Lecture Notes
Database Management
Patrick E. O'Neil

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, considering only queries, not updates.

Section 9.1. 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 weighted sum of these for the total cost, COST(PLAN):

$$COST(PLAN) = W_1 \cdot COST_{CPU}(PLAN) + W_2 \cdot COST_{I/O}(PLAN)$$

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.

Figure out various queries and rate of submission 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 $COST_{CPU}(Q_k)$ 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 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 some commercial database systems have no parallelism: all I/Os in sequence.

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 are trying hard to reduce response times.

Usually there exist parallel versions of these systems as well; worth it if extremely heavy I/O and response time is a problem, while 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 command. Fig. 9.4, pg. 538.

```
ANALYZE {INDEX | TABLE | CLUSTER}
    [schema.] {indexname | tablename | clustername}
   {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 a bit later.

Retrieving the Query Plans. In DB2 and ORACLE, perform SQL statement. In DB2:

```
  EXPLAIN PLAN [SET QUERYNO = n] [SET QUERYTAG = 'string'] FOR
       explainable-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" to represent individual procedural steps in a query plan. Can get rows back by:

```
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 we had in the first edition.

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 ORACLE8 documentation named in text.

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 not bitmap indexing or hashing capability.

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

Basic procedrual steps covered in the next few Sections (thumbnail):

Table Scan	Look through all rows of table
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 the steps we have listed access all the rows restricted by the WHERE clause in some single table query.

Need two tables in the FROM clause to require two steps of this kind and Joins come later. A multi-step plan for a single table query will also be covered later.

Such a multi-step plan on a single table is one that combines multiple indexes to retrieve data. Up to then, 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).

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, each row of 200 bytes, each 4 KByte page just 70% full. Thus 2800 usable pages, 14 rows/pg. Need CEIL(200,000/14) = 14,286 pages.

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 this query occurs with any frequency at all!)

In Table Space Scan, might not stop when find proper row, since statistics for a non-indexed column might not know socsecno is unique.

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

But if we assume random I/O, 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..

Assumptions about I/O

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 the first edition came out, 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 (expecially if there's ony one user process running), it doesn't really save any RESOURCE COST.

If it takes 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 there are few queries running, fewer than the number of disks, and there is extra CPU not utilized. Then can use more of the disk arms and CPUs with this sort of parallelism.

Parallelism shows up best when there is only one query running!

But 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.

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 would take about .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 it's 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 disk pages in sequence compared to randomly place I/O. (Accurate enough for rule of thumb.)

PUT 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 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)

IBM puts a lot of effort into making I/O requests sequentially in a query plan to gain this 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 table.

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 when really Random I/O.

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 14.

Exam 1.

Class 15.

9.3 Simple Indexed Access in DB2.

Index helps efficiency of query plan. There is a gread 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: ACCESSTYPE = 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. For each such entry, retrieve row pointed to. No assumption about clustering or non-clustering of rows here.

In Example 9.3.2, assume other restrictions in WHERE clause, but matching index scan used on C1X. Then other restrictions are validated as rows are accessed (row is ***qualified:*** look at row, check if matches restrictions).

Not all predicates are *indexable*. 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 <> 10 to be non-indexable. It is not impossible that an index will be used in a query with this predicate:

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

But the statistics usually weigh against it's use and so the query will be performed by a table space scan. More on indexable predicates later.

OK, now what about 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, will be like previous example with retrieved rows restricted by tests on other two predicates.

If have index combinx, created by:

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

Will be able to limit (filter) RIDs of rows to retrieve much more completely before going to data. Like books in a card catalog, looking up

