

Checkpoint 3

# **Self-organizing Tuple Reconstruction in Column-stores**

---

Team: Fsync, Manjeet Singh (50169105), Wasif Aleem (50161250)

[Source Code](#)

19th December, 2016

## Introduction : Background & Overview

Column-oriented database systems have recently become popular<sup>1</sup>. In C-store each column attribute is stored contiguously and are densely packed, as opposed to row oriented datastores. The main advantage of storing data column wise is it allows queries to directly access the columns of interest, suited for analytical workloads.

Sales			
saleid	prodid	date	region
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	8	8	8
9	9	9	9
10	10	10	10

Sales			
saleid	prodid	date	region
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

2


A traditional row store physically organizes and stores entire records contiguously, compared to column stores where each attribute of the record is stored physically as a separate contiguous column. For a given query which access a subset of attributes from a table, the column store only reads the columns of interest, on the other hand row stores needs to scan and extract needed attributes along with neighbouring attributes. However for a query that needs to access multiple attributes, column store suffers from scattered disk-seeks from column to column.

Since a column store physically stores data in column-wise manner in order to optimize read efficiency, this influences its architectural design. In a typical column store each attribute of a relation R is stored contiguously as a column on separate disk blocks.

The columns are densely packed and often avoid storing record IDs/tuple IDs by using a virtual identifier(position in column) to infer the record ID. They employ column specific compression techniques, substantially reducing the disk space needed.

<sup>1</sup> Monet DB, Apache Parquet, Amazon Redshift, Google BigQuery.

<sup>2</sup> The Design and Implementation of Modern Column-Oriented Database Systems, Abadi et al, Fig 1.1



Column stores have specialized scan operators that differ from row scan operators by leveraging the physical layout of column stores. Column scan operators combine and reconstruct partial or complete tuples from different columns for multi attribute queries.

This process is called as explicit tuple reconstruction which is a join between two columns using tuple IDs. This can incur substantial overhead in multi-attribute queries. Tuple reconstruction can be done in two ways, early and late tuple reconstruction. In early tuple reconstruction the required attributes are joined as early as possible during query processing(while columns are loaded), on the other hand late tuple reconstruction is delayed and is performed only when the attribute is required by the query plan.

Tuple reconstruction has low overhead when columns are already sorted in tuple-order. But during query processing(Joins, Order/Group By) this ordering is not always preserved. Preserving tuple order by pre-sorting the data requires idle-time and is not efficient under updates. This is suitable only for read-only workloads.

This high up-front cost of pre sorting can be avoided using self-organizing techniques introduced in database cracking<sup>[1]</sup>. The main idea is to physically reorganize column on each query in-order to speedup the future queries. This is done by making a copy of column and then creating an index(cracker index) of the copy, re-organizing the column and cracker-index on queries which are then utilized to speedup future queries. This avoids indexing data which is not queried, requires no knowledge of workloads also does not require idle time or index maintenance.

## Problem Statement

The main goal is to minimize cost of tuple-reconstruction in a self organizing way by making use of techniques proposed in Cracking. It aims to achieve performance similar to using presorted data, without requiring explicit upfront sorting cost and idle-time.

We focus on answering queries that require tuple-reconstruction.

- Multi-projection queries:
  - A selection query  $q$  that projects  $n$  attributes.
  - Ex: `SELECT B, C FROM R WHERE A < k`

Along with cracking we also implement a crack-sort hybrid approach to sideways cracking where we sort the cracked pieces if the piece-size is below a specific threshold.

## Outline: Approach

To achieve these goals we utilize sideways-cracking which is a self-organizing way for tuple reconstruction.

Sideways cracking performs tuple reconstruction by efficiently maintaining aligned auxiliary self-organizing data-structures called cracker-maps via cracking instead of using (random-access) position-based joins.

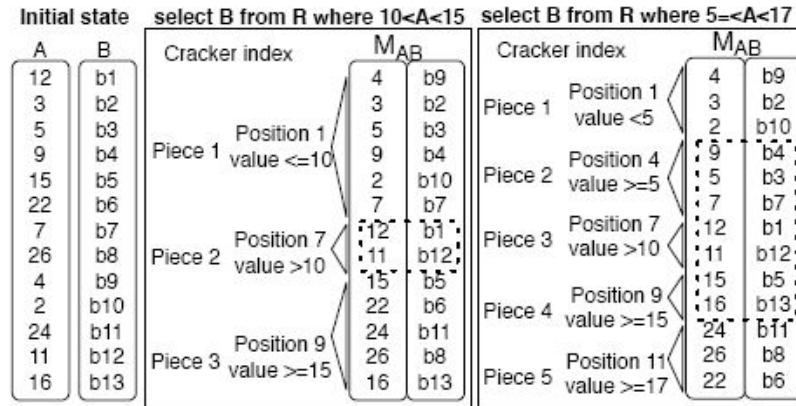


Figure 1: A simple example

3

Cracker-maps are created on demand. A cracker map  $M_{AB}$  is a two-column table over two attributes A and B of a relation R. Values of A are stored in the left column, while values of B are stored in the right column, called head and tail, respectively. All maps that have been created using A as head are collected in the map set  $S_A$  of R.A. For each cracker map  $M_{AB}$ , there is a cracker index that maintains information about how A values are distributed over  $M_{AB}$ .

