SQL Query Optimizer-based Query Progress Indicator

Dr.S.D.Joshi[#],Prof.L.V.Patil[*],Urmila Mane[$].

Department of Information Technology,SKN College of Engineering,Pune,India.

***Abstract*- Nowadays, widely used Business Intelligence(BI) and Data Warehousing(DW) technologies are mostly based on long-running and complex queries. So for this purpose it is important for users to have information about progress of query execution. Recently interest in the development of percent-done progress indicators has been increased. In this paper, we propose a method that constructs model of a percent-done progress indicators based on optimizer-based approach. Percent-done progress indicators basically used as a technique that graphically shows query execution time that means total and remaining or degree of completion. Also the proposed technique is based on postgerSQL database engine. PostgreSQL is a powerful, open source object-relational database system. Currently Postgres doesn't have SQL query progress indicator for long-running queries. With the help of user-system interaction (interface) the progress indicator show the progress of SQL queries through various phases like parsing, analyzing, rewrite, execution. The graphical user interface show all the queries running on system and their estimated time completion. The execution phase of query is critical phase and also the cost of query varies depending disk read time, type of join used, distribution or broadcast of table, order in which tables are joined, statistics information available.**

***Keywords*- ACID, BI, DW, GUI, PostgreSQL, RDBMSs, SQL, UNIX.**

I. INTRODUCTION

Progress indicators have been studied in various contexts (typical example is file transfer or file download) but there exists very limited work on this topic in case of data management context. In day to day life a typical progress indicator is used to estimate how much of the task has been completed and when the task will finish. Figure 1 shows an example of progress indicator which actually we are trying to develop for database queries.

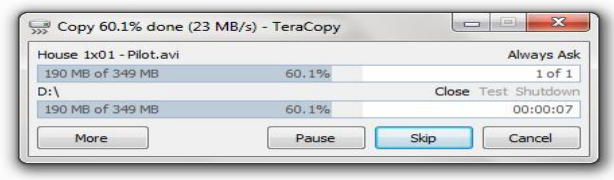

Fig.1. Typical File Transfer using TeraCopy.

In recent years, there has been increasing interest regarding development of progress indicators for SQL queries. A progress indicator in case of database queries is used to estimate precisely the value of a function that is related to the progress towards completion of a running query. For this purpose availability of such indicators can be of great help both to database administrators and end users. Given the complexity of any query in decision support or data warehousing applications, it is common for queries to take hours or days to terminate. During such cases, these indicators can greatly aid a user's understanding of the progress of a query towards completion and allow the user to plan accordingly for example, terminate the query and/or change the query parameters. Also from the point of view of administrators, unsatisfactory progress of queries may point to bad plans, poor tuning or inadequate access paths.

Many modern software systems nowadays provide progress indicators for long-running tasks. These progress indicators aim to make systems more user-friendly by helping the user quickly estimate how much of the task has been completed and when the task will finish. But already existing commercial RDBMSs provide progress indicator for long running queries which were not easy to prove.

Percent-done progress indicators basically used as a technique that graphically shows query execution time that means total and remaining or degree of completion. Also the progress indicator in proposed technique is based on PostgerSQL database engine. PostgreSQL is a powerful, open source object-relational database system. Currently PostgreSQL doesn't have SQL query progress indicator for long-running queries. With the help of user-system interaction (interface) the progress indicator show the progress of SQL queries through various phases like parsing, analyzing, rewrite, execution. The graphical user interface show all the queries running on system and their estimated time completion. The execution phase of query is critical phase and also the cost of query varies depending disk read time, type of join used, distribution or broadcast of table, order in which tables are joined, statistics information available.

Why use PostgreSQL?

PostgreSQL is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL: 2008 data types. It

also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC.
PostgreSQL prides itself in standards compliance. Its SQL implementation strongly conforms to the ANSI-SQL:2008 standard. It has full support for subqueries (including subselects in the FROM clause), read-committed and serializable transaction isolation levels. And while PostgreSQL has a fully relational system catalog which itself supports multiple schemas per database, its catalog is also accessible through the Information Schema as defined in the SQL standard.
The rest of the paper is organized as follows: Section II describes related work regarding the topic. Section III discusses proposed optimizer-based query progress indicator. Section IV concludes the paper. Finally section V describes future enhancement regarding this topic.

II. RELATED WORK

Gang Luo [14] proposed technique sufficient for implementing progress indicator for a large subset of RDBMS queries. They consider select-project-join queries, assume that the available join algorithms are hash-join, nested loops join, and sort-merge join, and those base relations can be accessed by either table-scans or index-scans. They collect statistics at some selected points of a query plan and use that improved and precise information to continuously refine the estimated cost of given query. Thus they estimate remaining execution time of query but don't deal with the percentage of work that has been completed. Also they do not provide estimates for some SQL queries which are non-trivial.

