“Explaining EXPLAIN”
php|works 2006 in Toronto

Lukas Kahwe Smith
smith@pooteeeweet.org
Agenda:

- Introduction
- Understanding Performance
- Simple Searches
- Joins and Subqueries
- Prepared Statements, Stored Routines
- Views, FROM Subqueries and Templates
- Reading EXPLAIN Output
- Optimal Execution Order
- SQL Query Visualization
- Controlling Execution Plans
- Example Optimization
Introduction: The "SQL" Standard

- Structured [English] Query Language
- Does not cover all behavioral aspects
  - Indexes
  - Algorithms
  - Caching
  - etc.
- Not all vendors chose the same ways to implement the standard
  - Do not expect things to work the same on different databases!
  - But the common ground is large enough
Introduction: EXPLAIN

- Show execution plan for a given query
  - How and in what order will the tables be read/scanned?
  - What indexes will be used?
  - What join algorithms will be used?
  - The [estimated] “execution cost“?
- Tool of choice for query optimizations
- Not part of the SQL standard
- All DBMS have some equivalent
  - SET EXPLAIN, SELECT .. PLAN, etc.
Introduction: Sakila and Pagila

- Most examples use the Sakila/Pagila sample database
  - Table and column names shortened
    - a is address
    - c is customer
    - a_id is address_id
    - date is rental_date
    - etc.
- Contains various tables, triggers, views, stored routines and sample data
- Files should be in your home dir!
Introduction: Example Query for Sakila

SELECT c.last_name, a.phone, f.title
FROM r
INNER JOIN c ON r.c_id = c.c_id
INNER JOIN a ON c.a_id = a.a_id
INNER JOIN i ON r.i_id = i.i_id
INNER JOIN f ON i.f_id = f.f_id
WHERE r.return_date IS NULL
AND r.date < (CURRENT_DATE - INTERVAL f.duration DAY)
AND a.phone LIKE '19%'
Introduction:
Example Query for Pagila

SELECT c.last_name, a.phone, f.title
  FROM r INNER JOIN c ON r.c_id = c.c_id
  INNER JOIN a ON c.a_id = a.a_id
  INNER JOIN i ON r.i_id = i.i_id
  INNER JOIN f ON i.f_id = f.f_id
WHERE r.return_date IS NULL
AND r.date < (CURRENT_DATE
  - (f.duration || ' DAY')::INTERVAL
AND a.phone LIKE '19%')
### Introduction:

**Example Output**

<table>
<thead>
<tr>
<th>last_name</th>
<th>phone</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREGORY</td>
<td>195003555232</td>
<td>BERETS AGENT</td>
</tr>
<tr>
<td>MYERS</td>
<td>196568435814</td>
<td>CLUB GRAFFITI</td>
</tr>
<tr>
<td>PATTERSON</td>
<td>198123170793</td>
<td>DOORS PRESIDENT</td>
</tr>
<tr>
<td>GREGORY</td>
<td>195003555232</td>
<td>FRIDA SLIPPER</td>
</tr>
<tr>
<td>HITE</td>
<td>191958435142</td>
<td>FROST HEAD</td>
</tr>
<tr>
<td>FORSYTHE</td>
<td>199514580428</td>
<td>GUNFIGHT MOON</td>
</tr>
<tr>
<td>WADE</td>
<td>192459639410</td>
<td>LUST LOCK</td>
</tr>
<tr>
<td>WADE</td>
<td>192459639410</td>
<td>PHILADELPHIA WIFE</td>
</tr>
</tbody>
</table>

8 rows in set (0.19 sec)
Introduction: EXPLAIN MySQL

*************** 1. row ***************
   id: 1
   select_type: SIMPLE
   table: film
   type: ALL
   possible_keys: PRIMARY
   key: NULL
   key_len: NULL
   ref: NULL
   rows: 1058
   Extra:

*************** 2. row ***************
   id: 1
   select_type: SIMPLE
   table: inventory
   type: ref
   possible_keys: PRIMARY,idx_fk_film_id
   key: idx_fk_film_id
   key_len: 2
   ref: sakila.film.film_id
   rows: 2
   Extra: Using index
### 3. row

