Query processing

Rasmus Pagh

Literature: RG 12.1,12.2,12.4,12.5,12.6,20

Today's lecture

First part:

- Strategies for query evaluation, by example.
- DBMS query evaluation algorithms.
- A primer on query optimization
- Making use of this knowledge: Schema tuning

Second part:

- Case study using MySQL

Recap: Indexing

- The choice of whether to use an index is made by the DBMS *for every instance of a query*
  - May depend on query parameters
  - Don’t have to take indexes into account when writing queries
- Clustering indexes store tuples that match a range condition together.
  - Only primary indexes can be clustering.
- Some queries can be answered looking **only** at the index (”covering index”).

Query optimization, query tuning

- **Query optimization** is the process where the DBMS tries to find the ”best possible” way of evaluating a given query.
- Standard approach builds on finding a ”good” relational algebra expression and then choosing how and in what order the operations are to be executed.
- **Query tuning** is a ”manual” effort to make query execution faster.

Query evaluation in a nutshell

- SQL can be rewritten to (extended) relational algebra
- The building blocks in DBMS query evaluation are algorithms that implement relational algebra operations.
- May be based on:
  - Sorting,
  - hashing, or
  - using existing indexes
- The DBMS knows the characteristics of each approach, and attempts to use the best one in a given setting.

Example 1

```
SELECT title
FROM (SELECT *
      FROM Movie
      WHERE studioName = 'Disney')
WHERE year = 1990;
```

Possible strategies:

1. Make a scan of the whole relation.
2. If available: Find movies from 1990 using index, then filter.
3. If available: Find Disney movies using index, then filter.
4. If available: Find Disney movies from 1990 **and** their titles in an index.

Example 2

```
SELECT *
FROM Movie M, Producer P
WHERE M.year=2015 AND
   P.birthdate<'1940-01-01' AND
  M.producer = P.id;
```

Possible strategies:

1. Use index to find 2015 tuples, use index to find matching tuples in Producer.
2. Use index to find producers born before 1940, use index to find matching movies.
3. Compute join of Movie and producer, then filter.

Problem session

```
SELECT * FROM Movie
WHERE studioName LIKE 'D%' AND
      year>1980 AND year<1990;
```

- Consider two scenarios:
  - Indexes on each of `studioName` and `year`.
  - Composite index on `(studioName,year)`
- In each scenario, what are possible evaluation strategies?

Query plans in MySQL

- `EXPLAIN <Query>`
- Always sequence of ”select types”
  - Simple (part of outermost SELECT)
  - Derived (=subquery)
  - Dependent subquery (=correlated subquery) …
- Specification of algorithms used:
  - ref (eq_ref): select or index nested loop join using (primary) index
  - range: index is used for range query
  - index: index-only (covering index) evaluation
  - index_merge: RowID intersection
  - ALL: full table scan …

Processing selection

- We consider the conjunction (”and”) of a number of equality and range conditions.
- Two main cases:
  - No relevant index. (What is that?)
    In this case, a full table scan is required.
  - One or more relevant indexes.
    a) There is a highly selective condition with a matching index.
    b) No single condition matching an index is highly selective.

Using a highly selective index

- Basic idea:
  - Retrieve all matching tuples (few)
  - Filter according to remaining conditions

- If index is clustered or *covering*: Retrieving tuples is particularly efficient, and the index does not need to be highly selective.

Using several less selective indexes

- For several conditions C_1, C_2,... matched by indexes:
  - Retrieve the RIDs R_i of tuples matching C_i.
  - Compute the intersection $R=R_1 \cap R_2 \cap \ldots$
  - Retrieve the tuples in R (in sorted order).

- Remaining problem:
  - How can we estimate the selectivity of a condition? Of a combination of conditions?
  - More on this in ”Database Tuning” or the Algorithms specialization.

Operations that require grouping

- Many operations are easy to perform once the involved tuples (in one or more relations) are grouped according to the values of some attribute(s):
  - Join with equality condition (group by join attributes)
  - Groupings and aggregations (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)
- Most (but not all) database systems implement efficient grouping

Sort-based grouping

Usual sorting algorithms are not optimized for large data sets.

Need to limit the number of times data is read/written to address I/O bottleneck.

Two-pass merge sort:

- Read chunks of data into memory, and output each in sorted order.
- Merge all chunks, keeping one block from each in RAM.

Hash-based grouping

- Split data into many chunks based on *hash value* of grouping attribute(s).
- Read one chunk into memory at a time (assuming it fits), and perform grouping.
- Uses less memory that sort-based join if one relation is smaller than the other.

Index nested loop join

- If there is an index that matches the join condition, the following algorithm can be considered:
  - For each tuple in R_1, use the index to locate matching tuples in R_2.
- Better than grouping if $|R_1|$ is small compared to #memory blocks of R_2.
  - MySQL currently implements **only** this join algorithm and a naïve alternative.
- If many tuples match each tuple, a clustered or covering index is preferable.

Indexes affect join order

- Flights from South America today:

```
select region, count(*)
from flights,country,airport
where region='SA' and dep=airport and
  airport.country=country.country and
  start_op<='2015-10-7' and end_op>='2015-10-7';
```

- With only primary key indexes:
  - Must start with flights (date condition), then join city, then join country (use region='SA').
- With indexes on `city(country)` and `flights(dep)` the "reverse" order can be used.
  - May mean less data is considered.

Database tuning

Two main techniques:

- Adding indexes (already discussed)
  - Distinction between primary and secondary indexes.
  - Used for selection, and for index nested loop join.
  - Some queries can be evaluated using an **index only**.
- Changing the schema/physical storage:
  - Denormalization
  - Partitioning

Denormalization

- Normalization reduces redundancy and avoids anomalies
- Normalization can **improve** performance
  - Less redundancy => more rows/page => less I/O
  - Decomposition => more tables => more clustered indexes => smaller indexes
- The price of normalization:
  - Need to do more joins.
  - Fewer indexing possibilities.

Denormalization and indexing

- `Customer(cno,name,country,type)`
- `Invoice(ino,cno,amount,country)`

redundant attribute

$$\pi_{\text{name,type,ino,amount}}(Customer \bowtie \sigma_{\text{country=''Sweden''} \wedge \text{amount}>10000}(Invoice))$$

- Can make a *covering index* on `Invoice(country,amount,cno,ino)`.

Partitioning of Tables

- A table might be a performance bottleneck
  - If it is heavily used, causing locking contention (more on this later in course)
  - If its index is deep (table has many rows or search key is wide), increasing I/O
  - If rows are wide, increasing I/O
- Table partitioning might be a solution to this problem.

Horizontal Partitioning

- If accesses are confined to disjoint subsets of rows, partition table into smaller tables containing the subsets
  - Geographically (e.g., by state), organizationally (e.g., by department), active/inactive (e.g., current students vs. grads)
- Advantages:
  - Spreads users out and reduces contention
  - Rows in a typical result set are concentrated in fewer pages
- Disadvantages:
  - Added complexity
  - Difficult to handle queries over all tables

Vertical Partitioning

- Split columns into two subsets, replicate key
- Useful when table has many columns and
  - it is possible to distinguish between frequently and infrequently accessed columns
  - different queries use different subsets of columns
- **Example**: Employee table
  - Columns related to compensation (tax, benefits, salary) split from columns related to job (department, projects, skills).
- DBMS trend (analytics): **Column stores**, where *full* vertical partitioning is done.

Conclusion

This lecture was related to a course goal:

- After the course the student should be able to *decide if a given index is likely to improve performance for a given query*.

Also appetizer for database specialization:

- Database Tuning (spring semesters)
- Big Data Management (fall semesters)