Query Processing

CISC637, Lecture #16
Ben Carterette

Basic Steps

- Given a SQL query:
  - parse and translate to relational algebra
  - optimize (form lowest-cost evaluation plan)
    - consider possible equivalent relational algebra queries
    - consider available indexes and file storage types
    - consider possible algorithms for each relational algebra operator in the query
  - evaluate
- Use `EXPLAIN` to view the DBMS' evaluation plan

Review

- Use EXPLAIN to view the DBMS' evaluation plan
- Projection:
  - type=ALL full file scan
  - type=index index-only scan
    - key=PRIMARY primary index scan
    - key=other secondary index scan
- Selection:
  - type=ALL full file scan
  - type=const exactly one match in PRIMARY index
    - ref=const, key=PRIMARY
  - type=ref = search, but more than one match
    - ref=const, key=PRIMARY or key=other
  - type=range <> search
    - If can be completed with index only, the Extra field will say so
- Use FORCE INDEX (index_name) or IGNORE INDEX (index_name) to give MySQL hints about how you want it to process queries

Approaches to Selection

- Full file scan
  - Needed when search is on non-indexed field
- Index lookup
  - Search is on indexed field (primary key or other)
  - Search is equality or range
    - For range, find first value in range, then scan forward
      - Or last value in range, then scan backward
  - Access full file only if necessary to complete query
- Combination of indexes + file access
  - Search includes multiple fields, some indexed, some not
  - Use most selective fields first, then scan records

Algorithms for Joins

- Index nested-loop join
- Block nested-loop join
- Merge-join

Index Nested-Loop Join

```
> SELECT * FROM takes JOIN student ON takes.ID = student.ID;
…
3540308 rows in set (6.77 sec)


> EXPLAIN SELECT * FROM takes JOIN student ON takes.ID = student.ID;
+----+-------------+---------+------+---------------+---------+---------+------------+--------+-------+
| id | select_type | table   | type | possible_keys | key     | key_len | ref        | rows   | Extra |
+----+-------------+---------+------+---------------+---------+---------+------------+--------+-------+
|  1 | SIMPLE      | student | ALL  | PRIMARY       | NULL    | NULL    | NULL       | 198376 |       |
|  1 | SIMPLE      | takes   | ref  | PRIMARY       | PRIMARY | 4       | student.ID |      9 |       |
+----+-------------+---------+------+---------------+---------+---------+------------+--------+-------+
```

simple selects

use primary key
for takes lookups

full table scan of student;
index lookup in takes
(based on value of student.ID)

Indexed Nested-Loop Join

- Joining tables T1, T2 on field K; table T2 has an index on K
- Algorithm:
  - Scan T1 block by block
  - For each block B_i read,
    - For each record t_j in B_i,
      - Look up values of $t_j[K]$ in T2's index
      - M = locations of blocks containing matching records
      - For each matching block M_k,
        - » For each record t_m in M_k,
          - If (t_j, t_m) matches join condition, add (t_j, t_m) to result set
- Block reads = O(B1 + R1×(H2 + M))
  - B1 = # of blocks in T1, R1 = # of records in T1
  - H2 = height of B+ tree index on T2, M = # of matching blocks in T2
- If both tables have indexes on K there is a choice:
  - Which table should be scanned fully (outer loop)?
  - Which should be accessed by index lookup (inner loop)?

