Database Cracking: Concept Evaluation

Zulsar Batmunkh
MIT

Dana Mukusheva
MIT

Abstract—abstract goes here.

1 Introduction

Database Cracking is a concept of partitioning and sorting the data on the fly based on the incoming query workloads. While traditional index creation and maintenance requires upfront knowledge of query patterns and workloads, as well as costly human supervision, database cracking treats index maintenance as a part of query processing. It continuously partitions the database into manageable pieces every time a new query is processed. Dynamic reorganization of data as the system is queried causes subsequent queries to run faster, irrespective of whether these queries had already been run. In our project, we analyzed the relative performance of cracking in a simple single-column database. We conducted multiple experiments, varying implementation details, workloads and query patterns. We are motivated by the Database Cracking paper from Ideros and will have advantages such as no need for upfront knowledge of query patterns and human supervision for index maintenance.

2 Background and related work

Blablabla one Nobody [?]. Blablabla two Nobody [?]. Blablabla three Nobody [?].

2.1 Database Cracking

2.2 Related Work

3 System Overview

In order to analyze the performance effects of the database cracking, we have implemented our own small database system MiniDB. In this section we will describe components and implementation details of MiniDB.

3.1 MiniDB data structures

MiniDB is a simple single column database implemented in Java, where each table consists of a single uniquely named column. Database keeps a hash map with (column name, column object) pairs. The database tuples are 32-bit positive integers. Column tuples are stored as a list. MiniDB has three column types: Simple Column, Sorted Column, and Cracker Column. Simple Column does not maintain a specific tuple order and inserts a tuple to the end of the tuple list. In Simple Columns tuple lookup requires linear scan of the tuples list. Sorted Column preserves the order in the tuples list. Tuple lookup takes logarithmic time and is implemented as a binary search. Cracker Column stores tuples in a partially sorted list, that is, once its tuples are reorganized and partitioned into two sublists (one with all tuples whose values are less than or equal to the partition value and one with all tuples whose values are greater than the partition value). CrackerColumns cannot exist independently in our database, they can only be coupled with Simple Columns that support cracking and initialized after the first query. Cracker Columns contain same values as their corresponding Simple Columns, but in a different and constantly changing order. For each Cracker Column, there is a Cracker Index instance. The Cracker Index instances are necessary to keep most-up-to date information about all partitions of the Cracker Column tuples. Particularly, Cracker Index stores (v, p) pairs, which indicate that all tuples located at the positions less than and including p have values less that and including v.

- *Zulsar Batmunkh is MIT undergraduate. email: zulsar@mit.edu*
- *Dana Mukusheva is MIT undergraduate. email: mukushev@mit.edu*

3.2 Query processing

MiniDB queries are the range scans that operate on a single column. Queries are implemented as RangeScan class, where each RangeScan instance stores the pointer to its column of interest

3.3 Cracker Index implementation

4 Experiments

4.1 Experiment Setup

Experiment Setup This section includes the detailed description of our experiment setup for analyzing the performances of different cracking index implementations. We implemented a simple single-column database implemented in Java for running our experiments on. Our main focus is to identify the impact of cracking only the SELECT operator. As mentioned in the previous section, we implemented three types of Cracker Indexes: AVL Tree, HashMap and Sorted List and analyzed the performances against simple scanning and sorted scanning techniques. We ran series of different sizes of queries on our dataset.

Hardware The experiments are ran on Amazon AWS server and MIT Athena computers.

Dataset Our simple database column contains one million distinct tuples of range 1 to 10^6.

Workload In analyzing the performances of different approaches, we explored workloads of size of 10000, 20000 and 50000 queries.

Query ranges We explored three different types of query plans. Open range. Each query has a single endpoint. Closed range. Each query has two endpoints. Mixed range. For each query, the range is randomly chosen to be either single or closed. The endpoints for the queries are chosen randomly from the range 1 to 10^6.

Selectivity We tried studying the selectivity effects on the performance by changing the result size to be random, 100 and 1000.

Cracking algorithms We recreated the two-piece cracking algorithm presented in the original paper with only single-sided predicate and apply the algorithm once on the column copy for open range queries and twice for closed range queries.

Minimum Partition Size The original paper mentioned about setting a minimum partition size threshold for cracker indexes, thus, no longer partitioning the column any further. The reason would be that the cracker index could become potentially huge same as the original column for non-repetitive queries and dividing columns further into small pieces would become highly costly. Following the idea, we implemented

Describe the hardware (server, memory, OS, number of cores, etc) Describe how we timed things

4.2 Performance Evaluation

4.3 Comparison to Sort based strategy and Traditional Indexes

Baseline For comparing the performances of our different cracking index implementations to traditional indexes and sort-based strategies, we further extended our database to have simple scanning and sorted column scanning to .

Simple Scanning.

In this approach, we do not create a copy of the column and simply scan all tuples in the column (would be O(N) linear time.)

Sorted Column Scanning

In this approach, we sort the column upon the first query and operates upon the sorted column copy for the queries. This would potentially take O(logN) time for finding the correct start and/or end ranges for the query.

5 Discussion

Talk about what needs to be done to improve the quality of our experiments, some assumptions we made and what we have neglected

6 Conclusion

References

[1] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[2] M. Kersten and S. Manegold. Cracking the database store. In *CIDR*, 2005.

[3] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking Management. *Proc. VLDB Endow.*, 7(2):97–108, Oct. 2013.