# Progressive Indexes

Timbó Holanda, P.T.

# CHAPTER 2

# Background

In this chapter, we will cover the basic knowledge necessary to read this thesis. We will start with an overview of Relational Database Systems (Section 1) and their physical layout (Section 1.1). We continue with an overview of Interactive Exploratory Data Analysis (Section 2) and give a general explanation of indexing techniques and how they can be used to boost interactive exploratory queries (Section 3).

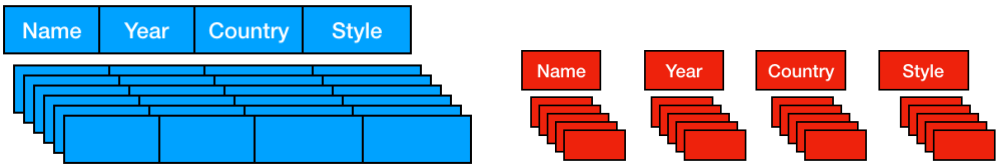
## 1 Relational Database Systems

Relational Database Systems (RDBMS) have been around since the early 70s. They are essential to any application that must access persistent data. They implement various techniques that guarantee data integrity, fast data access, transaction control and overall facilitate the development of a new application. The programmer does not need to worry about which data structures to represent his data, how to guarantee ACID (Atomicity, Consistency, Isolation, Durability) properties, or protecting his data against different types of corruption (e.g., hardware failures).

As an example, consider that a developer wants to create an online music store. He must store information about artists (e.g., their name, year they started, their country, and music style) and about their albums (e.g., album name, the year they were released, and the artist that made it). A simple way of storing this data would be to use text files (e.g., CSVs). However, the developer now has to implement methods to scan and write these files while being smart enough to store them on efficient data

structures. He also must use the correct algorithms to join the data in these files. And must deal with representation issues (e.g., how to store an album made as a collaboration of multiple artists?), transaction issues (e.g., what happens if two users alter the file simultaneously?) and data corruption (e.g., what happens if we are writing on the file and we experience a power shortage?). A more straightforward solution is to use RDBMSs, since they are designed to tackle these problems.

## 1.1 Physical Layout




(a) Row-Store. (b) Column-Store.

Figure 2-1: Physical layout of relational databases.

RDBMSs do not store data as text files but rather as a table composed of $n$ columns, where every row of this table represents a different entity with values for each of these columns (See Figure 2-1). An essential physical layout decision is choosing how the data should be partitioned, and there are two primary ways of doing it, a row-store or a column-store.

In the row-store model, data is partitioned in rows (i.e., the rows are stored consecutively in memory). This model is preferred for transactional workloads (i.e., when most queries update only a few tuples) since individual rows can be fetched computationally cheap. This model's main disadvantage is when you must retrieve a lot of data but not from all columns. Since rows are stored consecutively in memory, you will fetch data from columns you are not interested in, essentially wasting time. In a typical analytical scenario, the user is only interested in a small set of columns from the table, making this format unfit for data analysis.

In the column-store model, data is partitioned per column (i.e., the columns are stored consecutively in memory). This model is preferred for analytical queries since it is cheap to fetch individual columns, resulting in immense savings on disk access and memory bandwidth.

As an example, suppose our music-store table has 100 gigabytes of data, and different from the Figure 2-1 it is composed of 100 columns, also assume that the columns occupy the same amount of storage, 1 gigabyte per column. When executing an

analytical query interested in the number of albums released in 1980, the performance would significantly differ depending on the layout. In a row store, reading one column is equivalent to fetching all tuples, which at 100 megabytes per second (i.e., a typical hard-disk transfer speed) would take us about 17 minutes. In a column store, the same query can fetch the column that stores the albums' release date separately, so we only need to read 1 gigabyte of data, which takes about 10 seconds.

# 2 Interactive Exploratory Data Analysis

The workload from interactive data analysis is a type of analytical workload. The data scientist inspects a massive amount of data by issuing selective analytical queries (sometimes via a visualization tool) to test their hypothesis.

