

SQL QUERY EVALUATION

CS121: Relational Databases
Fall 2018 – Lecture 12

SQL Query Processing

4

- Databases go through three basic steps:
 - ▣ Parse SQL into an internal representation of a plan
 - ▣ Transform this into an optimized execution plan
 - ▣ Evaluate the optimized execution plan
- Execution plans are generally based on the extended relational algebra
 - ▣ Includes generalized projection, grouping, etc.
 - ▣ Also some other features, like sorting results, nested queries, LIMIT/OFFSET, etc.

Query Evaluation Example

5

- A simple query:

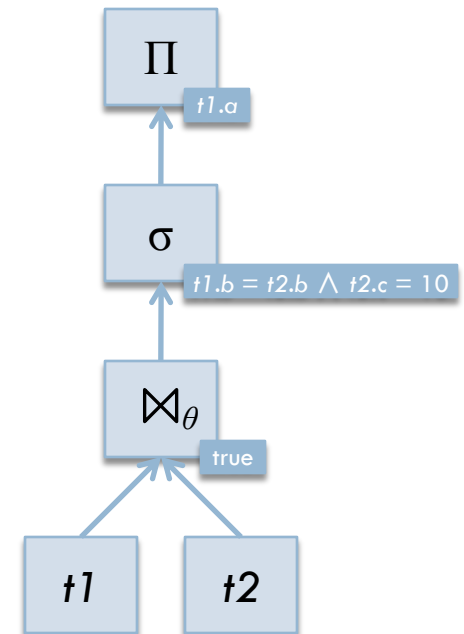
```
SELECT t1.a FROM t1, t2
WHERE t1.b = t2.b AND t2.c = 10;
```

- Translating directly into the relational algebra:

$$\Pi_{t1.a}(\sigma_{t1.b = t2.b \wedge t2.c = 10}(t1 \times t2))$$

- Database might create this structure:

- ▣ DBs usually implement common join operations with theta-join plan nodes
- ▣ Can be evaluated using a push- or a pull-based approach
- ▣ Evaluation loop retrieves results from top-level Π operation



Query Optimization

6

- Are there alternate formulations of our query?

$\Pi_{t1.a}(\sigma_{t1.b = t2.b \wedge t2.c = 10}(t1 \times t2))$

$\Pi_{t1.a}(t1 \bowtie_{t1.b = t2.b} (\sigma_{t2.c = 10}(t2)))$

$\Pi_{t1.a}(\sigma_{t2.c = 10}(t1 \bowtie_{t1.b = t2.b} t2))$

- *Which one is fastest?*

- The query optimizer generates many equivalent plans using a set of equivalence rules
 - Cost-based optimizers assign each plan a cost, and then the lowest-cost plan is chosen for execution
 - Heuristic optimizers just follow a set of rules for optimizing a query plan

Query Evaluation Costs

7

- A variety of costs in query evaluation
- Primary expense is reading data from disk
 - ▣ Usually, data being processed won't fit entirely into memory
 - ▣ Try to minimize disk seeks, reads and writes!
- CPU and memory requirements are secondary
 - ▣ Some ways of computing a result require more CPU and memory resources than others
 - ▣ Becomes especially important in concurrent usage scenarios
- Can be other costs as well
 - ▣ In distributed database systems, network bandwidth must be managed by query optimizer

Query Optimization (2)

8

- Several questions the optimizer has to consider:
 - ▣ How is a relation's data stored on the disk?
 - ...and what access paths are available to the data?
 - ▣ What implementations of the relational algebra operations are available to use?
 - Will one implementation of a particular operation be much better or worse than another?
 - ▣ How does the database decide which query execution plan is best?
- Given the answers to these questions, what can we do to make the database go faster?

Select Operation

9

- How to implement σ_P operation?
- Easy solution from last time: scan the entire data file
 - ▣ Called a file scan
 - ▣ Test selection predicate against each tuple in the data file
 - ▣ Will be slow, since every disk block must be read
- This is a *general* solution, but not a *fast* one.
- What is the selection predicate P?
 - ▣ Depending on the characteristics of P, might be able to choose a more optimal evaluation strategy
 - ▣ If we can't, just stick with the file scan

Select Operation (2)