```
authorlname = 'James' (c1 = 10) and authorfname between 'H' and 'K'
   and title begins with letter 'A'
```

Finally, we will cover the question of how to filter the RIDs of rows to retrieve if we have three indexes, C1X, C2X, and C3X. This is not simple.

See how to do this by taking out cards for each index, ordering by RID, then merge-intersecting.

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 CEIL(200,000/14) =14,286 data pages. Assume in index on eid, also have pctfree = 30, and eid||RID takes up 10 bytes, so 280 entries per pg, and CEIL(200,000/280) = 715 leaf level pages. Next level up CEIL(715/280) = 3. Root next level up. Write on board:

employees table: 14,286 data pages
index on eid, eidx: 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 about once every 120 seconds (popular pages) should stay in memory. Assume we have done this. If workload assumes 1 query per second of this on ename with eid = predicate (no others on this table), then leaf nodes and data pages not buffered, but upper nodes of eidx are. So really 2R is cost of query.

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

Class 16.

OK now we introduce a new table called prospects. Based on direct mail applications (junk mail). People fill out warrenty 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 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.

(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 255 RIDS (more?), so can fit 984 RIDs (or more) 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.

index on eid, eidx: 715 leaf nodes, 3 level 2 nodes, 1 root node.

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 assumes each of 100 hobbies equally likely (knows there are 100 from RUNSTATS), so restriction cuts 50M rows down to 500,000.

Walk down hobby index (2R for directory nodes) and across 500,000 entries (1000 per page so 500 leaf pages, sequential prefeth 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 = 625 seconds. (Yes, CPU is still ignored — in fact is relatively insignificant.)

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

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 Match. Consider the following query:

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

Recall CARD(zipcode) = 100,000. Range of zipcodes is 1000. Therefore, cut 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 55,000/500 = 137.5 seconds, a bit over 2 minutes, 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. Have to look at ACCESSNAME = addrx and note that this index is clustered, (clusterratio) whereas ACCESSNAME = hobbyx is not.

(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**. 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 components. Out of luck.

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

Indexes cost something. Disk media cost (not commonly crucial). With inserts or updates of indexed rows, lot of extra I/O (not common).

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. Generally assume uniform distribution, and get: FF(hobby = const) = 1/100 = .01.

And zipcode column has 100,000 values, FF(zipcode = const) = 1/100,000. FF(zipcode between 02159 and 03158) = 1000·(1/100,000) = 1/100.

How does the DB2 query optimizer make these estimates?

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
	FIRSTKEY-CARD	25	Number of distinct values in the first column, C1, of this key
	FULLKEY-CARD	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 =	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 ∝ const	Interpolation formula	"∝" is any comparison predicate other than equality; an example follows
Col < const or Col <= const	$\frac{(const - LOW2KEY)}{(HIGH2KEY-LOW2KEY)}$	LOW2KEY and HIGH2KEY are estimates for extreme points of the range of Col values
Col between const1 and const2	$\frac{(const2 - const1)}{(HIGH2KEY-LOW2KEY)}$	"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 Class 17.

Matching Index Scans with Composite Indexes

(Finish -> 9.6 Class 19, homework due Class 20 (Wed, April 12)

Assume new index mailx:

create index mailx on prospects (zipcode, hobby, incomeclass, age);

NOT clustered. column incomeclass has 10 distinct values, age has 50.

FULLKEYCARD(mailx) could be as much as

CARD(zipcode)·CARD(hobby)·CARD(incomeclass)·CARD(age) =
100,000·100·10·50 = 1,000,000,000.

Can't be that much, only 50,000,000 rows, so assume FULLKEYCARD is 50,000,000, with no duplicate rows. (Actually, 50M darts in 5G slots. About 1/100 of slots hit, so only about 1% duplicate keyvalues.)

Entries for mailx have length: 4 (integer zipcode) + 8 (hobby) + 2 (incomeclass) + 2 (age) + 4 (RID) = 20 bytes. So 200 entries per page.

NLEAF = 50,000,000/100 = 500,000 pages. Next level up has 5,000 nodes, next level 50, next is root, so NLEVELS = 4.

SYSINDEXES

NAME	TBNAME	NLEVELS	NLEAF	FIRSTKEY CARD	FULLKEY CARD	CLUSTER RATIO
. . .	. . .	. . .	. . .	. . .	. . .	. . .
mailx	prospects	4	250,000	100,000	50,000,000	0
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Example 9.5.1. Concatenated Index, Matching Index Scan.

select name straddr from prospects where
zipcode = 02159 and hobby = 'chess' and incomeclass = 10;

Matching Index Scan here means that the three predicates in the WHERE clause match the INITIAL column in concatenated mailx index.

Argue that *matching* means all entries to be retrieved are contiguous in the index.

Full filter factor for three predicates given is 1/100,000 ·1/100·1/10 = 1/100M, with 50M rows, so only 0.5 rows selected. 0.5R

Interpret this probabilistically, and expected time for retrieving rows is only 1/80 second. Have to add index I/O of course. 2R, .05 sec.

Example 9.5.2. Concatenated Index, Matching index scan.

