN, talk about $COST_{CPU}(PLAN)$ and $COST_{I/O}(PLAN)$.

But two plans $PLAN_1$ and $PLAN_2$ can have incomparable resource use with this pair of measures. See Figure 9.1, pg 535.

	$COST_{CPU}(PLAN)$	$COST_{I/O}(PLAN)$
$PLAN_1$	9.2 CPU seconds	103 reads
$PLAN_2$	1.7 CPU seconds	890 reads

Which is cheaper? Depends on which resources have a bottleneck. DB2 takes a convex combination of the resource uses for the total cost, $COST(PLAN)$:

$$COST(PLAN) = W_1 \cdot COST_{CPU}(PLAN) + W_2 \cdot COST_{I/O}(PLAN); \quad W_i \geq 0, \quad \text{SUM} = 1.0$$

Tradeoff between memory and I/O in Tx system too -- how much buffering?
 A DBA should try to avoid bottlenecks by understanding the *WORKLOAD* for a system. Estimated total CPU and I/O for all users at peak work time.

Buy more powerful CPU, more memory, or more disks? List all queries and rate submitted per second. Q_k , $RATE(Q_k)$ as in Figure 9.2. (pg. 537.)

Query Type	RATE(Query) in Submissions/second
Q1	40.0
Q2	20.0

Figure 9.2 Simple Workload with two Queries

Then work out $COST_{CPU}(Q_k)$ and $COST_{I/O}(Q_k)$ for plan that database system will choose. Now can figure out total average CPU or I/O needs per second:

$$RATE(CPU) = \sum_k RATE(Q_k) \cdot COST_{CPU}(Q_k)$$

Similarly for I/O. Tells us how many disks to buy, how powerful a CPU. (Costs go up approximately linearly with CPU power within small ranges).

Ideally, the weights W_1 and W_2 used to calculate $COST(PLAN)$ from I/O and CPU costs will reflect actual costs of equipment

Good DBA tries to estimate workload in advance to make purchases, have equipment (multiple CPUs and Disks) ready for a new application.

*** Note that one other major expense is response time. Could have poor response time (1-2 minutes) even on lightly loaded inexpensive system.

This is because many queries need to perform a LOT of I/O, and it's hard for a DBA to provide perfect I/O parallelism: many I/Os will occur one after another in sequence because of requests to same disk.

But long response time also has a cost, in that it wastes user time. Pay workers for wasted time. Employees quit from frustration and others must be trained. Vendors try hard to reduce response times.

Can run queries in parallel with careful loading; worth it if extremely heavy I/O and response time is a problem, assuming that the number of queries running at once is small compared to number of disks.

Note that in plans that follow, we will not try to estimate CPU. Too hard. Assume choose plan with best I/O and try to estimate that. Often total cost is proportional to I/O since each I/O entails extra CPU cost as well.

Statistics. Need to gather, put in System tables. System does not automatically gather them with table load, index create, updates to table.

In DB2, use Utility called RUNSTATS. Fig. 9.3, pg. 538.

```
RUNSTATS ON TABLE username.tablename
  [WITH DISTRIBUTION [AND DETAILED] {INDEXES ALL | INDEX indexname}]
  [other clauses not covered or deferred]
```

e.g.: runstats on table poneil.customers;

System learns how many rows, how many data pages, stuff about indexes, etc., placed in catalog tables.

ORACLE uses ANALYZE statement. Fig. 9.4, pg. 538.

```
ANALYZE {INDEX | TABLE | CLUSTER}
  [schema.] {indexname | tablename | clustertype}
  {COMPUTE STATISTICS | other alternatives not covered}
  {FOR TABLE | FOR ALL [INDEXED] COLUMNS [SIZE n]
  | other alternatives not covered}
```

Will see how statistics kept in catalog tables in DB2 on pg. 79, these Notes.