Once a map  $M_{AB}$  is created by a query, it is used to evaluate query by triggering physical reorganization (cracking) of  $M_{AB}$  based on the restriction applied to A. We use the two cracking algorithms (crackInTwo, crackInThree) to physically reorganize maps by splitting a piece of a map into two or three new pieces.

Maps which are cracked independently using different restrictions on A can cause the result tuples to be not positionally aligned. This can be solved by using adaptive/on-demand alignment which applies all physical-reorganizations in same order to all maps in  $S_A$ . This is done by maintaining a log termed cracker-tape  $T_A$  of  $S_A$  that records all selections that trigger reorganization.

<sup>3</sup> Fig from S. Idreos, M. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column Stores. SIGMOD 09.

## Outline: Implementation tasks

- Creating cracker-maps.
- Self-organizing cracker-maps using 3 way and 2 way cracking.
- Maintaining an index on head column of cracker-map using BST(TreeMap from Java Collections API).
- Scan operator on cracker-maps which returns an iterator on qualifying tuples.
- Creating and maintaining Cracker tape for performing alignments across map sets.
- Multi-projection queries.
- Hybrid Crack Sort.
- Run all experiments with seeded random provider for generating data-sets and query ranges.
- Run the experiments to compare pre-sorted, cracking and hybrid crack-sort performance.

## Metrics and Evaluation

We compare our implementation that uses sideways-cracking<sup>[2]</sup> against pre-sorted data and hybrid crack-sort. This is done by comparing response times for each approach by measuring total cost, which is cost for selection and tuple-reconstruction.

For presorted data, selections are very fast since the data is sorted, tuple-reconstructions are also fast since the projection attributes are already aligned with the selection result. However there is high up-front cost for first query because it sorts the data.

We expect our implementation to achieve similar performance to presorted data, since cracking continuously aligns and clusters relevant data together for selections and tuple-reconstructions. Since cracking performs minimal amount of work to answer queries, there will be no up-front costs.

### Experiment Setup:

We compare our implementation of selection and sideways cracking against pre-sorted data (using Java Collections.sort API). We use JDK 1.8 with heap size of 10GB (16GB physically available) and running with default garbage collector with pre-allocated<sup>4</sup> heap. We trigger manual GC<sup>5</sup> after each run and sleep for 5 seconds to allow for sufficient GC time.

We do a **warm-up** run before performing the actual experiment on a query size of 1000 to account for **JIT** in the experiments.

---

<sup>4</sup> -XX:+AlwaysPreTouch

<sup>5</sup> System.gc(): We confirmed the GC behaviour using Java Mission Control.

