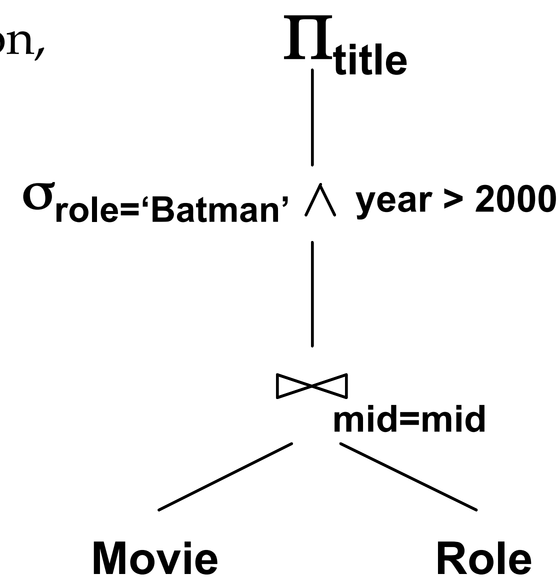
Overview of Query Evaluation

Chapter 12

Overview of Query Evaluation

- *Query:* SELECT M.title
  FROM Movie M, Role R
  WHERE M.mid=R.mid
  AND R.role='Batman' AND R.year > 2000
- *Plan:* *Tree of relational algebra ops, with an algorithm for each*
  - Each "pulls" tuples from tables via "access paths"
  - An access path might involve an index, iteration, sorting, or other approaches.
- Two main issues in query optimization:
  - For a given query, what plans are considered?
  - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find optimal plan.
- Practically: Want to avoid poor plans!



Some Common Techniques

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small subset of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

* *Watch for these techniques as we discuss query evaluation!*

Statistics and Catalogs

- Need information about the relations and indexes involved.
- ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Today's Working Example

- Consider database with the following two tables:

  Movie(*mid*: integer, *title*: string, *year*: integer, *rating*: text)
  Role(*mid*: integer, *aid*: integer, *role*: text, *billing*: integer)

- Assume each tuple of Role is 40 bytes, a page holds, at most, 100 records, each Movie tuple is 50 bytes, and a page holds no more than 80 records
- Furthermore, assume
  1000 pages of Roles (< 100,000 records), and
  500 pages of Movies (< 40, 000 records)

Example's Catalog

Attribute_Cat(*attr_name*: string, *rel_name*: string, *type*: string, *position*: integer)

- The system catalog is itself a collection of relations/tables (ex. Table attributes, table statistics, etc.)
- Catalog tables can be queried just like any other table
- Relational algebra operations can be used to examine Query evaluation tradeoffs

Attribute_Cat			
attr_name	*rel_name*	*type*	*position*
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
postion	Attribute_Cat	integer	4
mid	Movie	integer	1
title	Movie	string	2
year	Movie	integer	3
rating	Movie	integer	4
mid	Role	integer	1
aid	Role	integer	2
role	Role	string	3
billing	Role	integer	4

Access Paths

- An <u>access path</u> is a method of retrieving tuples:
  - File scan, or index search that matches the given query's selection
- A tree index <u>*matches*</u> (a conjunction of) terms that involve only attributes 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 <u>*matches*</u> (a conjunction of) terms that has a term *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, *or a*>5 *AND b*=3 *AND c*=5.

A Note on Complex Selections

(mid=130893 OR mid=320833) AND
(role='Batman' OR role='Robin')

- Convert selections to "sum of products" form

  (mid=130893 AND role='Batman') OR
  (mid=320833 AND role='Batman') OR
  (mid=130893 AND role='Robin') OR
  (mid=320833 AND role='Robin')

- "AND" terms allow us to optimally choose indices
  "OR" terms can be tested sequentially in iterations.

One Approach to Selections

- Find the *most selective access path*, retrieve tuples using it, and apply any remaining unmatched terms
  - *Most selective access path:* Either an index traversal or file scan that we *estimate* requires the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other unmatched terms are used to discard tuples, but do not affect number of tuples/pages fetched.
  - Consider *year > 2000 AND aid=70626 AND billing=1*.
    - A B+ tree index on *Movie(year)* can be used; then, *aid=70626* and *billing=1* checked after join with Role.
    - Similarly, a hash index on *Role(aid)* could be used; then filtered for *billing=1*, and joined with Movie

Which is faster?

