Overview of Query Evaluation

Midterm on Monday
6-8 pm in SN014

(I*f you need an alternative test time fill-out the on-line survey*)

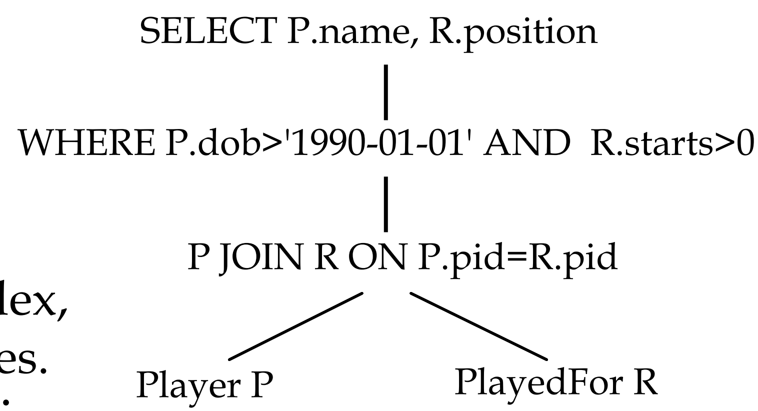
PS #3 due tonight before midnight

Overview of Query Evaluation

- *Query:*

```
SELECT P.name, R.position
FROM Player P, PlayedFor R
WHERE P.pid=R.pid
AND P.dob>'1990-01-01' AND R.starts>0
```

- *Plan:* *Tree of operations with an algorithm for each*
  - Each operation "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 queries use the same simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve a subset of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the search keys of 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 all the tables 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 and high key values (Low/High) for each tree index.
- Catalogs are updated regularly.
  - 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:

  Player(*pid*: int, *name*: string, *college*: string, *dob*: date)
  PlayedFor(*pid*: int, *tid*: int, *year*: int, *starts*: int)

- Assume each tuple of PlayedFor is 16 bytes, a page holds, at most, 250 rows, each Player tuple is 100 bytes, and a page holds no more than 40 rows
- Furthermore, assume
  400 pages of PlayedFor (< 100,000 records), and
  500 pages of Players (< 20,000 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
- These queries 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
pid	Player	integer	1
name	Player	string	2
college	Player	string	3
dob	Player	date	4
pid	PlayedFor	integer	1
tid	PlayedFor	integer	2
year	PlayedFor	integer	3
starts	PlayedFor	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

(dob>'1990-01-01' OR tid=1000 OR year=2018) AND (name='Chris Jones' OR tid=1000 OR year=2018)

- Selection conditions are first converted to "sum-of-products" form (ORs of AND clauses)

  (dob>'1990-01-01' AND name='Chris Jones')
  OR tid=1000 OR year=1995

- "AND" terms allow us to optimally choose indices "OR" terms can be generated as independent query evaluations over the same tables or a subset

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 *dob>'1990-01-01' AND name='Chris Jones'*.
    - A B+ tree index on *dob* can be used;
      then, *name* ccould be checked for each retrieved tuple.
    - Similarly, a hash index on <*name*> could be used;
      then *dob*<*2000-01-01* checked.

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 if table isn't clustered on search key).
  - Assume 10% of players were after before '1990-01-01'.
    - If the table is clustered by *dob* (unlikely), the cost is little more than (0.1 * 500) = 50 I/Os
    - If table isn't clustered by dob, then there are likely 4 per page requiring us to read all 500 pages!
    - In reality, players are clustered by the year that they entered the NFL, so the 50 I/Os might not be that far off since it is *correlated* with *dob*

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 if table isn't clustered on search key).
  - There are 8 players are named '*Chris Jones*'.
    - A single hash leads us to a hash bucket with 8 Player page ids
    - In the worse case the 8 are on different pages, requiring 8 I/Os.
    - The hash index on Player.name is very selective for this query
  - There are almost 300 players with name like '*Chris %*'
    - If these are distributed uniformly across the Player pages, we expect to read almost 300 of the 500 player blocks, making *dob* more selective

Selection

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

```
SELECT DISTINCT pid, tid
FROM   PlayedFor
```

- Sorting Approach
  - Sort on <pid, tid> and remove duplicates.
    (Can optimize by dropping unneeded attributes while sorting.)
- Hashing Approach
  - Hash on <pid, tid> during scan to create partitions.
    Ignore hash-key collisions.
- With an index containing both pid and tid, you can step through the leafs (if tree) compressing duplicates, or directory of a Hash, however, may be cheaper to sort data entries!