- **id**: 1
- **select_type**: SIMPLE
- **table**: rental
- **type**: ref
- **possible_keys**: rental_date, idx_fk_inventory_id, idx_fk_customer_id
- **key**: idx_fk_inventory_id
- **key_len**: 3
  - ref: sakila.inventory.inventory_id
- **rows**: 1
- **Extra**: Using where

### 4. row

- **id**: 1
- **select_type**: SIMPLE
- **table**: customer
- **type**: eq_ref
- **possible_keys**: PRIMARY, idx_fk_address_id
- **key**: PRIMARY
- **key_len**: 2
  - ref: sakila.rental.customer_id
- **rows**: 1
- **Extra**:  

Introduction: EXPLAIN MySQL (continued)

id: 1
select_type: SIMPLE
  table: address
    type: eq_ref
possible_keys: PRIMARY
  key: PRIMARY
  key_len: 2
    ref: sakila.customer.address_id
  rows: 1
  Extra: Using where
5 rows in set (0.00 sec)

Execution Order:
Film, Inventory, Rental, Customer, Address
Introduction: EXPLAIN PostgreSQL

Nested Loop (cost=359.64..366.64 rows=1 width=43)
  Join Filter: ("outer".rental_date < (('now':::text)::date - (("inner".rental_duration.."
  -> Nested Loop (cost=359.64..363.57 rows=1 width=35)
    -> Merge Join (cost=359.64..360.55 rows=1 width=37)
      Merge Cond: ("outer".customer_id = "inner".customer_id)
      -> Sort (cost=26.64..26.65 rows=3 width=29)
        Sort Key: customer.customer_id
        -> Nested Loop (cost=0.00..26.62 rows=3 width=29)
          -> Seq Scan on address (cost=0.00..17.54 rows=3 width=19)
            Filter: ((phone)::text ~~ '19%':::text)
          -> Index Scan using idx_fk_address_id on customer (cost=0.00.."
            Index Cond: (customer.address_id = "outer".address_id)
            -> Sort (cost=333.00..333.44 rows=176 width=14)
              Sort Key: rental.customer_id
              -> Seq Scan on rental (cost=0.00..326.44 rows=176 width=14)
                Filter: (return_date IS NULL)
                -> Index Scan using inventory_pkey on inventory (cost=0.00..3.01 rows=1 width=6)
                  Index Cond: ("outer".inventory_id = inventory.inventory_id)
                  -> Index Scan using film_pkey on film (cost=0.00..3.04 rows=1 width=24)
                    Index Cond: ("outer".film_id = film.film_id)"

Execution Order:
Address, Customer, Rental, Inventory, Film
Understanding Performance: Benchmarking

- Set of isolated performance test cases
- Indicator for how an application would perform if it were to use the given code
- Beware of caching
- Change one parameter at a time
- Store results for later reference
- Understand all aspects of benchmark before making conclusions!
- Tools: EXPLAIN and other DBMS tools, Super Smack, ApacheBench, etc.
Understanding Performance: Profiling

- Method of diagnosing the performance bottlenecks of a specific application
- Pin point trouble spots that to isolate, benchmark and tweak
- Focus on areas where application spends the most time in
- Profile real world user pattern
- Beware of caching
- Tools: user land profiler like APD, xDebug or Zend Server or GUI test tools
Understanding Performance: Optimizers

- Rule-based optimizers use non volatile data and fixed assumptions
- Cost-based optimizers additionally use table statistics and other volatile data
  - Biggest advantage for cost-based optimizers is for joins
- Physical I/O vs. Logical I/O
- Statistics and on disk representation of data and indexes may change over time
  - Use ANALYZE, OPTIMIZE, VACUUM etc.
Simple Searches: Index Basics

- Optimal search condition form
  - `<column> <operator> <literal>`
    - `c1 - 12 = c2 x 2 vs. c1 = (c2 x 2) + 12`
    - `c1 = c2 AND c1 = 12 vs. c1 = 12 AND c2 = 12`
  - Some DBMS allow indexes on expressions
  - Merging two indexes is expensive (*)

- Tablescan reading > 20% table rows
- Use index reading < 0.5% table rows
- No generic advice reading 0.5% - 20% table rows
  - Oracle 13%, MySQL 30%
Simple Searches: Index Types

