Data Storage and Formats, Fall 2008

Lecture 10
Query processing

Rasmus Pagh

Literature: KBL 10 and 12.2
(slide 30-35 from KBL, © Addison-Wesley)
Today’s lecture

• Problem session: Pretend to be a DBMS!
• DBMS query evaluation algorithms
  – External merge-sort
  – Algorithms for select, join, grouping,...
  – Use of indexes
• Query optimization, by example
• Making use of this knowledge:
  – Schema tuning
• ITU research on query evaluation (SQERD)
Sequential vs random access

• Recall from indexing lecture:
  – The speed of RAM access is 5-6 orders of magnitude higher than disk access (non-sequential).
  – The speed of reading RAM sequentially is 1-2 orders of magnitude higher than sequential disk access.

• RAM vs disk analogy:
  – Fly to Australia to borrow a cup of sugar if your neighbor is not home!
  – To pay for the trip, you’d better bring a lot of sugar!
Sequential vs random access

- Sequential is also better in RAM.
- My laptop can, in 1 second:
  - Perform up to 20 billion CPU instructions
  - Read 0.8 billion 4-byte words, sequentially
  - Read 0.034 billion words, random access

```
Rasmus-Paghs-computer-2:~/papers/sse/src pagh$ ./linear Generating 8388608 pointer-value pairs
100 linear probing sequences begin
End: 838860700
1.000000 seconds

100 double hashing sequences begin
End: 1677721400
24.320000 seconds
```
Problem session

Suppose you are a DBMS, and that all the relations queried below are very large (residing on external memory). They are stored as a sequence of tuples. How would you process each one of the queries?

1. SELECT *
   FROM Movie
   WHERE studioName = 'Disney' AND year = 1990;

2. SELECT *
   FROM (SELECT *
             FROM Movie
             WHERE studioName = 'Disney') S
   WHERE year = 1990;

3. SELECT *
   FROM Movie, MovieExec
   WHERE Movie.title = 'Star Wars' AND Movie.producerC = MovieExec.cert;

4. SELECT *
   FROM Ships, Classes, Outcomes
   WHERE (Outcomes.ship = Ships.name)
         AND (Classes.class = Ships.class);

5. SELECT *
   FROM Movie
   WHERE studioName LIKE 'M%' AND year>1980 AND year<1990;

6. SELECT *
   FROM Movie
   WHERE NOT EXISTS
       (SELECT *
        FROM Movie M
        WHERE M.year > Movie.year);
Recap: Choosing to use an index

• 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

• Estimating selectivity is done using statistics
  – In MySQL, statistics is gathered by executing statements such as ANALYZE TABLE actorInfo
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 (quicksort is not quick on disk!),
  – hashing, or
  – using existing indexes

• The DBMS knows the characteristics of each approach, and attempts to use the best one in a given setting.
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 help make query execution faster.
Next: Sorting

Why study sorting?

1. Basis for many efficient algorithms, especially in blocked memory.

2. Reminds us that massive data is a different world:
   - Bucket sorting may be worse than superlinear algorithms.

3. Sorting algorithms is a popular topic to test candidates in job interviews (in US). From “job interview” at Google: [http://www.youtube.com/watch?v=k4RRi_ntQc8](http://www.youtube.com/watch?v=k4RRi_ntQc8)
Analysis of disk-based algorithms

Two worlds:

- Assumption made in this lecture:
  - The algorithm decides when to read and write blocks (pages).

- DBMSs (and operating systems):
  - A buffer manager decides what pages are kept in memory.
  - Sometimes the buffer manager may be forced to write a page to disk.
  - Algorithms may prioritize data (memory is split into buffer pools).
Improving merge sort

• Make better use of internal memory:
  
  *Merge many runs instead of 2 (board)*

• In almost all settings: 2 phases!
  – Each piece of data read and written once.
Relational algebra operations

• Relational DBMSs compute query results by performing a sequence of relational algebra operations:
  – Selections ($\sigma$)
  – Projections ($\pi$)
  – Joins (⋈)
  – Groupings and aggregations ($\gamma$)
  – Set operations ($\cup, \cap, -$)
  – Duplicate elimination ($\delta$)