Index Nested-Loop Join: Alternative Table Order

```
> SELECT * FROM takes IGNORE INDEX (PRIMARY) JOIN student FORCE INDEX (PRIMARY) WHERE
    takes.ID=student.ID;
…
3540308 rows in set (6.45 sec)


> EXPLAIN SELECT * FROM takes IGNORE INDEX (PRIMARY) JOIN student FORCE INDEX (PRIMARY) WHERE
    takes.ID=student.ID;
+----+-------------+---------+--------+---------------+---------+---------+-----------------+--------+-------+
| id | select_type | table   | type   | possible_keys | key     | key_len | ref             | rows   | Extra |
+----+-------------+---------+--------+---------------+---------+---------+-----------------+--------+-------+
|  1 | SIMPLE      | takes   | ALL    | NULL          | NULL    | NULL    | NULL            | 554130 |       |
|  1 | SIMPLE      | student | eq_ref | PRIMARY       | PRIMARY | 4       | takes.ID        |      1 |       |
+----+-------------+---------+--------+---------------+---------+---------+-----------------+--------+-------+
```

simple selects

use primary key
for student lookups

full table scan of takes;
index lookup in student
(based on value of takes.ID)
eq_ref means there will be exactly 1
matching record

Block Nested-Loop Join

```
> SELECT * FROM takes IGNORE INDEX (PRIMARY) JOIN student IGNORE INDEX (PRIMARY) ON takes.ID =
    student.ID;
…
3540308 rows in set (6.77 sec)


> EXPLAIN SELECT * FROM takes IGNORE INDEX (PRIMARY) JOIN student IGNORE INDEX (PRIMARY) ON takes.ID =
student.ID;
+----+-------------+---------+------+---------------+---------+---------+--------------+---------+-------+
| id | select_type | table   | type | possible_keys | key     | key_len | ref          | rows    | Extra |
+----+-------------+---------+------+---------------+---------+---------+--------------+---------+-------+
|  1 | SIMPLE      | takes   | ALL  | NULL          | NULL    | NULL    | NULL         | 3049258 | Using where;
|  1 | SIMPLE      | student | ALL  | NULL          | NULL    | NULL    | NULL         |  198376 | Using join buffer
+----+-------------+---------+------+---------------+---------+---------+--------------+---------+(Block Nested Loop)
```

simple selects

full table scan of both tables

Block Nested-Loop Join

- Block nested-loop join (when there are no indexes):
  - Scan T1 block by block
  - For each block B_i read,
    - Scan T2 block by block
    - For each block B_j read,
      - For each record t_a in B_i,
        - » For each record t_b in B_j,
          - if (t_a, t_b) matches join condition,
            - add (t_a, t_b) to result set
  - Cost = $DB_1B_2 + CR_1R_2 = O(B_1B_2)$
- If possible, optimize by scanning all of T2 before looping
  - Only works if T2 can fit in memory
  - Cost = $D(B_1 + B_2)$

Merge Join

- Merge two tables record-by-record
  - Very much like mergesort's re-combine step
  - Requires that both relations are sorted on join key
  - Cost = $D(B_1 + B_2)$
- Any way to get benefit of that low cost if tables aren't sorted?
  - Before answering that, let's discuss GROUP BY

Aggregation (G)

```
SELECT field, COUNT(*) FROM table GROUP BY field
```

- Several different approaches
  - Choice depends on memory availability, whether index on grouping attribute is available, whether file is sorted by grouping attribute
- Best case is to be able to compute the aggregation function in one pass over the data

MySQL Query Processing: Aggregation

```
> SELECT ID, COUNT(*) FROM takes GROUP BY ID;
…
151840 rows in set (2.31 sec)


> EXPLAIN SELECT ID, COUNT(*) FROM takes GROUP BY ID;
+----+-------------+-------+-------+-------------------+---------+---------+------+---------+-------------+
| id | select_type | table | type  | possible_keys     | key     | key_len | ref  | rows    | Extra       |
+----+-------------+-------+-------+-------------------+---------+---------+------+---------+-------------+
|  1 | SIMPLE      | takes | index | PRIMARY,course_id | PRIMARY | 34      | NULL | 3049258 | Using index |
+----+-------------+-------+-------+-------------------+---------+---------+------+---------+-------------+
```

simple select

index scan

PRIMARY index

no other files required

Aggregation by Index Scan

- Used when there's an index on the GROUP BY field
  - and other fields in the table are not required
- Use the fact that search key values in B+ tree index are stored sequentially
  - Reading index records sequentially guarantees that you will get all records with same key value in a group
  - Compute aggregation with one scan of index
- Algorithm:
  - for each block B in index I:
    - read records until search key value changes
    - once it has changed, compute aggregation function on the rows that have been read, add result to result set