The amount of time required for complete execution of query would be reported to the user at any point during the query's execution. But any existing method which will provide such a measure will subject to the uncertainty arising from concurrent execution of other queries. Hence, due to this difficulty [13] focus on this problem of estimating the percentage remaining or equivalently completed of the given query, at any point during its execution. This paper also deals with the problem of reporting a "progress bar" for query execution. The follow-up work [11] proves that it is impossible for this proposed progress indicator to provide robust guarantees for the problem of progress estimation in the worst case. They provide estimates which are imprecise in certain cases.

Jeffrey F. Naughton [10] considers the problem of supporting the progress indicators for a wider class of SQL queries with more precise estimates. They also discuss and deal with the need of such a progress indicator which is not easy to prove. This paper aims to increase the coverage of progress indicator to large set of queries. They propose techniques to improve the accuracy of the estimates and also to provide new functionality that was not covered in previous work.

Before this all the previously proposed query progress indicators mainly consider each and every query in isolation and thus they ignore the impact of simultaneously running queries on each other's performance. For this purpose Gang Luo and Jeffrey F. Naughton [8] proposes technique to extend the single-query progress estimation to enable progress estimation for multiple queries. They explore a multi-query progress indicator, which deals with concurrently running queries and also queries predicted to arrive in the future at the time of producing its estimates. Also they extend the use of progress indicators beyond just being a GUI tool by showing how to apply that multi-query progress indicator to workload management.

Jiexing Li [1] implements a cost-based approach for query progress indicator with the help of two proposals which were proposed simultaneously and independently in [12, 13]. They summarize some common cases in which both are accurate and also some cases in which they fail to provide accurate and reliable estimates. This proposed query progress indicator is similar to these early progress indicators but without the uniform speed assumption. The previously proposed progress indicators make a common simplifying uniform future speed assumption. Also the developers of these progress indicators were aware that this assumption could cause errors but they did not explore how large those errors might be as well as they did not investigate the feasibility of removing that assumption.

III. OPTIMIZER-BASED QUERY PROGRESS INDICATOR

A. General Features of Progress Indicators

A Progress indicator for postgresql database will provide feedback to the user/DBA on.

- How much percentage of query is completed.
- How much percentage and time is required by the query to run to its completion.
- Current phase of executing query.
- Control over the query execution i.e. either allow the query to run to its completion or to abort the query.

The proposed system is having the following features To provide enhanced feedback to the user/DBA on how much of a SQL query execution has been completed i.e. phase of the query and how long it will take for query execution.

- **Multiple Query Progress Graph Display**: The system is designed to handle and display multiple queries progress in form of graphs. The graphs can be disguised by the distinct transaction-id and XY-Line color. The transaction-id is unique local transaction-id given by postgresql for every query.
- **Estimated Time for Query Completion**: The system gives the estimated time for query completion. The estimated time is dynamic i.e. it varies depending on the system load, resource etc.

- **History of Committed Queries**: The system is also featured with query history. It shows both last committed query and the list of committed queries.
- **Dynamic Variation of Y-Axis**: The Y axis is the time axis and is dynamic in nature as query completion time for different queries is different i.e. one query may commit early and other may long time to complete.
- **Client-Server Implementation**: The system is implemented in 2 tier architecture i.e. client-server .From client side user can fire the query and GUI of query progress will be at client side. At server side query execution is done by the database.

B. Model and Implementation

The architecture shown below, describes how the different components of the system interact and there working collaboratively to achieve the desired functionality of the system. The system mainly consists of user/dba, postgresql database, and the GUI which shows the progress of the query and all these components interact with each other.

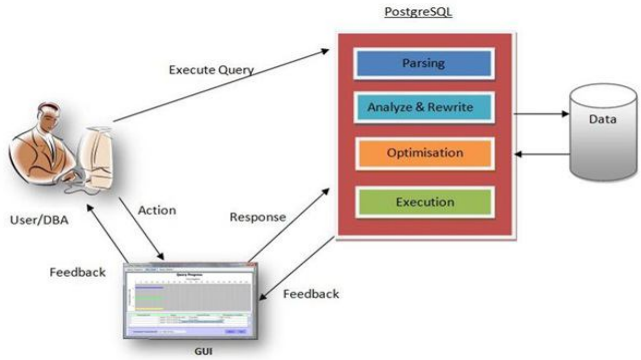

Fig.2.General Outline of system.

