Query Execution and Optimization

Execution plans
Logical optimization
Physical Optimization
Query Execution: Parameters

Design parameters for physical schema:

- Hardware type and structure
- Data volume: # pages for relations
- Typical operations: types of queries
- Indexes to the data
  - Index types
  - Index implementation and organization
  - Index utilization
Query Execution: Example

- Number of tuples in relations:

  1 Mio Movies

  ```sql
  CREATE TABLE Movie(
    id INTEGER,
    title VARCHAR(60),
    category CHAR(10),
    year DATE,
    director VARCHAR(30),
    pricePDay DECIMAL(4,2),
    length INTEGER,
    ...);
  ```

  3 Mio Tapes

  ```sql
  CREATE TABLE Tape(
    id INTEGER,
    format CHAR(5),
    movie_id INTEGER,
    ...);
  ```

  500000 Rents

  ```sql
  CREATE TABLE Rental(
    tape_id INTEGER,
    mem_no INTEGER,
    from_date DATE,
    until_date DATE,
    ...);
  ```
## Query Execution: Example

- **Data volume**
  - pages à 4KB, PCTFREE = 30%, ~2800Bytes usage

- **Movie:**
  - ~120B / row, 1 Mill tuples, ~120 MB
  - 23 records per page, 43,479 pages

- **Tape:**
  - ~20 B / row, 3 Mill tuples, ~60 MB
  - 143 records per page, 20,980 pages

- **Rents:**
  - ~20 B / row, 500,000 tuples, ~10 MB
  - 143 records per page, 4,597 pages
Query Execution: Example

- Typical data update rates

- **Movie:**
  - Low update frequency, high read load,
  - Medium growth

- **Tape**
  - Low update frequency, high read load,
  - Medium growth

- **Rental**
  - High update frequency,
  - High growth rate
Query Execution: Example

- Typical operations

- Browse or query the movie table
  - Very high frequency

- Rent a tape:
  - access Customer (by name or id), access Tape (by tape-id),
  - Insert into Rental table
  - High frequency (10 / minute ?)

- Return a tape:
  - Access rentals, access Movie table to calculate the price
  - update Rentals
  - High frequency
Query Execution: Structure

- Query execution sequence:
  1. User-defined query
  2. Parsing
  3. Query validation
  4. View resolution
  5. Query optimization
  6. Execution plan creation
  7. Code creation
  8. Execution
  9. Query result

- Main difference to language compilers: translation is data dependent
Query Execution: Principles

- Set-oriented query logic compiled onto record interface of DBS
- Record interface defined by set of operators
- Important operators:
  - Single row table access
  - Table scan: reads a table sequentially
  - Index scan: reads index tree leaves sequentially from $x \leq$ key value to $y \geq$ key value
  - Join operator for two streams of rows (records)
  - Filter operators
  - Projection

- Order of execution defined by execution plan
Query Execution: Execution plan example

- Structures: Index on Movie(category)

- Query:
  
  ```sql
  SELECT title
  FROM Movie
  WHERE category = 'SciFi'
  AND pricePDay < 3.00;
  ```

- Execution plan
  
  - Index access: take rowIds x from index one after the other for keyVal = 'SciFi'
  - Table access: for each x fetch the tuple
  - Filter: if it passes the filter pricePDay < 3
  - Projection: project on title
Query Execution: Execution plan

- Often represented as tree
- Nodes building blocks of operations to
  - access data (scan of base table or index) and / or
  - transform data (project, sort, join, ...)

