Lecture #11: System Catalogs and Database Compression

15-721 Advanced Database Systems (Spring 2018)
http://15721.courses.cs.cmu.edu/spring2018/
Carnegie Mellon University
Prof. Andy Pavlo

1 System Catalogs

Almost every DBMS stores their database catalog in itself. This requires specialized code for "bootstrapping" catalog tables. The meta-data are tuples stored in the database, but the DBMS code itself is written an imperative language. Thus, we need to wrap object abstraction around tuples.

The entire DBMS should be aware of transactions in order to automatically provide ACID guarantees for DDL commands and concurrent transactions. Actions that modify the catalog should be treated like transactions to allow for ACID guarantees.

1.1 Schema Changes

Add Column:

- NSM: Copy tuples into new region in memory.
- DSM: Just create the new column segment.

Drop Column:

- NSM # 1: Copy tuples into new region of memory.
- NSM # 2: Mark column as "deprecated, clean up later.
- DSM: Drop the column and free memory.

Change Column:

- Check whether the conversion is allowed to happen.
- Depends on default values.

Create Index

- Scan the entire table and populate the index. Can use one or multiple threads. This is done in the context of a transaction.
- While the system is building the index, it also has to record changes made by transactions that modified the table while another transaction was building the index.
- When the scan completes, the index building transaction locks the table and resolve changes that were missed after the scan started. This ensures that there are no false negatives or false positives.

Drop Index

- Drop the index logically from the catalog
- It only becomes "invisible" when the transaction that dropped it commits
- All existing transactions will still have to update it

1.2 Sequences

- Typically stored in the catalog and used for maintaining a global counter. Also called "auto-increment" or "serial" keys.
- Sequences are not maintained with the same isolation protection as regular catalog entries. This is because rolling back a transaction that incremented a sequence does not rollback the change to that sequence.

2 Database Compression

I/O is (almost) always the main bottleneck if the DBMS has to fetch data from disk. Thus, compression in these systems (almost) improves performance.

In-memory DBMSs are more complicated since they do not have to fetch data from disk to execute a query. But compressing the database reduces DRAM requirements and processing. They have to strike a balance between **speed** vs. **compression ratio**. In-memory DBMSs always choose speed.

There are key properties of real-world data sets that are amenable to compression:

- Data sets tend to have highly **skewed** distributions for attribute values.
- Data sets tend to have high **correlation** between attributes of the same tuple.

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

- Must produce fixed-length values.
- Allow the DBMS to postpone decompression as long as possible during query execution.
- 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.

2.1 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. This approach can target multiple attributes for the same tuple. It is possible to
- **Columnar Level:** Compress multiple values for one or more attributes stored for multiple tuples (DSM only).

3 Naive Compression

Compress data using a general purpose algorithm (e.g., gzip). Scope of compression is only based on the data provided as input.

The data **must** be decompressed first before it can be read and (potentially) modified. This limits the scope of the compression scheme. These schemes also do not consider the high-level meaning or semantics of the data

There are two types of compression algorithms:

- **Entropy Encoding:** More common sequences use less bits to encode, less common sequences use more bits to encode.
  Example: Huffman Coding

- **Dictionary Encoding:** Build a data structure that maps data segments to an identifier. Replace those segments in the original data with a reference to the segments position in the dictionary data structure. Example: Most general purpose compression algorithms.

4 Encoding Schemes

The alternative to a naive compression scheme is one where the DBMS encodes the database itself in a compression format. The DBMS's query processing algorithms need to be aware of the data's encoding scheme in order to decipher it.

4.1 Run-Length Encoding (RLE)

Compress runs of the same value in a single column into triplets [3].

- The value of the attribute
- The start position in the column segment
- The # of elements in the run

Requires the columns to be sorted intelligently to maximize compression opportunities.

4.2 Bitmap Encoding

Store a separate Bitmap for each unique value for a particular attribute where an offset in the vector corresponds to a tuple [2].

- The i^{th} position in the Bitmap corresponds to the i^{th} tuple in the table
- Typically segmented into chunks to avoid allocating large blocks of contiguous memory

Only practical if the value cardinality is low.

4.3 Delta Encoding

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. Can be combined with RLE to get even better compression ratios.

4.4 Incremental Encoding

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.

4.5 Mostly Encoding

When the values for an attribute are "mostly" less than the largest size, you can store them as a smaller data type. The remaining values that cannot be compressed are stored in their raw form in a separate space. From **Amazon Redshift**.

5 Dictionary Compression

The most common database compression scheme is dictionary encoding [1]. The DBMS replaces frequent pattern with smaller codes. A dictionary compression scheme needs to support fast encoding/decoding, as well as range queries.

5.1 Dictionary Construction

Approach #1 – All-at-Once:

- Compute the dictionary for all the tuples at a given point of time.

- New Tuples must use a separate dictionary or all the tuples must be recomputed.

Approach #2 – Incremental:

- Merge new tuples in with an existing dictionary.
- Likely requires re-encoding to existing tuples.

5.2 Dictionary Scope

Approach #1 – Lock Level:

- Only include a subset of tuples within a single table.
- Build dictionaries per block.
- Potentially lower compression ratio, but adding new tuples is easier.

Approach #2 – Table Level:

- Construct a dictionary for the entire table.
- Better compression ratio, but expensive to update.

Approach #3 – Multi-Table:

- Can be either subset or entire tables.
- Sometimes helps with joins and set operations. For example, if two tables are connected via a foreign key and if the columns in both tables use the same dictionary, them the DBMS does not have to re-encode the values when it joins those tables together.

5.3 Encoding and Decoding

A dictionary needs to support two operations:

- **Encoding**: For a given uncompressed value, convert it into its compressed form.
- **Decode**: For a given compressed value, convert it back into its original form.
- No magic hash function will do this for us.
- Requires two data structures to support operations in both directions. The encoded values need to support sorting in the same order as original values.

5.4 Dictionary Implementations

Approach #1 – Hash Table:

- Fast and compact.
- Unable to support range and prefix queries.

Approach #1 – B+ Tree:

- Slower than a hash table and takes more memory.
- Can support range and prefix queries.
- Use a shared leaf node design to minimize the amount of redundant data [1].

References

[1] C. Binning, S. Hildenbrand, and F. Farber. Dictionary-based order-preserving string compression for main memory column stores. *SIGMOD '09: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 283–296, 2009. URL https://dl.acm.org/citation.cfm?id=1559877.

[2] P. E. O'Neill. Model 204 architecture and performance. In *High Performance Transaction Systems*, pages 39–59. 2005. URL https://link.springer.com/chapter/10.1007%2F3-540-51085-0_42#citeas.

[3] A. M. Roth and S. J. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3):31–39, Sept 2013. URL https://dl.acm.org/citation.cfm?id=163096.