Lecture #5: Storage Models & Compression

15-445/645 Database Systems (Fall 2023)
https://15445.courses.cs.cmu.edu/fall2023/
Carnegie Mellon University
Andy Pavlo + Jignesh Patel

1 Database Workloads

OLTP: Online Transaction Processing

An OLTP workload is characterized by fast, short running operations, repetitive operations and simple queries that operate on single entity at a time. OLTP workloads typically handle more writes than reads, and only read/update a small amount of data each time.

An example of an OLTP workload is the Amazon storefront. Users can add things to their cart and make purchases, but the actions only affect their account.

OLAP: Online Analytical Processing

An OLAP workload is characterized by long running, complex queries and reads on large portions of the database. In OLAP workloads, the database system is often analyzing and deriving new data from existing data collected on the OLTP side.

An example of an OLAP workload would be Amazon computing the most bought item in Pittsburgh on a day when it's raining.

HTAP: Hybrid Transaction + Analytical Processing

A new type of workload (which has become popular recently) is HTAP, where OLTP and OLAP workloads are present together on the same database.

2 Storage Models

There are different ways to store tuples in pages. We have assumed the n-ary storage model so far.

N-Ary Storage Model (NSM)

In the n-ary storage model, the DBMS stores all of the attributes for a single tuple contiguously in a single page. This approach is ideal for OLTP workloads where requests are insert-heavy and transactions tend to operate only an individual entity. It is ideal because it takes only one fetch to be able to get all of the attributes for a single tuple.

Advantages:

- Fast inserts, updates, and deletes.
- Good for queries that need the entire tuple.

Disadvantages:

- Inefficient for scanning large portions of the table and/or a subset of the attributes.

Decomposition Storage Model (DSM)

In the decomposition storage model, the DBMS stores a single attribute (column) for all tuples contiguously in a block of data. Thus, it is also known as a "column store." This model is ideal for OLAP workloads with many read-only queries that perform large scans over a subset of the table's attributes.

Advantages:

- Reduces the amount of I/O wasted because the DBMS only reads the data that it needs for that query.
- Better query processing because of increased locality and cached data reuse.
- Better data compression.

Disadvantages:

- Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.

To put the tuples back together when using a column store, there are two common approaches: The most commonly used approach is fixed-length offsets. Here, the value in a given column will belong to the same tuple as the value in another column at the same offset. Therefore, every single value within the column will have to be the same length.

A less common approach is to use embedded tuple ids. Here, for every attribute in the columns, the DBMS stores a tuple id (ex: a primary key) with it. Then, the system would also store a mapping to tell it how to jump to every attribute that has that id. Note that this method has a large storage overhead because it needs to store a tuple id for every attribute entry.

Partition Attributes Across (PAX)

In the hybrid Partition Attributes Across storage model, the DBMS vertically partitions attributes within a database page. The goal of doing so is to get the benefit of faster processing on columnar storage while retaining the spatial locality benefits of row storage.

In PAX, the rows are horizontally partitioned into groups of rows. Within each row group, the attributes are vertically partitioned into columns. Each row group is similar to a column store for its subset of the rows.

A PAX file has a global header containing a directory with offsets to the file's row groups, and each row group maintains its own header with meta-data about its contents.

3 Database Compression

Compression is widely used in disk-based DBMSs as disk I/O is (almost) always the main bottleneck. It is especially popular in systems that have read-only analytical workloads. The DBMS can fetch more useful tuples if they have been compressed beforehand at the cost of greater computational overhead for compression and decompression.

In-memory DBMSs are more complicated since they do not have to fetch data from disk to execute a query. Memory is much faster than disks, but compressing the database reduces DRAM requirements and processing. They have to strike a balance between **speed** vs. **compression ratio**. Compressing the database reduces DRAM requirements. It may decrease CPU costs during query execution.

If data sets were completely random bits, there would be no way to perform compression. However, there are key properties of real-world data sets that are amenable to compression:

- Highly *skewed* distributions for attribute values (e.g., Zipfian distribution of the Brown Corpus).

- High *correlation* between attributes of the same tuple (e.g., Zip Code to City, Order Date to Ship Date).

