**Query - Optimization and Planning**

**Aman Gour 120050030**

**Introduction**

For any production database, SQL query performance becomes an issue sooner or later. Having long-running queries not only consumes system resources that makes the server and application run slowly, but also may lead to table locking and data corruption issues. So, *query optimization* becomes an important task.

|  |  |
| --- | --- |
| The life cycle of any query made to the database follows these 5 stages.   1. Transmission of the query to database backend 2. Parsing of the query string 3. Bind variables on the first invocation 4. **Planning of the query to optimize retrieval of data** 5. Execution of query and retrieval of data from hardware   *Query Planning* and *Analysis* lies at the 3rd logical stage (and 4th in the diagram shown).  At this step only all the queries are planned and optimised. It also determines the most efficient way of retrieving data. | QueryExecution.png |

### In this essay we will see the application of various tuning tools by using one of the most important method of query optimization that helps access in fast access to data and also minimizes the amount of data accessed - *Index*. A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.

### 

### Useful Tuning Tools - Postgres

The most useful tool in tuning your database is the SQL command EXPLAIN ANALYZE. This allows you to profile each SQL query your application performs and see exactly how the PostgreSQL planner will process the query. Postgres is able to analyze and plan queries efficiently without user direction. Postgres has a great ability to show you how it will actually execute a query under the covers. This is known as an execution plan and which is exposed by **explain**. Understanding this tells you how you can optimize your database with indexes to improve performance.

## Explain: This tool is used to examine how a query will be executed by the database system. It can be thought of as a debugger. It gives an estimate for the various important parameters in the query optimization :

* + It shows how long it get access the first row of data
  + It long it will take to get all the rows of data
  + Number of rows that will be returned
  + Size of a row in bytes
* **Analyze:** After you make some optimization to query, say we introduce a new index. If you now rerun your query again, you won’t see any noticeable change in the output. This is because PostgreSQL has not yet re-analyzed the data and determined that the new index may help for this query. For this we need the tool called analyze that re-analyzes the data.

Typically the best way to optimize your queries is to use indexes on specific columns and combinations of columns to correspond to often used queries.

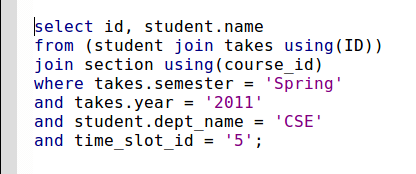
**Demonstration on university schema on next page**

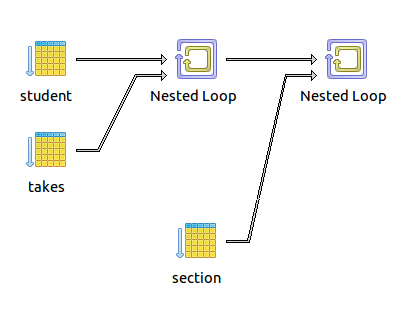
**Reference**

* <http://www.postgresql.org/docs/9.2/static/sql-explain.html>
* <http://www.mohawksoft.org/?q=node/56>
* https://wiki.postgresql.org/wiki/Using\_EXPLAIN

**Demonstration on University Schema**

**Query 1**





**Query Plan**

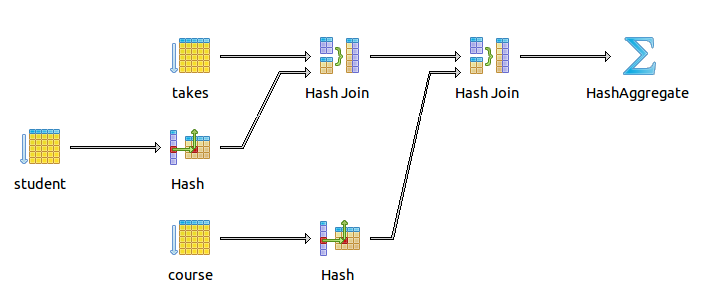
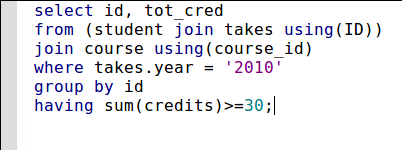
|  |
| --- |
| **Nested Loop (cost=4.27..19.24 rows=1 width=82) (actual time=0.042..0.042 rows=0 loops=1)**  **-> Nested Loop (cost=0.00..9.61 rows=1 width=116) (actual time=0.041..0.041 rows=0 loops=1)**  **-> Seq Scan on takes (cost=0.00..1.33 rows=1 width=58) (actual time=0.040..0.040 rows=0 loops=1)**  **Filter: (((semester)::text = 'Spring'::text) AND (year = 2011::numeric))**  **-> Index Scan using student\_pkey on student (cost=0.00..8.27 rows=1 width=82) (never executed)**  **Index Cond: ((id)::text = (takes.id)::text)**  **Filter: ((dept\_name)::text = 'CSE'::text)**  **-> Bitmap Heap Scan on section (cost=4.27..9.61 rows=2 width=34) (never executed)**  **Recheck Cond: ((course\_id)::text = (takes.course\_id)::text)**  **Filter: ((time\_slot\_id)::text = '5'::text)**  **-> Bitmap Index Scan on section\_pkey (cost=0.00..4.27 rows=2 width=0) (never executed)**  **Index Cond: ((course\_id)::text = (takes.course\_id)::text)**  **Total runtime: 0.135 ms** |