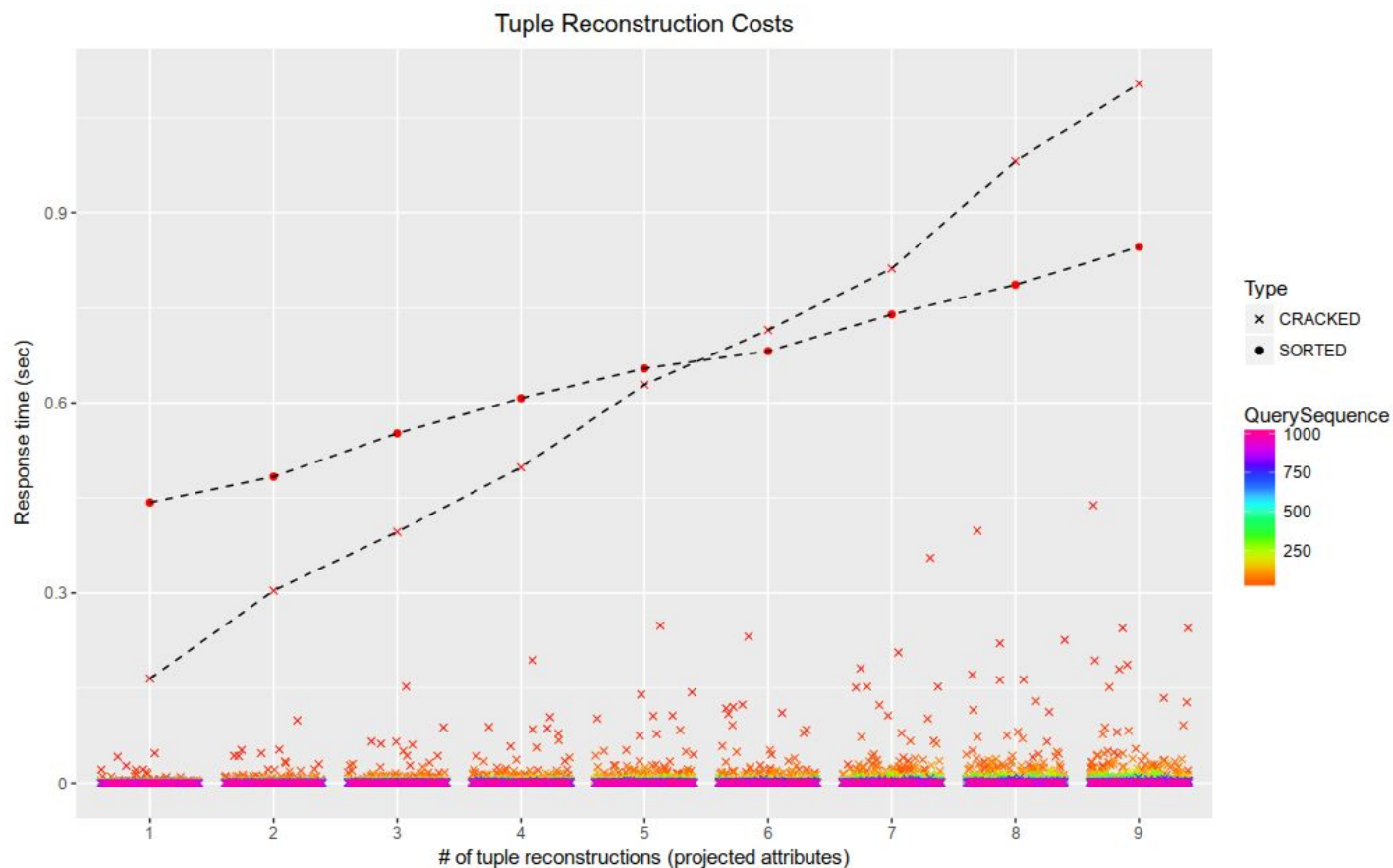
## Experiment 1: Varying Tuple Reconstruction

This experiment demonstrates the behavior for queries with one selection but multiple tuple reconstructions.

Query -> **select** A1, A2, ....., A9 **from** R **where** v1 < A0 < v2

We test for queries with 1 to 9 attributes in the select clause. For each case, we run 1000 queries requesting random ranges with selectivity of 1000, on a data-set of  $10^6$ .

The following figure shows the costs for tuple reconstructions. The **costs for first query is shown as dotted lines** in each sorted and cracked case.