Retrieving Query Plans. in DB2 and ORACLE. Perform SQL statement. In DB2:

```
EXPLAIN PLAN [SET QUERYNO = n] [SET QUERYTAG = 'string'] FOR
  explainable-sql-statement;
```

Can do this before running the SQL statement; For example:

```
explain plan set queryno = 1000 for
  select * from customers
  where city = 'Boston' and discnt between 12 and 14;
```

The Explain Plan statement puts rows in a "plan_table" (created by user) to show procedural steps in query plan. Can get rows back with this query:

```
select * from plan_table where queryno = 1000;
```

Recall that a Query Plan is a sequence of procedural access steps that carry out a program to answer the query. Steps are peculiar to the DBMS.

From one DBMS to another, difference in steps used is like difference between programming languages. Can't learn two languages at once.

We will stick to a specific DBMS in what follows, MVS DB2, so we can end up with an informative benchmark.

But we will have occasional references to ORACLE, DB2 UDB. Here is the ORACLE Explain Plan syntax.

```
EXPLAIN PLAN [SET STATEMENT_ID = 'text-identifier'] [INTO  
[schema.]tablename]  
FOR explainable-sql-statement;
```

This inserts a sequence of statements into a user created DB2/ORACLE table known as PLAN_TABLE one row for each access step. To learn more about this, see ORACLE9 documentation on cs634 Web Page.

Need to understand what basic procedural access steps ARE in the particular product you're working with.

The set of steps allowed is the "bag of tricks" the query optimizer can use. Think of these procedural steps as the "instructions" a compiler can use to create "object code" in compiling a higher-level request.

A system that has a smaller bag of tricks is likely to have less efficient access plans for some queries.

MVS DB2 (and the architecturally allied DB2 UDB) have a wide range of tricks, but can't do bitmap indexing or hashing.

Still, very nice capabilities for range search queries, and probably the most sophisticated query optimizer.

Basic procedural steps covered in the next few Sections, pg. 539:

Tablespace Scan	Look through all rows of table
Index Scan	Retrieve rows through an index lookup
Unique Index Scan	Retrieve row through unique index
Unclustered Matching Index Scan	Retrieve multiple rows through a non-unique index, rows not same order
Clustered Matching Index Scan	Retrieve multiple rows through a non-unique clustered index
Index-Only Scan	Query answered in index, not rows

Note that each step we have listed above will access all the rows restricted by the WHERE clause in any single table query using that step

Need two tables in FROM clause to require two steps of this kind

We will cover joins later. We will also cover multi-step plans for a single table query. Such multi-step plans combine multiple indexes to retrieve data. In simple steps above, only one index per table can be used.

9.2 Table Space Scans and I/O

Single step. The plan table (plan_table) will have a column ACCESSTYPE with value R (ACCESSTYPE = R for short) for Table Space Scan.

Example 9.2.1. Table Space Scan Step. Look through all rows in table to answer query, maybe because there is no index that will help.

Assume in DB2 an employees table with 200,000 rows, rows of 200 bytes, 4 KByte pages just 70% full. Thus 2800 usable pages, 14 rows/pg. Need $\text{CEIL}(200,000 \text{ rows/table} / 14 \text{ rows/page}) = 14,286 \text{ pages/table}$.

Consider the query:

```
select eid, ename from employees where socsecno = 113353179;
```

If there is no index on socsecno, only way to answer query is by reading in all rows of table. (Stupid not to have an index if query occurs with any frequency! But index exists if socsecno declared Unique or Primary Key.)

Therefore have to read all pages in table in from disk, and $\text{COST}_{I/O}(\text{PLAN}) = 14286 \text{ I/Os}$. Does this mean 14286 random I/Os? Maybe not.

But if we assume random I/O on a single disk, then at 80 I/Os per second, need about $14286/80 = 178.6$ seconds, a bit under three minutes. This dominates CPU by a large factor and would predict elapsed time quite well. □

Homework 4, non-dotted Exercises through Exercise 9.9. This is due when we finish Section 9.6. Maybe more later.

Assumptions about I/O REVIEW: Quick

We are now going to talk about I/O assumptions a bit. First, there might be PARALLELISM in performing random I/Os from disk.

