Introduction to Database Systems

Intro to Query Processing

Literature: PDBM 13.1

Björn Þór Jónsson

IT UNIVERSITY OF COPENHAGEN

First Things First

- **All exercises now ONLINE on Discord!**
  - See post: https://piazza.com/class/kdy92u8mvyh2s6?cid=29
- Homework 3 due tomorrow
  = DDL + Normalization + Indexing (+ SQL)
  - Question 2.1: See video: https://use.vg/DtvIyrpjS7Zm
- **Homework 2: Review next Monday at 18:00**
  - On the regular Zoom meeting!
- Now, let's finish last week's slides!

Recap: Indexing

- Indexes are data structures that facilitate access to data from disk
  - … if conditions are a prefix of indexed attributes
  - *Clustered* indexes store tuples that match a range condition together
  - Some queries can be answered looking **only** at the index (a *covering* index for query)
  - Indexes slow down updates and insertions
- 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 name indexes when writing queries

Today's Lecture

Part A:

- Processing simple selections
- Processing complex selections
- Intro to join evaluation algorithms
- (Super brief) Intro to grouping/aggregations

Part B:

- Short intro to database tuning
  - A very interesting and important topic!
  - Not necessary for the course, and will not get to it
  - I'll leave the slides in for those interested

Part A: Query Processing

Processing Simple Selections

- We discussed this earlier, to summarise:

- Point and range queries on the attribute(s) of the **clustered index** are almost always best performed using an index scan
- **Unclustered** indexes should only be used with **high selectivity queries**
- Exception: **Covering index** is good for any selectivity
- If no index exists, a full table scan is required!

Processing Complex Selections

- We consider the conjunction (”and”) of equality and range conditions.
- No relevant index: Full table scan
- One index relevant:
  - Highly selective: Use that index
  - If not: Full table scan
- Multiple relevant indexes:
  - One is highly selective: Use that index
  - No single condition matching an index is highly selective: Can “intersect” the returned sets

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 addresses R_i of tuples matching C_i.
    - The addresses are in the index leaves!
  - Compute the intersection $R = R_1 \cap R_2 \cap \ldots$
  - Retrieve the tuples in R from disk (in sorted order)

- Remaining problem:
  - How can we estimate the selectivity of a condition? Of a combination of conditions?
  - Use some stats and probabilistic assumptions…

Example

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine *two* indexes to identify rows fulfilling both conditions.
5. Use *one* composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 *and* their titles in a composite *covering* index.

Example

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine *two* indexes to identify rows fulfilling both conditions.
5. Use *one* composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 *and* their titles in a composite *covering* index.

```
CREATE INDEX yearIdx
ON Movie(year)
```

```
CREATE INDEX studIdx
ON Movie(studioName)
```

```
CREATE INDEX yearStudIdx
ON Movie(year, studioName)
```

```
CREATE INDEX coveringIdx
ON Movie(year, studioName, title)
```

Example – Variant 1

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

Available Indexes:

```
CREATE INDEX yearIdx
ON Movie(year)

CREATE INDEX studIdx
ON Movie(studioName)
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine *two* indexes to identify rows fulfilling both conditions.
5. Use *one* composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 *and* their titles in a composite *covering* index.

Which strategies are possible and which index would be used?

Example – Variant 2

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine *two* indexes to identify rows fulfilling both conditions.
5. Use *one* composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 *and* their titles in a composite *covering* index.

Available Indexes:

```
CREATE INDEX yearIdx
ON Movie(year)
```

```
CREATE INDEX yearStudIdx
ON Movie(year, studioName)
```

Which strategies are possible and which index would be used?

Example – Variant 3

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine *two* indexes to identify rows fulfilling both conditions.
5. Use *one* composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 *and* their titles in a composite *covering* index.

Available Indexes:

```
CREATE INDEX yearIdx
ON Movie(year)
```

```
CREATE INDEX yearStudIdx
ON Movie(year, studioName)
```

```
CREATE INDEX coveringIdx
ON Movie(year, studioName, title)
```

Which strategies are possible and which index would be used?

Example – Variant 4