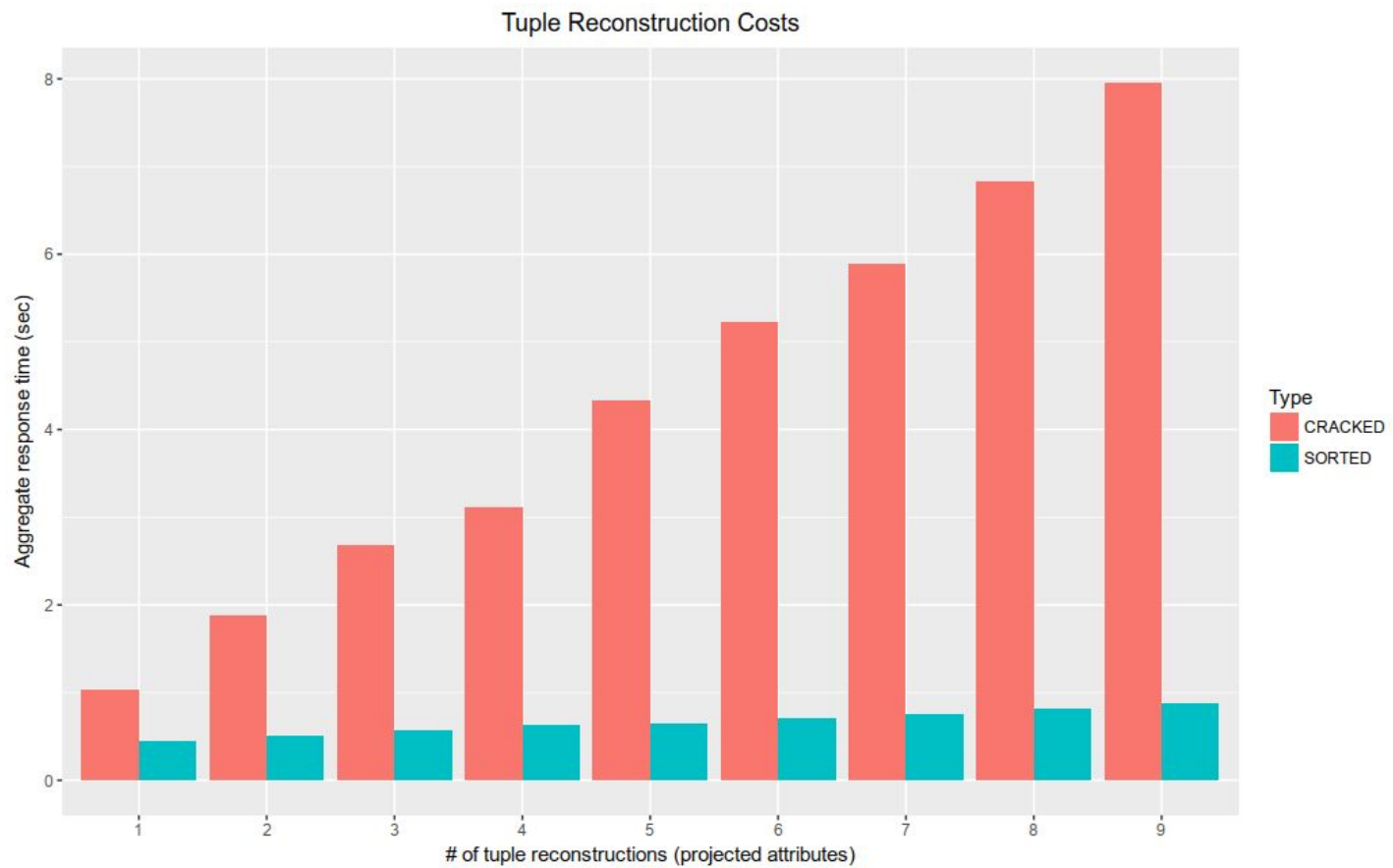


We see the cost for first query is higher in sorted dataset, this is because **first query** operation on sorted data requires creation of cracker-maps by copying data, sorting it and then perform the selection. The cost for **first query** in cracking case is small, this is because cracking performs the minimal amount of work to crack only the requested ranges. The first few cracker