The pages of a table might be *striped* across several different disks, with the database system making requests in parallel for a single query to keep all the disk arms busy. See Figure 9.5, pg. 542.

When DB2 was first documented it was rare for most DBMS systems to make multiple requests at once (a form of parallelism), now it's common.

DB2 has a special form of sequential prefetch now where it stripes 32 pages at a time on multiple disks, requests them all at once.

While parallelism speeds up TOTAL I/O per second (especially if there's only one user process running), it doesn't really save any RESOURCE COST.

If it takes $1/80$ sec = 12.5 ms (0.0125 seconds) to do a random I/O, doesn't save resources to do 10 random I/Os at once on 10 different disks.

Still have to make all the same disk arm movements, cost to rent the disk arms is the same if there is parallelism: just spend more per second.

Will speed things up if queries running are fewer than the number of disks, and there is extra CPU not utilized in a non-parallel system. Then can use more of the disk arms and CPUs with this sort of multi query parallelism.

Clustered parallelism shows up best when there is only one query running!

If there are lots of queries compared to number of disks and accessed pages are randomly placed on disks, probably keep all disk arms busy already without any parallelism in each single query.

But there's another factor operating. Two disk pages that are close to each other on one disk can be read faster because there's a shorter seek time.

Recall that the system tries to make extents contiguous on disk, so I/Os in sequence are faster. Thus, a table that is made up of a sequence of (mainly) contiguous pages, one after another within a track, will take much less time to read in.

In fact it seems we should be able to read in successive pages at full transfer speed should take about $1/800 = .00125$ secs per page.

Used to be that by the time the disk controller has read in the page to a memory buffer and looked to see what the next page request is, the page immediately following has already passed by under the head.

But now with multiple requests to the disk outstanding, we really COULD get the disk arm to read in the next disk page in sequence without a miss.

Another factor supports this speedup: the typical disk controller buffers an entire track in its memory whenever a disk page is requested.

Reads in whole track containing the disk page, returns the page requested, then if later request is for page in track doesn't have to access disk again.

So when we're reading in pages one after another on disk, it's like we're reading from the disk an entire track at a time.

I/O is about TEN TIMES faster for many disk pages in sequence compared to randomly placed I/O. (Accurate enough for rule of thumb.)

ROUGH RULE OF THUMB ON BOARD: We can do 800 I/Os per second when pages in sequence (S) instead of 80 for randomly placed pages (R). Sequential I/O takes 0.00125 secs instead of 0.0125 secs for random I/O.

DB2 Sequential Prefetch (S) makes this possible even if turn off buffering on disk (which actually hurts performance of random I/O, since reads whole track it doesn't need: adds 0.008 sec to random I/O of 0.0125 sec)

These performance numbers are out of date but IBM still puts effort into making I/O requests sequentially in a query plan to gain I/O advantage!

Example 9.2.2. Table Space Scan with Sequential Advantage. The 14286R of Example 9.2.1 becomes 14286S (S for Sequential Prefetch I/O instead of Random I/O). And 14286S requires $14286/800 = 17.86$ seconds instead of the 178.6 seconds of 142286R. Note that this is a REAL COST SAVINGS, that we are actually using the disk arm for a smaller period. Striping reduces elapsed time but not COST. □

Cover idea of List Prefetch. 32 pages, not in perfect sequence, but relatively close together. Difficult to predict time.

We use the rule of thumb that List Prefetch reads in 200 pages per second.

See Figure 9.10, page 546, for R, S, and L type I/O.

Plan table row for an access step will have PREFETCH = S for sequential prefetch, PREFETCH = L for list prefetch, PREFETCH = blank if random I/O. See Figure 9.10. And of course ACCESSTYPE = R for tablespace scan.

Note that sequential prefetch is just becoming available on UNIX database systems. Often just put a lot of requests out in parallel and depend on smart I/O system to use arm efficiently

Class 15.

Exam 1.

Class 16.

