15 OCTOBER 2024

TOASTed JSONB data in PostgreSQL: performance tests of different compression algorithms

TOAST (The Oversized Attribute Storage Technique) is PostgreSQL's mechanism for handling large data objects that exceed the 8KB data page limit. Introduced in PostgreSQL 7.1, TOAST is an improved version of the out-of-line storage mechanism used in Oracle databases for handling large objects (LOBs). Both databases store variable-length data either inline within the table or in a separate structure. PostgreSQL limits the maximum size of a single tuple to one data page. When the size of the tuple, including compressed data in a variable-length column, exceeds a certain threshold, the compressed part is moved to a separate data file and automatically chunked to optimize performance.

TOAST can be used for storing long texts, binary data in bytea columns, JSONB data, HSTORE long key-value pairs, large arrays, big XML documents, or custom-defined composite data types. Its behavior is influenced by two parameters: TOAST_TUPLE_THRESHOLD and TOAST_TUPLE_TARGET. The first is a hardcoded parameter defined in PostgreSQL source code in the heaptoast.h file, based on the MaximumBytesPerTuple function, which is calculated for four toast tuples per page, resulting in a 2000-byte limit. This hardcoded threshold prevents users from storing values that are too small in out-of-line storage, which would degrade performance. The second parameter, TOAST_TUPLE_TARGET, is a table-level storage parameter initialized to the same value as TOAST_TUPLE_THRESHOLD, but it can be adjusted for individual tables. It defines the minimum tuple length required before trying to compress and move long column values into TOAST tables.

In the source file heaptoast.h, a comment explains: "If a tuple is larger than TOAST_TUPLE_THRESHOLD, we will try to toast it down to no more than TOAST_TUPLE_TARGET bytes through compressing compressible fields and moving EXTENDED and EXTERNAL data out-of-line. The numbers need not be the same, though they currently are. It doesn't make sense for TARGET to exceed THRESHOLD, but it could be useful to make it be smaller." This means that in real tables, data stored directly in the tuple may or may not be compressed, depending on its size after compression. To check if columns are compressed and which algorithm is used, we can use the PostgreSQL system function pg_column_compression. Additionally, the pg_column_size function helps check the size of individual columns. PostgreSQL 17 introduces a new function, pg_column_toast_chunk_id, which indicates whether a column's value is stored in the TOAST table.

In the latest PostgreSQL versions, two compression algorithms are used: PGLZ (PostgreSQL LZ) and LZ4. Both are variants of the LZ77 algorithm, but they are designed for different use cases. PGLZ is suitable for mixed text and numeric data, such as XML or JSON in text form, providing a balance between compression speed and ratio. It uses a sliding window mechanism to detect repeated sequences in the data, offering a reasonable balance between compression speed and compression ratio. LZ4, on the other hand, is a fast compression method designed for real-time scenarios. It offers high-speed compression and decompression, making it ideal for performance-sensitive applications. LZ4 is significantly faster than PGLZ, particularly for decompression, and processes data in fixed-size blocks (typically 64KB), using a hash table to find matches. This algorithm excels with binary data, such as images, audio, and video files.

In my internal research project aimed at understanding the performance of JSONB data under different use cases, I ran multiple performance tests on queries that process JSONB data. The results of some tests showed interesting and sometimes surprising performance differences between these algorithms. But presented examples are anecdotal and

cannot be generalized. The aim of this article is to raise an awareness that there can be huge differences in performance, which vary depending on specific data and use cases and also on specific hardware. Therefore, these results cannot be applied blindly.

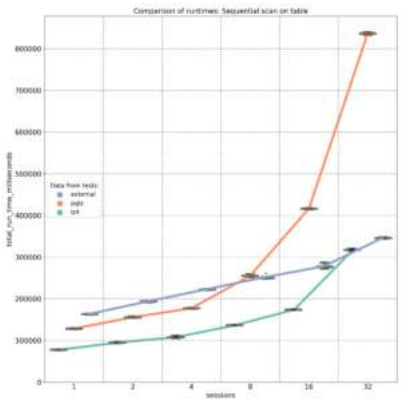
JSONB data is stored as a binary object with a tree structure, where keys and values are stored in separate cells, and keys at the same JSON level are stored in sorted order. Nested levels are stored as additional tree structures under their corresponding keys from the higher level. This structure means that retrieving data for the first keys in the top JSON layer is quicker than retrieving values for highly nested keys stored deeper in the binary tree. While this difference is usually negligible, it becomes significant in queries that perform sequential scans over the entire dataset, where these small delays can cumulatively degrade overall performance.