• Next: A bit on how to perform each operation.
Selection

- We focus on 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

• *Examples from problem session:*

1. SELECT *
   FROM Movie
   WHERE studioName = 'Disney'
   AND year = 1990;

2. SELECT *
   FROM (SELECT *
         FROM Movie
         WHERE studioName = 'Disney') S
   WHERE year = 1990;
Using several less selective indexes

- For several conditions $C_1, C_2, \ldots$ 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 DBT (spring 2010?)
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):
  - Projections (group by output attributes)
  - Join with equality condition (group by join attributes)
  - Groupings and aggregations (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)
Two principles for grouping

• Sorting the tuples
  – All the mentioned operations can be performed during the final phase, i.e., no need to materialize the sorted list.

• Hashing the tuples
  – Hash to as many buckets as memory allows (need one output buffer for each bucket).
  – If each bucket fits in memory, grouping can be done by reading each bucket.
Pros and cons

• Sorting-based grouping is *deterministic*, i.e., no chance of bad behaviour.

• Sorting-based grouping outputs the result in sorted order
  – For union, intersection, and projection we may freely choose the order.

• Hashing-based grouping uses less memory for joins.
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 $R_2$.

• In general, the cost is at least 1 I/O (hash index) for each tuple in $R_1$.
  – Use only if $|R_1|$ is small compared to $B(R_2)$.
  – (Why are we interested in small relations?)

• If many tuples may match each tuple, a clustered index is preferable.
Corellated subqueries

- When a subquery refers to a tuple variable from the surrounding query, it will often be executed many times.
  - The SQL programmer should consider if the corellation can be avoided.
  - **Example:** Rewriting

```
SELECT * FROM Movie
WHERE NOT EXISTS
  (SELECT * FROM Movie M
   WHERE M.year > Movie.year);
```
Problem session

• How can we avoid corellation in:
  
  ```sql
  SELECT ssnum
  FROM Employee E1
  WHERE salary = (SELECT max(salary)
                  FROM Employee E2
                  WHERE E1.dept=E2.dept)
  ```

• **Hint**: Create a temporary table.
Summary

• Several algorithms possible for each operator:
  – Use index or not (selection, join)?
  – Use several indexes (selection)?
  – Sort- or hash-based?

• Choosing the best, or at least “sufficiently good”, is complicated in general.
  – We will not go into how this is done, but simply give an illustrative example.
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 evaluation
  - index_merge: RID intersection
  - ALL: full table scan ...
Query optimization example

Schema:

• Sailors(sid, sname, rating, age)
  – 40 bytes/tuple, 100 tuples/page, 1000 pages

• Reserves(sid, bid, day, rname)
  – 50 bytes/tuple, 80 tuples/page, 500 pages

Query:

```
SELECT S.sname
FROM (Reserves NATURAL JOIN Sailors)
WHERE bid=100 AND rating>5
```
Example, cont.

- Simple logical query plan:

\[ \pi_{sname}(\sigma_{bid=100 \land rating > 5}(Reserves \bowtie Sailors)) \]

- Physical query plan:
  - Nested loop join.
  - Selection and projection "on the fly" (pipelined).

- Cost: Around 500*1000 I/Os.
Example, cont.

• New logical query plan (*push* selects):
  \[ \pi_{sname}(\sigma_{bid=100}(Reserves) \bowtie \sigma_{rating>5}(Sailors)) \]

• Physical query plan:
  – Full table scans of Reserves and Sailors.
  – Sort-merge join of selection results

• Cost:
  – 500+1000 I/Os plus sort-merge of selection results
  – Latter cost should be estimated!
Example, cont.

• Another logical query plan:

\[ \pi_{sname}(\sigma_{\text{rating}>5}(\sigma_{\text{bid}=100}(\text{Reserves} \bowtie \text{Sailors}))) \]

• Assume there is an index on bid and sid.

• Physical query plan:
  - Index scan of Reserves.
  - Index nested loop join with Sailors.
  - Final select and projection “on the fly”.

• Cost:
  - Around 1 I/O per matching tuple of Reserves for index nested loop join.
More MySQL query plans

We consider again how MySQL executes the queries from the problem session.