Turn back Exam 1.

9.3 Simple Indexed Access in DB2. All on DB2 Now!!!

Index helps efficiency of query plan. There is a great deal of complexity here. Remember, we are not yet covering queries with joins: only one table in FROM clause and NO subquery in WHERE clause.

Examples with tables: T1, T2, . . . , columns C1, C2, C3, . . .

Example 9.3.1. Assume index C1X exists on column C1 of table T1 (always a B-tree secondary index in DB2). Consider:

```
select * from T1 where C1 = 10;
```

This is a *Matching Index Scan*. In plan table: ACESSTYPE = I, ACCESSNAME = C1X, MATCHCOLS = 1. (MATCHCOLS might be >1 in multiple column index.)

Perform matching index scan by walking down B-tree to LEFTMOST entry of C1X with C1 = 10. Retrieve row pointed to.

Loop through entries at leaf level from left to right until run out of entries with C1 = 10 (follow sibling ptrs). For each such entry, retrieve row pointed to. No assumption about clustering or non-clustering of rows here. □

Example 9.3.2, pg 548. Assume matching index scan used on C1X; additional restrictions on other columns are then validated as rows are accessed (row is *qualified*: look at row, check if matches restrictions).

Not all predicates are *indexable* (**bad name**). In DB2, indexable predicate is one that can be used in a *matching index scan*, i.e. a lookup that uses a contiguous section of an index. Covered in full in Section 9.5.

For example, looking up words in the dictionary that start with the letters 'pre' is a matching index scan. Looking up words ending with 'tion' is not.

DB2 considers the predicate $C1 \neq 10$ to be non-indexable. It is not impossible that an index could be useful in a query with this predicate:

```
select * from T1 where C1 <> 10;
```

E.g., if 99% of rows have $C1 = 10$; but the statistics usually weigh against index use and so the query will be performed by a table space scan. More on indexable predicates later.

OK, now what about this query?

```
select * from T1 where C1 = 10 and C2 between 100 and 200  
and C3 like 'A%';
```

These three predicates are all indexable. If have only C1X, retrieval might be like previous example with rows qualified on other two predicates.

If have index combinx, created by:

```
create index combinx on T1 (C1, C2, C3) . . .
```

Will be able to filter RIDs of rows much better on ranges on C1, C2 AND C3 before going to data. Like books in a card catalog, looking up author & title:

```
lname = 'James' (c1 = 10) and fname between 'H' and 'K'  
and title begins with letter 'A';
```

Later we will cover the question of how to filter the RIDs of rows to retrieve if we have three different indexes, C1X, C2X, and C3X.

We can do this by taking out "cards" for each index range, ordering by RID, then merge-intersecting these sequences. It is an interesting query optimization problem whether this is worth it.

OK, now some examples of simple index scans.

Example 9.3.3. Index Scan Step, Unique Match. Continuing with Example 9.2.1, employees table with 200,000 rows of 200 bytes and pctfree = 30, so 14 rows/pg and $\text{CEIL}(200,000/14) = 14,286$ data pages.

Assume index on eid, eid_x, also has pctfree = 30, and eid_lIRID takes up 10 bytes, so 280 entries/pg, and $\text{CEIL}(200,000/280) = 715$ leaf level pages. Next level up $\text{CEIL}(715/280) = 3$. Root next level up. Write on board:

employees table: 14,286 data pages
index on eid, eid_x: 715 leaf nodes, 3 level 2 nodes, 1 root node.

Now query: select ename from employees where eid = '12901A';

Root, on level 2 node, 1 leaf node, 1 data page. Seems like 4R. But what about buffered pages?

Five minute rule says should purchase enough memory so pages referenced more frequently than once every K seconds (popular pages; originally 5 minutes but increasing as memory becomes cheaper relative to disk arm cost) should stay in memory.

Assume we have done this. If workload has 1 query per second retrieving ename with eid = random-const, then leaf nodes and data pages not buffered, but upper nodes of eid_x are. So really 2R is cost of query.