```
SELECT title
FROM Movie
WHERE year = 1990
  AND studioName = 'Disney';
```

Examples of strategies:

1. Make a scan of the whole relation.
2. Find movies from 1990 using index, then filter.
3. Find Disney movies using index, then filter.
4. Combine *two* indexes to identify rows fulfilling both conditions.
5. Use *one* composite index to find Disney movies from 1990.
6. Find Disney movies from 1990 *and* their titles in a composite *covering* index.

Available Indexes:

CREATE INDEX idIdx
ON Movie(id)

CREATE INDEX titleIdx
ON Movie(title)

Which strategies are possible and which index would be used?

Processing Complex Selections Revisited

- We have considered the conjunction (”and”) of a number of equality and range conditions.

- What about disjunctive (”or”) selections?
  - One full table scan
    OR
  - Multiple “and” queries

Query Evaluation in a Nutshell

- SQL rewritten to (extended) relational algebra
- The building blocks in DBMS query evaluation are algorithms that implement relational algebra operations.
  - **Join** is the most important one!
- May be based on:
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- The DBMS optimizer knows the characteristics of each approach, and attempts to use the best one in a given setting

Join Evaluation in a Nutshell

- Join is the most important operation!
- May be based on:
  - Reading everything / Sorting / Hashing
  - Using indexes can sometimes help!
- We consider a simple join:

```
R JOIN S ON S.ID = R.ID
```

  - Extends to more complex joins in a straightforward way

Nested Loops Join

- The following basic algorithm can be used for **any** join:

for each tuple in R
    for each tuple in S
        if r.ID = s.ID
        then output (r, s)

- If the join condition is complex/broad, sometimes this is the only/best choice

```
R JOIN S ON S.ID <> R.ID
```

Role of Index in Nested Loops Join

- If there is an index that matches the join condition, the following algorithm can be considered:
  - For each tuple in R
    use the index to locate matching tuples in S
- Good if |R| is small compared to |S|
- If many tuples match each tuple, a clustered or covering index is preferable.

- MySQL currently implements only this join algorithm and the previous naïve alternative.

Example

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

Some 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. NL join `Movie` and `Producer`, then filter.

Problem session

What would be good indexes for these queries?

```
1. SELECT firstNames
   FROM person
   WHERE gender='m'
     AND firstnames LIKE 'Maria%’;