- **Btree indexes**
  - Best general purpose index type
  - Sorting, equality and range searches
    - \(\text{bday} = \text{CURRENT\_DATE AND name LIKE 'T%'}\)
- **Bitmap indexes**
  - Equality searches with multiple indexes (*)
  - Distinct values should be < 1% of rowcount
- **Hash indexes**
  - Equality searches
- **Custom index types**
  - GiST (PostgreSQL), Fulltext (MySQL)
Simple Searches: Covering and Compound Indexes

- **Covering Index**
  - DBMS skips reading table when index contains all data required from the table
    - SELECT indexed_col FROM t1 WHERE indexed_col = 'A%';
    - PostgreSQL must read table due to their MVCC

- **Compound Index**
  - Index (c1, c2, c3) implies (c1, c2) and (c1)
    - SELECT * FROM t1 WHERE c1 = 'A%';
  - Index (c1, c2, c3) not usable in this case
    - SELECT * FROM t1 WHERE c2 = 'A%';
    - Oracle supports "index skip scan"
<table>
<thead>
<tr>
<th>Operator</th>
<th>Points</th>
<th>Operand</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>10</td>
<td>Literal alone</td>
<td>10</td>
</tr>
<tr>
<td>&gt;</td>
<td>5</td>
<td>Column alone</td>
<td>5</td>
</tr>
<tr>
<td>&gt;=</td>
<td>5</td>
<td>Parameter alone</td>
<td>5</td>
</tr>
<tr>
<td>&lt;</td>
<td>5</td>
<td>Multiop. Expression</td>
<td>3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>5</td>
<td>Exact numeric type</td>
<td>2</td>
</tr>
<tr>
<td>LIKE</td>
<td>3</td>
<td>Other numeric type</td>
<td>1</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>0</td>
<td>Temporal type</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Character type</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NULL</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Searches: Code Points Examples

• WHERE some_char = 'The answer: 42!'
  - Left side
    • 0 Points for "character type"
    • 5 Points for "column alone"
  - Operator
    • 10 Points for "equal"
  - Right side
    • 10 Points for "literal alone"
  - 25 Points Total
Simple Searches: Code Points Examples (continued)

- WHERE some_int <= another_int + 23
  - Left side
    - 2 Points for "exact numeric type"
    - 5 Points for "column alone"
  - Operator
    - 5 Points for "smaller or equal"
  - Right side
    - 3 Points for "multi operand expression"
    - 2 Points for "exact numeric type"
  - 17 Points Total
Joins and Subqueries: Nested Loop Joins

for (each row in outer_table) {
    for (each row in inner_table) {
        if (join column matches) {
            pass;
        } else {
            fail;
        }
    }
}
Joins and Subqueries: Nested Loop Joins (continued)

- Stable performance and memory usage
- Outer table
  - Table with most restrictive/expensive WHERE clause
  - Table that allows fewer rows through filter
- Inner table
  - Table with a good index
  - Small table that fits into memory
- Join Condition
  - Should be done on indexes
  - Should be done on same data type and size
**Joins and Subqueries: Hash Joins**

- Fast when joining a large table with a small table on an equality condition
- Fall back from nested loop joins when
  - Inner table hash fits into memory
  - No index for join condition on the inner table
  - No restrictions on large outer table
- Disadvantages
  - Memory requirements
  - Hash generation overhead
Joins and Subqueries: Sort Merge Joins

sort (t1); sort (t2); // <- expensive
get first row (t1); get first row (t2);
while (rows in t1 || rows in t2) {
    if (join-col in t1 < join-col in t2) {
        get next row (t1);
    } elseif (join-col in t1 > join-col in t2) {
        get next row (t2);
    } elseif (join-col in t1 = join-col in t2) {
        pass;
        get next row (t1); get next row (t2);
    }
}
Joins and Subqueries: Sort Merge Joins (continued)

- Only single pass when data is presorted
- Fall back for nested loop joins and hash joins when
  - Both tables are about equal in size
  - Both tables are large
- Disadvantages
  - Startup time and memory cost for the initial sorting
Joins and Subqueries: Join Advantages over Subquery

- Optimizer has more choices
  - Correlated subqueries force a nested loop
  - More freedom in the execution order
- Ability to include columns from both tables in the select list
- Due to their greater popularity they are used more and therefore optimized more in DBMS
  - Some DBMS can parallelize joins better
  - Subqueries in MySQL 4.1 – 5.0.x often slow
Joins and Subqueries: Subquery Advantages over Join