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - For example, assuming uniform distribution of roles, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

```
SELECT  *
FROM    Roles R
WHERE   R.role < 'C%'
```

Projection

- Expensive part is eliminating duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified.

```
SELECT   DISTINCT mid, aid
FROM     Roles
```

- Sorting Approach
  - Sort on <mid, aid> and remove duplicates.
    (Can optimize by dropping unwanted attributes while sorting.)
- Hashing Approach
  - Hash on <mid, bid> during scan to create partitions.
    Tracks <mid,bid> tuples seen so far, compares to only a subset.
- With a B^+-tree indexed on <mid, aid>, you can step through the leafs compressing duplicates
- With a Hash Index on <mid, aid>, scan page for copies

Join: Index Nested Loops

foreach tuple r in R:
  foreach tuple s in S:
    if r_i ***op*** s_j add <r, s> to result

- If there is an index on the join attribute of one relation (say S), make it the *inner loop* to exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
  - M=#pages of R, p_R=# R tuples per page
- For each R tuple, cost of probing S index is ~1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O total (typical)
  - Unclustered: upto 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *mid* of Movie (as inner):
  - Scan Roles: 1000 page I/Os, 100*1000 tuples.
  - For each Roles tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (exactly one) matching Movie tuple.
  - Total: 1000 + (1+1.2)*100000 = 221,000 I/Os.
- Hash-index (Alt. 2) on *mid* of Roles (as inner):
  - Scan Movie: 500 page I/Os, 80*500 tuples.
  - For each Movie tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Roles tuples. Assuming uniform distribution, 2.5 roles per movie (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.
  - Total: 500 + (1.2 + 1)*40000 = 88,500 I/Os (clustered)
500 + (1.2 + 2.5)*40000 = 148,500 I/Os (unclustered)

Join: Sort-Merge ($R \bowtie_{i=j} S$)

- Sort R and S on the join column
- Scan them while "merging" (on join col.) and outputting resulting tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

mid	title	year	rating
3102	The Dark Knight Rises	2012	PG-13
34102	The Dark Knight	2008	PG-13
51427	Batman Begins	2005	PG-13
152371	Batman Revealed	2012	
320833	Batman Evolution	2014	

mid	aid	role	billing
3102	70626	Bruce Wayne	1
34102	70626	Bruce Wayne	1
48670	70626	Jack Kelly	1
51427	70626	Bruce Wayne	1
113303	70626	John Conner	1
142162	70626	John Preston	1
142164	70626	John Miller	1

Note importance of out-of-core external sorting (Next lecture's topic)

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100, or 300 buffer pages, both Roles and Movie can be sorted in 2 passes; total join cost: 7500.

Highlights of Query Optimization

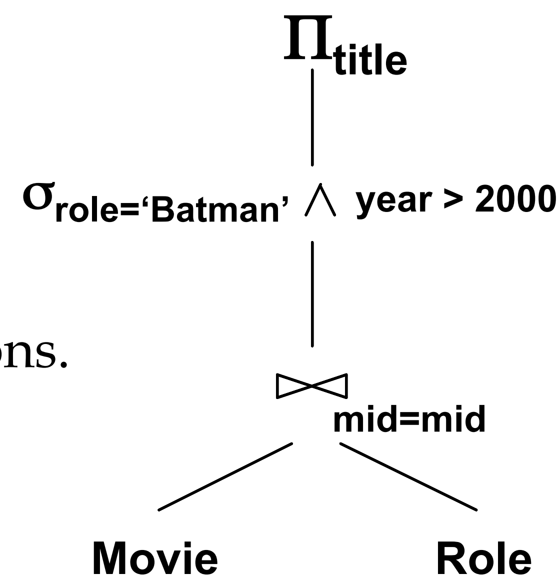
- Cost estimation: Approximate art at best.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- Plan Space: Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Actual Cartesian products avoided.