Example:

```sql
SELECT title
FROM Movie
WHERE category = 'SciFi'
AND pricePDay < 3.00;
```

![Diagram of query execution plan with nodes for project, filter, table access, and index scan.]
Query Execution: Execution plan

- **Oracle**: EXPLAIN command
  - creates a table with information about query plan

- **Example**: `CREATE INDEX movie_category_index
  ON movie(category);`

```
SELECT title
FROM Movie
WHERE category = 'SciFi'
AND pricePDay < 3.00;
```

---

**Query Plan**

```
SELECT STATEMENT   Cost =
	TABLE ACCESS BY INDEX ROWID MOVIE
	INDEX RANGE SCAN MOVIE_CATEGORY_INDEX
```
Oracle:
- Create plan_table
- Explain query
- Query plan_table

Plan_table
- Stores explain results
- Created once

CREATE TABLE plan_table
(statement_id VARCHAR2(30),
timestamp DATE,
remarks VARCHAR2(80),
operation VARCHAR2(30),
options VARCHAR2(30),
object_node VARCHAR2(128),
object_owner VARCHAR2(30),
object_name VARCHAR2(30),
object_instance NUMERIC,
object_type VARCHAR2(30),
optimizer VARCHAR2(255),
search_columns NUMERIC,
id NUMERIC,
parent_id NUMERIC,
position NUMERIC,
cost NUMERIC,
cardinality NUMERIC,
bytes NUMERIC,
other_tag VARCHAR2(255),
other LONG) ;
Query Execution: Oracle explain usage

Example:
- Explain query:

```sql
EXPLAIN PLAN
SET STATEMENT_ID = 'te1'
FOR SELECT title
FROM Movie
WHERE category = 'SciFi'
AND pricePDay < 3.00;
```

Query plan_table:

```sql
SELECT LPAD(' ',2*(LEVEL-1))||operation||'
||options ||' '||object_name ||' '||
DECODE(id, 0, 'Cost = '||position) "Query Plan"
FROM plan_table
START WITH id = 0 AND statement_id = 'te1'
CONNECT BY PRIOR id = parent_id
AND statement_id = 'te1';
```
Query Execution: Optimization Example

- Query:
  
  ```sql
  Select t.id, m.title, r.mem_no
  From movie m, rental r, tape t
  Where t.id=r.tape_id
  And t.movie_id=m.id;
  ```

  $$\left(\prod_{\text{tape.id, title, mem_no}}(\text{rental } \bowtie_{\text{tape_id}=\text{id}} \text{ tape } \bowtie_{\text{movie_id}=\text{movie.id}} \text{ movie})\right)$$

- Why optimization?
  ** Variant 0:** Join(Join(movie, rental), tape): creates 1 Mill * 500000 tuples (cross product), then deletes unwanted tuples

  ** Variant 1:** Join(Join(movie, tape), rental): creates 1 Mill * x tuples, then deletes unnecessary tuples

  ** Variant 2:** Join(Join(rental, tape), movie): only rented tapes, only movies to rented tapes
Query Execution: Optimization

- High level (logical) optimization
  - dictionary independent (algebraic, "logical")
  - transformation of the query according to the algebraic laws of relational algebra

- Low level (physical) optimization
  - physical level using indexes ("internal")
  - cost based selection of optimal plan using database statistics
Normalizes statements:
- Eliminate ANY / ALL operators

```
SELECT ...
WHERE val  >  ANY (x, y);

⇒ val > x OR val > y
```

```
SELECT ...
WHERE x > ALL(SELECT y FROM R WHERE z=10);

⇒ NOT ( x <= ANY (SELECT ...) )
⇒ NOT EXISTS (SELECT y
FROM R WHERE z= 10 AND x <= y)
```
Query Execution: Logical Optimization

- Normalizes statements:
  - other ‘additional’ constructs (BETWEEN)
  - Evaluate expressions as far as possible

  SELECT ...
  WHERE x > 0.5 * z/100 * 4

  ⇒ x > z/200

- Focus on applying algebraic transformation rules
  - Commutative selection and join
  - Distributive selection, projection
  - ...