- ANY or EXISTS can break out early
- Column type mismatches are less costly
- Only recently DBMS are adding the ability to join in UPDATE/DELETE
  - MySQL limits subqueries in UPDATE/DELETE
- Simpler to read ("modular")
  - Many RDBMS rewrite subqueries where possible to JOINs internally
Prepared Statements and Stored Routines Execution Plans:

- MySQL disable query cache and prevent use of some statements
- Oracle execution plan are generated
  - < 9i at prepare time
  - since 9i with first bound values
- PostgreSQL execution plan generated
  - named prepared statements at prepare time
  - unnamed prepared statements with first bound values
- Similar issues for stored routines
Views, FROM Subqueries and Templates:

- Control over execution plan is limited by the underlying view defining query
  - Any change may affect any number of other queries that use the given view
- Some view/subquery using queries cannot be translated into a simple query
  - Especially the case for outer joins to views or views with UNION and GROUP BY
- Lead to redundant or unnecessary work
  - SELECT .. FROM c LEFT OUTER JOIN a ON s.a_id = a.a_id WHERE a.phone = '555'
MySQL EXPLAIN Output: MySQL EXPLAIN Columns

- id
- select_type
- table
- type
- possible_keys
- keys
- key_len
- ref
- rows
- EXTRA

- Sequential numer
- SIMPLE, SUBQUERY ..
- Table name
- const, *ref*, index, ALL ..
- List of possible indexes
- Index that is used
- Length if the index used
- Expression compared
- Expected read rowcount
- Using index, where, filesort, temporary etc.
Example EXPLAIN

*************** 2. row ***************

id: 1
select_type: SIMPLE
table: inventory
type: ref
possible_keys: PRIMARY,idx_fk_film_id
key: idx_fk_film_id
key_len: 2
ref: sakila.film.film_id
rows: 2
Extra: Using index
Optimal Execution Order: Robust Plan Characteristics

- Cost is proportional to rowcount returned
- Require little sort or hash memory
- Require no changes when table sizes grow
- Have moderate sensitivity to distribution
- Are not necessarily the fastest, but usually pretty close to the fastest, execution plan
Optimal Execution Order: Robust Plan Requirements

• Prefer very selective filters
  - Initial driving table is the most important choice
• Drive using nested loop joins on indexes
  - Only consider tables that join previous tables
• Drive to primary keys first
  - Keep number of rows low as long as possible
Optimal Execution Order: Further Optimization Strategies

• Only when basic robust plan rules do not give the required performance
  – Prefer smaller tables/expensive filters
    • But make a very small table inner most table
  – Try to join to very selective filters earlier
    • Jump to single row join nodes
    • Join to tables with similar filter ratios
  – Hash joins for joining a large table with a small rowcount who's hash fits into memory
  – Sort merge joins when data is presorted or both table have equally large rowcounts
  – etc ..
SQL Query Visualization:
Example Query

SELECT c.last_name, a.phone, f.title
FROM r INNER JOIN c ON r.c_id = c.c_id
INNER JOIN a ON c.a_id = a.a_id
INNER JOIN i ON r.i_id = i.i_id
INNER JOIN f ON i.f_id = f.f_id
WHERE r.return_date IS NULL
AND r.date < (CURRENT_DATE - INTERVAL f.duration DAY)
AND a.phone LIKE '19%'
Q1: SELECT COUNT(*) FROM rental WHERE return_date IS NULL => 183
Q2: SELECT COUNT(*) FROM rental => 16044
Q3: SELECT COUNT(*) FROM customer => 599
Q4: SELECT COUNT(*) FROM address a, customer c WHERE a.address_id = c.address_id => 599
Q5: SELECT COUNT(*) FROM address WHERE phone LIKE '19%' => 8
Q6: SELECT COUNT(*) FROM address => 603
SQL Query Visualization: Deducing Execution Plan

- Driving table choice
  - rental or address
  - similar filter ratio but address produces lower rowcount

- Best plan
  1. address
  2. customer
  3. rental
  4. inventory
  5. film
Controlling Execution Plans: Strategy Overview

• SQL Level
  - Add SQL hints or (bogus) information
  - Rewrite SQL
• Statistics Level
  - ANALYZE table and indexes
  - Fake statistics
• Server Configuration
  - Enable/disable features
  - Set buffer sizes
• Schema Level
  - Denormalization
Controlling Execution Plans: SQL Level

- SQL hints for MySQL
  - SELECT SQL_SMALL_RESULT .. FROM ..
  - SELECT .. FROM t1 FORCE INDEX (idx1) ..
  - SELECT .. FROM t1 STRAIGHT_JOIN t2 ..