10

- Most select predicates involve a binary comparison
 - ▣ “Is an attribute equal to some value?”
 - ▣ “Is an attribute less than some value?”
- If data file was ordered, could use a binary search...
 - ▣ Would substantially reduce number of blocks read
 - ▣ Maintaining the logical record ordering becomes very costly if data changes frequently
- Solution:
 - ▣ Continue using heap file organization for table data
 - ▣ For important attributes, build indexes against the data file
 - Index provides a faster way to find specific values in the data file

Select Operation

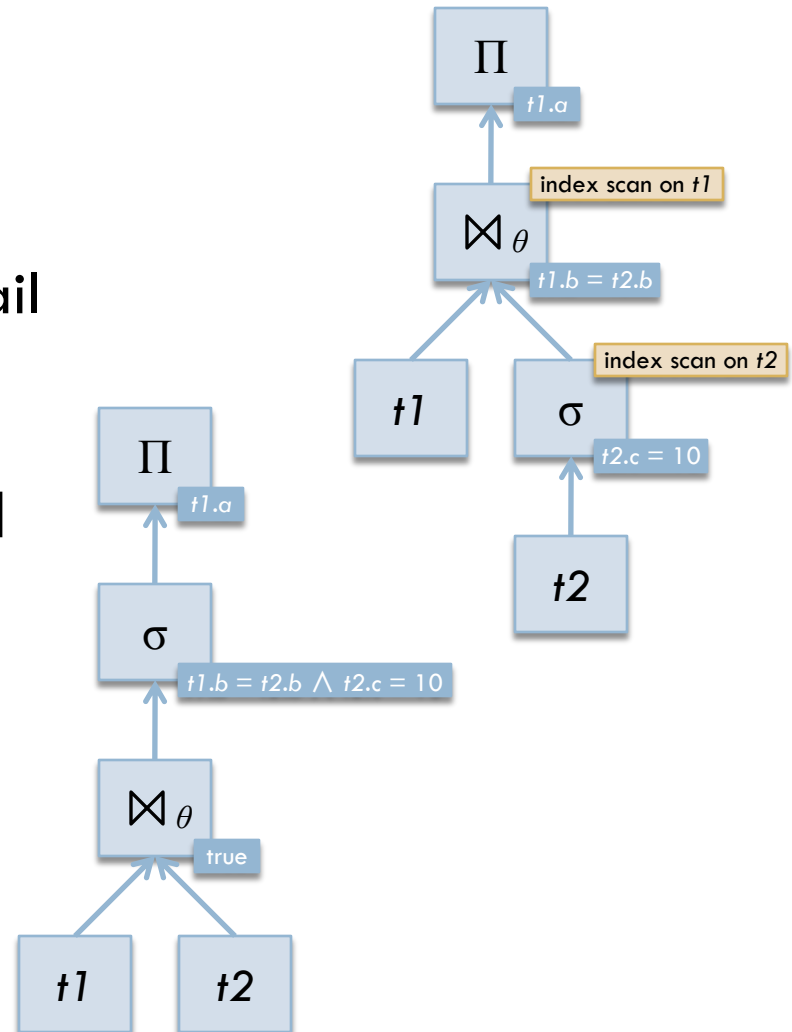
11

- Query planner/optimizer looks at all access paths available for a given attribute
- For select operations:
 - ▣ If select predicate is an equality test and an index is available for that attribute, can use an index scan
 - ▣ Can also use index scan for comparison/range tests if an ordered index is available for the attribute
- For more complicated tests, or if no index is available for attributes being used:
 - ▣ Use the simple file scan approach

Query Optimization Using Indexes

12

- Database query optimizer looks for available indexes
 - ▣ If a select/lookup operation can use an index, execution plan is annotated with this detail
 - ▣ Overall plan cost is computed including these optimizations
- Indexes can only be exploited in certain circumstances
 - ▣ Typically, only by plan nodes that directly access the table
 - ▣ e.g. original plan can't really exploit indexes at all ☹️



Project Operation

13

- Project operation is simple to implement
 - ▣ For each input tuple, create a new tuple with only the specified attributes
 - ▣ May also involve computed values
- Which would be faster, in general?
 - $\Pi_{balance}(\sigma_{balance < 2500}(account))$
 $\sigma_{balance < 2500}(\Pi_{balance}(account))$
 - ▣ Want to project as few rows as possible, to minimize CPU and memory usage
 - Do select first: $\Pi_{balance}(\sigma_{balance < 2500}(account))$
 - ▣ Good heuristic example: “Do projects as late as possible.”

