Overview of Query Processing

Query

Query Compiler

Execution Plan

Query Engine

Result

Query processing flow diagram:
- Query
- Query Compiler
- Execution Plan
- Query Engine
- Result
Compilation

- SQL is declarative
  - Query has to be translated into a procedural program that can be run on the query engine
- DBMSs translate SQL into another format
  - A widely-used approach is the translation into the operations of relational algebra
- relational algebra includes operations such as
  - select tuples from $R$ that satisfy predicate $p$: $\sigma_p(R)$
  - project tuples from $R$ over attributes $A_1, \ldots, A_n$: $\pi_{A_1,\ldots,A_n}(R)$
  - join relation $R$ and $S$ based on predicate $p$: $R \bowtie_p S$
  - take the Cartesian product of $R$ and $S$: $R \times S$
Canonical Translation

There is a standard way of translating SQL into relational algebra.

Expressions in relational algebra are often represented graphically.

```
select  A_1, \ldots, A_n 
from    R_1, \ldots, R_m 
where  p
```
Optimisation

- Canonical plan is not very efficient (e.g. it contains Cartesian products)
- Most DBMSs have query optimisers rewriting the plan to make it more efficient
- Query rewriting and optimisation is a complex problem
Why bother about query optimisation as an ordinary user?

Sometimes the query optimiser produces bad plans

Nevertheless, most DBMSs allow a user to take a look at the generated execution plans

Plans can be analysed and query can be changed to make it more efficient

Most systems implement the SQL “explain” statement which returns an explanation of the query execution plan
Consider the query:
```
select pname, sname, jname
from Part, Supplier, Project, Supply
where Part.pnum=Supply.pnum and
Supplier.snum=Supply.snum and Project.jnum=Supply.jnum;
```

We can simply add `explain` before `select`

Output tells us how MySQL will execute the query:

- Read each row in `Project`
- Use an index on `jnum` in `Supply` to find matching rows in `Supply`
- Use an index on `pnum` in `Part` to find matching row in `Part` (primary key)
- Use an index on `snum` in `Supplier` to find matching row in `Supplier` (primary key)
Query Optimisation

- A DBMS can estimate the execution costs of a plan with the help of
  - cost models
  - statistics
- Investigating all possible plans is usually too expensive
- An optimiser uses heuristics to optimise queries
- Optimisation can take place on different levels:
  - Logical level
  - Physical level
Logical Level

- Starting point is the canonical relational algebra expression
- What can we do?
  - Transformation of algebraic expression into equivalent ones (ideally into faster ones)
  - A rule of thumb is to minimise the input/output of the individual operators
Logical Level (2)

- Basic techniques
  - Breaking up selections
  - Pushing “down” selections
  - Converting selection and Cartesian products into joins
  - Determining join order

- The tree (plan) is processed bottom-up (from leaves to root)

- So pushing down selections reduces the size of intermediate results as they pass up the tree
Example Query

```
select S.name, P.name
from student S, attends A, lecture L, professor P
where S.stud_no = A.stud_no
and A.course_no = L.course_no
and L.prof_pers_no = P.pers_no
and S.date_of_birth > 1991-01-01
and P.name = 'G..;
```
(Optimised) Query Plan

\[
\pi_{S.name, P.name} \\
\sigma_{S.date_of_birth > 1991-01-01} \land \land A.stud_no = S.stud_no \\
\land L.cno = A.cno \land P.pers_no = L.prof_pers_no \\
\sigma_{P.name='G.'} \land L \\
P
\]
Physical Optimisation

- Up to now, we have only looked at relational algebra operators on a logical level
- However, an operator can be implemented in different ways
- Optimisation is not only about rewriting the plan, but about choosing an implementation for each one
Physical Optimisation (2)

- Other aspects of physical optimisation include:
  - Decide whether to use *indexes* or not (and which ones)
  - Whether to materialise intermediate results or not
  - When to sort and eliminate duplicates
  - Which cost models and statistics to use
Benefit of using an index

Consider an organisation, such as Amazon, with hundreds of millions of products

Say the database system needs to retrieve a single product with \textit{id} value 123 from a table

It could

1. \textit{scan} the table, one row at a time, checking each time if $id = 123$
2. use an \textit{index} on \textit{id} to find the row(s) with value 123

first method costs hundreds of millions of operations

second method (typically) costs a few operations
Hierarchical Indexes

- There are many different hierarchical index structures
- We will consider only $B$-trees (and $B^+$-trees)