```
select name straddr from prospects
   where  zipcode between 02159 and 04158
   and hobby = 'chess' and incomeclass = 10;
```

Now, important. Not one contiguous interval in index. There is one interval for: z = 02159 and h = 'c' and inc = 10, and then another for z = 02160 and h = 'c' and inc = 10, and . . . But there is stuff between them.

Analogy in telephone directory: last name between 'Sma' and 'Smz' and first name 'John'. Lot of directory to look through, not all matches.

Query optimizer here traverses from leftmost z = 02159 to rightmost z = 04158 and uses h = 'c' and inc = 10 as screening predicates.

We say the first predicate is a MATCHING predicate (used for cutting down interval of index considered) and other two are SCREENING predicates.

(This MATCHING predicate is what we mean by Matching Index Scan.)

So index traversed is: (2000/100,000) (filter factor) of 500,000 leaf pages, = 10,000 leaf pages. Query optimizer actually calculates FF as

(04158-02159)(HIGH2KEY-LOW2KEY) = 2000/(99998-00001)
= 200/99997 or approximately 2000/100,000 = 1/50

Have to look through 1/50·NLEAF = 5000 pages, I/O cost is 5,000S with elapsed time: 5,000/400 = 12.5 seconds.

How many rows retrieved? (1/50)(1/100)(1/10) = (1/50,000) with 50M rows, so 1000 rows retrieved. Sequential, class?

No. 1000R, with elapsed time 1000/40 = 25 seconds. Total elapsed time is 37.5 secs.

Example 9.5.3. Concatenated Index, Non-Matching Index Scan.

```
select name straddr from prospects where
   hobby = 'chess' and incomeclass = 10 and age = 40;
```

Like saying First name = 'John' and City = 'Waltham' and street = 'Main'. Have to look through whole index, no matching column, only screening predicates.

Still get small number of rows back, but have to look through whole index. 250,000S. Elapsed time 250,000/400 = 625 seconds, about 10.5 minutes.

Number of rows retrieved: (1/100)(1/10)(1/50)(50,000,000) = 1000. 1000R = 25 seconds.

In PLAN TABLE, for Example 9.5.2, have ACCESSTYPE = I, ACCESSNAME = mailx, MATCHCOLS = 1; In Example 9.5.3, have MATCHCOLS = 0.

Definition 9.5.1. Matching Index Scan. A plan to execute a query where at least one indexable predicate must match the first column of an index (known as matching predicate, matching index). May be more.

What is an indexable predicate? Equal match predicate is one: Col = const See Definition 9.5.3. Pg. 565

Say have index C1234X on table T, composite index on columns (C1, C2, C3, C4). Consider following compound predicates.

C1 = 10 and C2 = 5 and C3 = 20 and C4 = 25 (matches all columns)

C2 = 5 and C3 = 20 and C1 = 10 (matches first three: needn't be in order)

C2 = 5 and C4 = 22 and C1 = 10 and C6 = 35 (matches first two)

C2 = 5 and C3 = 20 and C4 = 25 (NOT a matching index scan)

Screening predicates are ones that match non-leading columns in index. E.g., in first example all are matching, in second all are matching, in third, two are matching, one is screening, and one is not in index, in fourth all three are screening.

Finish through Section 9.6 by next class. Homework due next class (Wednesday after Patriot's day). NEXT homework is rest of Chapter 9 non-dotted exercises if you want to work ahead.

Definition 9.5.2. Basic Rules of Matching Predicates

(1) A *matching predicate* must be an *indexable* predicate. See pg. 560, Table 9.14 for a list of indexable predicates.

(2) Matching predicates must match successive columns, C1, C2, . . . of an index. Procedure: Look at index columns from left-to right. If find a matching predicate for this column, then this is a matching column. As soon as column fails to be matching terminate the search.

Idea is that sequence of matching predicates cuts down index search to smaller contiguous range. (One exception: In-list predicate, covered shortly).