Cost Estimation

```
SELECT  M.title
FROM  Movie M, Role R
WHERE  M.mid=R.mid AND
  R.role='Batman' AND M.year>2000
```

- For each plan considered, we must estimate cost:
  - *Cost* of each operation in tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

RA Tree:



Size Estimation and Reduction Factors

- Consider a query block:

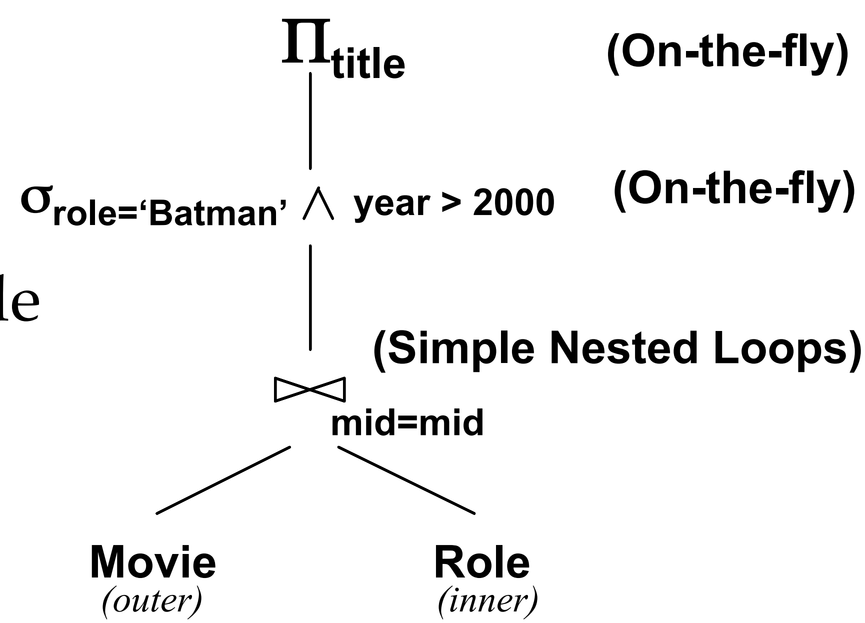
```
SELECT attribute list
FROM relation list
WHERE term1 AND ... AND termk
```

- Maximum # tuples in result is the product of the cardinalities of relations in the **FROM** clause.
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * RF_1 * RF_2 * … RF_k.
  - Implicit assumption that *terms* are independent!
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col1=col2* has RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*

Motivating Example

```
SELECT  M.title
FROM  Movie M, Role R
WHERE  M.mid=R.mid AND
  R.role='Batman' AND M.year>2000
```

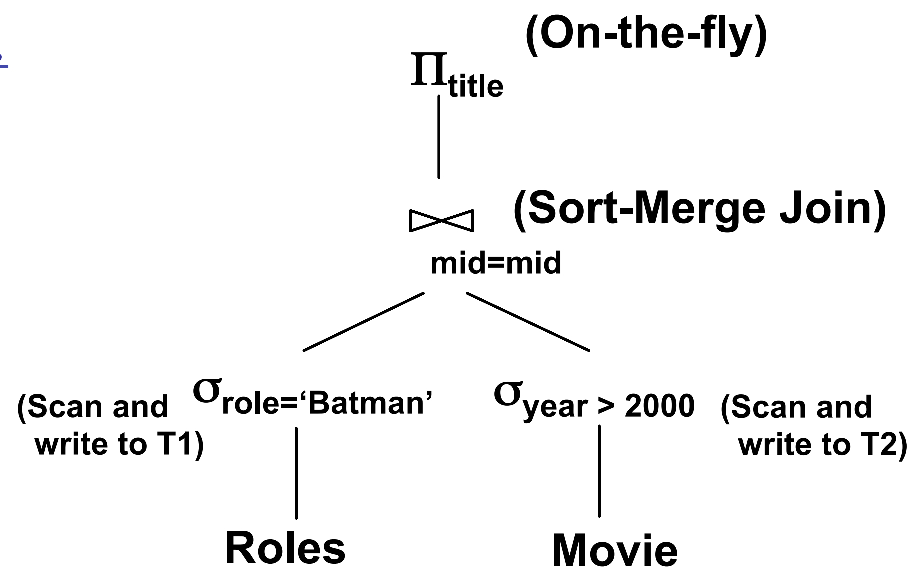
Plan:



- Cost: 500+500*1000 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been "pushed" earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* To find more efficient plans that compute the same answer.

