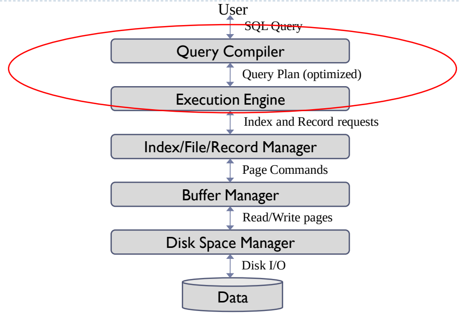
Query Evaluation Overview, cont.

Lecture 9 Feb. 29, 2016

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Architecture of a DBMS



A first course in database systems, 3rd ed, Ullman and Widom

The two major parts of the DB engine

- QP = query processor, top two boxes on last slide
- Storage manager = rest of boxes
- See "index and record requests" flowing between
- Can be more specific, see list, pg. 283:
- Actions on "files": file scan, search with equality selection, search with range selection, insert record, delete record
- Files listed: heap files, sorted files, clustered files, heap file with unclustered tree index, heap file with unclustered hash index. An index on its own is a sorted file.
- A file is something that the storage engine can process via an ISAM-like API
- A table can be accessed as a file: pick an index for it (or not)

Storage Engine API

- If a QP and storage engine hue to an API, then different storage engines can be "plugged in" to the database
- Example: MS SQL Server can access Excel files via the OLE-DB API. Also via ODBC.
  - That is, there is an Excel OLE-DB "provider" (you don't need the whole Excel GUI).
- Example: MySQL has various storage engines—MyISAM and Innodb, etc.
  - New one (Nov '12): ClouSE uses Amazon S3 cloud storage.

MySQL Storage Engine API

Top-level API (subset) from internals manual

Note handoff to TABLE object for data actions:

```
int (*commit)(THD *thd, bool all);
int (*rollback)(THD *thd, bool all);
int (*prepare)(THD *thd, bool all);
int (*recover)(XID *xid_list, uint len);
handler *(*create)(TABLE *table);   ←next slide
void (*drop_database)(char* path);
bool (*flush_logs)();
```

MySQL Storage Engine API: TABLE API

- 22.18.1 bas_ext
- 22.18.2 close
- 22.18.3 create
- 22.18.4 delete_row
- 22.18.5 delete_table
- 22.18.6 external_lock
- 22.18.7 extra
- 22.18.8 index_end
- 22.18.9 index_first
- 22.18.10 index_init Set current index
- 22.18.11 index_last
- 22.18.12 index_next
- 22.18.13 index_prev

(22.18.11–22.18.13: Index scan)

Scan: iteration over rows, see "next' methods

- 22.18.14 index_read
- 22.18.15 index_read_idx
- 22.18.16 index_read_last
- 22.18.17 info
- 22.18.18 open
- 22.18.19 position
- 22.18.20 records_in_range
- 22.18.21 rnd_init
- 22.18.22 rnd_next
- 22.18.23 rnd_pos
- 22.18.24 start_stmt
- 22.18.25 store_lock
- 22.18.26 update_row
- 22.18.27 write_row Insert row

(22.18.21–22.18.23: Table scan)

Access Paths

- An access path is a method of retrieving tuples:
  - File scan (AKA table scan if on a table)
  - Index scan using an index that matches a condition
- A tree index matches (a conjunction of) terms that involve every attribute in a prefix of the search key
  - E.g., tree index on <a, b, c> matches the selection a=5 AND b=3, and a=5 AND b>6, but not b=3
- A hash index matches (a conjunction of) terms attribute = value for every attribute in the search key of the index
  - E.g., hash index on <a, b, c> matches a=5 AND b=3 AND c=5
  - but it does not match b=3, or a=5 AND b=3

Example of matching indexes

Pg. 399: fix error Sailors→ Reserves on line 8

Reserves (sid: integer, bid: integer, day: dates, *rname*: string) ← rname column added here

with indexes:

- Index1: Hash index on (rname, bid, sid)
  - Matches: rname='Joe' and bid = 5 and sid=3
  - Doesn't match: rname='Joe' and bid = 5
- Index2: Tree index on (rname, bid, sid)
  - Matches: rname='Joe' and bid = 5 and sid=3
  - Matches: rname='Joe' and bid = 5, also rname = 'Joe'
  - Doesn't match: bid = 5
- Index3: Tree index on (rname)
- Index4: Hash index on (rname)
  - These two match any conjunct with rname='Joe' in it