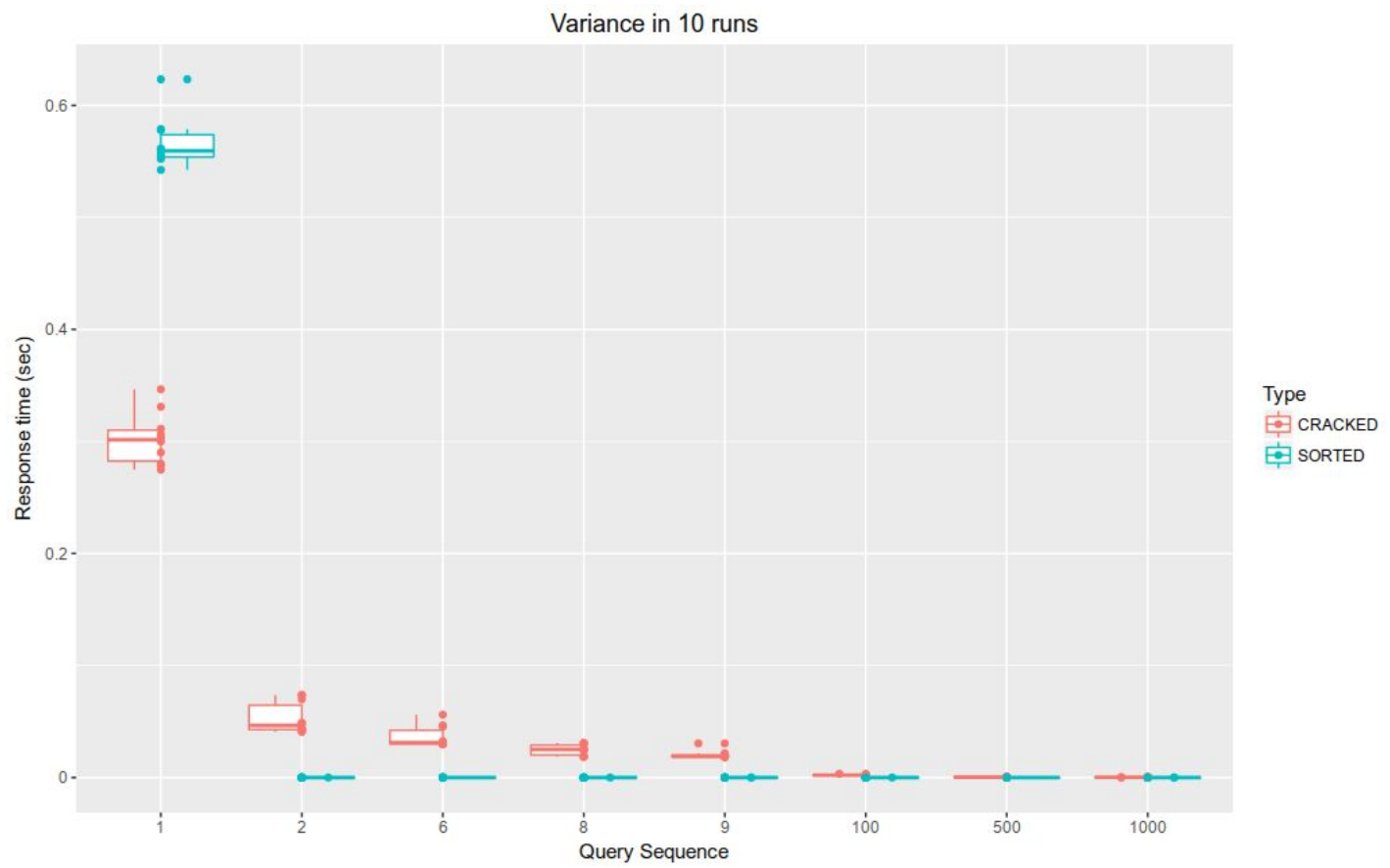


queries (in red) have high cost, this is either due to crack-in-three or expensive crack-in-two on both high and low partitions.

Following bar-plot shows **aggregate costs for tuple-reconstructions**. We can clearly see cracking cost increases linearly with size of projected attributes.



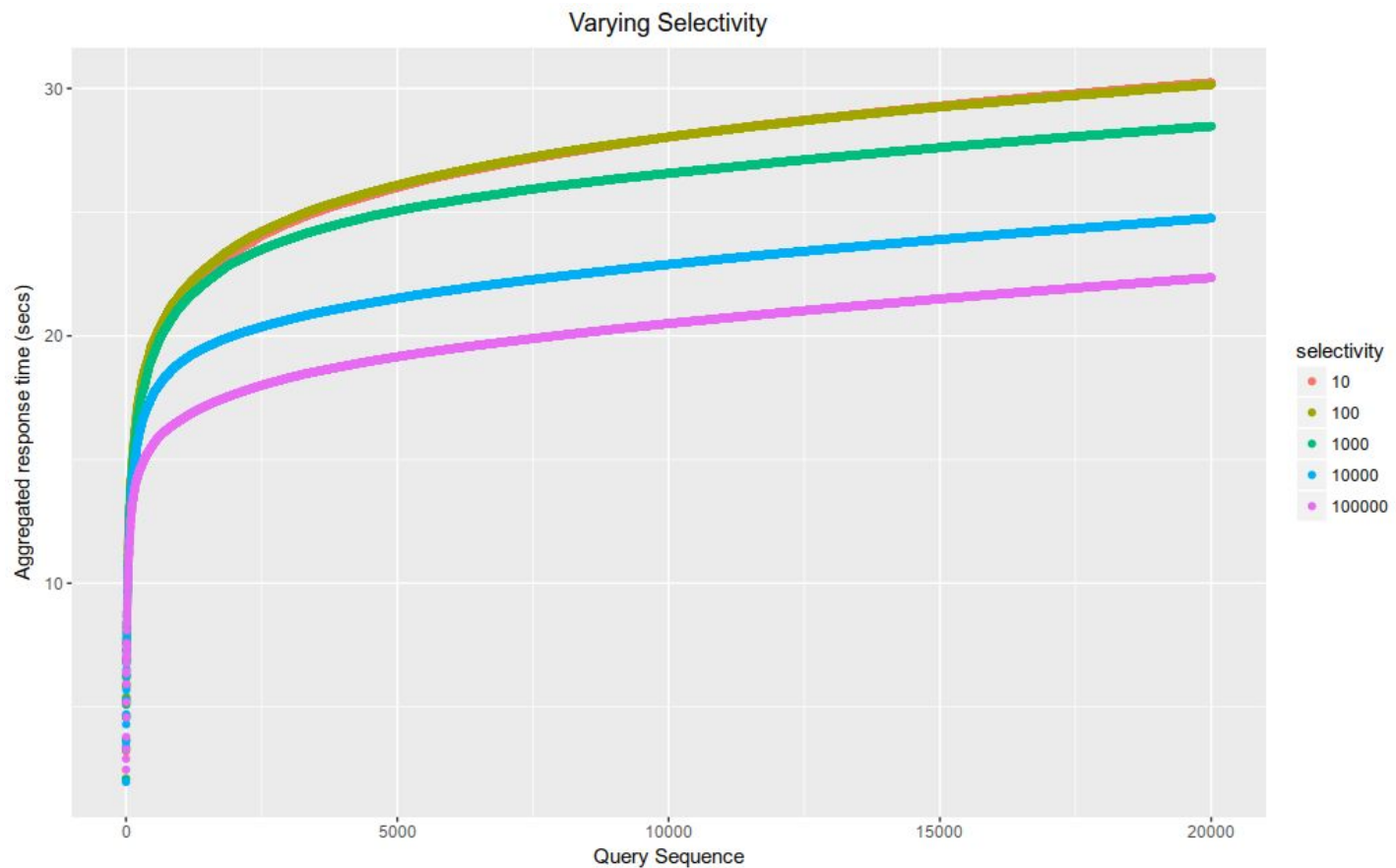
Following is a **box-plot** for tuple-reconstructions costs for 2 columns in 10 different runs on identical data and query sequence. We can see that the variance in costs for each query sequence between different runs is relatively small, demonstrating consistent behaviour between different runs.