Join: Index Nested Loops

```
foreach tuple r in R:
    foreach tuple p in P:
        if r_i op p_j :
            add <r, p> to result
```

```
foreach tuple p in P:
    foreach tuple r in R:
        if r_i op p_j :
            add <r, p> to result
```

- If there is an index on the attribute of one relation (say P), if we make it the *inner loop* to exploit the index.
  - Cost: M + ($(M*p_R)$ * cost of finding matching P tuples)
  - M= #pages of R, p_R=# tuples per R 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 on *name* of Player:
  - Scan PlayedFor: 400 page I/Os, 250*400 tuples.
  - For each PlayedFor tuple: 1.2 I/Os to get bucket index, plus 1 I/O to get a matching Player tuple.
  - Total: 400 + (1+1.2)*100000 = 220,400 I/Os.
- Tree-index on *dob* of Player:
  - Scan Player via TreeIndex: traverse tree (3 page I/Os), scan subset of Player tuples (80 page I/Os, assumes 10% and correlation with *dob*)
  - For each surviving Player tuple: Scan the PlayedFor records
  - Total: 83 + (80*40)*400 = 1,280,083 I/Os
  - Of course, another index on PlayedFor would help here
  - BTW, if the dob filtering was 1%, Total: 83 + (8*40)*400 = 128,083 I/Os

Join: Sort-Merge (*R* JOIN *S* ON *i=j*)

- First, Sort R and S on the join attribute
- Scan both sorted tables while "merging" to output result tuples.
  - Advance scan of R until current R-tuple >= current P tuple, then advance scan of P until current P-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in R_i (*current R group*) and all S tuples with same value in S_j (*current S group*) <u>*match*</u>; output <r_i, s_j> 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. (Repeated scaning of S group is likely to find needed pages in buffer.)

Example of Sort-Merge Join

pid	name	college	dob
29010	Austin Shepherd	Alabama	1992-05-28
29011	Josh Shirley	Nevada-Las Vegas	1992-01-04
29012	Jameill Showers	Texas-El Paso	--
29013	Trevor Siemian	Northwestern	1991-12-26
29014	Ian Silberman	Boston College	1992-10-10
29015	Shayne Skov	Stanford	1990-07-09

pid	tid	year	starts
29010	1032	2015	0
29011	1006	2015	0
29011	1001	2016	0
29012	1012	2015	0
29013	1004	2015	0
29013	1004	2016	14
29013	1004	2017	10
29013	1032	2018	0
29013	1019	2019	0

We'll use "out-of-core" external sorting (Next lecture's topic)

Pass 1: Read P in 10, 50 block chunks, sort each one, and then write them back, then read R in 8, 50 block chunks, sort each, and write them back (2(400+500))

Pass 2: Read in the head blocks of the 10 sorted P chunks and the heads of 8 sorted R chunks. Merge the tops of the 10 into one block and the tops of the 8 into another (refill any head block when it is exhasted). These two merged blocks are then scanned for matching keys (400+500).

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- Using only 50 buffer pages, both Players and PlayedFor can be sorted in 2 passes; total join cost: 3(400+500) = 1800 I/Os.

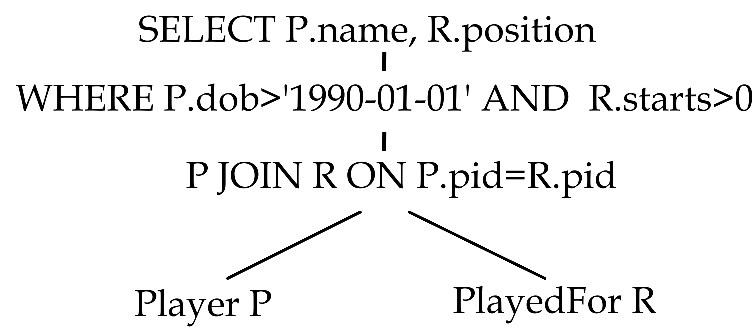
Highlights of Query Optimization

- **Cost estimation:** Approximations are an art.
  - 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

- For each plan considered, we must estimate cost:
  - *Cost* of each operation in plan 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.