(3) A non-matching predicate in an index scan can still be a screening predicate.

Look at rule (1) again. This is actually a kind of circular definition. For a predicate to be *matching* it must be *indexable* and:

Definition 9.5.3: An *indexable* predicate is one that can be used to match a column in a matching index scan.

Calling such a predicate *indexable* is confusing. Even if a predicate is not indexable, the predicate can use the index for screening.

Would be much better to call such predicates *matchable*, but this nomenclature is embedded in the field for now.

When K leading columns of index C1234X are matching for a query, EXPLAIN into plan table, get ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = K. When non-matching index scan, MATCHCOLS = 0.

Recall Indexable Predicates in Figure 9.14, pg. 560, and relate to telephone directory. Does the predicate give you a contiguous range?

Col > const? between? In-list is special. like 'pattern' with no leading wild card? Col1 like Col2? (same middle name as street name) Predicate and? Predicate or? Predicate not?

OK, a few more rules on how to determine matching predicates. Page 566, **Def 9.5.4**. Match cols. in index left to right until run out of predicates. But
(3) Stop at first range predicate (between, <, >, <=, >=, like).
(4) At most one In-list predicate.

In-list is special because it is considered a sequence of equal matching predicates that the query optimizer agrees to bridge in the access plan

C1 in (6, 8, 10) and C2 = 5 and C3 = 20 is like C1 = 6 and . . .;

Plan for C1 = 8 and . . .; then C1 = 10 and . . .; etc.

But the following has only two matching columns since only one in-list can be used.

C1 in (6, 8, 10) and C2 = 5 and C3 in (20, 30, 40)

When In-list is used, say ACCESSTYPE = 'N'.

Example 9.5.4. In following examples, have indexes C1234X, C56X, Unique index C7X.

(1) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C3 <> 9;
ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 2

(2) select C1, C5, C8 from T where C1 = 5 and C2 >= 7 and C3 = 9;
C3 predicate is indexable but stop at range predicate.
ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 2

(3) select C1, C5, C8 from T
where C1 = 5 and C2 = 7 and C5 = 8 and C6 = 13;
We ignore for now the possibility of combining multiple indexes
Note, we don't know what QOPT will choose until we see plan table row
ACCESSTYPE = I, ACCESSNAME = C56X, MATCHCOLS = 2

(4) select C1, C4 from T
where C1 = 10 and C2 in (5, 6) and (C3 = 10 or C4 = 11);
C1 and C2 predicates are matching. The "or" operator doesn't give indexable predicate, but would be used as screening predicate (not mentioned in plan table, but all predicates used to filter, ones that exist in index certainly would be used when possible). ACCESSTYPE = 'N',

ACCESSNAME = C1234X, MATCHCOLS = 2 Also: INDEXONLY = 'Y'

(5) select C1, C5, C8 from T where C1 = 5 and C2 = 7 and C7 = 101;
ACCESSTYPE = I, ACCESSNAME = C7X, MATCHCOLS = 1
(Because unique match, but nothing said about this in plan table)

(6) select C1, C5, C8 from T
where C2 = 7 and C3 = 10 and C4 = 12 and C5 = 16;
Will see can't be multiple index. Either non-matching in C1234X or matching on C56X. ACCESSTYPE = I, ACCESSNAME = C1234X, MATCHCOLS = 2

Some Special Predicates

Pattern Match Search. Leading wildcards not indexable.

"C1 like 'pattern'" with a leading '%' in pattern (or leading '_'?) like looking in dictionary for all word ending in 'tion'. Non-matching scan (non-indexable predicate).

Exist dictionaries that index by backward spelling, and DBA can use this trick: look for word with match: Backwards = 'noit%'

Expressions. Use in predicate makes not indexable.