2. SELECT A.street, A.streetno
   FROM person P
      JOIN address A ON A.person_id=P.id
   WHERE P.lastname='Bohr’
     AND P.firstnames LIKE 'Niels%';
```

Merge Join

Consider `R JOIN S on R.ID = S.ID`

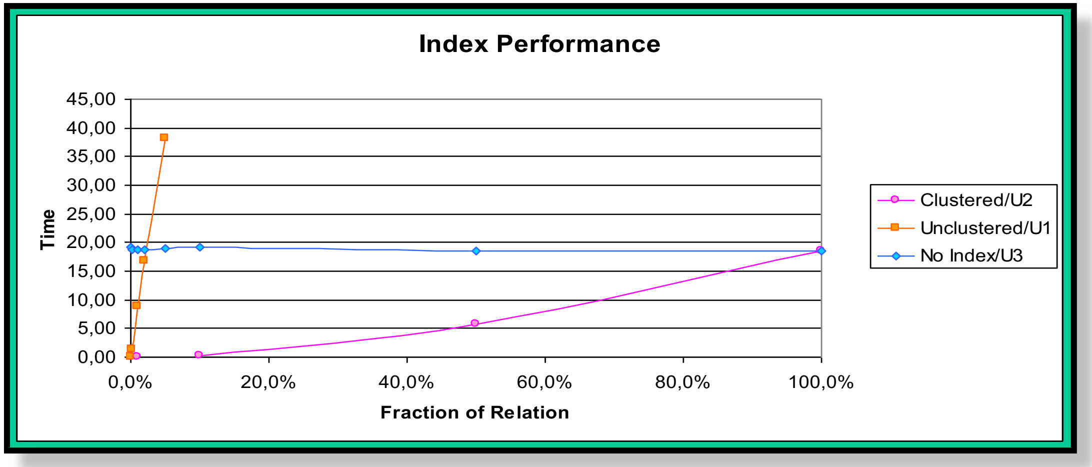
- Step 0: Sort R and S on ID
- Step 1: Merge the sorted R and S

- Cost:
  - If already sorted: $O(|R| + |S|)$
  - Can we do better?

  - If not sorted:
    $O(|R| \log |R| + |S| \log |S| + |R| + |S|)$

Role of Indexes in Merge Joins

- Indexes can be used to read data in sorted order
- When is this a win?
  - Index is clustered
  - Index is covering
- When is this a loss?
  - Index is unclustered



Hash Join

Consider `R JOIN S on R.ID = S.ID`

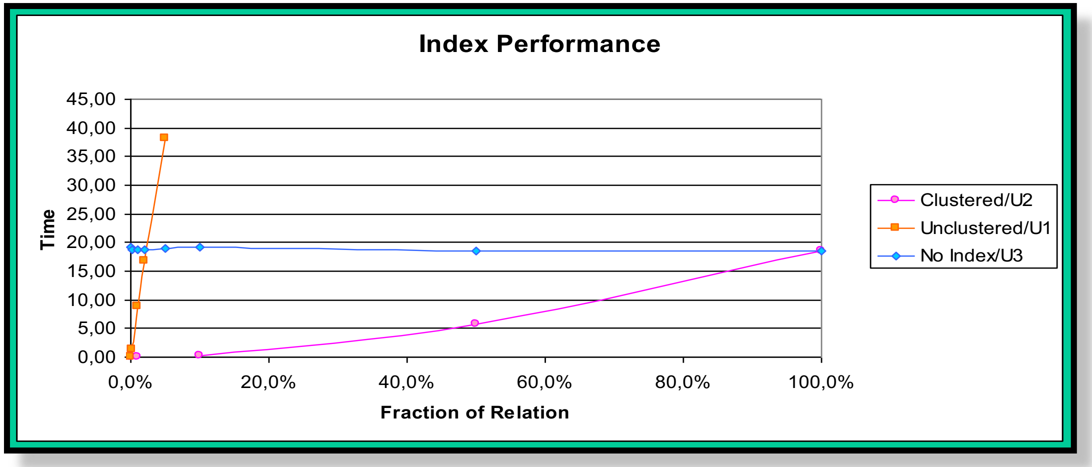
- **Best if S fits in RAM**

- Step 0: Create a good hash function for ID
- Step 1: Create a hash table for S in memory
- Step 2: Scan R and look for matching tuples in the hash table

- Cost: O(|R| + |S|)
  - Can we do better?
  - What is S does not fit in RAM?

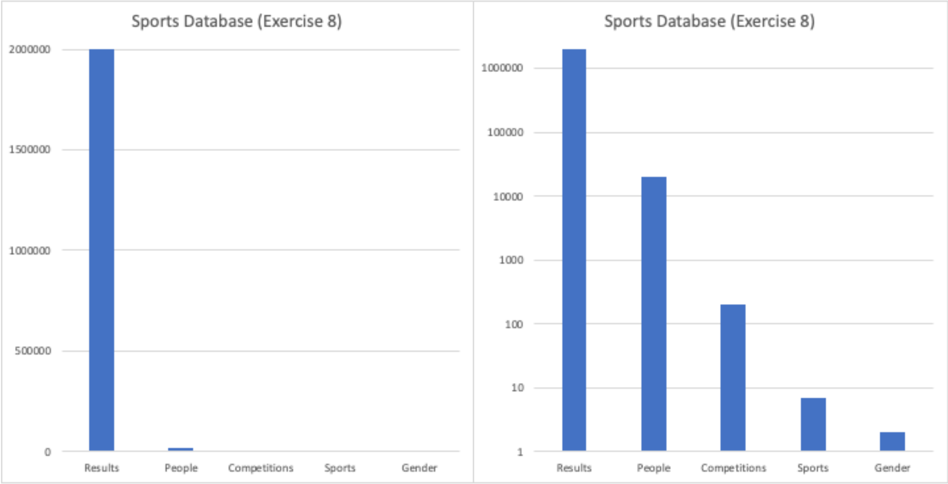
Role of Indexes in Hash Joins

- Hash joins read all the relations
- How can indexes be useful?
  - Apply to non-join conditions
  - Index is covering



Comparison of Join Algorithms

- Nested loops join:
  - Very costly O(|R|*|S|)
  - Works for any condition → sometimes only option
- Merge join:
  - Works well if data is well clustered
  - Works well if relations are large and similar in size
- Hash join:
  - Works well if one relation is small
  - Is that often the case?



Grouping Operations

- Many operations are based on grouping records (of one or more relations) according to the values of some attribute(s):
  - Join (group by join attributes, see above)
  - Group by and aggregation (obvious)
  - Set operations (group by all attributes)
  - Duplicate elimination (group by all attributes)
- Most database systems implement such grouping efficiently using sorting or hashing

Part B: Intro to Database Tuning

Database Tuning

- Done: ER design, database schema, views
  - Logical schema
  - External schema
- Next: Reach performance goals
  - Before throwing hardware at it...
  - ... consider the physical schema

<u>Two main tuning techniques</u>:

- Adding indexes
- Changing the schema/physical storage

Need to Understand the Workload!

- The major queries + frequency
  - Which tables and columns are read
  - Which columns appear in selections/joins? What is the likely reduction factor of the selections?
  - Why? Because indexes can speed up queries
    - Find records quickly !

- The major updates + frequency
  - As before +
  - Update type (UPDATE/INSERT/DELETE) and updated columns
  - Why updates? Because indexes can speed up OR slow down updates

Query Optimization vs 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 Opt vs Query Tuning

Query Optimisation

- SELECT a
  FROM R
  WHERE b IN (
  SELECT b FROM S);

- SELECT DISTINCT a
  FROM R, S
  WHERE R.b = S.b;

From Lecture 3

Query Tuning

- select R.peopleID, R.sportID, R.result
  from Results R
  where R.result = (
  select max(R1.result)
  from Results R1
  where R1.sportID = R.sportID));

- select R.peopleID, R.sportID, R.result
  from Results R
  where (R.sportID, R.result) in (
  select R1.sportID, max(R1.result)
  from Results R1
  group by R1.sportID));

From Homework 1

Indexing Decisions

- Starting point:
  - Keys (primary key / unique) are enforced using indexes
- Read/Write ratios
  - OLTP = 60/40 ←→ OLAP = 98/2

- Which tables should be indexed?
- Should we create multiple indexes?
- Which columns should become search keys?
- Should the indexes be clustered or not?

Choosing Columns

- Candidates for index search keys
  - Columns in WHERE clauses
  - Columns in GROUP BY clauses
  - Columns in ORDER BY clauses
- Columns that are rarely candidates
  - Large columns (too much space)
  - Frequently updated columns (too much maintenance)
  - Columns in SELECT clauses (not used to find tuples)
    - ... but see covering indices!

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.