## Experiment 2: Varying Selectivity

In this experiment, we study the effects of varying selectivity and compare aggregated response times for cracking. We vary the selectivity by requesting such ranges that the query size will always be **S** tuples.

On a column size of  $10^7$  tuples, a series of queries is fired. The experiment is repeated for queries with **S** ranging between  $[10, 10^5]$ . As in the previous experiment, all query ranges are randomly generated by a seeded provider.

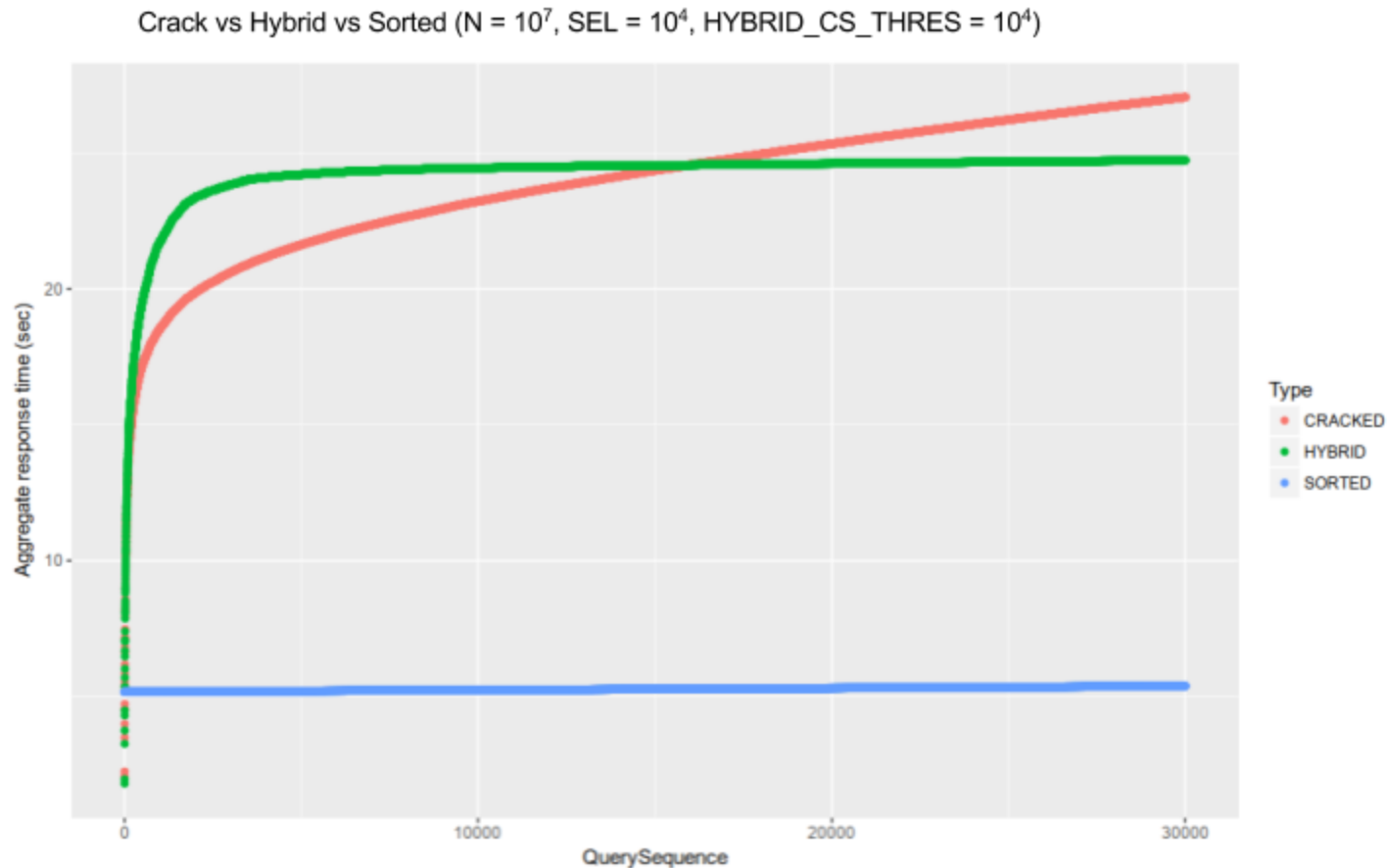


The above figure shows **aggregated** response times relative to selectivity **S** for each query sequence. This shows the cracking cost varies with selectivity. Higher selectivities leads to faster convergence and lower aggregated response times, whereas lower selectivities have high aggregate response times and convergence cost is higher compared to high selectivities.