When user/DBA fires a query then it passes through different phases i.e. parsing, analyze, rewrite, planning, execution of postgresql and at every phase it gives the feedback to the user/dba through the GUI .The feedback is about how much percent of query is completed , how long it will take for query to run to its execution. Also the user/DBA can interact with the GUI during execution by aborting the query in between and the DBA can see at what percentage of the query it is aborted. Aborting the query in between will not harm the data as the kill signal is sent which cause the shutdown of query execution i.e. data integrity is maintained. Effect is only reflected into the database when the execution of the query is complete. GUI also handles the history of committed queries.

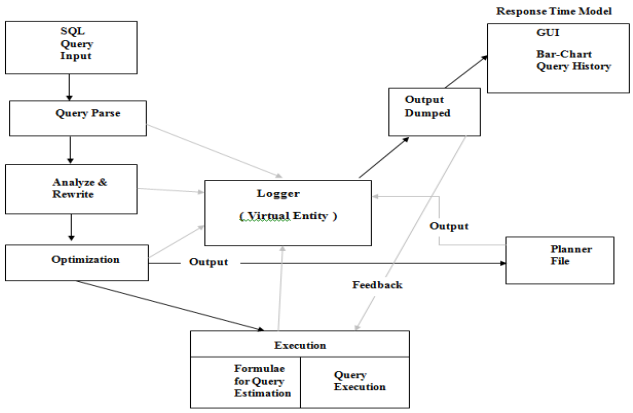

Fig.3. Working of Proposed System.

1) SQL Query Execution Plan-background

SQL divides a query plan for each query [1]. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex *planner* that tries to select good plans.

The structure of a query plan is a tree of plan nodes. Nodes at the bottom level are table scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes "atop" the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. The first line (topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like.

EXPLAIN SELECT * FROM tenk1;

QUERY PLAN

Seq Scan on tenk1 (cost=0.00…458.00, rows=10000, width=244)

The numbers that are quoted by EXPLAIN are:

- Estimated total cost (If all rows were to be retrieved, though they might not be: for example, a query with a LIMIT clause will stop short of paying the total cost of the Limit plan node's input node.).

- Estimated number of rows output by this plan node (Again, only if executed to completion.).
- Estimated average width (in bytes) of rows output by this plan node.

The costs are measured in arbitrary units determined by the planner's cost parameters. Traditional practice is to measure the costs in units of disk page fetches; that is, sequential page cost is conventionally set to 1.0 and the other cost parameters are set relative to that.

It is important to note that the cost of an upper-level node includes the cost of all its child nodes. It is also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client which could be an important factor in the true elapsed time; but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same row set, we trust.).

Rows output is a little tricky because it is not the number of rows processed or scanned by the plan node. It is usually less, reflecting the estimated selectivity of any WHERE-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

2) Mathematical Model

The planner cost and rows output will be used to estimate the query completion time. The cost estimates are expressed in arbitrary units, but thing to pay attention to is ratios of actual time taken by query and estimated planner cost is somewhat consistent.

The mathematical model for query completion time estimate would be based on planner cost, rows output and feedback mechanism.

2.1 Feedback Mechanism

As explained in the previous section, the query plan is divided into number of nodes (for large query) and each node has cost/"rows output". We will extrapolate planner cost and rows output (and some heuristic, which will be based on testing of large TPCH queries) of all the nodes in plan to come with rough query completion estimates, when query start execution.

As query progresses we will go on refining estimates based on actual time taken by each node (sometimes also called as snippet). The current execution node estimates will be then taken based on above feedback and planner cost/"rows output". Please note we are considering that query is going to take maximum (around 90%) time in execution phase and very less time in parsing, analyze, rewrite, planning, optimization phase.

2.2 Plan Tree Walker

The structure of a query plan is a tree of plan nodes. We will walk the entire plan tree to come up with rough query estimates at start and then go on refining the estimates based on above mathematical model. The planner has different types of nodes based on kind of operation node is going to perform. For example, there are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes "atop" the scan nodes to perform these operations.

We will walk the entire query plan tree to get rough total query completion estimate at start. During query execution, execution engine walks through all the plan nodes sequentially. At each plan node we will use our feedback model to refine the particular node's estimate and also total query completion estimates.