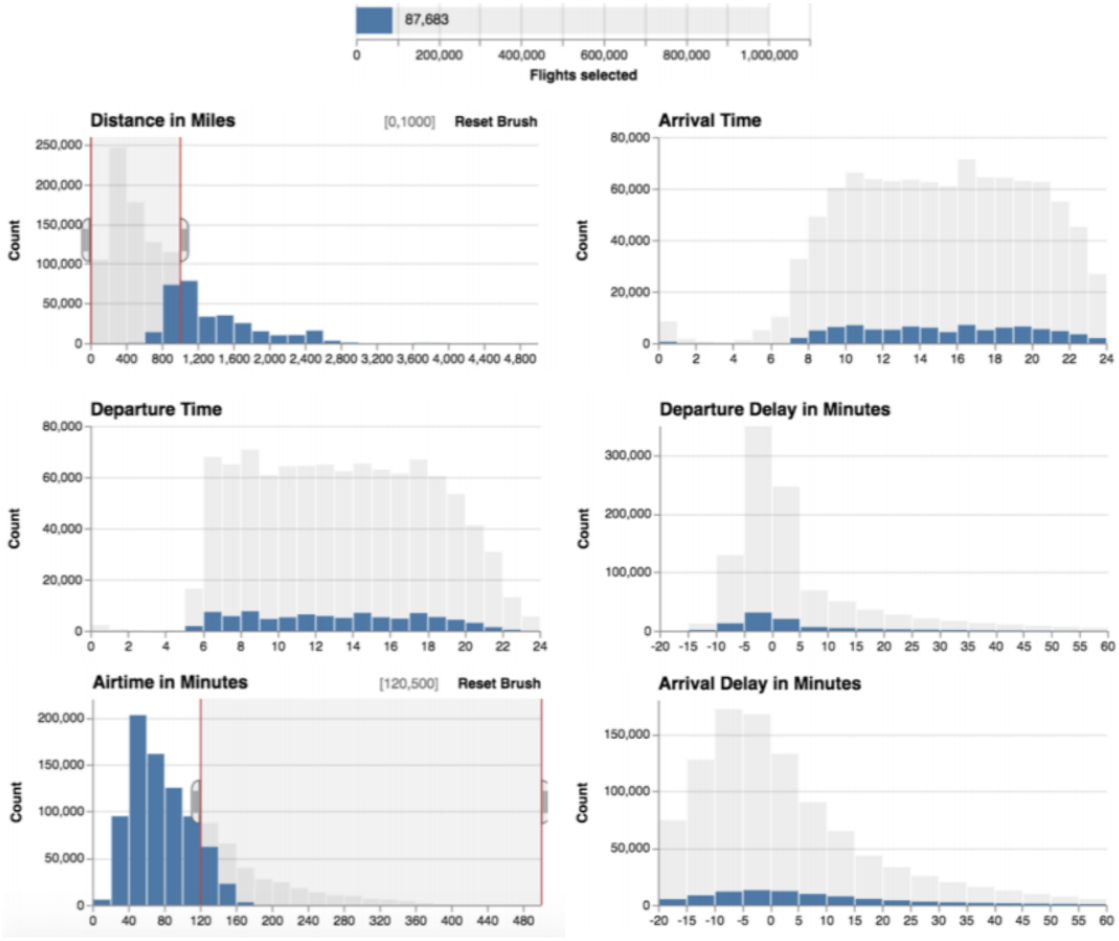



Figure 2-2: Interactive Data Analysis Example [4]

In [4], Battle et al. present cross-filter applications as the classical scenario of interactive data analysis. These applications consist of aggregate-filter-group queries

with users expecting almost immediate responses from the system.

Figure 2-2 depicts an example of a cross-filter application. It presents a dataset that contains flight information with six different attributes. The idea is that the data scientist can visualize each attribute as one of the histogram figures (e.g., the distance in miles histogram presents, from our selected flight, the number of flights that traveled a given amount of miles). The data scientist must interact with the range slider on top, and these figures are automatically updated depending on the filter's new inputs. It is easy to imagine that it will be quite frustrating if these figures are not immediately updated when changing the filter.

Since these workloads are dependent on a filter, when applying selective filters (e.g., wanting to know the information of a small number of flights), aggressive data skipping techniques like secondary index structures can significantly influence the query performance.

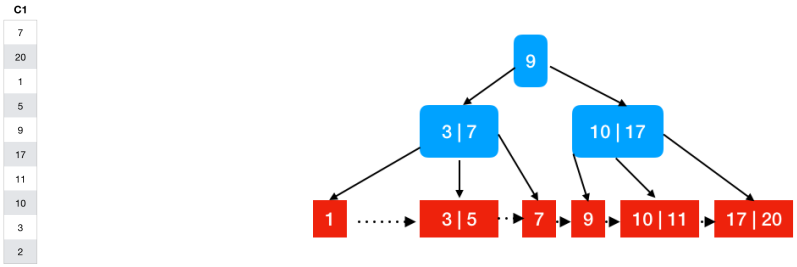
Let's go back to our music-store example from section 1.1, and let's assume that we want to know the quantity of all albums released in 1980 (also assuming that the selectivity is around 0.1% of all our data). When no index is present, a full scan of the column must be performed, which takes approximately 10 seconds. When using an index, we can access just the data that match our filter. Hence we only scan 0.1% of our data, with our query taking around 0.01 seconds to be fetched.

