A Comparison of Algorithms for Self Tuning Histograms

Julián Hernández, Rodrigo Ipince, José Muñiz

December 12, 2008

Abstract

In this paper, we explore the feasibility of implementing self-tuning histograms for OLTP databases.

1 Introduction

Traditional database management systems based on the System-R design often include an optimizer. As explained in [2], this optimizer is responsible for coming up with a low-cost query plan to execute. In order to estimate the costs of the set of feasible plans for a given query, the optimizer requires a selectivity.

A common approach for estimating selectivities is to use a histogram associated with each field on each relation. The histogram captures the frequency of different values over a number of ranges. Current commercial database systems like Oracle, Microsoft SQL Server, and Sybase, as well as open source alternatives like PostgreSQL and mySQL, all use this approach.

Estimating the selectivity from a given query is easy and requires almost no overhead when consulting the histogram. However, the process of building a histogram is usually expensive. The previously mentioned database use a mechanism of statistical analysis by which they analyze the entire table after a threshold of insertions is met. This mechanism is, however, prohibitive when large-sized tables are constantly updated. This situation is common in OLTP transactions, and therefore is worth discussing.

This paper is organized into seven parts. Section 2 describes how histograms are used in database systems. Section 3 shows some previous work on using results from queries to improve future cost estimates. Section 4 then explains our modifications to the previous algorithms. Finally, Sections 5, 6, and 7 describe our framework for testing improvements for cost estimation over transactional loads.

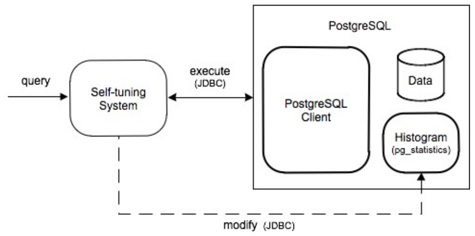

Figure 1: High level overview of the benchmarking system

2 Histogram querying and traditional building

HI

As described in [3], a statistical approach to building histograms consists of

3 Previous self tuning histograms

3.1 Curve fitting

[1]

3.2 Aboulnaga

4 Modifications to proposed algorithms

5 Benchmark setup

In order to test the accuracy of the previously discussed algorithms for self-tuning histograms, we built a system as shown in Figure 5

This system represents an intermediate layer between the end user and PostgreSQL, which intercepts

EmpID	**Country**
1	3
2	2
3	2
4	1

Table 1: Table for **Employees**

DeptID	**Country**
1	3
2	3
3	1

Table 2: Table for **Departments**

the results from the database to the user and uses the cardinality to build a histogram. The different histograms can then be compared by their accuracy.

This layer is responsible for receiving user queries. The information is then fetched back to Postgres, which responds to the middle system with the query plan to be executed. The benchmark tool then simulates the query plan, by reducing it into a family of nested loop joins and sequential scans with a set of conditions. For each query, a collection of `receive` calls is sent to the histogram builder. For each scan, the executor makes one `receive` call. In the case of a join, for each element in the outer loop, the executor can make a `receive` call as it matches the join field in the outer table with the tuples in the inner table. For example, if Tables 5 and 2 are joined by a natural join, where **Employees** is the outer table, then we get the following three distinct `receive` calls:

```
receive(``Dept.Country'', [3...3], 2)
receive(``Dept.Country'', [2...2], 0)
receive(``Dept.Country'', [1...1], 1)
```

The system assumes independence between selectivities of multiple conditions, either over a single table or over several tables. In other words, if a query of the form `SELECT ... WHERE` p_1 `and` p_2, we assume that $\sigma_{p_1} \times \sigma_{p_2} = \sigma_{p_1 \wedge p_2}$. For this reason, some joins operators might be able to send `receive` messages even when facing multiple conditions, especially when one is a join condition and the other is a filter on the data.

We chose not to modify the source code for Postgres directly for several reasons:

- The s
-

However, the benchmark still makes sense since all of Postgres implementation of joins, merge joins, hash joins and nested loops joins can all be modified to emit appropriately.

Finally, it should be noted that we choose to measure our results in terms of row estimate errors, instead of query execution times. This decision was taken for several reasons :

- Modifying the `pg_statistic` table in Postgres,which is responsible for holding the histogram infromation, is not modifiable through an external interface. Although this histogram is stored in a table
- Given that we do not consider the effect of join selectivity estimate errors from this paper, we establish that a measurement of row selectivity in errors.
- hej

6 Choice of Queries and Insetions

We use a framework of insertions in the spirit of the TPC-C benchmark. We include the following tables:

1. Customers
2. Orders
3. Products

7 Results

8 Conclusion

9 Bibliography

References

[1] Aboulnaga, A., and Chaudhuri, S. Self-tuning histograms: Building histograms without looking at data. pp. 181–192.

[2] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J., Griffiths, P. P., III, W. F. K., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (1976), 97–137.

[3] Gibbons, P. B., Matias, Y., and Poosala, V. Fast incremental maintenance of approximate histograms. pp. 466–475.