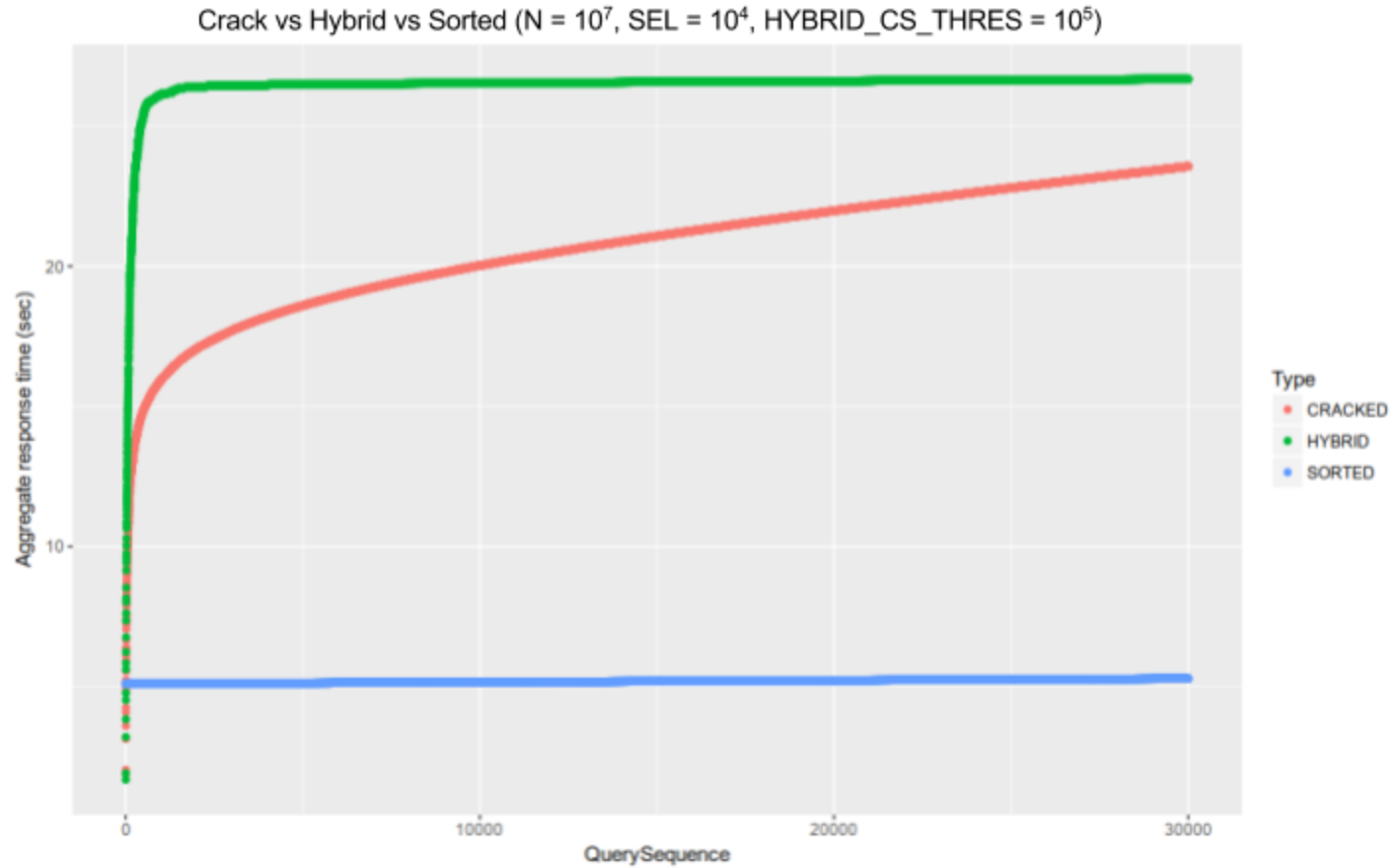
### Experiment 3: Accumulated query response time for Cracked, Sorted and Hybrid CS

In this experiment, we study the time to convergence between cracking, sorting and hybrid crack-sort.