**Create Index yearIndex on takes.year**

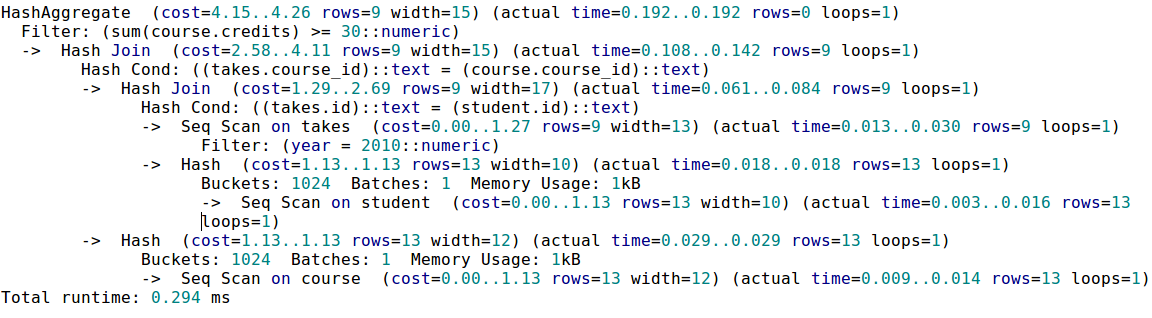
**Query Plan[[1]](#footnote-0)**

|  |
| --- |
| **Nested Loop (cost=0.00..3.71 rows=1 width=12) (actual time=0.022..0.022 rows=0 loops=1)**  **Join Filter: ((takes.course\_id)::text = (section.course\_id)::text)**  **-> Nested Loop (cost=0.00..2.51 rows=1 width=19) (actual time=0.021..0.021 rows=0 loops=1)**  **Join Filter: ((student.id)::text = (takes.id)::text)**  **-> Seq Scan on student (cost=0.00..1.16 rows=1 width=12) (actual time=0.021..0.021 rows=0 loops=1)**  **Filter: ((dept\_name)::text = 'CSE'::text)**  **-> Seq Scan on takes (cost=0.00..1.33 rows=1 width=13) (never executed)**  **Filter: (((semester)::text = 'Spring'::text) AND (year = 2011::numeric))**  **-> Seq Scan on section (cost=0.00..1.19 rows=1 width=7) (never executed)**  **Filter: ((time\_slot\_id)::text = '5'::text)**  **Total runtime: 0.093 ms** |

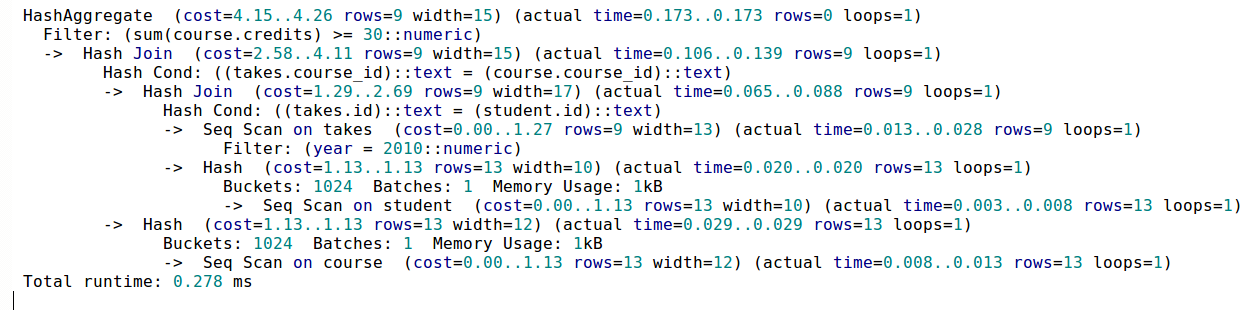
**Query 2:**



**Query Plan**



**Create Index yearIndex on takes.year**



**Analysis**

**Query 1:** We can see that after adding the index on the year attribute of table there is decrease in the time taken for this query. Also the sequence in which the filtering is also done is also changed to make the query more efficient in the previous case the filtering was done first on the dept and then on the year which is then changed because now the year field has an index.

**Query 2:** Here also a significant decrease in time is observed and the order is changed by PostgreSQL because it knows that the query is more efficiently executed when the order of the instructions is changed.

1. Analysis for both queries on next page [↑](#footnote-ref-0)