2.3 Percentage Completion

```
// Calculate the percentage or contribution with respect to total cost of tree.

// percentage_so_far: stores the accumulated percentage

// final_cost: total cost of plan tree. Calculated prior at the end of planning phase.

// value: cost of the current executing node.

// Used "90" based on heuristic – considering the fact that execution phase going to eat most of // the time

    percentage_so_far = percentage_so_far + (value * 90) / final_cost;
```

2.4 Estimate updating based on feedback

Below calculation will be done by execution engine during each plan node execution.

```
// Take feedback into account. We have taken actual time taken by query so far (for estimated // cost so far) into account to project remaining time query will probably take

// final_cost is global variable and its updation will reflect in all the alogorithms

final_cost =  final_cost * actual_time_so_far / total_cost_so_far

current_cost = current_node_cost * actual_time_so_far / total_cost_so_far

total_cost_so_far = total_cost_so_far + current_cost
```

IV. CONCLUSION

The SQL progress indicators for long-running queries are nowadays becoming a desirable user-interface tool to monitor progress of executing query in RDBMSs. But all the previously proposed techniques for supporting the construction of progress indicators for SQL queries are having very limited functionality and accuracy. In this paper, we have implemented a technique based on query optimizer which can be used for the development of query progress indicator. Also we have modeled some of the features of the proposed system along with its general architecture and principle working.

V. FUTURE ENHANCEMENT

As we know that today's world is completely dependent on the internet and online tools. We can enhance our idea and can make our tool as web portal, so that anyone can use it at any time. We can also send the progress status of the query through email or the sms to the DBA. So that he can know the progress of the query without running the GUI and sitting in front of the machine. So like this possibilities are endless.

REFERENCES

(1) Jiexing Li, Rimma V. Nehme , Jeffrey Naughton; "GSLPI: a Cost-based Query Progress Indicator"; 2012 IEEE 28th International Conference on Data Engineering.

(2) Basit Raza, Abdul Mateen, M M Awais and Muhammad Sher; "Survey on Autonomic Workload Management: Algorithms,Techniques and Models"; Journal of computing, volume 3,Issue 7,July 2011, ISSN 2151-9617.

(3) Kristi Morton, Abram Friesen, Magdalena Balazinska, Dan Grossman; "Estimating the Progress of MapReduce Pipelines";ICDE Conference 2010.

(4) Elnaz Zafarani, Mohammad_Reza Feizi_Derakhshi, Hasan Asil, Amir Asil; "Presenting a New Method for Optimizing Join Queries Processing in Heterogeneous Distributed Databases"; 2010 Third International Conference on Knowledge Discovery and Data Mining.

(5) Mario Milicevic, Krunoslav Zubrinic, Ivona Zakarija; "Dynamic Approach to the Construction of Progress Indicator for a Long Running SQL Queries"; international journal of computers issue 4, volume 2, 2008.

(6) Mario Milicevic, Krunosla V Zubrinic, Ivona Zakarija; "Adaptive Progress Indicator for Long Running SQL Queries"; Proceedings of the 8th WSEAS International Conference on Applied Computer Science(ACS'08).

(7) Chaitanya Mishra, Nick Koudas; "A Lightweight Online Framework For Query Progress Indicators"; 2007 IEEE.

(8) Gang Luo , Jeffrey F. Naughton , and Philip S. Yu;" Multi-query SQL Progress Indicators"; Y. Ioannidis et al. (Eds.): EDBT 2006, LNCS 3896, pp. 921 – 941, 2006, Springer-Verlag Berlin Heidelberg 2006.

(9) Christian M. Garcia-Arellano, Sam S. Lightstone, Guy M. Lohman, Volker Markl, and Adam J. Storm; "Autonomic Features of the IBM DB2 Universal Database for Linux, Unix, and Windows"; IEEE Transactions on systems, MAN, And Cybernetics Part C:Applications And Reviews, Vol.36,No.3, May 2006.

(10) Gang Luo, Jeffrey F, Naughton, Curt J. Ellmann, Michael W. Watzke; "Increasing the Accuracy and Coverage of SQL Progress Indicators"; Proceedings of the 21st International Conference on Data Engineering (ICDE 2005).

(11) S. Chaudhuri, R. Kaushik, and R. Ramamurthy, "When can we trust progress estimators for SQL queries?" in SIGMOD, 2005.

(12) DB2, "IBM DB2 query monitor for z/OS," ftp://ftp.software.ibm.com/software/data/db2imstools/whitepapers/db2querymon-wp05.pdf, 2005.

(13) Suraji Chaudhuri, Vivek Narasayya, Ravishankar Ramamurthy; "Estimating Progress of Execution for SQL Queries"; *SIGMOD 2004*, June 13–18, 2004, Paris, France, 2004 ACM.

(14) Gang Luo , Jeffrey F. Naughton , Curt J. Ellmann , Michael W. Watzke; "Toward a Progress Indicator for Database Queries"; ACM SIGMOD 2004, June 13–18, 2004, Paris, France,2004 ACM.

(15) Chaitanya Mishra, Nick Koudas; "A Lightweight Online Framework For Query Progress Indicators"; 2002 ACM.