Sorting

14

- SQL allows results to be ordered
- Databases must provide sorting capabilities in execution plans
 - ▣ Data being sorted may be much larger than memory!
- For tables that fit in memory, traditional sorting techniques are used (e.g. quick-sort)
- For tables that are larger than memory, must use an external-memory sorting technique
 - ▣ Table is divided into runs to be sorted in memory
 - ▣ Each run is sorted, then written to a temporary file
 - ▣ All runs are merged using an N-way merge sort

Sorting (2)

15

- In general, sorting should be applied as late as possible
 - ▣ Ideally, rows being sorted will fit into memory
- Some other operations can also use sorted inputs to improve performance
 - ▣ Join operations
 - ▣ Grouping and aggregation
 - ▣ Usually occurs when sorted results are already available
- Could also perform sorting with an ordered index
 - ▣ Scan index, and retrieve each tuple from table file in order
 - ▣ With magnetic disks, seek-time usually makes this prohibitive
 - (solid-state disks don't have this issue!)

Join Operations

16

- Join operations are very common in SQL queries
 - ▣ ...especially when using normalized schemas
- Could also potentially be a very costly operation!
 - ▣ $r \bowtie s$ defined as $\sigma_{r.A = s.A}(r \times s)$
- A simple strategy for $r \bowtie_{\theta} s$:
 - for each** tuple t_r **in** r **do begin**
 - for each** tuple t_s **in** s **do begin**
 - if** t_r, t_s satisfy condition θ **then**
 - add $t_r \cdot t_s$ to result
 - end**
 - end**
 - end**
- $t_r \cdot t_s$ denotes the concatenation of t_r with t_s

Nested-Loop Join

17

- Called the nested-loop join algorithm:
 for each tuple t_r **in** r **do begin**
 for each tuple t_s **in** s **do begin**
 if t_r, t_s satisfy condition θ **then**
 add $t_r \cdot t_s$ to result
 end
 end
 end
- A very slow join implementation
 - ▣ Scans r once, and scans s once for each row in r !
 - ▣ Not so horrible if s fits entirely in memory
- But, it can handle arbitrary conditions
 - ▣ For some queries, the only option is a nested-loop join!

Indexed Nested-Loop Join

18

- Most join conditions involve equalities
 - ▣ Called equijoins
- Indexes can speed up table lookups...
- Modify nested-loop join to use indexes in inner loop:
 for each tuple t_r **in** r **do begin**
 use index on s to retrieve tuple t_s
 if t_r, t_s satisfy condition θ **then**
 add $t_r \cdot t_s$ to result
 end
- Only an option for equijoins, where an index exists for the join attributes

MySQL Join Processor

19

- MySQL join processor is based on nested-loop join algorithm
 - ▣ Instead of joining two tables, can join N tables at once

```
for each tuple  $t_r$  in  $r$  do begin
  for each tuple  $t_s$  in  $s$  do begin
    for each tuple  $t_t$  in  $t$  do begin
      if  $t_r, t_s, t_t, \dots$  satisfy condition  $\theta$  then
        add  $t_r \cdot t_s \cdot t_t \cdot \dots$  to result
      end
    end
  end
end
```
- Employs many optimizations
 - ▣ When possible, outer table is processed in blocks, to reduce number of iterations over inner tables
 - ▣ Indexes are exploited *heavily* for finding tuples in inner tables.
 - ▣ If a subquery can be resolved into a constant, it is.

MySQL Join Processor (2)

20

- Since MySQL join processor relies so heavily on indexes, what kinds of queries is it bad at?
 - ▣ Queries against tables without indexes... (duh)
 - ▣ **Queries involving joins against derived relations (ugh!)**
 - ▣ MySQL isn't smart enough to save the derived relation into a temporary table, then build an index against it
 - A common technique for optimizing complex queries in MySQL
- For more sophisticated queries, really would like more advanced join algorithms...
 - ▣ Most DBs include several other very powerful join algorithms
 - ▣ (Can't add to MySQL easily, since it doesn't use relational algebra as a query-plan representation...)

