Let's take a deeper look at query processing and optimization
Show some example queries on sakila and compare their execution times

Basic database performance-tuning concepts

- the goal of database performance improvement is to execute as many queries as fast as possible
- the activities we'll talk about today all aim to reduce database response time.
- There are a lot of moving parts that must be examined

- Obvious things like the fastest system with the most memory and fastest hard disk spare space along with a high performance network and an operating system and applications can handle the kinds of heavy traffic we expect.
- a lot of the time much of the sluggishness of a database can be traced back to the application's particular SQL queries.
– On the client side the focus is on generating a SQL query that the database can efficiently to obtain the correct answer quickly. It is also important that it utilize a minimum amount of resources at the server.

– On the database server side, the focus is on responding to the client requests as fast as possible while using the least amount of resources.

– There is no magic here: the data in a database or stored in files. As the data grows in size, additional space is allocated in increments as extends.

– These data files a group to file groups or table spaces. These are a logical grouping of several data files stored data similar characteristics such as System information, User data, index data, etc.

– Data buffer caches are also primary means of performance. They store the most recently accessed data blocks in memory.

  • SQL itself maintains a cache containing the most recently executed SQL statements or procedures or functions.
  • The database will often retrieve data from permanent storage (the disc) and stage it in main memory for fast access.
  • This in memory data cache is a lot faster than working with files.
  • Most of our optimizations aim to reduce input/output operations with the disk since it is a lot slower than the computer.

SLIDE 11–1
- Often the query optimizations are related to the optimal order of processing the steps of the query for the faster execution time
- In distributed databases, the selection of sites and servers is important to minimizing network costs.

  • Some optimizations are actually handled by the DB itself - automatically - while others can only be done manually
    - The first of these automatic optimizations is static query optimization, which is done when the SQL is “compiled” by the DB
    - Other automatic optimizations occur at DB run time, since that’s is when the actual query sequences appear. This allows the DBMS to leverage the most up-to-date information about the database.

The DBMS maintains statistics about the database
- number & size of tuples
- previous average access times
- number of users accessing it

These statistics are used by the DBMS to determine the best access strategy.

SLIDE 11–2 /Table 11.2/

How a DBMS processes SQL queries

SLIDE 11–3 /Figure 11.2 /

1. Parse the SQL Query
  * Usual parsing techniques are employed - break it down into smaller parts
  * Rewrite the SQL in relational algebra expressions which are much easier to work with
  * Apply transformation rules to those expressions to find equivalent, but more efficient, queries
  * Devise a plan for the most efficient way to access the data, including the complex I/O operations required.

  * IMPORTANT
  + A key technique for optimizing queries is to determine whether an equivalent access plan already exists in the SQL Cache
  + If not, the optimizers evaluates the alternative plans and chooses one to use AND store in the SQL cache.

1. Run the Query
Acquire needed locks
Fetch the data into the data cache
Execute transactions

2. Fetch the result and return it.
   - Temporary table space may have been used
   - Copy result into Client cache (if there is one)
   - Initiate return of results to Client

What kinds of I/O plans might be used?

The types of decisions the query optimizer has to make

So how do databases find stuff so fast?
Their indexes play a key (ha!) role

- full table scans are too slow, even with caches
  - How would you even decide what would to put in the cache?
- help with searching, sorting, joins, and aggregate functions like $AVG$
- Just like the index in a book, a DB index is an ordering of one or more attributes

We will explore in depth the various kinds of indexes in lecture 13

The importance of indexes in query processing
About the importance of indexes in query processing

Relational algebra expressions sometimes have equivalents that are simpler.
One job of the query optimizer is to find those equivalents and choose the least expensive one.

Query Opt

Once the query is represented in RelAlg expressions, the optimizer tried to find faster equivalent ones.

RelAlg expressions are equivalent if they produce the same set of tuples on all possible valid instances for the database.

Lets take a look at some of the equivalence rules employed.

$\sigma_{B_1 \land B_2}(E) = \sigma_{B_1}(\sigma_{B_2}(E))$
1. Selection operators are commutative

\[ \sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E)) \]

2. In a sequence of Projection operators, only the last one is needed

\[ \Pi_{L}(\Pi_{L_2} \ldots (\Pi_{L_n}(E))) = \Pi_{L}(E) \]

3. Selection can be combined with Cross products and theta joins
   a. \( \sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie E_2 \)
   b. \( \sigma_{\theta}(E_1 \bowtie_2 E_2) = E_1 \bowtie_{1 \land \theta_2} E_2 \)

4. Theta joins are commutative

\[ E_1 \bowtie_0 E_2 = E_2 \bowtie_0 E_1 \]

SLIDE 12 - 8

1. Natural joins are a special case of theta joins, so they are also commutative
   a. \( (E_1 \bowtie_0 E_2) \bowtie_0 E_3 = E_1 \bowtie_0 (E_2 \bowtie_0 E_3) \)
   b. Theta joins are also associative sometimes:

\[ (E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \land \theta_3} E_3 = E_1 \bowtie_{\theta_1 \lor \theta_2} (E_2 \bowtie_{\theta_3} E_3) \]

   where \( \theta_2 \) only involves attributes from \( E_2 \) and \( E_3 \).

2. Selection distributes over theta-join sometimes:
   a. When \( \theta_1 \) only involves attributes of \( E_1 \)

\[ \sigma_{\theta_1}(E_1 \bowtie_0 E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_0 E_2 \]

   b. When \( \theta_1 \) only involves attributes of \( E_1 \) and \( \theta_2 \) involves only attributes of \( E_2 \)

\[ \sigma_{\theta_1 \land \theta_2}(E_1 \bowtie_3 E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_3 (\sigma_{\theta_2}(E_2)) \]

3. and several more.