This Query Plan is a single step, with ACCESSTYPE = I, ACCESSNAME = eid_x, MATCHCOLS = 1. □

Class 17.

OK now we introduce a new table called prospects. Based on direct mail applications (junk mail). People fill out warranty cards, name hobbies, salary range, address, etc.

50M rows of 400 bytes each. FULL data pages (pctfree = 0) and on all indexes: 10 rows on 4 KByte page, so 5M data pages.

prospects table: 5M data pages

Now: create index addrx on prospects (zipcode, city, straddr) cluster . . .;

zipcode is integer or 4 bytes, city requires 12 bytes, straddr 20 bytes, RID 4 bytes, and assume NO duplicate values so no RID-lists (compression).

Thus each entry requires 40 bytes, and we can fit 100 on a 4 KByte page. With 50M total entries, that means 500,000 leaf pages. 5000 directory nodes at level 2. 50 level 3 node pages. Then root page. Four levels.

Also assume a nonclustering hobbyx index on hobbies, 100 distinct hobbies (. . . cards, chess, coin collecting, . . .). We say CARD(hobby) = 100. (Look ahead to pg. 558 for statistics gathering.)

(Like we say CARD(zipcode) = 100,000. Not all possible integer zipcodes can be used, but for simplicity say they are.)

Duplicate compression on hobbyx, each key (8 bytes?) amortized over 1K RIDs, so can fit 984 (if 255 RIDs per block) per 4 KByte page, call it 1000.

Thus 1000 entries per leaf page. With 50M entries, have 50,000 leaf pages. Then 50 nodes at level 2. Then root.

prospects table	addrx index	hobbyx index
50,000,000 rows	500,000 leaf pages	50,000 leaf pages
5,000,000 data pages	5,000 level 3 nodes	151 level 2 nodes
(10 rows per page)	50 level 2 nodes	1 root node
	1 root node	(1000 entries/leaf)
	CARD(zipcode)= 100,000	CARD(hobby)=100

Figure 9.12. Some statistics for the prospects table, page 552

Example 9.3.4. Matching Index Scan Step, Unclustered Match.
Consider the following query:

```
select name, straddr from prospects where hobby = 'chess';
```

Query optimizer might assume each of 100 hobbies equally likely, so restriction cuts 50M rows down to 500,000. (Assumption depends on RUNSTATS measured histogram, but we generally assume equal likely.)

Walk down hobby index (2R for directory nodes) and across 500,000 entries (1000 per page so 500 leaf pages, sequential prefetch so 500S).

For every entry, read in row -- non clustered so all random choices out of 5M data pages, 500,000 distinct I/Os (not in order, so R), 500,000R.

Total I/O is 500S + 500,002R. Time is $500/800 + 500,002/80$, about $500,000/80 = 6250$ seconds. Or about 1.75 hours (2 hrs = 7200 secs). □

Really only picking up 500,000 distinct pages, will lie on less than 500,000 pages (out of 5 M). Would this mean less than 500,000 R because buffering keeps some pages around for double/triple hits?

VERY TRIVIAL EFFECT! Hours of access, 120 seconds pages stay in buffer.

Can generally assume that upper level index pages are buffer resident (skip 2R) but leaf level pages and maybe one level up are not. Should calculate index time and can then ignore it if insignificant.

If we used a table space scan for Example 9.3.4, qualifying rows to ensure hobby = 'chess, how would time compare to what we just calculated?

Simple: 5M pages using sequential prefetch, $5,000,000/800 = 6250$ seconds. (Yes, CPU is still ignored — in fact is relatively insignificant.)

But this is the same elapsed time as for indexed access of 1/100 of rows!!

Yes, surprising. But 10 rows per page so about 1/10 as many pages hit, and S is 10 times as fast as R.

Query optimizer compares these two approaches and chooses the faster one. Would probably select Table Space Scan here But minor variation in CARD(hobby) could make either plan a better choice.