- Add more information
  - Add implicit join condition
    - WHERE a.postcal_code = 34221
      AND store.a_id = a.a_id AND staff.a_id = a.a_id
      AND staff.a_id = store.a_id
  - Improve driving table filter ratio for inner joins by applying the master join ratio early
    - t1.FKt2 IS NOT NULL
Controlling Execution Plans: SQL Level (continued)

- Add bogus information
  - Disable index
    - rental_duration + 0 = :int
    - title || "" = :title
    - COALESCE(last_name, last_name)
  - Force join order
    - Add bogus filter to make it appear like having a restriction so that it is favored as the outer table
      - AND table1.column1 > ""
    - Force staff table to join before address table
      - AND store.manager_staff_id = staff.staff_id
      - AND store.address_id + (0*staff.staff_id) = address.address_id
Controlling Execution Plans: SQL Level (continued)

- Convert single SELECT into a UNION ALL to enable easier index use
  - SELECT .. FROM f
    WHERE (title = :1 OR lang_id = :2)
  - SELECT .. FROM f WHERE title = :1
    UNION ALL
    SELECT .. FROM f WHERE lang_id = :2

- Convert multiple queries (or a CURSOR) into a single query using CASE
  - \( r = \text{CASE WHEN } r > 2 \text{ THEN } r * 0.90; \text{ ELSE } r * 1.10 \text{ END}; \)
Controlling Execution Plans: SQL Level (continued)

• EXISTS may be expressed with an equivalent IN (same for NOT variants)
  – SELECT .. FROM inventory i WHERE EXISTS (SELECT NULL FROM rental WHERE i.inventory_id = rental.inventory_id)
    • Use to drive from inventory to rental
  – SELECT .. FROM .. i WHERE inventory_id IN (SELECT inventory_id FROM rental)
    • Use to drive from rental to inventory

• INTERSECT/EXCEPT may be expressed with an equivalent EXISTS/NOT EXISTS
Controlling Execution Plans: Statistics and Configuration Level

- PostgreSQL statistics
  - `ANALYZE [ table [ (column [, ...] ) ] ]`
  - Statistics are stored in `pg_statistics`
    - May be manipulated as needed
    - Will be overwritten with the next `ANALYZE`

- PostgreSQL query planner configuration
  - `SET SESSION ENABLE_HASHJOIN TO OFF`
  - `SET CPU_OPERATTOR_COST TO 0.003`
  - `SET GEQO_THRESHOLD TO 9`
Controlling Execution Plans: Schema Level

- Merge One-One relationships
- Split tables into One-One relationships
- Denormalization
  - Add (Join-)Indexes, Materialized Views
  - Cache data in application memory
  - Cache aggregate results in memory/DBMS
  - Move “inherited” properties to detail tables
    - SELECT country.country FROM city INNER JOIN country ON city.country_id = c.country_id
    - Copy country column from to city table
      - SELECT city.country FROM city
Example Optimization: Force Execution Order with a Hint

```
SELECT c.last_name, a.phone, f.title
FROM a STRAIGHT_JOIN c
    ON a.a_id = c.a_id
INNER JOIN r ON c.c_id = r.c_id
INNER JOIN i ON r.i_id = i.i_id
INNER JOIN f ON i.f_id = f.f_id
WHERE r.return IS NULL
    AND r.date < CURRENT_DATE
    - INTERVAL f.duration DAY
    AND a.phone LIKE '19%';
```
References:

- These slides
- “SQL Performance Tuning”
  by Peter Gulutzan and Trudy Pelzer
- “SQL Tuning” by Dan Tow
- Benchmarking and Profiling
- Sakila 0.8.0
  - http://www.openwin.org/mike/download/sakila-0.8.zip
- Pagila 0.8.0
  - http://pgfoundry.org/frs/download.php/919/pagila-0.8.0.zip
Thank you for listening ..
Comments? Questions?

smith@pooteewleet.org