1. SELECT *
   FROM Movie
   WHERE studioName = 'Disney' AND year = 1990;

2. SELECT *
   FROM (SELECT *
       FROM Movie
       WHERE studioName = 'Disney') S
   WHERE year = 1990;

3. SELECT *
   FROM Movie, MovieExec
   WHERE Movie.title = 'Star Wars' AND Movie.producerC = MovieExec.cert;

4. SELECT *
   FROM Ships, Classes, Outcomes
   WHERE (Outcomes.ship = Ships.name)
     AND (Classes.class = Ships.class);

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

6. SELECT *
   FROM Movie
   WHERE NOT EXISTS
     (SELECT *
      FROM Movie M
      WHERE M.year > Movie.year);
Next: tuning

Two main techniques:

• Adding indexes (already discussed)
  – Important 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
Normalization

- Normalization can **decrease** performance.

**Example:** Transcript\((StudId, CrsCode, Semester, Grade)\)
  - Functional dependency: \(StudId \rightarrow Name\)
  - Key of Transcript = \((StudId, CrsCode, Semester)\)
  - If \(Name\) were an attribute of Transcript it would not be in BCNF or 3NF, *but* ...
  - a join required to list names of students with A in CS305

```
SELECT S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
   AND T.Grade = 'A'
```

- ... and join is expensive
Denormalize

• Add attribute Name to Transcript

```
SELECT T.Name
FROM Transcript T
WHERE T.CrsCode = 'CS305' AND T.Grade = 'A'
```

• Join avoided, but added redundancy:
  - Slows data modification (update to redundant attribute has to be performed in two tables)
  - Increases size of table
  - Introduces the possibility of inconsistent data
Partitioning of Tables

• A table might be a performance bottleneck
  – If it is heavily used, causing locking contention
  – If it’s 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:** Column stores, where full vertical partitioning is done.
Next

• ITU research in databases:
  - Scalable computation of acyclic joins.
  - Binary joins - when can it be done faster?
  - Hashing on external memory:
    • Can it be improved?
    • \textit{Without} a hash function?
  - Algorithms for combinations of relational algebra operations:
    • 3-way joins
    • Join+project
Acyclic joins

• **Intuition.** Consider the graph with:
  - Relations as vertices.
  - An edge between two vertices if there is an equality condition between the relations.

• This is called the *join graph*.

• If the join graph is *acyclic*, there is an efficient procedure for eliminating all "dangling" tuples that will not contribute to the final result.
Complexity of acyclic join

- Suppose we want to join $k$ relations of $n$ blocks in total, and that the final result has size $z$ blocks.
- What can we say about the I/O cost in general, for the classical query optimization approach?
- **Bad case:** Star schema. Computation may require $\Omega(zk)$ I/Os.
- **Result [PP06]:** Can always get by with $O(n+z)$ I/Os, assuming 2-pass sorting.
Hashing without a hash function?

- **Problem with hashing:** May fail to have good performance. Would like deterministic data structure with same performance.
- Most experts believe this is not possible on a “normal computer”.
- But what if we have more resources - e.g., an array of 64 disks that we can access in parallel? (Such storage systems exist.)
Deterministic load balancing [BHPPRT06]

Idea:

- Associate (in a clever way) with each possible key $x$ a **fixed** set of blocks where it may be stored.
- One possible block on each disk - lookup in 1 parallel I/O.
- When inserting a key $x$, check all possible positions (in parallel) and insert $x$ in the *least full* block.
- This **always** works if (roughly):
  - Blocks have room for $\log N$ keys.
  - There are $\log(#\text{possible keys})$ disks.
Better query evaluation algorithms
On-going work with Rasmus Resen Amossen

• Consider SQL statements such as:
  – `SELECT DISTINCT a,c FROM R1, R2 WHERE R1.b=R2.b;`
  – `SELECT * FROM R1,R2,R3 WHERE R1.b=R2.a AND R2.b=R3.a AND R3.b=R1.a`

• It turns out that traditional DBMSs (that evaluate queries using a sequence of relational algebra operations), can be outperformed.
  – At least in worst case...
  – Practical impact: Future work!