Example 9.3.5. Matching Index Scan Step, Clustered Index Scan.
Pg 553. Consider the following query:

```
select name, straddr from prospects          Recall pg 75 cluster by
  where zipcode between 02159 and 03158;    addrx: starts with zip
```

Recall $CARD(\text{zipcode}) = 100,000$. Range of zipcodes is 1000. Assume this cuts number of rows down by a factor of 1/100. SAME AS 9.3.4.

Bigger index entries. Walk down to leaf level and walk across 1/100 of leaf level: 500,000 leaf pages, so 5000 pages traversed. I/O of 5000S.

And data is clustered by index, so walk across 1/100 of 5M data pages, 50,000 data pages, and they're in sequence on disk, so 50,000S.

Compared to Nonmatching index scan of Example 9.3.4, walk across 1/10 as many pages and do it with S I/O instead of R. Ignore directory walk.

Then I/O cost is 55,000S, with elapsed time $55,000/800 = 68.75$ seconds, a bit over 1 minute, compared with 1.75 hrs for unclustered index scan. □

The difference between Examples 9.3.4 and 9.3.5 doesn't show up in the PLAN table (pg 547). Must look at ACCESSNAME = addrx and note that this index is clustered, (clusterratio) whereas ACCESSNAME = hobbyx is not.

_ See SYSINDEXES = addrx, pg 558

(1) Clusterratio determines if index still clustered in case rows exist that don't follow clustering rule. (Inserted when no space left on page.)

(2) Note that entries in addrx are 40 bytes, rows of prospects are 400 bytes. Seems natural that 5000S for index, 50,000S for rows.

Properties of index:

1. Index has directory structure, can retrieve range of values
2. Index entries are ALWAYS clustered by values
3. Index entries are smaller than the rows.

Example 9.3.6. Concatenated Index, Index-Only Scan. Pg 554.
Assume (just for this example) a new index, naddrx:

```
create index naddrx on prospects (zipcode, city, straddr, name)
  . . . cluster . . . ;
```

Now same query as before:

```
select name, straddr from prospects where zipcode  
between 02159 and 03158;
```

Can be answered in INDEX ONLY (because find range of zipcodes and read name and straddr off components of index: Show components:

```
naddrx keyvalue: zipcodeval.cityval.straddrval.nameval
```

This is called an Index Only scan, and with EXPLAIN plan table gets new column: INDEXONLY = Y (ACCESSTYPE = I, ACCESSNAME = naddrx). Previous plans had INDEXONLY = N.

(All these columns always reported; I just mention them when relevant.)

Time? Assume naddrx takes 60 bytes instead of 40 bytes, then amount read in index, instead of 5000S is 7500S, elapsed time $7500/800 = 9.4$ seconds. Compare to 62.5 seconds with Example 9.3.5. □

Valuable idea, Index Only. Select count(*) from . . . is always index only if index can do in a single step at all, since count entries. But can't build index on the spur of the moment. If don't have needed one already, out of luck. E.g., consider query:

```
select name, straddr, age from prospects where zipcode  
between 02159 and 02258;
```

Now naddrx doesn't have all needed columns, so need to access rows!

If try to foresee all needed components in an index, might be essentially duplicating the rows, so will lose performance boost from size.

Indexes cost something. Disk media cost is not high, but with inserts or updates of indexed rows, lot of extra I/O, and this is quite expensive (but this is not a common problem).

With read-only, like prospects table, load time increases. Still often have every col of a read-only table indexed.

Chapter 9.4. Filter Factors and Statistics

Recall, estimated probability that a random row made some predicate true. By statistics, determine the fraction (FF(pred)) of rows retrieved.

E.g., hobby column has 100 values. We generally assume uniform distribution, and get: $FF(\text{hobby} = \text{const}) = 1/100 = .01$. (In modern systems the query plan is based on a histogram, but we assume equal probability.)

And zipcode column has 100,000 values, $FF(\text{zipcode} = \text{const}) = 1/100,000$.
 $FF(\text{zipcode between } 02159 \text{ and } 03158) = 1000 \cdot (1/100,000) = 1/100$.