The dataset used for the tests consisted of GitHub historical events available as JSON objects from gharchive.org covering the first week of January 2023. I tested three different tables: one using PGLZ, one using LZ4, and one using EXTERNAL storage without compression. A Python script downloaded the data, unpacked it, and loaded it into the respective tables. Each table was loaded separately to prevent prior operations from influencing the PostgreSQL storage format.

The first noteworthy observation was the size difference between the tables. The table using LZ4 compression was the smallest, around 38GB, followed by the table using PGLZ at 41GB. The table using external storage without compression was significantly larger at 98GB. As the testing machines had only 32GB of RAM, none of the tables could fit entirely in memory, making disk I/O a significant factor in performance. About one-third of the records were stored in TOAST tables, which reflected a typical data size distribution seen by our clients.

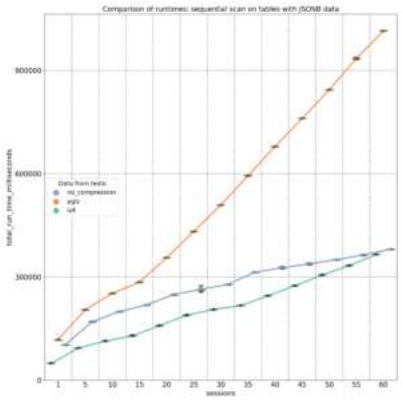
To minimize caching effects, I performed several tests with multiple parallel sessions running testing queries, each with randomly chosen parameters. In addition to use cases involving different types of indexes, I also ran sequential scans across the entire table. Tests were repeated with varying numbers of parallel sessions to gather sufficient data points, and the same tests were conducted on all three tables with different compression algorithms.

The first graph shows the results of select queries performing sequential scans, retrieving JSON keys stored at the beginning of the JSONB binary object. As expected, external storage without compression (blue line) provides nearly linear performance, with disk I/O being the primary factor. On an 8-core machine, the PGLZ algorithm (red line) performs reasonably well under smaller loads. However, as the number of parallel queries reaches the number of available CPU cores (8), its performance starts to degrade and becomes worse than the performance of uncompressed data. Under higher loads, it becomes a serious bottleneck. In contrast, LZ4 (green line) handles parallel queries exceptionally well, maintaining better performance than uncompressed data, even with up to 32 parallel queries on 8 cores.



The second test targeted JSONB keys stored at different positions (beginning, middle, and end) within the JSONB binary object. The results, measured on a 20-core machine, demonstrate that PGLZ (red line) is slower than the uncompressed table right from the start. In this case, the performance of PGLZ degrades linearly, rather than geometrically, but still

lags significantly behind LZ4 (green line). LZ4 consistently outperformed uncompressed data throughout the test.



But if we decide to change the compression algorithm, simply creating a new table with the default_toast_compression setting set to “lz4” and running INSERT INTO my_table_lz4 SELECT * FROM my_table_pglz; will not change the compression algorithm of existing records. Each already compressed record retains its original compression algorithm. You can use the pg_column_compression system function to check which algorithm was used for each record. The default compression setting only applies to new, uncompressed data; old, already compressed data is copied as-is.

To truly convert old data to a different compression algorithm, we need to recast it through text. For JSONB data, we would use a query like: INSERT INTO my_table_lz4 (jsonb_data, ...) SELECT jsonb_data::text::jsonb, ... FROM my_table_pglz; This ensures that old data is stored using the new LZ4 compression. However, this process can be time and resource-intensive, so it’s important to weigh the benefits before undertaking it.

To summarize it – my tests showed significant performance differences between the PGLZ and LZ4 algorithms for storing compressed JSONB data. These differences are particularly pronounced when the machine is under high parallel load. The tests showed a strong degradation in performance on data stored with PGLZ algorithm, when the number of parallel sessions exceeded the number of available cores. In some cases, PGLZ performed worse than uncompressed data right from the start. In contrast, LZ4 consistently outperformed both uncompressed and PGLZ-compressed data, especially under heavy loads. Setting LZ4 as the default compression for new data seems to be the right choice, and some cloud providers have already adopted this approach. However, these results should not be applied blindly to existing data. You should test your specific use cases and data to determine if conversion is worth the time and resource investment, as converting data requires re-casting and can be a resource-intensive process.

ABOUT THE AUTHOR

Josef Machytka

View posts