Alternative Plan 1 (No Indexes)

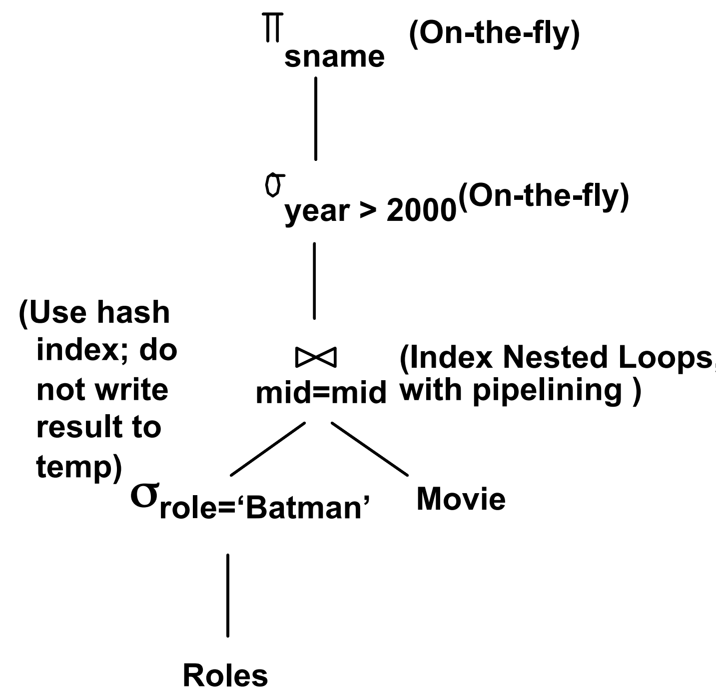
- ***Main difference:*** *Push selects.*
- With 5 buffers, cost of plan:
  - Scan Roles (1000) + write temp T1 (3 pages, we have 237 "Batman" roles).
  - Scan Movie (500) + write temp T2 (250 pages, about half of pages).
  - Sort T1 (3), sort T2 (2*4*250), merge (3+250)
  - Total: 2256 page I/Os.



- If we used BNL join, join cost = 3+4*250, total cost = 1003.
- If we `push' projections, T1 has only *mid*, T2 only *mid* and *title*:
  - T1 fits in 1 page, cost of BNL drops to under 250 pages, total < 250.

Alternative Plan 2 (With Indexes)

- With clustered index on *mid* of Roles, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- INL with ***pipelining*** (outer is not *materialized*).
    - Projecting out unnecessary fields from outer doesn't help.
- Join column *mid* is a key for Movie.
    - At most one matching tuple, unclustered index on *mid* OK.
- Decision not to push *rating*>5 before the join is based on availability of *mid* index on Movie.
- Cost: Selection of Roles tuples (10 I/Os); for each, must get matching Movie tuple (1000*1.2); total 1210 I/Os.



Practical Example

```
$ sqlite3 movies.db
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> EXPLAIN QUERY PLAN
   ...>     SELECT R.role, A.first, A.last, M.title
   ...>     FROM Role R, Actor A, Movie M
   ...>     WHERE R.aid=A.aid AND R.mid=M.mid AND R.role like "%Batman%";
0|0|0|SCAN TABLE Role AS R (~500000 rows)
0|1|1|SEARCH TABLE Actor AS A USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
0|2|2|SEARCH TABLE Movie AS M USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
sqlite> EXPLAIN QUERY PLAN
   ...>     SELECT R.role, A.first, A.last, M.title
   ...>     FROM Role R, Actor A, Movie M
   ...>     WHERE R.aid=A.aid AND R.mid=M.mid AND M.title="Batman";
0|0|2|SCAN TABLE Movie AS M (~100000 rows)
0|1|0|SEARCH TABLE Role AS R USING INDEX RoleMid (mid=?) (~10 rows)
0|2|1|SEARCH TABLE Actor AS A USING INTEGER PRIMARY KEY (rowid=?) (~1 rows)
sqlite>
```

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.