DB2 statistics.

See Figure 9.13, pg. 558. After use RUNSTATS, these statistics are up to date. (Next pg. of these notes) Other statistics as well, not covered.

DON'T WRITE THIS ON BOARD -- SEE IN BOOK

Catalog Name	Statistic Name	Default Value	Description
SYSTABLES	CARD	10,000	Number of rows in the table
	NPAGES	$CEIL(1+CARD/20)$	Number of data pages containing rows
SYSCOLUMNS	COLCARD	25	Number of distinct values in this column
	HIGH2KEY	n/a	Second highest value in this column
	LOW2KEY	n/a	Second lowest value in this column
SYSINDEXES	NLEVELS	0	Number of Levels of the Index B-tree
	NLEAF	$CARD/300$	Number of leaf pages in the Index B-tree
	FIRSTKEYCARD	25	Number of distinct values in the first column, C1, of this key
	FULLKEYCARD	25	Number of distinct values in the full key, all components: e.g. C1.C2.C3
	CLUSTER-RATIO	0% if CLUSTERED = 'N' 95% if CLUSTERED = 'Y'	Percentage of rows of the table that are clustered by these index values

Figure 9.13. Some Statistics gathered by RUNSTATS used for access plan determination

Statistics gathered into DB2 Catalog Tables named. Assume that index might be composite, (C1, C2, C3)

Go over table. CARD, NPAGES for table. For column, COLCARD, HIGH2KEY, LOW2KEY. For Indexes, NLEVELS, NLEAF, FIRSTKEYCARD, FULLKEYCARD, CLUSTERRATIO. E.g., from Figure 9.12, statistics for prospects table (given on pp. 552-3). Write these on Board.

SYSTABLES

NAME	CARD	NPAGES
.
prospects	50,000,000	5,000,000
.

SYSCOLUMNS

NAME	TBNAME	COLCARD	HIGH2KEY	LOW2KEY
.
hobby	prospects	100	Wines	Bicycling
zipcode	prospects	100000	99998	00001
.

SYSINDEXES

NAME	TBNAME	NLEVELS	NLEAF	FIRSTKEY CARD	FULLKEY CARD	CLUSTER RATIO
.
addrx	prospects	4	500,000	100,000	50,000,000	100
hobbyx	prospects	3	50,000	100	100	0
.

CLUSTERRATIO is a measure of how well the clustering property holds for an index. *With 80 or more*, will use Sequential Prefetch in retrieving rows.

Indexable Predicates in DB2 and their Filter Factors

Look at Figure 9.14, pg. 560. QOPT guesses at Filter Factor. Product rule assumes independent distributions of columns. Still no subquery predicate.

Predicate Type	Filter Factor	Notes
Col = const	1/COLCARD	"Col <> const" same as "not (Col = const)"
Col \propto const	Interpolation formula	" \propto " is any comparison predicate other than equality; an example follows
Col < const or Col <= const	$\frac{(\text{const} - \text{LOW2KEY})}{(\text{HIGH2KEY} - \text{LOW2KEY})}$	LOW2KEY and HIGH2KEY are estimates for extreme points of the range of Col values
Col between const1 and const2	$\frac{(\text{const2} - \text{const1})}{(\text{HIGH2KEY} - \text{LOW2KEY})}$	"Col not between const1 and const2" same as "not (Col between const1 and const2)"
Col in list	(list size)/COLCARD	"Col not in list" same as "not (Col in list)"
Col is null	1/COLCARD	"Col is not null" same as "not(Col is null)"
Col like 'pattern'	Interpolation Formula	Based on the alphabet
Pred1 and Pred2	FF(Pred1)·FF(Pred2)	As in probability
Pred1 or Pred2	FF(Pred1)+FF(Pred2) -FF(Pred1)·FF(Pred2)	As in probability
not Pred1	1 - FF(Pred1)	As in probability

Figure 9.20. Filter Factor formulas for various predicate types