```
select * from T where 2*C1 <= 56;
```

DB2 doesn't do algebra. You can re-write: where C1 <= 28;

Never indexable if two different columns are used in predicate: C1 = C2.

One-Fetch Access. Select min/max . . .

```
select min(C1) from T;
```

Look at index C1234X, leftmost value, read off value of C1. Say have index C12D3X on T (C1, C2 DESC, C3). Each of following qs has one-fetch access.

```
select min(C1) from T where C1 > 5; (NOT obviously 5)
select min(C1) from T where C1 between 5 and 6;
select max(C2) from T where C1 = 5;
select max(C2) from T where C1 = 5 and C2 < 30;
```

```
select min(C3) from T where C1 = 6 and C2 = 20 and C3 between 6 and 9;
```

Class 18.

9.6 Multiple Index Access

Assume index C1X on (C1), C2X on (C2), C345X on (C3, C4, C5), query:

(9.6.1) select * from T where C1 = 20 and C2 = 5 and C3 = 11;

By what we've seen up to now, would have to choose one of these indexes. Then only one of these predicates could be matched.

Other two predicates are not even screening predicates. Don't appear in index, so must retrieve rows and validate predicates from that.

BUT if only use FF of one predicate, terrible inefficiency may occur. Say T has 100,000,000 rows, FF for each of these predicates is 1/100.

Then after applying one predicate, get 1,000,000 rows retrieved. If none of indexes are clustered, will take elapsed time of: 1,000,000/40 = 25,000 seconds, or nearly seven hours.

If somehow we could combine the filter factors, get composite filter factor of (1/100)(1/100)(1/100) = 1/1,000,000, only retrieve 100 rows.

Trick. Multiple index access. For each predicate, matching on different index, extract RID list. Sort it by RID value.

(Draw a picture - three extracts from leaf of wedge into lists)

Intersect (AND) all RID lists (Picture) (easy in sorted order). Result is list of RIDs for answer rows. Use List prefetch to read in pages. (Picture)

This is our first multiple step access plan.
See Figure 9.15. Put on board.

V Note

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C1X	S	1
T	MX	1	C2X	S	2
T	MX	1	C345X	S	3
T	MI	0			4
T	MI	0			5

Figure 9.15 Plan table rows of a Multiple Index Access plan for Query (9.6.1)

Row M, start Multiple index access. Happens at the end, after Intersect Diagram steps in picture. RID lists placed in RID Pool in memory.

Note Plan acts like reverse polish calculation: push RID lists as created by MX step; with MI, pop two and intersect , push result back on stack.

MX steps require reads from index. MI steps require no I/O: already in memory, memory area known as RID Pool.

Final row access uses List prefetch, program disk arms for most efficient path to bring in 32 page blocks that are not contiguous, but sequentially listed. Most efficient disk arm movements.

(Remember that an RID consists of (page_number, slot_number), so if RIDs are placed in ascending order, can just read off successive page numbers).

Speed of I/O depends on distance apart. Rule of thumb for Random I/O is 40/sec, Seq. prefetch 400/sec, List prefetch is 100/sec.

Have mentioned ACCESSTYPEs = M, MX, MI. One other type for multiple index access, known as MU (to take the Union (OR) of two RID lists). E.g.

(9.6.2) select * from T where C1 = 20 and (C2 = 5 or C3 = 11);

Here is Query Plan (Figure 9.16):

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	C1X	S	1
T	MX	1	C2X	S	2
T	MX	1	C345X	S	3
T	MU	0			4
T	MI	0			5

Figure 9.16 Plan table rows of a Multiple Index Access plan for Query (9.6.2)

Actually, query optimizer wouldn't generate three lists in a row if avoidable. Tries not to have > 2 in existence (not always possible).

Figure 9.17. 1. MX C2X, 2. MX C345X, 3. MU, 4. MX C1X, 5. MI

Example 9.6.1. **Multiple index access**. prospects table, addrx index, hobbyx index (see Figure 9.11, p 544), index on age (agex) and incomeclass (incomex). What is NLEAF for these two, class? (Like hobbyx: 50,000.)

```
select name, straddr from prospects
   where zipcode = 02159 and hobby = 'chess' and incomeclass = 10;
```

Same query as Example 9.5.1 when had zhobincage index, tiny cost, 2R index, only .5R for row. But now have three different indexes: PLAN.

TNAME	ACCESSTYPE	MATCHCOLS	ACCESSNAME	PREFETCH	MIXOPSEQ
T	M	0		L	0
T	MX	1	hobbyx	S	1
T	MX	1	addrx	S	2
T	MI	0			3
T	MX	1	incomex	S	4
T	MI	0			5

Figure 9.18 Plan table rows of Multiple Index Access plan for Example 9.6.1

Calculate I/O cost. FF(hobby = 'chess') = 1/100, on hobbyx (NLEAF = 50,000), 500S (and ignore directory walk).

For MIXOPSEQ = 2, FF(zipcode = 02159) = 1/100,000 on addrx (NLEAF = 500,000), so 5S.

Intersect steps MI requires no I/O.

FF(incomeclass = 10) = 1/10, on incomex (NLEAF = 50,000), so 5,000S.

Still end up with 0.5 rows at the end by taking product of FFs for three predicates: (1/100,000)(1/100)(1/10) = 1/100,000,000

Only 50,000,0000 rows; ignore list prefetch of .5L.

So 5,505, elapsed time 5,505/400 = 13.8 seconds.

List Prefetch and RID Pool

All the things we've been mentioning up to now have been relatively universal, but RID list rules are quite product specific. Probably won't see all this duplicated in ORACLE.

-> We need RID extraction to access the rows with List Prefetch, but since this is faster than Random I/O couldn't we always use RID lists, even in single index scan to get at rows with List Prefetch in the end?

YES, BUT: There are restrictive rules that make it difficult. Most of these rules arise because RID list space is a scarce resource in memory.

-> The RID Pool is a separate memory storage area that is 50% the size of all four Disk Buffer Pools, except cannot exceed 200 MBytes.

Thus if DBA sets aside 2000 Disk Buffer Pages, will have 1000 pages (about 4 MBytes) of RID Pool. (Different memory area, though.)

Def. 9.6.1. Rules for RID List Use.

(1) QOPT, when it constructs query plan, predicts size of RIDs active at any time. Cannot use > 50% of total capacity. If guess wrong, abort during runtime.

(2) No Screening predicates can be used in an Index scan that extracts a RID List.

(3) An In-list predicate cannot be used when extract a RID list.

Example 9.6.3. RID List Size Limit. addrx, hobbyx, incomex, add sexx on column sex. Two values, 'M' and 'F', uniformly distributed.