Query Execution: Logical Optimization

- Heuristic for logical optimization

1. Split conjunctive selections to enable shift of partial selection predicates
2. Shift selections towards the leaves of the execution tree
3. Rearrange leaves such that most restrictive selections are far left in the tree
4. Split projections (and create new ones) and move towards leaves (but beware of mutating joins to cross products)
Example:

\[(\Pi_{\text{tape.id, title, mem_no}}(\text{rental} \bowtie_{\text{tape_id=id}} \text{tape} \bowtie_{\text{movie_id=movie.id}} \text{movie}))\]
Query Execution: Physical Optimization

- Choose access path to result data according to heuristic rules for physical structures
  - Rule based
  - Cost based

- Rule based: Precedence among access operators
  - Optimizer always chooses lowest rank operator independent from data
  - **Oracle:**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Access Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single row access by rowid</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>Single row by primary key</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>Single column index</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>Full table scan</td>
</tr>
</tbody>
</table>
Query Execution: Rule-based Optimization

- Simple rule based example 1:

  ```sql
  SELECT * 
  FROM movie 
  WHERE id = 200;
  ```

- Execution plan:

  ```sql
  Query Plan
  ------------------------------------------
  SELECT STATEMENT  Cost =
  TABLE ACCESS BY INDEX ROWID MOVIE
  INDEX  UNIQUE SCAN SYS_C002773
  ```
Query Execution: Rule-based Optimization

- Simple rule based example 2:

```sql
SELECT *
FROM movie
WHERE id BETWEEN 0 AND 1000;
```

- Execution plan:

```
Query Plan
------------------------------------------
SELECT STATEMENT  Cost =
 TABLE ACCESS BY INDEX ROWID MOVIE
 INDEX  RANGE SCAN SYS_C002773
```

- Depends on internal (physical) schema of DB
- No cost estimation
Query Execution: Cost-based Optimization

- Utilizes statistical properties of the DB state
- Tries to weigh I/O and CPU costs

- Among the top secrets of DB vendors

- Basic principle: make the typical (simple) case fast

- Primary influence factors
  - Number of page I/Os
  - Data volume to transfer
  - CPU cost (path length of query: i.e. number of instructions)
  - Resource allocation, e.g. buffer pools required
Query Execution: Cost-based Optimization

- Find a formula for

\[
\text{cost (execution plan)} = \text{Weight}_1 \times \text{page accesses (execution plan)} \\
+ \text{Weight}_2 \times \text{cpuCost (execution plan)}
\]

- Based on data statistics
  - DBS Static parameters
  - Data dependent parameters

- Static parameters
  - Block size, device characteristics
  - min/max # entries in B-tree nodes
  - ...

Query Execution: Cost-based Optimization

- **Data dependent parameters**
  - Number of tuples $n$
  - Number of different values $colCard$ for each column
  - Selectivity

- **Selectivity of predicates**
  $\begin{align*}
  a = \text{val} : & \quad s(a,v,=) = \frac{1}{colCard} \\
  a < \text{val} : & \quad s(a,v,<) = \frac{v - \text{LowKey}}{\text{HighKey} - \text{LowKey}} \\
  \text{Interpolation for composites: } & \quad s(P1 \text{ AND } P2) = s(P1) \times s(P2) \\
  \end{align*}$

- Strong assumptions on distribution of values and independence of attributes
Query Execution: Cost-based Optimization

- Histogram based data
  - Assumption "uniform distribution of data" mostly wrong
  - Dynamically collected histograms allow for much better predictions

- In most cases gathered offline from time to time
- Collect new statistics after big changes in data quantities or distribution
Query Execution: Cost-based Optimization

- **Oracle: analyze command**

- **Levels of analyzing tables:**
  - for table and indexes:
    analyze table tt estimate statistics
  - for table only:
    analyze table tt estimate statistics for table
  - for indexes only:
    analyze table tt estimate statistics for all indexes
  - for index histograms:
    analyze table tt compute statistics for all indexed columns
  - Everything:
    analyze table tt compute statistics for table for all indexes for all indexed columns
  - ...

