Progressive Indexing for Interactive Data Analysis

Pedro Holanda
CWI
holanda@cwi.nl

ABSTRACT

Index creation is one of the major difficult decisions in database schema design. Based on the expected queries, the database administrator needs to decide whether creating a specific index is worth the overhead of creating and maintaining it. This considerable up-front cost creates a trade-off that requires careful consideration and experimentation.

Index creation is especially challenging in exploratory and interactive data analysis, where workload patterns change frequently and queries are not known in advance. In these scenarios, it is not certain whether or not creating a complete index is worth the large initial cost. In spite of these challenges, indexing remains crucial for improving database performance. When no indexes are present, even simple point and range selections require expensive full table scans. When these operations are performed many times, indexes are essential to ensure fast query response times.

Adaptive indexing techniques seem like a promising solution. Techniques such as database cracking automatically build an index as a side effect of querying the data. An index is built for a column when it is first queried. As the column is queried more, the index is refined until it approaches the speed of a full index. This way, the cost of creating an index is smeared out over the cost of querying the data many times, and there is no large initial overhead for creating the index.

However, database cracking has several limitations. The first query that is issued is heavily penalized. In its first iteration, database cracking immediately requires the entire column to be copied, requiring $O(n)$ additional storage and time. Afterwards, it performs a potentially large amount of reshuffling around the query predicate. Because of this large initial investment, database cracking relies on the column being queried many times before paying for itself. If the column is only queried once, a large price is paid to create an index that is never used. It also introduces a large amount of unpredictability in query running time.

Database cracking is also not robust against malicious querying patterns. Because the data is pivoted around the query predicate, querying the data in an ordered fashion (from lowest value to highest value) will lead to cracking effectively performing an $O(n^2)$ sort while almost not improving the performance of subsequent queries. This querying pattern is also not particularly unusual; scanning a column from earliest to latest date will result in database cracking having extremely poor performance.

In this work, we introduce several novel progressive indexing techniques (e.g., progressive quicksort, progressive mergesort, progressive radixsort) that attempt to solve these issues of database cracking. We investigate their robustness in the face of different querying patterns and data distributions, and analyze their behavior by creating a cost model that describes the costs of performing different operations on these indices.

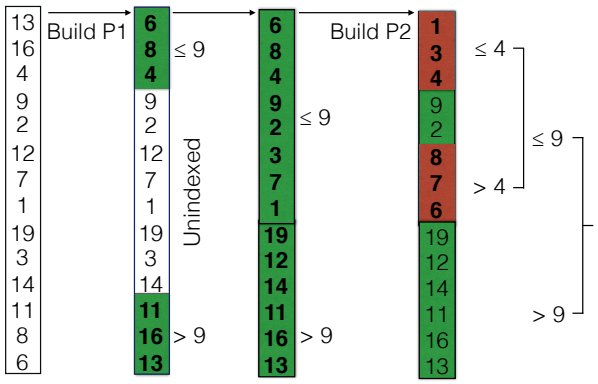

Figure 1: Progressive Quicksort.

Progressive indexing methods work by indexing a fraction Δ of the data every time a query is issued. This fraction Δ can either be fixed, in which case the queries will get progressively faster over time, or it can increase, in which case the index converges faster but queries only start to get faster after the index has been fully completed.

The main idea behind progressive indexing is that the fraction of the data that has already been indexed does not need to be scanned again, thus speeding up subsequent queries.

An example of one of our progressive indexing solutions, called progressive quicksort, is depicted in Figure 1. When the first query is issued, the build phase starts and the data is scanned from both the start and end of the column. We swap elements around, taking a random pivot in consideration, until a total of Δ elements have been scanned. While building, we also search for any elements that fulfill the query predicate. After the build phase is completed, we scan the unindexed $n - \Delta$ remaining elements to compute the answer to the query. In this example $\Delta = 6$ and the random pivots are 9 and 4.