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 Previous work

Database cracking is an active research topic, which is led by a group of researchers from CWI Amsterdam. The concept of dynamic and query-considerate data partitioning was first introduced in [2]. It was then further developed into a cracking system with mature architecture and describe in [1]. This system was built on top of MonetDB, a column oriented database system. The authors also presented cracking algorithms that are used to partition the data, as well as conducted experiments and obtained evidence that cracking improves query execution performance in a long term. The performance was measured as cumulative time of query execution. In [3], the authors raised questions regarding the effects on the performance of cracking algorithm selection and cracker index implementation. The database cracking has a lot of open questions and requires more experiments and research.
The previous work done in the field defined the experimental setup of our project. Particularly, we have focused on column store databases in order to qualitatively compare our results with ones obtained in [1], which was also used as a source of the cracker algorithm. Moreover, we conducted experiments to analyze the performance of different cracker index implementations to prove the hypothesis stated in [3].

2.2 Database Cracking Mechanism

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 would be required for preliminary creation of traditional indexes for the columns of interests. The database creates a copy of a column that was queried for the first time, we will call the copy column a cracker column. Tuple relocation and swapping only happen in the cracker column, leaving the original column intact. Each subsequent query results in further data partitioning and spends less time on cracking the values, as the partition ranges become smaller and smaller.

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

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.

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.

	AVL Tree	Hash Map	Sorted List
Pros	Logarithmic insert, lookup, predecessor and successor search	Constant time lookup	Logarithmic lookup and constant prede
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.

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, 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

This section includes the detailed description of our experiments conducted to analyze the impact of database cracking on query execution performance. We use a similar setup for all performance evaluations. Our main focus is to study the impact of cracking the SELECT operator in memory. In order to identify relative performances, we ran same queries on the same column for all cracker index implementations and compared the results to two baselines: (1) simple scanning which scans the entire column and filters tuples satisfying the query and (2) sorting upfront

4.1 Experiment Setup

Hardware: Amazon AWS server (8 GB memory) and MIT Athena computers (40 GB memory).

Dataset: An array with one million distinct tuples of range 1 to 10^6.

Workload: Randomly generated 20000 queries with varying selectivity.

Query ranges: Open (single predicate), closed (lower and upper predicates) and mixed (randomly chosen to be either open or closed). The predicates are chosen randomly from the range 1 to 10^6.

Selectivity: 0.01 and 0.001.

Minimum Partition Size: 100 and 1000.The cracker index could potentially become very large for non-repetitive queries and dividing columns further into small pieces would become highly costly. Therefore, we experimented with mentioned minimum partition sizes and not cracking the column any further.

4.2 Performance of cracking index implementations

include graph- ——-

4.2.1 Varying minimum partition size

This approach simulates a sort based strategy. Upon a first query, the data column 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.

4.2.2 Varying selectivity

Simple Scanning

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.