Example

ANALYZE table movie estimate statistics;

EXPLAIN PLAN
  SET STATEMENT_ID = '3'
  FOR SELECT *
  FROM movie
  WHERE id BETWEEN 0 AND 1000;

Query Plan

SELECT STATEMENT  Cost = 1
  TABLE ACCESS FULL MOVIE

Currently only 7 movies in table ⇒ one block access
How can joins be efficiently implemented?
- No predefined ‘connections’ between join partner tuples

Base algorithm: Nested Loop Join

Consider \( R \bowtie_{r.a \ \theta \ s.b} S \)

for each of the \( n \) records of \( R \) {
    for each of the \( m \) records of \( S \) {
        if \( r.a \ \theta \ s.b \) then resultSet \( \leftarrow \) \( (r,s) \)
    }
}

\( O(n \times m) \)

Improvement: inner loop - relation with higher cardinality

Improvement: block-based reading
Query Execution: Joins

- **Two pass algorithm: Sort - Merge Join**
  - Consider $R \bowtie_{r.a} s.b S$
  - First step: sort $R$ on $a$ ; sort $S$ on $b$
  - Second step: join to sorted streams of data

- Idea: Reduce number of comparison steps
- Very efficient, if one relation already sorted according to join attribute (index exists for $a$) and $S$ is small.

$$..., 22, 18, 15, 12, 8$$

$$(22, 22), (15, 15), (15, 15), ...$$

$$..., 22, 19, 15, 15, 13, 9$$
Query Execution: Joins

- **Two pass algorithm: Hash join**
  - Consider $R \bowtie_{r.a \theta s.b} S$, $a$, $b$ keys in $R$, $S$
  - Idea: Reduce number of comparison steps
  - Create hash index on both $a$ and $b$

  - **First phase:** for each $r \in R$
    - compute hash position $h(a)$ in hash table $B$
    - $B[h(a)] \leftarrow r$.rowid
Query Execution: Joins

- **Two pass algorithm: Hash join**

  - **Second phase:**
    
    for each $s \in S$
    
    if $B[h(b)]$ not empty{
      
      fetch $s$;
      
      for each $r.rowid \in B[h(b)]$
      
      if $r.a = s.b$ resultSet $\leftarrow (r,s)$
      
    }
  

Example:

```
SELECT title, tape.id
FROM Movie, Tape
WHERE category = 'SciFi'
AND pricePDay < 3.00
AND movie.id=tape.movie_id;
```
Example:

```sql
SELECT title, tape.id
FROM Movie, Tape
WHERE category = 'SciFi'
AND pricePDay < 3.00
AND movie.id=tape.movie_id;
```

Query Plan

```
<table>
<thead>
<tr>
<th>SELECT STATEMENT</th>
<th>Cost =</th>
</tr>
</thead>
<tbody>
<tr>
<td>NESTED LOOPS</td>
<td></td>
</tr>
<tr>
<td>TABLE ACCESS FULL TAPE</td>
<td></td>
</tr>
<tr>
<td>TABLE ACCESS BY INDEX ROWID MOVIE</td>
<td></td>
</tr>
<tr>
<td>INDEX UNIQUE SCAN SYS_C002773</td>
<td></td>
</tr>
</tbody>
</table>
```
Example:  
ANALYZE table movie estimate statistics;  
ANALYZE table tape estimate statistics;  

SELECT title, tape.id  
FROM Movie, Tape  
WHERE category = 'SciFi'  
AND pricePDay < 3.00  
AND movie.id=tape.movie_id;  

Query Plan

---

SELECT STATEMENT  Cost = 3

HASH JOIN

TABLE ACCESS FULL MOVIE
TABLE ACCESS FULL TAPE
Query Execution: Short summary

- Performance depends on execution plan
- Logical optimization
- Physical optimization
  - Rule-based
  - Cost-based
- Different Table/Index access methods
- Different Join implementations
  - Nested loop
  - Hash-based
  - Many more (bitmap join index)
- Optimization highly vendor dependent