Making SQL great again

A little bit of complexity theory

Estimating Time Complexity of Your Query Plan

- The execution plan defines, among other things, what algorithm is used for each operation, which makes that every query execution time can be logically expressed as a function of the table size involved in the query plan, which is referred to as a complexity function.

- There are four main types of time complexity:
  - Constant.
  - Linear.
  - Logarithmic.
  - Quadratic.

O(1): Constant Time

- An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. When you're talking about a query, it will run in constant time if it requires the same amount of time irrespective of the table size.

- These type of queries are not really common, but here is an example:

```sql
SELECT TOP 1 t.*
FROM t
```

- The time complexity is constant because you select one arbitrary row from the table. Therefore, the length of the time should be independent of the size of the table.

Linear Time: O(n)

- An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases.

- For databases, this means that the time execution would be directly proportional to the table size: as the number of rows in the table grows, the time for the query grows.

Linear Time: O(n) (cont.)

- **Example:** a query with a WHERE clause on a un-indexed column:
  - A **full table scan** is needed.
  - This means that every row needs to be read to find the one with the right ID.
  - You don'€™t have a limit at all, so every row does need to be read, even if the first row matches the condition.

Linear Time: O(n) (cont.)

Consider also the following example of a query that would have a complexity of O(n) if there is no index on `i_id`:

```sql
SELECT i_id
FROM item;
```

- Other queries, such as count queries like `COUNT(*) FROM TABLE`; will have a time complexity of $O(n)$.

Linear Time: JOIN

- A hash join has an expected complexity O(M + N).

- The hash join algorithm first prepares a hash table of the smaller table.

- Once the hash table is built, the larger table is scanned and the relevant rows from the smaller table are found by looking in the hash table.

Linear Time: JOIN (cont.)

- Merge joins generally have a complexity of O(M+N) but it will heavily depend on the indexes on the join columns and, in cases where there is no index, on whether the rows are sorted according to the keys used in the join:
    - If both tables that are sorted according to the keys that are being used in the join, or if both tables have an index on the joined columns, then the complexity will be O(M + N).
    - If neither table has an index on the joined columns, a sort of both tables will need to happen first so that the complexity will look more like O(M log M + N log N).
    - For nested joins, the complexity is generally O(MN). This join is efficient when one or both of the tables are extremely small (for example, smaller than 10 records).

Logarithmic Time: O(log (n))

- An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size.

- For queries, this means that they will run if the execution time is proportional to the logarithm of the database size.

- **Example:** Suppose there is an index on `i_i`d. The following query has a complexity of $O(\log(n))$:

```sql
SELECT i_stock
FROM item
WHERE i_id = N;
```