Sort-Merge Join

21

- If tables are already ordered by join attributes, can use a merge-sort technique
 - ▣ Must be an equijoin!
- Simple high-level description:
 - ▣ Two pointers to traverse tables in order:
 - p_r starts at first tuple in r
 - p_s starts at first tuple in s
 - ▣ If one pointer's tuple has join-attribute values less than the other pointer, advance that pointer
 - ▣ When pointers have the same value of the join attribute, generate joins using those rows
 - If p_r or p_s points to a run of records with the same value, must include all of these records in the join result

Sort-Merge Join (2)

22

- Much better performance than nested-loop join
 - ▣ Dramatically reduces disk accesses
 - ▣ Unfortunately, relations aren't usually ordered
- Can also enhance sort-merge joins when at least one relation has an index on the join attributes
 - ▣ e.g. one relation is sorted, and the unsorted relation has an index on the join attributes
 - ▣ Traverse unsorted relation's index in order
 - ▣ When rows match, use index to pull those tuples from disk
 - ▣ Disk seek cost must be managed carefully with this technique
 - e.g. can sort record pointers before reading the tuples from disk, to minimize the overall seek time

Hash Join

23

- Another join technique for equijoins
- For tables r and s :
 - ▣ Use a hash function on the join attributes to divide rows of r and s into partitions
 - Use same hash function on both r and s , of course
 - Partitions are saved to disk as they are generated
 - Aim for each partition to fit in memory
 - r partitions: $H_{r1}, H_{r2}, \dots, H_{rn}$
 - s partitions: $H_{s1}, H_{s2}, \dots, H_{sn}$
 - ▣ Rows in H_{ri} will only join with rows in H_{si}

Hash Join (2)

24

- After partitioning:

- for** $i = 1$ **to** n **do**

- build a hash index on H_{si}

- (using a second hash function!)*

- for each** row t_r **in** H_{ri}

- probe hash index for matching rows in H_{si}

- for each** matching tuple t_s **in** H_{si}

- add $t_r \cdot t_s$ to result

- end**

- end**

- end**

- Very fast and efficient equijoin strategy

- Very good for joining against derived relations!

- Can perform badly when rows can't be hashed into partitions that fit into memory

Outer Joins

25

- Join algorithms can be modified to generate left outer joins reasonably efficiently
 - ▣ Right outer join can be restated as left outer join
 - ▣ Will still impact overall query performance if many rows are generated
- Full outer joins can be significantly harder to implement
 - ▣ Sort-merge join can compute full outer join easily
 - ▣ Nested loop and hash join are much harder to extend
 - ▣ Full outer joins can also impact query performance heavily

Other Operations

26

- Set operations require duplicate elimination
 - ▣ Duplicate elimination can be performed with sorting or with hashing
- Grouping and aggregation can be implemented in several ways
 - ▣ Can sort results on the grouping attributes, then compute aggregates over the sorted values
 - All rows in a given group are adjacent to each other, so uses memory very efficiently (at least, after the sorting step...)
 - MySQL uses this approach by default
 - ▣ Can also use hashing to perform grouping and aggregation
 - Hash tuples on the grouping attributes, and compute each group's aggregate values incrementally

Optimizing Query Performance

27

- To improve query performance, you must know how the database actually runs your query
- Discussed the “explain” statement last time
 - ▣ Runs planner and optimizer on your query, then outputs the plan and corresponding cost estimates
- Using this information, you can:
 - ▣ Create indexes on tables, where appropriate
 - ▣ Restate the query to help the DB pick a better plan
- Harder cases may require multiple steps:
 - ▣ Generate intermediate results more well-suited for the desired query
 - ▣ Then, use intermediate results to generate final results

Query Execution Example

28

- For each assignment, finds the average size of the last submission from students for that assignment:

```
SELECT shortname,  
       AVG(last_submission_size) AS  
         avg_last_submission_size  
FROM assignment NATURAL JOIN  
     submission NATURAL JOIN  
     (SELECT sub_id,  
            total_size AS last_submission_size  
     FROM fileset NATURAL JOIN  
       (SELECT sub_id, MAX(sub_date) AS sub_date  
        FROM fileset GROUP BY sub_id  
       ) AS last_sub_dates  
     ) AS last_sub_sizes  
GROUP BY shortname;
```

Find the date of the last fileset submitted for each student's submission. Name the result columns to allow a natural join against the fileset table.

Query Execution Example (2)

29

- For each assignment, finds the average size of the last submission from students for that assignment:

```
SELECT shortname,  
       AVG(last_submission_size) AS  
         avg_last_submission_size  
FROM assignment NATURAL JOIN  
  submission NATURAL JOIN  
  (SELECT sub_id,  
         total_size AS last_submission_size  
   FROM fileset NATURAL JOIN  
     (SELECT sub_id, MAX(sub_date) AS sub_date  
      FROM fileset GROUP BY sub_id  
     ) AS last_sub_dates  
  ) AS last_sub_sizes  
GROUP BY shortname;
```

Join the derived result against fileset so we can retrieve the total size of the submitted files.