Executing Selections

- Find the *most selective access path*, retrieve tuples using it
  - Then, apply any remaining terms that don't match the index
- *Most selective access path:* index or file scan estimated to require the fewest page I/Os
  - Consider *day<8/9/94 AND bid=5 AND sid=3*
- If we have B+ tree index on *day*, use that access path
  - Then, *bid=5* and *sid=3* must be checked for each retrieved tuple
  - *day* condition is primary conjunct
- Alternatively, use hash index on *<bid, sid>* first
  - Then, *day<8/9/94* must then be checked

Example Schema

Sailors (sid: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (sid: integer, bid: integer, day: dates, *rname*: string)

- Similar to old schema; *rname* added
- Reserves:
  - 40 bytes long tuple, 100K records, 4KB pages
  - So 100K*40 = 4MB data, 4MB/4KB = 1000 pages
  - Assume 4000 bytes/pg, so100 tuples per page
- Sailors:
  - 50 bytes long tuple, 40K tuples, 4KB pages
  - So 80 tuples per page, 500 pages

Using an Index for Selections

- Cost influenced by:
  - Number of qualifying tuples
  - Whether the index is clustered or not
  - Ex:

```
SELECT  *
FROM    Reserves R
WHERE   R.rname < 'C%'
```

- Assuming uniform distribution of names, 2/26 ~10% of tuples qualify, that is 10000 tuples (pg. 401)
  - With a clustered index, cost is little more 100 I/Os:
    - 10000*40 = 400KB data, in 100 data pages, plus a few index pgs
  - If not clustered, up to10K I/Os!
    - About 10000 data pages accessed, each with own I/O (unless big enough buffer pool)
    - Better to do a table scan: 1000 pages, so 1000 I/Os.

Executing Projections

- Expensive part is removing duplicates
  - DBMS don't remove duplicates unless DISTINCT is specified

```
SELECT  DISTINCT R.sid, R.bid
FROM    Reserves R
```

- Sorting Approach
  - Sort on <sid, bid> (or <bid, sid>) and remove duplicates
  - Avoidable if an index with R.sid and R.bid in the search key exists
- Hashing Approach
  - Hash on <sid, bid> to create partitions (buckets)
  - Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates

Executing Joins: Index Nested Loops

```
foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result
```

- Cost = $(M*p_R)$ * (cost of finding matching inner-table tuples)
- M = number of pages of R, p_R = number of R tuples per page
- If relation has index on join attribute, make it inner relation
  - For each outer tuple, cost of probing inner index is 1.2 for hash index, 2-4 for B+, plus cost to retrieve matching S tuples
  - Clustered index ~~typically single I/O~~ no more I/O (unless many matching S tuples)
  - Unclustered index 1 I/O per matching S tuple

Duplicate keys in indexes

- B trees: see Sec. 10.7 Duplicates: two ways to go—
  - Overflow pages, but not "typical"
  - Just sequential entries with the same key (we'll assume this)
- Extendible Hashing: uses overflow pages (pg. 379)
- Linear Hashing: uses multiple entries in the main pages.
  - May involve "extra" overflow pages, since splitting doesn't help with a long sequence of same-key entries.
- Shouldn't use a hash index on a low-cardinality column. B-tree is OK (esp. Alt. 3). (Bitmap index is best.)
- Cost of access for all dups of one key: calculate number of pages of duplicate index entries

Example of Index Nested Loops (1/2)

Example: Reserves JOIN Sailors (natural join on sid)

Case 1: Hash-index (Alternative 2) on *sid* of Sailors

- Choose Sailors as inner relation
- Scan Reserves: 100K tuples, 1000 page I/Os
- For each Reserves tuple
  - 1.2 I/Os to get data entry in index (see pg. 412)
  - 1 I/O to get (the exactly one) matching Sailors tuple (primary key)
- Total: 221,000 I/Os

Example of Index Nested Loops (2/2)

Example: Reserves JOIN Sailors (natural join on sid)

Case 2: Hash-index (Alternative 1 or 2) on *sid* of Reserves

- Choose Reserves as inner
- Scan Sailors: 40K tuples, 500 page I/Os
- For each Sailors tuple
  - 1.2 I/Os to find index page with data entries
  - Assuming uniform distribution, 2.5 matching records per sailor
  - Cost of retrieving records is nothing further (Alt. 1, clustered) or 2.5 I/Os (Alt. 2)
- Total: 88,500 I/Os (clustered) or 148,500 I/Os (unclustered

Executing Joins: Sort-Merge

- Sort R and S on the join column
  - Then scan them to do a merge on join column
- R is scanned once
- Each S group is scanned once per matching R tuple
  - Multiple scans per group needed only if S records with same join attribute value span multiple pages
  - Multiple scans of an S group are likely to find needed pages in buffer
- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N worst case (very unlikely!)