Alternate Evaluation Trees:

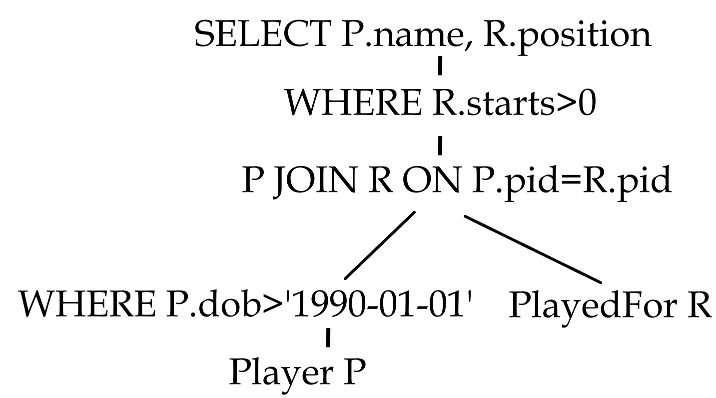


Scan 500 Player blocks and for each scan 400 PlayedFor blocks

Cost Estimation

- For each plan considered, we must estimate cost:
  - *Cost* of each operation in plan 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.

Alternate Evaluation Trees:

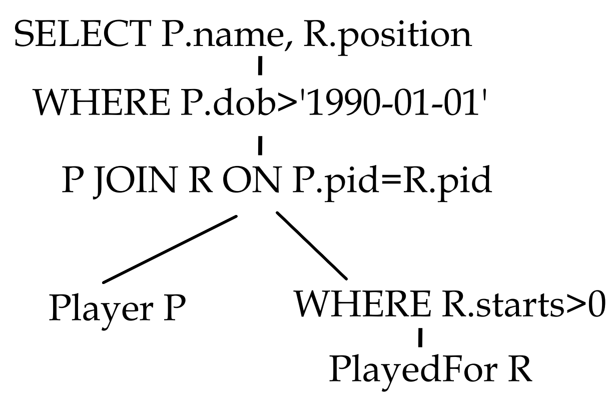


An index on dob allows us to consider around 10% of Players

Cost Estimation

- For each plan considered, we must estimate cost:
  - *Cost* of each operation in plan 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.

Alternate Evaluation Trees:

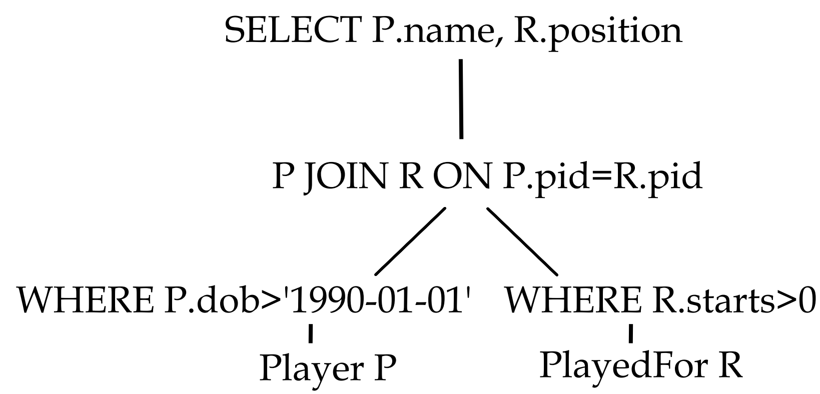


FYI: Only 44% of players on a team's roster ever start a game in a given season

Cost Estimation

- For each plan considered, we must estimate cost:
  - *Cost* of each operation in plan 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.

Alternate Evaluation Trees:



10% of Players joined with 44% of PlayedFor, but how are these "non-starters" distributed?

Size Estimation and Reduction Factors

- Consider a query block:

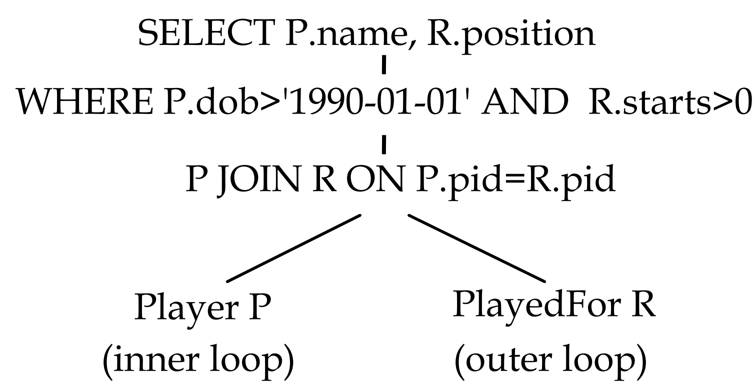
```
SELECT  attribute list
FROM  relation list
WHERE  term_1 AND ... AND term_k
```