```
SELECT S.Name, T.Grade
FROM   Student S, Transcript T
WHERE  S.Id = T.StudId
  AND  T.CrsCode = 'CS305'
  AND  T.Semester = 'S2002'
```

Denormalization

- **Tradeoff:** *Judiciously* introduce redundancy to improve performance of certain queries
- **Example:** Add attribute *Name* to Transcript

```
SELECT T.Name, T.Grade
FROM   TranscriptDenorm  T
WHERE  T.CrsCode = 'CS305'  AND  T.Semester = 'S2002'
```

  – Join is avoided
  – If queries are asked more frequently than Transcript is modified, added redundancy might improve average performance
  – But, Transcript′ is no longer in BCNF since key is (*StudId, CrsCode, Semester*) and *StudId* → *Name*

Problem Session

- Schema:
  - `Customer(cno, name, country, type)`
  - `Invoice(ino, cno, amount)`
  - Additional indexes on `country` and `amount`.
- What are possible query plans for this?
- Can denormalization help?

```
SELECT ino, amount
FROM customer, invoice
WHERE country="Sweden"
  AND amount > 10000
  AND customer.cno = invoice.cno
```

Denormalization

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

redundant attribute

- Query can run on the new `Invoice`:

```
SELECT ino, amount
FROM invoice
WHERE country=“Sweden”
  AND amount > 10000
```

- What speaks against such denormalization?

Materialized/Indexed Views

- Some systems (including PostgreSQL) allow storing (materializing) views as tables
  - System keeps the view up to date
  - System automatically chooses to use the view

```
CREATE MATERIALIZED VIEW StudTransJoin
AS
SELECT S.Name, T.Grade
FROM   Student S, Transcript T
WHERE  S.Id = T.StudId
```

```
SELECT S.Name, T.Grade
FROM   Student S, Transcript T
WHERE  S.Id = T.StudId
  AND  T.CrsCode = 'CS305'
  AND  T.Semester = 'S2002'
```

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, organizationally, active/inactive
- 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 (for analytics):
  - **Column stores**, with *full* vertical partitioning.
  - More on this next week.

Take Aways

- Performance difference between well-tuned and poorly-tuned applications can be massive!
- The DBMS does its best to optimize queries, but sometimes it needs help!
  - Query tuning – rewrite as joins or non-correlated subqueries
  - Indexes – solve 90+% of all other performance problems
- If that is not sufficients:
  - Materialized views / Partitioning / Denormalization
  - Beyond the scope of this course!