System R Optimizer

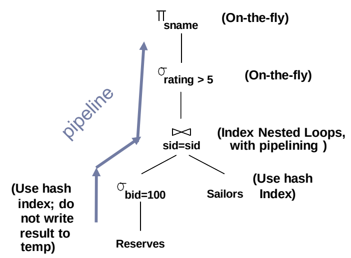
- Developed at IBM starting in the 1970's
  - Most widely used currently; works well for up to 10 joins
- Cost estimation
  - Statistics maintained in system catalogs
  - Used to estimate cost of operations and result sizes
  - Considers combination of CPU and I/O costs
- Query Plan Space
  - Only the space of left-deep plans is considered
  - Cartesian products avoided

Left Deep Trees

- Consider nested-loop joins
- Inner tables need to be materialized because they are probed repeatedly for each row of the outer table
  - Materialized means available as a table, not just a stream of rows, so can be probed by PK index.
- Left table = outer table
- Left table can be pipelined: rows used one at a time in order (i.e., doesn't need to be materialized)
- So Left-deep plans allow output of each operator to be pipelined into the next operator without storing it in a temporary relation

i.e., Left Deep trees can be "fully pipelined"

Example of join with left table pipelined and right table materialized



Cost Estimation

For each plan considered, must estimate:

- Cost of each operator in plan tree
  - Depends on input cardinalities
  - Operation and access type: sequential scan, index scan, joins
- Size of result for each operation in tree
  - Use information about the input relations
  - For selections and joins, assume independence of predicates

Size Estimation and Reduction Factors

SELECT attribute list
FROM relation list
WHERE $term_1$ AND … AND $term_k$

- Maximum number of tuples is cardinality of cross product
- Reduction factor (RF) associated with each *term* reflects its impact in reducing result size
  - Implicit assumption that *terms* are independent!
  - *col = value* has RF =1/NKeys(*I*), given index *I* on *col*
  - *col1 = col2* has RF = 1/max(NKeys(*I1*), NKeys(*I2*))
  - *col > value* has RF = (High(*I*)-value)/(High(*I*)-Low(*I*))

Example Schema

Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

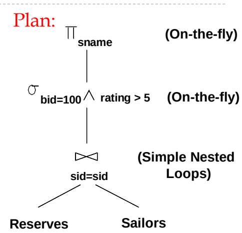
- Similar to old schema; *rname* added
- Reserves:
  - 40 bytes long tuple, 100K records, 100 tuples per page, 1000 pages
- Sailors:
  - 50 bytes long tuple, 40K tuples, 80 tuples per page, 500 pages

Evaluation Example

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5

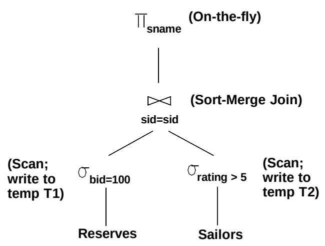
Cost: 1000+500*1000 I/Os

Plan:



- By no means the worst plan!
- Misses several opportunities:
  - Selections could have been `pushed' earlier
  - No use of any available indexes

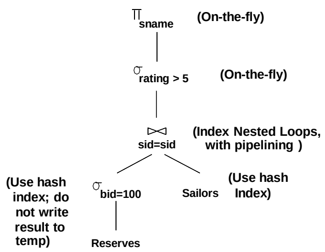
Alternative Plan 1 (No Indexes)



Alternative Plan 1 (No Indexes)

- Main difference: push down selections
- Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution)
- Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings)
- Sort-merge join T1 and T2
  - Assume there are 5 buffers:
  - Sort T1 (2*2*10), Sort T2 (2*4*250), Merge (10+250)
  - Total: 4060 page I/Os

Alternative Plan 2 (With Indexes)



Alternative Plan 2 (With Indexes)

- With clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages
- Inner Nested Loop join with **pipelining** (result not materialized)
- Join column *sid* is a key for Sailors
  - At most one matching tuple, unclustered index on *sid* OK
- Decision not to push *rating>5* before the join is based on availability of *sid* index on Sailors
- Cost:
  - Selection of Reserves tuples 10 I/Os
  - For each, must get matching Sailors tuple (1000*1.2)
  - Total 1210 I/Os

Summary

- There are several alternative evaluation algorithms for each relational operator.
- A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.
- Must understand query optimization in order to fully understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.