# 3 Index Structures

From our previous example, it becomes clear that, for highly selective queries (i.e., queries that filter most of the data), an index structure can significantly impact query performance. This impact exists because index structures can skip data that is not relevant to our query (i.e., not reading data that does not match our filter predicates).

Index structures come in all shapes and forms, covering different use cases. For example, the Adaptive Radix Tree (ART) [2, 40, 8] is designed to produce a compact index structure that is efficient for point-queries (i.e., queries with equality filters) and updates. At the same time, the B+Tree [22, 60] is optimized to execute range queries while not being as efficient as the ART for point-queries and updates.

Figure 2-3a depicts an example where the original data is composed of one column with unordered integers, and Figure 2-3b depicts a B+tree index created on this column. Note that the B+tree has the original data sorted in its leaves (i.e., red nodes) while the inner-nodes (i.e., blue nodes) are used to navigate the tree efficiently. When executing the following query `SELECT SUM(R.C1) FROM R WHERE R.C1 BETWEEN 3`




(a) Original Column (Column-Store). (b) B+-Tree Index.

Figure 2-3: Scan Vs Index.

`AND 6`, if we do not have an index, that means we must scan all the elements from our original column. However, if a B+Tree exists, we can quickly navigate the inner nodes and scan only the leaves with relevant data.

# 4 Index Selection Problem

A natural question arises after understanding the benefits of indexes. Why not create all possible indexes to speed up all possible filter queries? Although indexes boost query execution since they skip data that does not match filter predicates, they impost three different penalties to the DBMS. Indexes have a creation cost, a maintenance cost (i.e., every time an update happens, the index must be updated as well), and a storage cost (i.e., secondary indexes materialize a copy of the original data). Hence, the DBA must decide which indexes to create for a given database.

The DBA's goal is to decide a set of indexes to create for a table that will execute the workload as fast as possible while considering the amount of available memory. To do so, the database administrator must follow four steps: (1) Identify a relevant workload, (2) Create a search space with indexes that can potentially speed up this workload, (3) Perform a careful analysis on the maintenance and speed up trade-offs, (4) Assess the impact on the available memory.

Even when workloads are well known, selecting the optimal set of indexes is an NP-Hard problem [15], since it represents an analysis on all possible combinations of indexes that can be helpful to the workload. When the querying pattern is not known in advance, optimal a-priori index creation is impossible. To facilitate this process, two different types of solutions have been proposed. (1) automatic index selection and (2) adaptive index creation.

## 4.1 Automatic Index Selection

Automatic index selection techniques [1, 14, 58, 23, 13, 44, 53, 11] attempt to automatize the index selection process either completely or by giving hints of what indexes to create or drop to the DBA. In general, they work by capturing the workload, finding a set of indexes that optimize it, and either suggesting them for the DBA to create or by automatically creating them.

The process of finding a set of indexes can be driven by machine learning algorithms [44], or by the what-if architecture [13]. In the what-if architecture, the DBMS' query optimizer is used to predict the workload boost and the extra costs of maintaining and creating indexes using hypothetical indexes (i.e., it only creates the index's meta-data to force the optimizer to predict the costs if the index existed).

These solutions are well suited for the classical data warehouse scenario since the data warehouse scenario has a well-defined workload that rarely changes and has maintenance times (i.e., hours when the database is not being queried). The DBMS can exploit the maintenance time to perform full index creation. Since self-tuning tools can only be used when the system's workload is stable and known, they present several problems for interactive data analysis workloads. In an interactive environment, the workload is unknown or rapidly changes beyond what is known upfront. Besides, there is no specific idle time to invest in upfront full index creation. Hence automatic index selection techniques do not offer much help.

## 4.2 Adaptive Index Creation

Adaptive indexing techniques are an alternative to a-priori index creation. Instead of constructing the index upfront, the index is built as a by-product of querying the data. These techniques are designed for scenarios where the workload is unknown, and there is no idle time to invest in index creation. Their main goal is to smear out the high investment of creating an up-front full index over the execution of several queries.

Database Cracking [36] (also known as "Standard Cracking") is the original adaptive indexing technique. It works by physically reordering the index while processing queries. It consists of two data structures: a cracker column (i.e., a copy of the original column) and a cracker index (i.e., a binary search tree that holds information on where pieces offsets and maximum value).

Each incoming query cracks the column into smaller pieces and then updates the cracker index concerning those pieces. As more queries are processed, the cracker index converges towards a full index.