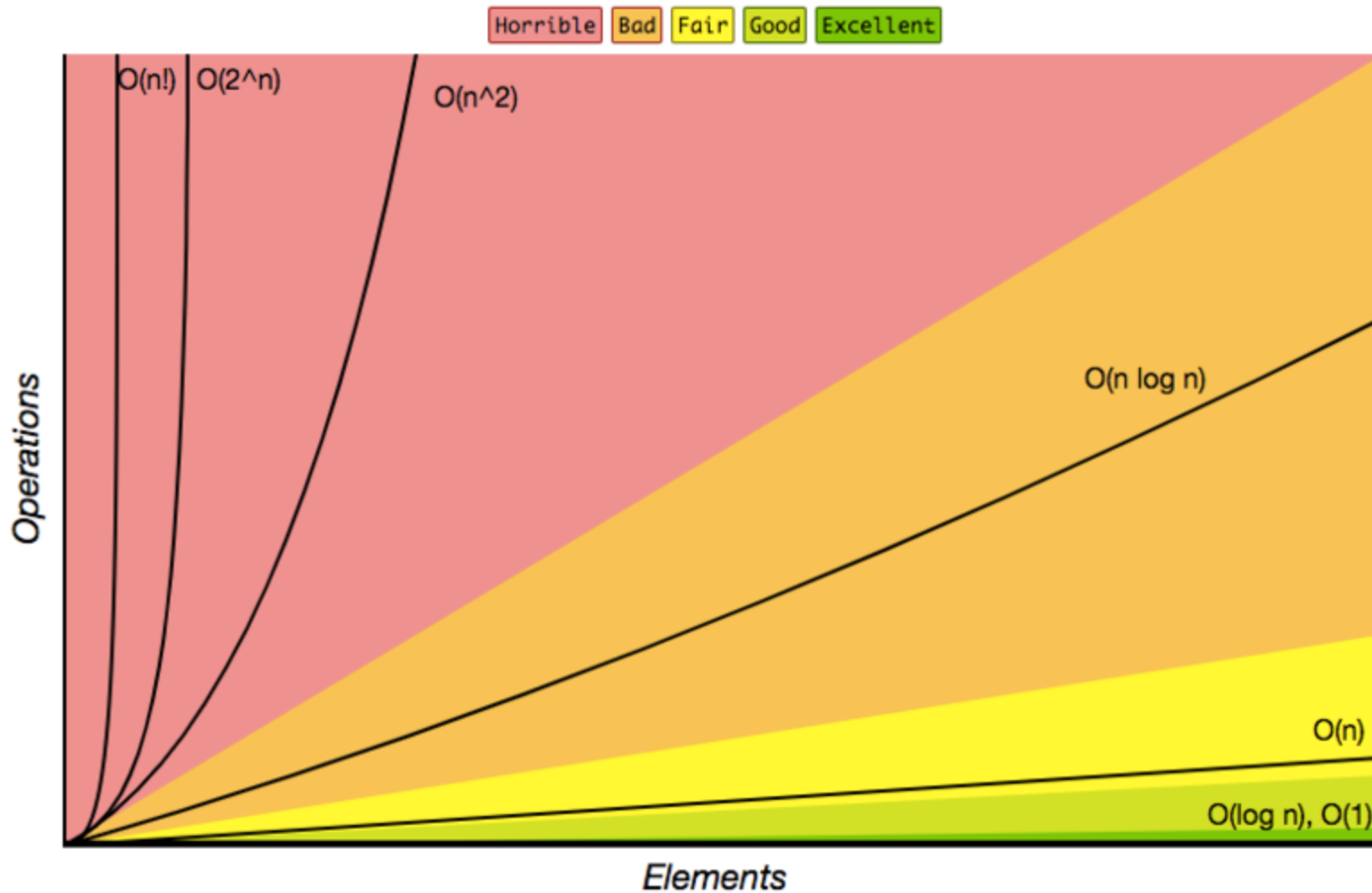- Note that without the index, the time complexity would have been O(n).

Quadratic Time: O(n^2)

- An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size.

- Once again, for databases, this means that the execution time for a query is proportional to the square of the database size.

- **Example**:

```sql
SELECT *
FROM item, author
WHERE item.i_a_id=author.a_id
```

- This query has O(n^2) complexity as a worst-case scenario, depending on the index structure.

Big-O Complexity Chart

Horrible Bad Fair Good Excellent

O(n!) O(2^n)   O(n^2)

O(n log n)

O(n)

O(log n), O(1)

Operations

Elements

- First things to look after:
  - `JOIN ... ON` and index structure of the involved tables.
  - `WHERE` clauses.

Collecting Statistics

Why do we need statistics

- The execution strategy is based on the statistics collected on the tables used within the SQL query. Statistics on the table is collected using the `COLLECT STATISTICS` command.

- Optimizer requires environment information and data demographics to come up with the optimal execution strategy.

Information collected

- **Environment Information**
    - Number of Nodes, AMPs and CPUs.
    - Amount of memory.
- **Data Demographics.**
    - Number of rows.
    - Row size.
    - Range of values in the table.
    - Number of rows per value.
    - Number of Nulls.

Approaches

There are three approaches to collect statistics on the table.

- Random AMP Sampling.

- Full statistics collection.

- Using `SAMPLE` option.