```
select name, straddr from prospects where zipcode between
  02159 and 02358 and incomeclass = 10 and sex = 'F';
```

Try multiple index access. Extract RID lists for each predicate and intersect the lists.

But consider, sexx has 50,000 leaf pages. (Same as hobbyx, from 50,000,000 RID size entries.) Therefore sex = 'F' extracts 25,000 page RID list. 4 Kbytes each page, nearly 100 MBytes.

Can only use half of RID Pool, so need 200 MByte RID Pool.

Not unreasonable. A proof that we should use large buffers.

Example 9.6.5. No Index Screening. Recall zhobincage index in Example 9.3.8.

```
select name, straddr from prospects where zipcode between 02159
  and 04158 and hobby = 'chess' and incomeclass = 10;
```

Two latter predicates were used as screening predicates. But screening predicates can't be used in List Prefetch. To do List Prefetch, must resolve two latter predicates after bring in rows.

The 'zipcode between' predicate has a filter factor of 2000/100,000 = 1/50, so with 50 M rows, get 1M rows, RID list size of 4 MBytes, 1000 buffer pages, at least 2000 in pool. Assume RID pool is large enough.

So can do List Prefetch, 1,000,000/100 = 10,000 seconds, nearly 3 hours.

If use screening predicates, compound FF is (1/50)(1/100)(1/10) = 1/50,000. End with 1000 rows. Can't use List Prefetch, so 1000R but 1000/40 is only 25 seconds. Same index I/O cost in both cases. Clearly better not to use RID list.

What is reason not to allow List Prefetch with Screening Predicates? Just because resource is scarce, don't want to tie up RID list space for a long time while add new RIDs slowly. If refuse screening, QOPT might use other plans without RIDs.

Point of Diminishing Returns in MX Access.

Want to use RID lists in MIX. >> Talk it

Def. 9.6.2. (1) In what follows, assume have multiple indexes, each with a disjoint set of matching predicates from a query. Want to use RID lists.

(2) Way QOPT sees it: List RID lists from smallest size up (by increasing Filter Factor). Thus start with smaller index I/O costs (matching only) and larger effect in saving data page I/O.

(3) Generate successive RID lists and calculate costs; stop when cost of generating new RID list doesn't save enough in eventual data page reads by reducing RID set.

Example 9.6.6.

```
select name straddr from prospects
   where prospects between o2159 and 02659
   and age = 40 and hobby = 'chess' and incomeclass = 10;
```

(1) FF(zipcode between 02159 and 02658) = 500/100,000 = 1/200.
(2) FF(hobby = 'chess') = 1/100
(3) FF(age = 40) = 1/50 (Typo in text)
(4) FF(incomeclass = 10) = 1/10

Apply predicate 1. (1/200) (50M rows) retrieved, 250,000 rows, all on separate pages from 5M, and 250,000L is 2500 seconds. Ignore the index cost.

Apply predicate 2 after 1. Leaf pages of hobbyx scanned are (1/100) (50,000) = 500S, taking 500/400 = 1.25 seconds. Reduce number of rows retrieved from 250,000 to about 2500, all on separate pages, and 2500L

takes about 25 seconds. So at cost of 1.25 seconds of index I/O, reduce table page I/O from 2500 seconds to 25 seconds. Clearly worth it.

Apply predicate 3 after 1 and 2. Leaf pages of agex scanned are (1/50) (50,000) = 1000S, taking 1000/400 = 2.5 seconds. Rows retrieved down to (1/50) (2500) = 50, 50L is .5 seconds. Thus at cost of 2.5 seconds of index I/O reduce table page I/O from 25 seconds to 0.5 seconds. Worth it.

With predicate 4, need index I/O at leaf level of (1/10) (50,000) = 5000S or 12.5 seconds. Not enough table page I/O left (0.5 seconds) to pay for it. Don't use predicate 4. We have reached point of diminishing returns.