Recall that database data is stored on disk in fixed size blocks or *pages*

- Typically a large number of tuples will fit into a single page
- Main idea is:
  - Sort tuples according to indexed attribute
  - Arrange them into a hierarchy of pages, leaving space for insertions
B-Tree Example

► Let’s have a look at an example:

Let’s search for the data item 21 in the tree
B-Tree Search

- We start at the root of the tree, looking for 21
- 21 is less than 34 so we move to the left child
- 21 is between 16 and 26, so we follow that pointer to the page starting with item 17
- We now find 21 as the third item on this page
Properties of a B-tree nodes

Assume we have an inner node looking like this:

\[ p_0 \quad k_1 \quad p_1 \quad k_2 \quad \ldots \quad k_n \quad p_n \]

where \( p_j \) is a pointer, \( k_j \) a (key) value

Then the following holds:

- The subtree pointed to by \( p_0 \) contains values less than \( k_1 \)
- \( p_j \) points to a subtree with values between \( k_j \) and \( k_{j+1} \)
- The subtree referenced by \( p_n \) contains values greater than \( k_n \)
B-Tree Insertion

Consider our example tree:

Let’s insert the data item 14 into the tree
B-Tree Insertion (2)

This leads to a page overflow: median value (12) on the page is pushed upwards and page is split:

Let's insert another data item: 20
B-Tree Insertion (2)

- This leads to a page overflow: median value (12) on the page is pushed upwards and page is split:

Let’s insert another data item: 20
B-Tree Insertion (3)

- This time, we have a page overflow on an index page
  - Treated in the same way: push median up, split page
Properties of a B-tree

- Every path from the root to a leaf has the same length
  - So it’s a balanced tree

- Every node (except the root) has at least $i$ and at most $2i$ entries
  - In the previous examples, $i = 2$

- The entries in every node are sorted

- Every node with $n$ entries (except leaves) has $n + 1$ children
B-tree Deletion

- Deletion of a value in a leaf is simple: just delete the value.
- An inner node needs to stay properly connected to its children.
- For example, we want to delete the value 16 from an inner node (distinguishes values in leaf nodes):

```
3 5 8 9 12 15 17 19 21 25 32 33
    7 16 26
  3 5 8 9 12 15
    17 19 21 25
       32 33
```
B-tree Deletion (2)

- Solution: we delete it and move up the next bigger value from the right child node

- Alternatively, we can move up the previous smaller value (15) from the left child node
B-tree Deletion (3)

▶ After the deletion of a value, a node can have too few entries (fewer than $i$)
▶ In that case we can merge a node with one of its neighbours (pulling down a value from the parent node)
  ▶ This can cause a parent node to have too few entries, i.e., we have to merge the parent node with a neighbour
  ▶ We’re not going into details here
    ▶ As databases tend to grow (and not shrink), many databases don’t even implement this merging
B+-Trees

- B+-trees are an important variant of B-trees
- The performance of a B-tree depends heavily on the height of the tree
  - The deeper a tree, the more page lookups (on secondary storage) we need to reach a leaf
- So what can we do to “flatten” B-trees?
B⁺-Trees (2)

- If we can increase the branching (number of pointers) in inner nodes, then the tree will become “flatter”
- Instead of storing data in inner nodes, we only store search keys (take up less space ⇒ more room for pointers)
- We also link all the leaf nodes, allowing a fast sequential search
Schema of a $B^+$-Tree

$p_i$ = pointer

$S_i$ = search key

$R_i$ = record

entry point

sequential search

$p_0$, $S_1$, $p_1$, $S_2$, ..., $S_n$, $p_n$, free

$R_1$, ..., $R_j$, free
Implementing Selection

- We want to select tuples from a relation $R$ satisfying predicate $p$, e.g., $p$ is $\text{age} \geq 60$
- Two usual methods
  - we can fetch each tuple in turn and check if it satisfies predicate $p$
  - we can use an index if one exists, e.g., on $\text{age}$
Implementing Join

- Say we want to join \( R \) and \( S \) based on \( R.A = S.B \)
- An implementation that always works is a nested-loop join:

\[
\text{for each } r \in R \\
\quad \text{for each } s \in S \\
\quad \quad \text{if } r.A = s.B \text{ then} \\
\quad \quad \quad \text{res} := \text{res} \cup (r \times s)
\]
Implementing Join