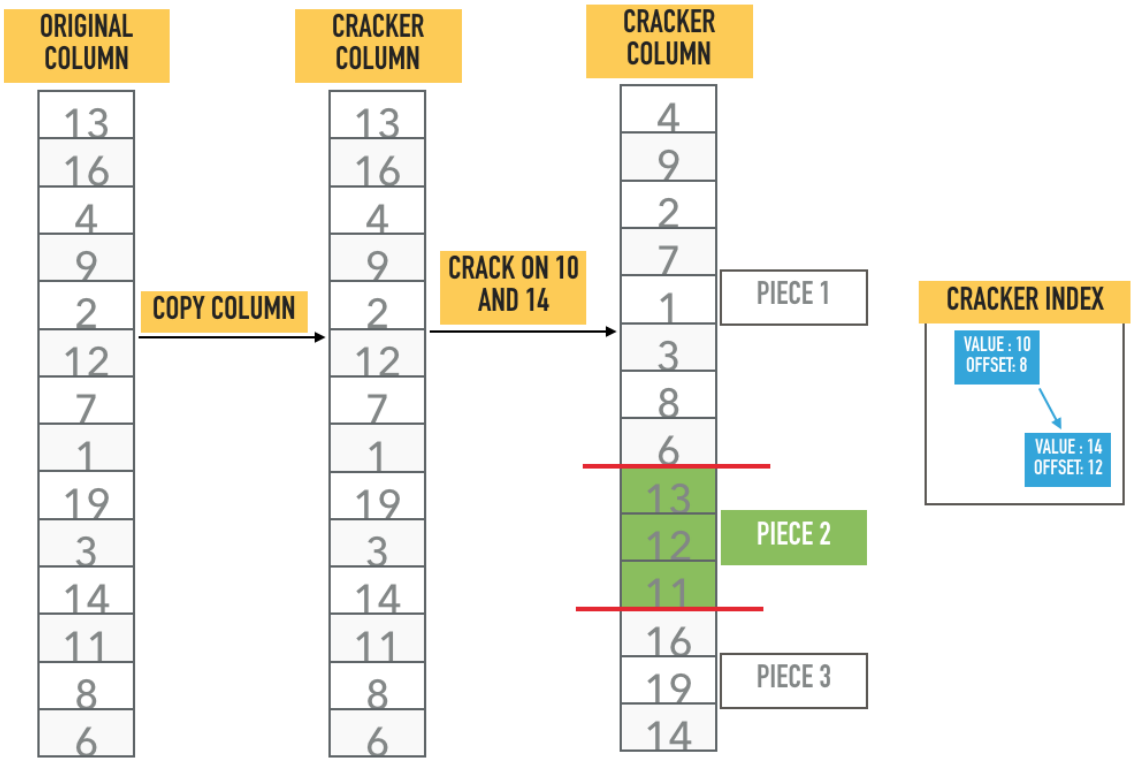



Figure 2-4: Standard Cracking executing filter $C > 10$ and $C < 14$.

Figure 2-4 depicts an example of standard cracking executing a query that requests all values higher than 10 and lower than 14, and the original column has no index yet. When this query is executed, it triggers the first step of database cracking, which performs a full copy of the original column. After copying it to a structure called *cracker column*, it performs two quick-sort iterations using, as quick-sort pivots, the query predicates 10 and 14. This results in a cracker column cracked into three pieces. Where *Piece 1* has all elements up to our first query predicate (i.e., 10), *Piece 2* all elements between our query predicates (i.e., 10 and 14), and *Piece 3* with all elements above or equal to the second predicate (i.e., 14). The information regarding the pieces (i.e., where each piece start and the highest element within that piece is stored in an AVL-Tree [6] (i.e., a binary search tree with self-balancing properties) called *cracker index*. When the next query is executed, the system can already take advantage of this index (e.g., if a query only has one filter $c > 18$, only *Piece 3* needs to be checked). After the first query, the pieces are refined even further by performing new quick-sort iterations with pivots equal to the currently executing filter predicates.

While database cracking accomplishes its mission of constructing an index as a by-product of querying, it suffers from several problems that make it unsuitable for interactive data analysis: (1) it adds a significant overhead over naive scans in the first

iterations of the algorithm, (2) the performance of cracking is not robust, as sudden changes in workload cause spikes in performance, and (3) convergence towards a full index is slow and workload-dependent.

There is a large body of work on extending and improving database cracking. These improvements include better convergence towards a full index [21, 50], more predictable performance [49, 26], more efficient tuple reconstruction [35, 37, 50], better CPU utilization [46], other cracking engines [47, 25], predictive query processing [57], using modern hardware to boost query execution [39], using mediocre elements as cracking pivots [62], creating multidimensional adaptive indexes [45], generalizing database cracking [49] and handling updates [34, 29].