How about some examples:

SLIDE 11–5

SLIDE 11–6 Easy music query

SLIDE 11–9,10 harder music query

This final version pushes the select down to operating on two smaller tables separately.

SLIDE 11–11

**Evaluation plans**

Evaluation plans specify the algorithm and parameters for each operation required, and how those operations are coordinated and ordered.
The cost differences can be quite large
Let’s look at the example from the book:

*write on the board*

```
SELECT P_Code, P_Descript, P_Price, V_Name, V_State
FROM Product, Vendor
```

1. the **Product** table has 7,000 rows
2. the **Vendor** table has 300 rows
3. 10 Vendors are in FL
4. 1,000 products come from the vendors in FL

Without doing a query, the optimizer only knows 1 and 2.

Assuming all I/O costs are 1 and no indexes, this table shows two candidate access plans:

**SLIDE 11–12**

For Plan **A**:

- **A1:**
  - the cross product costs 7000+300 reads (using the cache for subsequent reads of those entries),
  - 7000x300=2,100,000 temporary table entries

- **A2:**
  - search across all 2,100,000 entries for those with equal **V_Code**
  - resulting in 7,000 temporary table entries

- **A3:**
  - search across all 7,000 entries for those with **V_State='FL'**
  - resulting in 1,000 result table entries.

Thus the optimizer would choose plan **B**.

**What about indexes?**

An **Index** is an ordered array of index key values and row ID values (think of them as pointers). Indexes are typically used to speed up and facilitate data access.

Indexes are great, but you don’t want a lot of them.

- They take up disk space
- They require updating when data is modified
- They take up cache space
Instead of indexes on every column (NO!) try a few indexes, especially for commonly queried composites (e.g., `firstName,MI,lastName`) and see how it goes.

Transaction oriented DB’s might be better with fewer indexes, and business information DB’s might have more.

More on indexes in Lecture 13.

### Some common practices used to write efficient SQL code

Some suggestions from Burak Guzel and others:

1.“USE the cache, Luke!”
MySQL will try to reuse results in the cache as much as possible. So you should try to make similar queries consistent to use the cache. However, if you call a nondeterministic function in the query, MySQL can’t depend on results being the same. So try to avoid this example from [Burak Guzel]:

```sql
// query cache does NOT work
$r = mysql_query("SELECT username FROM user WHERE signup_date >= CURDATE()");

// query cache works!
$today = date("Y-m-d");
$r = mysql_query("SELECT username FROM user WHERE signup_date >= '{$today}'");
```

2. When you’re only looking for a single, specific tuple, let SQL know [Burak Guzel]:

```sql
// do I have any users from Alabama?

// what NOT to do:
$r = mysql_query("SELECT * FROM user WHERE state = 'Alabama'");  
  if (mysql_num_rows($r) > 0) {
    // ...
  }  

// much better:
$r = mysql_query("SELECT 1 FROM user WHERE state = 'Alabama' LIMIT 1");
  if (mysql_num_rows($r) > 0) {
    // ...
  }
```

3. Whenever possible (and enforceable) use `CHAR(n)` instead of `VARCHAR(n)` (and TEXT and BLOB) since fixed-size attributes are always faster.

4. Keep your primary keys integers whenever you can.

5. Don’t use DISTINCT when you have or could use GROUP BY. [MySQL]

6. Avoid wildcard characters at the beginning of LIKE clauses. [MySQL]

7. Avoid `SELECT *` [Burak Guzel]:
8. Use Prepared Statements whenever you need to get input from the user [Burak Guzel]:

```php
// create a prepared statement
if ($stmt = $mysqli->prepare("SELECT username FROM user WHERE state=?")) {
    // bind parameters
    $stmt->bind_param("s", $state);
    // execute
    $stmt->execute();
    // bind result variables
    $stmt->bind_result($username);
    // fetch value
    $stmt->fetch();
    printf("%s is from %s\n", $username, $state);
    $stmt->close();
}
```

9. Transform conditional expressions to use literals (constants) [Coronel]

```php
/* not good as a condition*/
P_Price = 10 = 7;
/* better */
P_Price = 17;
/* not good as a condition*/
P_QOH < P_MIN AND P_MIN = P_REORDER AND P_QOH = 10
/* better */
P_QOH = 10 AND P_MIN = P_REORDER AND P_MIN > 10
```

10. Always test equality conditions first - they’re the easiest to process. [Coronel]
11. Numeric field comparisons are always faster than character, date, and NULL comparisons. [Coronel]
12. Functions are convenient, but using them in a conditional can be very expensive especially for larger tables. [Coronel]
13. When using multiple OR conditions, put the one most likely to be true first (this is a good thing to do in any programming language). [Coronel]
14. Similarly, when using multiple AND conditions, put the one most likely to be false first (also a good thing to do in programming
15. Avoid the use of the NOT logical operator when possible.
16. Use the `DESCRIBE` to learn about tables and `EXPLAIN` to understand

**DBMS performance tuning**

**Caches**
- Data cache
  - stores recent data results
  - shared space among all database users
- SQL cache
  - stores the most recently executed SQL statements
  - reasoning is for large multi-user databases many queries will be similar or the same
- Sort cache
  - temporary storage area used by `ORDER BY` and `GROUP BY` operations.

In-memory databases becoming popular with users who have extremely high-performance needs, and sometimes even short-term memory requirements (think stock trading)

**RAID solutions for physical storage**

Allocate separate disk volumes for specific tasks, to reduce contention.
- System data dictionaries - most frequently used space
- User data space - the more volumes here the better.
- Index table space - should be separate from user and system data areas
- Temporary table space
- Rollback table space

in addition, monitor usage to identify “hot spots” of database access and migrate those to their own volumes

**Distributed DB’s**

- ch12 Coronel (Distr. DBs)