- Say we want to join \( R \) and \( S \) based on \( R.A = S.B \)
- An implementation that always works is a nested-loop join:

\[
\text{for each } r \in R \\
\quad \text{for each } s \in S \\
\quad \quad \text{if } r.A = s.B \text{ then} \\
\quad \quad \quad \text{res} := \text{res} \cup (r \times s)
\]

- But this is expensive
- If \( R \) has \( n \) tuples and \( S \) has \( m \) tuples, then at least \( n \times m \) operations
(Sort-)Merge-Join

- Prerequisite: $R$ and $S$ are sorted
  - You may need to do a sort operation before joining

Example:

<table>
<thead>
<tr>
<th>$R$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>$B$</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

-pointer$_r$     pointer$_s$-

$R$ and $S$ are sorted.

You may need to do a sort operation before joining.

MySQL example:

You can use MySQL to perform joins efficiently.
Index-Join

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th>S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... → B-tree
MySQL example

- Recall the MySQL plan from earlier:
  - Read each row in Project
  - Use an index on jnum in Supply to find matching rows in Supply
  - Use an index on pnum in Part to find matching row in Part (primary key)
  - Use an index on snum in Supplier to find matching row in Supplier (primary key)

- Uses index-join

- If indexes don’t exist, it falls back on nested-loop join
Estimating Costs

- In order to make informed decisions, an optimiser has to employ cost models.
- A cost model estimates the cost of an execution plan based on a number of input parameters:
  - the relational algebra expression
  - data on indexes
  - size of relations
  - attribute value distributions
  - ...
- Estimating the execution time is far from trivial.
MySQL Example

- Two tables representing wi-fi usage at a number of locations
  - wifiDetails table: information about wi-fi usage
  - locations table: information about locations that users visit
- wifiDetails has 3671218 rows
- locations has 204 rows
- wifiDetails has attributes such as: day, hour, minutes, seconds, User_Email, Calling_Station_Id, Called_Station_Id, Aruba_Location_Id
- locations has attributes: Id, Aruba_Location
MySQL Example (2)

- The example query joins the two tables to obtain the name of each location and the corresponding number of visitors:

```
SELECT t2.Aruba_Location as Location,
       COUNT(DISTINCT t1.Calling_Station_Id) as NumOfUsers
FROM wifiDetails t1, locations t2
WHERE t1.Aruba_Location_Id = t2.Id
GROUP BY t2.Aruba_Location;
```

- Plan 1 takes 1 min 38 secs = 98 secs
- If add an index to wifiDetails as follows:
  ```
  CREATE INDEX Aruba_Location_Id_index ON wifiDetails (Aruba_Location_Id) USING BTREE;
  ```
- Plan 2 takes 18 secs (over 5 times faster)
MySQL Example — Plan 1

<table>
<thead>
<tr>
<th>table</th>
<th>t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>index</td>
</tr>
<tr>
<td>key</td>
<td>PRIMARY</td>
</tr>
<tr>
<td>ref</td>
<td>NULL</td>
</tr>
<tr>
<td>rows</td>
<td>3671218</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>table</th>
<th>t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>eq_ref</td>
</tr>
<tr>
<td>key</td>
<td>PRIMARY</td>
</tr>
<tr>
<td>ref</td>
<td>t1.Aruba_Location_Id</td>
</tr>
<tr>
<td>rows</td>
<td>1</td>
</tr>
</tbody>
</table>

- This output from EXPLAIN means:
  - The system uses the primary key index on the table wifiDetails (t1) to read each row
  - For each, it looks up the (single) matching value of Id using the primary index on the table locations (t2)
This output from EXPLAIN means:

- The system reads each row of the table locations (t2)
- and uses the Aruba_Location_Id index on table wifiDetails to find the matching Id values
Summary

- Answering user queries *efficiently* is important for a DBMS.
- Most DBMSs have a query optimiser that tries to find an efficient execution plan for every query.
- Usually hidden from the eyes of a user.
- However, knowing about this helps a user in analysing bottlenecks.
For more information

See, e.g.,

- Chapters 11 to 13 of [Silberschatz et al.].
- Chapter 23 of [Connolly and Begg].