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 removes the need for human administration and efficiently manages database workload environments by treating 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 this paper, we analyze the relative performance of cracking in a simple single-column database. We conducted multiple experiments, with varying implementation details, workloads and query patterns.

Section 2 presents the background on the database cracking and related works. Section 3 includes our system overview and implementation details regarding our simplified database. We details our experiment setup in Section 4 and discuss the results of our findings and comparisons to already existing strategies in Section 5. Finally, Section 6 includes our conclusion.

2 Background

2.1 Database Cracking Overview

Database cracking is a concept of adaptive column partitioning and sorting based on the query workloads and patterns. The core idea of database cracking is to partition the column according to each incoming query, therefore, effectively reducing the size of partitions as more queries arrive. Such self-organizational behavior is desired when there is no upfront knowledge of query patterns, which is required to preliminary create traditional indexes of the columns of interest.

2.2 Previous work

Database cracking is an active research topic, which is led by a group of researchers from CWI Amsterdam. The concept of self-organizing columns and query-dependent column partitioning was first presented in by Kersten and Manegold [2]. Blablabla one Nobody [3]. Blablabla two Nobody [1]. Blablabla three Nobody [2].

2.3 Cracking Algorithms

3 System Overview

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

MiniDB is a single column database implemented in Java, where each table consists of a single uniquely named column. The database keeps a mapping between a column name and a column object in a hash map. The database table stores tuples as a list, and each tuple is a 32-bit integer. MiniDB maintains the following data structures: Simple Column, Sorted Column, Cracker Column, Cracker Index, Range Scan.

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

3.1 MiniDB columns and indexes

Simple Columns represent database tables and store tuples. They do not maintain a specific tuple order and insert a tuple to the end of the tuple list. In Simple Columns tuple lookup requires linear scan of the tuples list.

Sorted Column is another data structure that represents database tables which preserves the order of the tuples. Tuple lookup takes logarithmic time and is implemented as a binary search.

Cracker Column is a data structure that cannot exist independently in the database, it can only be coupled with Simple Column that supports cracking, They are initialized and attached to a Simple Column instance after the first query. Cracker Columns contain same values as their corresponding Simple Columns, but in a different and constantly changing order. Cracker Columns store 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) Each Cracker Column is supplemented with a Cracker Index instance.

The Cracker Index instances are the data structures that are necessary to keep most-up-to date information about all partitions of the Cracker Column tuples. Particularly, Cracker Index stores pairs in the form (v, p), where v indicates that all tuples located at the positions less than and including p have values less that and including v.

3.2 Query processing

MiniDB queries are the Range Scan objects that operate on a single column. Each Range Scan instance stores the pointer to its column of interest, endpoints of the value ranges, and the range sign, either one of $\leq, <, \geq, >, <<, \leq <, < \leq, \leq\leq$. Range Scan objects are essentially iterators on the values of their columns of interests. If the column does not support cracking, the iterator goes over all tuples and returns only those whose values belong to the specified range. Otherwise, they use cracking and simply iterate over all values that lie in a specified partition.

3.3 Cracker Index implementation

Cracker Index functionality requires methods such as insert value-position pair, find predecessor of value v, find successor of value v and lookup position p of value v. Predecessor/successor search is used to define partitions with the closest boundaries, which is the case when a new unseen query arrives. We have implemented Cracker Index using three different underlying representations, AVL Tree, Sorted List and HashMap. Each of them has advantages and disadvantages in different scenarios.

AVL Tree stores value v as a node key and a position p as a node data. All of the operations on AVL Tree have logarithmic cost, which makes it a good candidates for large workloads. However, at the same time, the size of the tree grows with a number of queries, and maintaining the balance of the large tree might be costly.

Hash Map directly stores mapping between the value v and the position p. The HashMap implementation is beneficial when all incoming queries are repetitive,and additional cracking is unnecessary,

	AVL Tree	Hash Map	Sorted List
Pros	Logarithmic insert, lookup, predecessor and successor search	Constant time lookup	Logarithmic lookup and constant predecessor and successor search
Cons	Cost for balance maintenance	Linear predecessor and successor search	Linear insert

Table 1. Summary of cost-related advantages and disadvantages for each Cracker Index implementation.

since the correct partition already exists. However, adding a new partition info into the index is costly, as predecessor/successor search takes linear time.

Sorted List stores value-position pairs a list sorted on values. Preserving the order of the list is costly and takes linear time, however, cheap successor and predecessor search balances the query cost.

In Table 1 we summarized pros and cons for each Cracker Index implementation.

4 Experiments

4.1 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

In consistent with findings already existing in the original paper, we found the cracking approach to outperform both sort bases strategy and traditional index under changing workload. As more and more queries arrive, the cost of physically reorganizing the data is spread over all the queries. In our workload environment, we found each of the different cracker index implementations to have certain pros and cons over each other, meanwhile all equally performing much faster to the baseline.

Overall, by using the cracker index the database automatically adapts to any workload and creates a re-organization that serve subsequent queries faster and faster. We believe that the database cracking idea has a great potential to revolutionize the traditional approach.

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.