Query Execution Example (3)

30

- For each assignment, finds the average size of the last submission from students for that assignment:

```
SELECT shortname,  
       AVG(last_submission_size) AS  
         avg_last_submission_size  
FROM assignment NATURAL JOIN  
   submission NATURAL JOIN  
   (SELECT sub_id,  
          total_size AS last_submission_size  
    FROM fileset NATURAL JOIN  
      (SELECT sub_id, MAX(sub_date) AS sub_date  
       FROM fileset GROUP BY sub_id  
      ) AS last_sub_dates  
   ) AS last_sub_sizes  
GROUP BY shortname;
```

Outermost query finds the averages of these last submissions,
and also incorporates the short-name of each assignment.

MySQL Execution and Analysis

31

- MySQL executes this query rather slowly*
 - ▣ About 3 sec on a server with 8GB RAM, RAID1 mirroring
 - ▣ Intuitively makes sense...
 - Joins against derived relations, non-index columns, etc.
 - All the stuff that MySQL isn't so good at handling

□ EXPLAIN output:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	1506	Using temporary; Using filesort
1	PRIMARY	submission	eq_ref	PRIMARY	PRIMARY	4	last_sub_sizes.sub_id	1	
1	PRIMARY	assignment	eq_ref	PRIMARY	PRIMARY	4	donnie_db.submission.asn_id	1	
2	DERIVED	<derived3>	ALL	NULL	NULL	NULL	NULL	1506	
2	DERIVED	fileset	ALL	NULL	NULL	NULL	NULL	2799	Using where; Using join buffer
2	DERIVED	submission	eq_ref	PRIMARY	PRIMARY	4	last_sub_dates.sub_id	1	Using index
3	DERIVED	fileset	ALL	NULL	NULL	NULL	NULL	2799	Using temporary; Using filesort

- Confirms our suspicions
- Can optimize by storing innermost results in a temp table, and creating indexes on (sub_id, sub_date)

* Test was performed with MySQL 5.1; MariaDB 5.5 executes this query extremely quickly.

PostgreSQL Execution/Analysis (1)

32

- Postgres executes this query instantaneously. On a laptop.
 - ▣ Fundamental difference: more sophisticated join algorithms
 - Specifically hash join, which is very good at joining relations on non-indexed attributes
- **EXPLAIN** output:

```
HashAggregate (cost=221.38..221.39 rows=1 width=8)
-> Nested Loop (cost=144.28..221.37 rows=1 width=8)
    -> Nested Loop (cost=144.28..213.09 rows=1 width=20)
        -> Nested Loop (cost=144.28..212.81 rows=1 width=20)
            -> Hash Join (cost=144.28..204.53 rows=1 width=12)
                Hash Cond: ((fileset.sub_id = fileset.sub_id) AND ((max(fileset.sub_date)) = fileset.sub_date))
                -> HashAggregate (cost=58.35..77.18 rows=1506 width=12)
                    -> Seq Scan on fileset (cost=0.00..44.57 rows=2757 width=12)
                -> Hash (cost=44.57..44.57 rows=2757 width=16)
                    -> Seq Scan on fileset (cost=0.00..44.57 rows=2757 width=16)
            -> Index Scan using submission_pkey on submission (cost=0.00..8.27 rows=1 width=8)
                Index Cond: (submission.sub_id = fileset.sub_id)
        -> Index Scan using assignment_pkey on assignment (cost=0.00..0.27 rows=1 width=8)
            Index Cond: (assignment.asn_id = submission.asn_id)
    -> Index Scan using submission_pkey on submission (cost=0.00..8.27 rows=1 width=4)
        Index Cond: (submission.sub_id = fileset.sub_id)
```

- As expected, Postgres uses a hash join to join the derived relation against **fileset** table on non-index columns

PostgreSQL Execution/Analysis (2)