Given this, we want a database compression scheme to have the following properties:

- Must produce fixed-length values. The only exception is variable length data stored in separate pools. This is because the DBMS should follow word-alignment and be able to access data using offsets.
- Allow the DBMS to postpone decompression as long as possible during query execution (**late materialization**).
- Must be a **lossless** scheme because people do not like losing data. Any kind of **lossy** compression has to be performed at the application level.

Compression Granularity

The kind of data we want to compress greatly affects which compression schemes can be used. There are four levels of compression granularity:

- **Block Level:** Compress a block of tuples for the same table.
- **Tuple Level:** Compress the contents of the entire tuple (NSM only).
- **Attribute Level:** Compress a single attribute value within one tuple. Can target multiple attributes for the same tuple.
- **Columnar Level:** Compress multiple values for one or more attributes stored for multiple tuples (DSM only). This allows for more complicated compression schemes.

4 Naive Compression

The DBMS compresses data using a general purpose algorithm (e.g., gzip, LZO, LZ4, Snappy, Brotli, Oracle OZIP, Zstd). Although there are several compression algorithms that the DBMS could use, engineers often choose ones that often provides lower compression ratio in exchange for faster compression/decompression.

An example of using naive compression is in **MySQL InnoDB**. The DBMS compresses disk pages, pads them to a power of 2KBs and stores them into the buffer pool. However, every time the DBMS tries to read/modify data, the compressed data in the buffer pool must first be decompressed.

Since accessing data requires decompression of compressed data, this limits the scope of the compression scheme. If the goal is to compress the entire table into one giant block, using naive compression schemes would be impossible since the whole table needs to be compressed/decompressed for every access. Therefore, MySQL breaks the table into smaller chunks since the compression scope is limited.

Another problem is that these naive schemes also do not consider the high-level meaning or semantics of the data. The algorithm is oblivious to both the structure of the data, and how the query is planning to access the data. Thus, this eliminates the opportunity to utilize late materialization, since the DBMS cannot tell when it can delay the decompression of data.

5 Columnar Compression

Run-Length Encoding (RLE)

RLE compresses runs (consecutive instances) of the same value in a single column into triplets:

- The value of the attribute
- The start position in the column segment

- The number of elements in the run

The DBMS should sort the columns intelligently beforehand to maximize compression opportunities. This clusters duplicate attributes, thereby increasing the compression ratio. Note that the effectiveness of RLE greatly depends on the underlying data characteristics (e.g. number and frequency of attributes in each data).

Bit-Packing Encoding

When all values for an attribute are less than the value's declared largest size, store them with fewer bits.

Mostly Encoding

Bit-packing variant that uses a special marker to indicate when a value exceeds the largest size and then maintains a look-up table to store them.

Bitmap Encoding

The DBMS stores a separate bitmap for each unique value of a particular attribute where an offset in the vector corresponds to a tuple. The i^{th} position in the bitmap corresponds to the i^{th} tuple in the table that indicates whether that value is present or not. The bitmap is typically segmented into chunks to avoid allocating large blocks of contiguous memory.

This approach is only practical if the value cardinality is low, since the size of the bitmap is linearly proportional to the cardinality of the attribute value. If the cardinality of the value is high, then the bitmap can become larger than the original data set.

Delta Encoding

Instead of storing exact values, record the difference between values that follow each other in the same column. The base value can be stored in-line or in a separate look-up table. We can also use RLE on the stored deltas to get even better compression ratios.

Incremental Encoding

This is a type of delta encoding whereby common prefixes or suffixes and their lengths are recorded so that they need not be duplicated. This works best with sorted data.

Dictionary Compression

The most common database compression scheme is dictionary encoding. The DBMS replaces frequent patterns in values with smaller codes. It then stores only these codes and a data structure (i.e. the dictionary) that maps these codes to their original value. A dictionary compression scheme needs to support fast encoding/decoding, as well as range queries.

Encoding and Decoding: The dictionary needs to decide how to **encode** (convert uncompressed value into its compressed form) and **decode** (convert compressed value back into its original form) data. As such, it is not possible to use hash functions.

The encoded values also need to support sorting in the same order as original values (i.e. **order-preserving encodings**). This ensures that the compressed queries run on compressed data return results that are consistent with uncompressed queries run on the original data. This order-preserving property allows operations to be performed directly on the codes.

Note: Columnar compression schemes work best with read heavy workloads and may need additional support for writes.