We run 30,000 queries on a data-set of size  $10^7$  and a selectivity of  $10^4$ . In case of hybrid CS we run experiments for sorting threshold for  $10^4$  and  $10^5$ .



For the above graph, we compare **accumulated response times (secs)** for cracking, hybrid CS and sorting. We can see that for sorted case (blue) we have a high initial cost after which the cost for future queries is small and constant. In case of cracking (red), since we organize on each query the cost increase with number of queries. In case of hybrid crack-sort (green) with sorting threshold  $10^4$  we see that it has high initial sorting costs compared to cracking after which the future queries have small response time similar to the sorted case (indicated by flat-line).



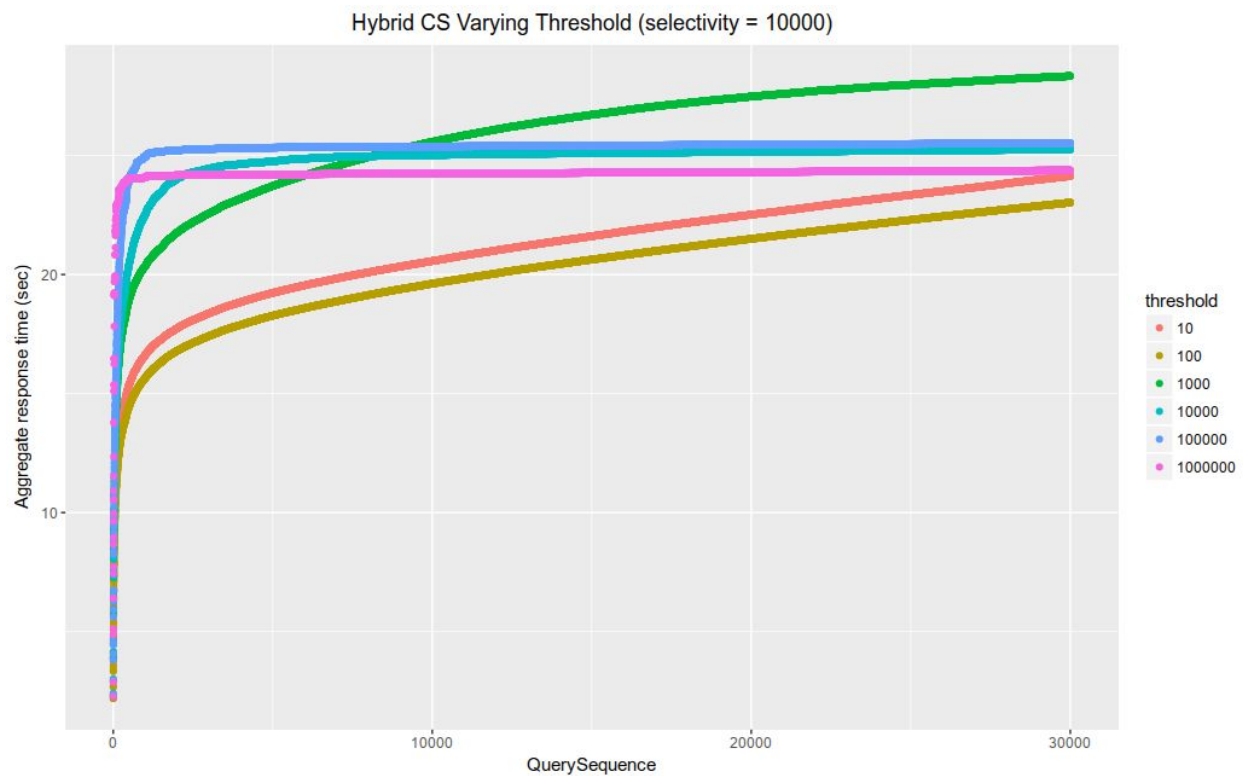
The above graph is similar to previous one but we increase the hybrid crack-sort sorting threshold to  $10^5$  from the previous  $10^4$ . We observe the initial sorting cost is higher than previous case but approaches convergence in fewer number of queries than previous case.

#### Experiment 4: Comparing different Hybrid CS thresholds

In this experiment, we study the time to convergence for hybrid crack-sort with varying thresholds.

We run 30,000 queries on a data-set of size  $10^7$  and a selectivity of  $10^4$ . We vary sorting threshold in range  $[10, 10^6]$ .

In the following graph we compare aggregated response times for varying threshold. We observe that for lower thresholds the initial queries are much faster but the cost increases with the number of queries since we have to sort more number of pieces, whereas for higher thresholds we have higher initial costs for sorting larger pieces but we quickly reach convergence after few queries.





## References

1. S. Idreos, M. Kersten, and S. Manegold. Database Cracking. CIDR 2007.
2. S. Idreos, M. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column Stores. SIGMOD 09.
3. D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. 2013