Performance Monitoring

Top 5 reasons for slow queries

1. Bad Query

- This is the most common reason for long running query.
- Your query is not using any kind of index: Primary Index, Secondary Index or Partition Primary Index.
- The tables are getting redistributed and those big tables taking long time in it.

2. Delay Time

- Check for Delay time in DBQL (`dbc.dbqlogtbl`) for your query.
- It may happen that query execution time is in seconds however it was in Delay queue for hours.

3. Blocking

- It may happen that your query is blocked by some other query.
- Also watch out for TDM jobs. Such jobs generally apply exclusive locks on the table, thereby blocking all other queries requesting the same table.
- You can check this Teradata Viewpoint Monitor Portlet.

4. Server State

- Generally Teradata Server status is classified as Healthy, Degraded, Critical and Down.

- As the number of concurrent users increase the load on server also increases.

- With more load on server and AMP doing more work, your query may take more time than usual time.

5. Skewness

- Check skewness factor and change the PI if needed.

Prevention

Good Schema Design

- In a perfect world, we don't need to optimize our queries because the schema is just great.

- Data is **normalized**: tables are organized to reduce disk space *and* redundancy.

- We want to avoid situations like having `EmployeeSalary` in a `Customers` table. Or, even worse, a single, huge table with everything.

- **More rows, less columns.**

Normalization

- Different rules for normalization.
- Depending how many of those are followed, we say that the database is in First Normal Form, Second Normal Form (5 and counting).
- Third Normal Form (3NF) is the standard recommended by Teradata, and usually the last necessary.

First Normal Form

- Eliminate columns with repeated information in individual tables.

- Create a separate table for each set of related data.

- Identify each set of related data with a primary key.

Second Normal Form

- Create separate tables for sets of values that apply to multiple records.

- Relate these tables with a foreign key.

- **Example:** `CustomerAddress` . You need this information in your `Orders` , `Invoices` and `Customers` table. Instead of duplicating it, it should be only on `Customers` table.

Third Normal Form

- Eliminate fields that do not depend on the key.

- In the previous example, that could mean storing `CustomerAddress` in its own `Addresses` table.

- Theoretical advantages, but maybe not practical: it would mean having a lot of small tables in some cases, which can degrade performance.

Normalizing an Example Table

These steps demonstrate the process of normalizing a fictitious student table.

Unnormalized table:

StudentId	Advisor Adv-Room	Class1	Class2	Class3
1022	Jones	412	101-07	143-01
4123	Smith	216	201-01	11-02

First Normal Form: No duplicate columns

StudentId	Advisor	Adv-Room	ClassId
1022	Jones	412	101-07
1022	Jones	412	143-01
1022	Jones	412	159-02
4123	Smith	216	201-01
4123	Smith	216	211-02
4123	Smith	216	214-01

Second Normal Form: Separate tables

StudentId	Advisor	Adv-Room
1022	Jones	412
4123	Smith	216

StudentId	ClassId
1022	101-07
1022	143-01
1022	159-02
4123	201-01
4123	211-02
4123	214-01

Third Normal Form

StudentId	Advisor
1022	Jones
4123	Smith

Name	Room	Dept
Jones	412	42
Smith	216	42

Why not to normalize?

- Joins are expensive.
- Prototyping should be *quick and dirty*.
- If using NoSQL databases.

Other tips

- Primary key *need not be* a business attribute (e.g. ID Number). These may change!

- Multi-column Primary Key = Bad idea in general.

Other tips (cont.)

- Add constraints on the database, specially unique (when relevant).
- **Avoid bad names,**
  - `id` in `Customers` and `product_id` in `Products`.
  - Non-ASCII characters, whitespace, reserved words.
  - Mixing plural and singular.
  - Or mixing_underscore and CamelCase
- Hard Delete vs Soft Delete: Flag as 1/0 for active/inactive instead of deleting.

Identify suspect queries

Beyond `EXPLAIN`

- **Product Join Indicator:** the ratio of CPU Seconds to IO for a query. (`AMPCPUTime` * 1000) / `TotalIOCount`
  - $\geq$ 3: the query should be reviewed.
  - $\geq$ 6: potentially an unnecessary product join.

Beyond `EXPLAIN` (cont.)

- **Unnecessary IO Indicator:** is the ratio of IO to CPU Seconds, `TotalIOCount` /(`AMPCPUTime` * 1000)
  - Number of rows read / number of rows processed.
  - ≥ 3: the query should be reviewed to eliminate full-table scans and possibly redistribution steps.
  - Good indicator

Beyond `EXPLAIN` (cont.)

- **Product Join Indicator**, (`AMPCPUTime` * 1000)/ `TotalIOCount`
- PJI is the measure of how CPU intensive your query is.
    - If PJI $\geq$ 3 you should review the query, PJI $\geq$ 6 it might be a product join.
    - This value is high during a product join, but not only.

Beyond `EXPLAIN` (cont.)

- Both metrics are available in Viewpoint's Query Monitor portlet, and every individual query has these values displayed when you click on the session ID.

- You can also track them using `dbqlogtbl` & `dbqlsqltbl` tables in `dbc`.

Set up `dbql`

- You can check if this is activated:

  `SELECT * FROM dbc.dbqlrulesv;`

- If not activated, it can be set up by the DBA in `BTEQ`.

```
begin query logging with objects,
sql, usecount, utilityinfo LIMIT SQLTEXT=0 on all;

begin query logging with objects,
sql limit threshold = 5 elapsedsec
and sqltext=0 on VIEWPOINT;
```

Monitoring

```sql
SET QUERYBAND = 'Version=1' FOR SESSION;

SELECT
AMPCPUTIME,
(FIRSTRESPTIME-STARTTIME DAY(2) TO SECOND(6)) RUNTIME,
SPOOLUSAGE/1024**3 AS SPOOL_IN_GB,
AMPCPUTIME*1000/TOTALIOCOUNT AS PJI,
TOTALIOCOUNT/(AMPCPUTIME*1000) AS UIOI
FROM DBC.DBQLOGTBL
WHERE QUERYBAND = 'Version=1';
```

Monitoring (cont.)

The query will return:

- Total CPU Usage
- Spool Space needed
- PJI/LHR (ratio between CPU and IO usage)
- CPU Skew
- Skew Impact on the CPU
- **Goal:** cut total CPU usage, consumed spool space and skew.