Collecting and viewing statistics

```
/* Retrieve statistics */
collect statistics
on tutorial.salestransaction column(customerid);

/* Show collected statistics*/
help statistics tutorial.salestransaction;
show statistics on tutorial.salestransaction;
```

Recommendations for stats collection

- At least on Primary Index column/columns.

- Update collect stats after 10% change in data of a table.

- Non-indexed columns that are frequently used in `WHERE` and `JOIN` clauses.

- All **sets of columns** involved on a `JOIN`.

- **But** statistics require significant amount of system resources, collect wisely and greedily.

Too much to remember? No worries

```
/* Suggest which statistics to collect */
DIAGNOSTIC HELPSTATS ON FOR SESSION;
explain select sum(noofitems)
from tutorial.salestransaction sales
left join  tutorial.soldvia sold
on sales.tid = sold.tid;
```

- Collect the stats only if the optimizer is recommending high confidence.

Set how statistics will be sampled

```
/* Random sample */
COLLECT STATISTICS
USING SAMPLE 10 PERCENT
COLUMN productprice
ON tutorial.productid;

/* Start with full-table scan, but stop at some point*/
COLLECT STATISTICS
USING SYSTEM SAMPLE
COLUMN productprice
ON tutorial.productid;

/*Full-table scan to collect the specified stats.*/
COLLECT STATISTICS
USING NO SAMPLE
COLUMN productprice
ON tutorial.productid;
```

EXPLAIN

Explaining `EXPLAIN`

- Explain plan is the step-by-step description of a query plan generated by the parsing engine.

- Explain plan can tell you how Optimizer will execute a query.

- Although you can not directly modify it, you can modify your query to influence the execution plan.

Explaining `EXPLAIN` (cont.)

- **Locking Information:** Explain plan provides information about locking.

- **Row retrieval Strategy:** Either full table scan, using primary index, using secondary index or any other access path.

- **Time & size estimation information:** Estimated row counts and estimated time to complete a particular step and query.

Explaining `EXPLAIN` (cont.)

- **Join information:** what kind of join operation strategy is chosen by the optimizer base on the situation.

- **Confidence level:** In the explain of a query, you will find something like high confidence, low confidence, no confidence. These are obtained through the statistics collection phase.

- **AMPs involvement information:** During any kind of operation like retrieving rows, joining tables, aggregation. Depends on the work, it can be single AMP, group AMP or all AMPs.

Things to look after in `EXPLAIN`

- Join strategy used ($O(mn)$ complexity if product join).

- All-rows scan ($O(n)$ complexity).

- Confidence level for large tables, or recently modified tables (that XIIth century map).

General rules

- **Know your data:** Understand the relationships between tables, know your data, and *fetch only the required columns.*

- `JOINs:` Be reasonable with `CROSS JOIN` or `LEFT` joins. Bad joins with skew values result in high CPU and IO queries.

- **Data types:** Ensure appropriate data types for columns on the table(s). For example, don't define a `VARCHAR` data type for a `Salary` column.

- **Pre-aggregate data:** To avoid carrying a large number of rows, do aggregations and then join. Extra points if you do those in a volatile table.

- **Prioritize optimization:** A moderately slow query repeated often is a better candidate for tuning than a query running once a year that takes several hours.

More performance tips

- If `LIKE` is used in a `WHERE` clause, it is better to try to use as many leading characters as possible.

- Avoid use of large list of values in `IN/NOT IN` clauses. Store them in some temporary table and use that table in the query.

- If values are nearly unique values then `DISTINCT` clause may outperform `GROUP BY`. When there are many duplicate value then `GROUP BY` performs better than `DISTINCT`.

- When using `CASE`, start with the most frequent values to avoid checking all conditions.

Exercise

From the `ratings` write a query that returns:

userId	movieTitle	rating

for the 5 favorite movies of a user.

The goal is to compare different execution plans.

- **Some things you can try:**
    i. Accidental product join (`select .. from a,b`).
    ii. Create a fancy index (join/aggregate).
    iii. Before and after statistics collection.