33

- Can disable various join algorithms in Postgres 😊
 - ▣ `SET enable_hashjoin = off;`
- **EXPLAIN** output:

```
HashAggregate (cost=422.68..422.69 rows=1 width=8)
-> Nested Loop (cost=373.85..422.67 rows=1 width=8)
    -> Nested Loop (cost=373.85..414.39 rows=1 width=20)
        -> Nested Loop (cost=373.85..414.11 rows=1 width=20)
            -> Merge Join (cost=373.85..405.83 rows=1 width=12)
                Merge Cond: ((fileset.sub_id = fileset.sub_id) AND (fileset.sub_date = (max(fileset.sub_date))))
                -> Sort (cost=202.12..209.01 rows=2757 width=16)
                    Sort Key: fileset.sub_id, fileset.sub_date
                    -> Seq Scan on fileset (cost=0.00..44.57 rows=2757 width=16)
                -> Sort (cost=171.73..175.50 rows=1506 width=12)
                    Sort Key: fileset.sub_id, (max(fileset.sub_date))
                    -> HashAggregate (cost=58.35..77.18 rows=1506 width=12)
                        -> Seq Scan on fileset (cost=0.00..44.57 rows=2757 width=12)
            -> Index Scan using submission_pkey on submission (cost=0.00..8.27 rows=1 width=8)
                Index Cond: (submission.sub_id = fileset.sub_id)
        -> Index Scan using assignment_pkey on assignment (cost=0.00..0.27 rows=1 width=8)
            Index Cond: (assignment.asn_id = submission.asn_id)
    -> Index Scan using submission_pkey on submission (cost=0.00..8.27 rows=1 width=4)
        Index Cond: (submission.sub_id = fileset.sub_id)
```

- Sort + sort-merge join is still faster than nested loops!!

PostgreSQL Execution/Analysis (3)

34

- Now, disable sort-merge joins too:
 - ▣ **SET enable_mergejoin = off;**
- Finally, Postgres performance is closer to MySQL
- **EXPLAIN** output:

```
HashAggregate (cost=103956.21..103956.23 rows=1 width=8)
-> Nested Loop (cost=93.75..103956.21 rows=1 width=8)
    -> Nested Loop (cost=93.75..103947.93 rows=1 width=20)
        -> Nested Loop (cost=93.75..103947.65 rows=1 width=20)
            -> Nested Loop (cost=93.75..103939.37 rows=1 width=12)
                Join Filter: ((fileset.sub_id = fileset.sub_id) AND (fileset.sub_date = (max(fileset.sub_date))))
                -> Seq Scan on fileset (cost=0.00..44.57 rows=2757 width=16)
                -> Materialize (cost=93.75..108.81 rows=1506 width=12)
                    -> HashAggregate (cost=58.35..77.18 rows=1506 width=12)
                        -> Seq Scan on fileset (cost=0.00..44.57 rows=2757 width=12)
            -> Index Scan using submission_pkey on submission (cost=0.00..8.27 rows=1 width=8)
                Index Cond: (submission.sub_id = fileset.sub_id)
        -> Index Scan using assignment_pkey on assignment (cost=0.00..0.27 rows=1 width=8)
            Index Cond: (assignment.asn_id = submission.asn_id)
    -> Index Scan using submission_pkey on submission (cost=0.00..8.27 rows=1 width=4)
        Index Cond: (submission.sub_id = fileset.sub_id)
```


Query Estimates

35

- Query planner/optimizer must make *estimates* about the cost of each stage
- Database maintains statistics for each table, to facilitate planning and optimization
- Different levels of detail:
 - ▣ Some DBs only track min/max/count of values in each column. Estimates are very basic.
 - ▣ Some DBs generate and store histograms of values in important columns. Estimates are much more accurate.
- Different levels of accuracy:
 - ▣ Statistics are expensive to maintain! Databases update these statistics relatively infrequently.
 - ▣ If a table's contents change substantially, must recompute statistics

Table Statistics Analysis

36

- Databases also frequently provide a command to compute table statistics
- MySQL command:
`ANALYZE TABLE assignment, submission, fileset;`
- PostgreSQL command:
`VACUUM ANALYZE;`
 - for all tables in database`VACUUM ANALYZE tablename;`
 - for a specific table
- These commands are expensive!
 - ▣ Perform a full table-scan
 - ▣ Also, typically lock the table(s) for exclusive access

Review

37

- Discussed general details of how most databases evaluate SQL queries
- Some relational algebra operations have several ways to evaluate them
 - ▣ Optimizations for very common special cases, e.g. equijoins
- Can give the database some guidance
 - ▣ Create indexes on tables where appropriate
 - ▣ Rewrite queries to be more efficient
 - ▣ Make sure statistics are up-to-date, so that planner has best chance of generating a good plan