MySQL Query Processing: Aggregation

```
> SELECT ID, MAX(grade) FROM takes GROUP BY ID;
…
151840 rows in set (2.59 sec)

> EXPLAIN SELECT ID, MAX(grade) FROM takes GROUP BY ID;
+----+-------------+-------+-------+-------------------+----------------+---------+------+---------+-------+
| id | select_type | table | type  | possible_keys     | key            | key_len | ref  | rows    | Extra |
+----+-------------+-------+-------+-------------------+----------------+---------+------+---------+-------+
|  1 | SIMPLE      | takes | index | PRIMARY,course_id | PRIMARY        | 34      | NULL | 3049258 |       |
+----+-------------+-------+-------+-------------------+----------------+---------+------+---------+-------+
```

simple select

index scan

primary index

since it doesn't say "Using index", other file access necessary

same as previous algorithm, except requires following index pointers to full records

MySQL Query Processing: Aggregation

```
> SELECT grade, COUNT(*) FROM takes GROUP BY grade;
…
5 rows in set (3.23 sec)

> EXPLAIN SELECT grade, COUNT(*) FROM takes GROUP BY grade;
+----+-------------+---------+------+---------------+------+---------+------+--------+---------------------------------+
| id | select_type | table   | type | possible_keys | key  | key_len | ref  | rows   | Extra                           |
+----+-------------+---------+------+---------------+------+---------+------+--------+---------------------------------+
|  1 | SIMPLE      | takes   | ALL  | NULL          | NULL | NULL    | NULL | 3049258| Using temporary; Using filesort |
+----+-------------+---------+------+---------------+------+---------+------+--------+---------------------------------+
```

simple select

full table scan

sort takes file into a temporary table

Aggregation Algorithms

- Three possible approaches:
  - Index scan only, when all needed fields are in index
  - Index scan + file access, when GROUP BY fields are in index
  - Filesort, when GROUP BY fields are not in index
- Having a sequential file makes aggregation very easy
- If the file is not already sequential, it often makes sense to re-sort it
  - Sort in memory if it can fit
  - Sort on disk if not
    - External mergesort

External Merge Sort

- Suppose we have a file that requires B blocks of storage, and we have M blocks of usable memory ($M < B$)
  - For i from 1 to $\lceil B/M \rceil$
    - read M blocks
    - sort all records in those blocks in memory
    - write M blocks to temporary file F_i
- When done, we will have $\lceil B/M \rceil$ files in which all records are in sorted order by the sort key

External Merge Sort

- Merge those files
  - read first block of each temporary file into memory buffer ($\lceil B/M \rceil$ blocks total, one for each file)
  - create output block in memory
  - repeat:
    - take first record in sorted order among all blocks in memory
      - this record will come from one of the temp files, call it F_j
    - write that record to the output block (in memory)
      - when output block is full, write it to disk and start a new one
    - advance to next record in that block of F_j
    - if at end of block, replace it in memory with next block of F_j
  - until no more blocks for any partial relation
- When done, output file will be sorted by some field

Let's Go Back to Merge-Join

- *Sort*-Merge Join
- Join two tables on some field
  - We saw that merge join works well when both tables are sorted on that field
  - What if only one of them is sorted?
- If either relation is unsorted, sort it
  - Worst case: both have to be sorted externally
  - Cost? $O(2B \log_2 B)$ to sort one
    - Total with sorting both: $O(2B_1 \log_2 B_1 + 2B_2 \log_2 B_2 + B_1 + B_2)$
  - Compare to $O(B_1 + B_2)$ for merge-join
  - Compare to $O(B_1 + R_1 \times (h+M))$ for indexed nested-loop join