- 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 $* RF_1 * RF_2 * \ldots RF_k$.
  - 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 P.name, R.position
FROM Player P, PlayedFor R
WHERE P.pid=R.pid
AND P.dob>'1990-01-01' AND R.starts>0
```

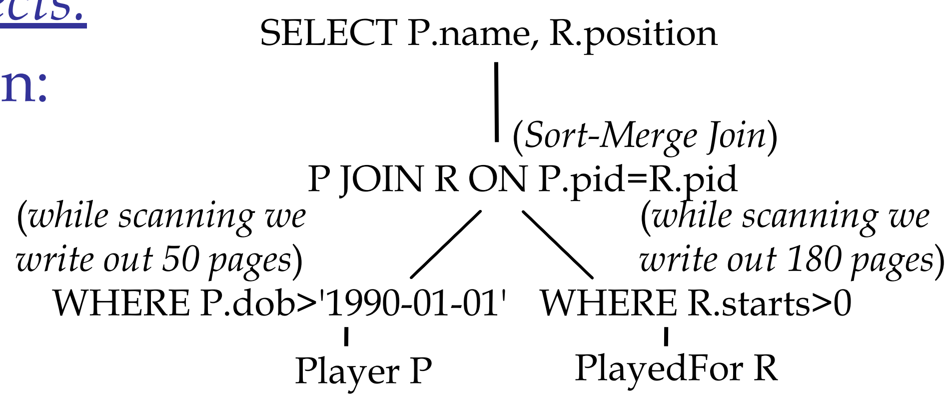
- Cost: 400+400*500 = 200,400 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been “pushed” earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* To find more efficient plans that compute the same answer.



We made the outer loop the one with the fewest blocks, not the fewest records

Alternative Plan 1 (No Indexes)

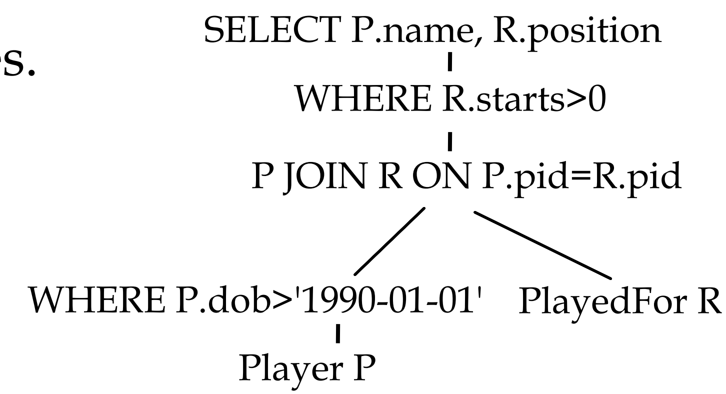
- ***Main difference:*** ***Push selects.***
- With 5 buffers, cost of plan:
  - Scan Player (500) + write temp T1 (50 pages).
  - Scan PlayedFor (400) + write temp T2 (180 pages, 44% of records).
  - Sort T1 (2*50), sort T2 (2*4*45), merge (50+180)
  - Total: 1820 page I/Os.



- If we "push" projections, T1 needs only (*pid, name*),
  T2 needs only (*pid, position*):
  - Thus T1 fits in 15 pages, and T2 fits in 90 cost drops to under 1500 pages.

Alternative Plan 2 (With Indexes)

- With a clustered index on *pid* of PlayedFor, we find that the 10% of pids born after '1990-01-01' fall in the last 80 of 400 pages.



- Join column *sid* is a key for Player.
  - –At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *R.starts>0* before the join is based on availability of PlayedFor's *pid* index.
- Cost: Selection of Player tuples with dob > '1990-01-01' (2 for *dob* index + 80 get the pages) I/Os;
- For each, must get matching tuple (80*40*(1.2 pid index)) total 3922 I/Os. But if dob was more selective (2%) we'd get (2+16)+(16*40*1.1)= 786 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.