Tailoring TSXOR for Compression of Regular Interval Time Series in Timescale

Alexander Stiwi[1,*], Manoel Rüffer[1], Hanna Köpcke[2] and Tobias Teich[1]

[1]University of Applied Sciences Zwickau, 08066 Zwickau, Germany
[2]University of Applied Sciences Mittweida, 09648 Mittweida, Germany

Abstract

This paper presents a relational integration of the TSXor compression algorithm for regularly sampled time-series data. We design and implement a schema and compression pipeline within TimescaleDB, allowing lossless compression and in-database decompression using SQL and PL/pgSQL. Our approach replaces full timestamps with delta-based offsets and encodes floating-point values using reference-based XOR encoding. We evaluate the system on multiple real-world datasets, showing moderate compression ratios (up to 23%) and fast decompression performance. While the SQL-based implementation introduces storage and execution overhead, the integration enables transparent querying and lays the foundation for further database-native optimizations of time-series compression.

Keywords

Time-series compression, TSXor, TimescaleDB, in-database processing, delta encoding

1. Introduction

Time-series data, composed of chronologically ordered observations, underpin a wide range of modern applications. From real-time patient monitoring in healthcare to sensor-driven industrial automation and financial trend analysis, time series have become a fundamental data type in both operational and analytical systems [1]. The rapid growth of sensors and devices has generated large volumes of time-stamped data, creating a data deluge that challenges database storage and query performance.

Relational time-series databases like TimescaleDB offer strong integration and query features, including continuous aggregates and chunked storage. Yet, they face limitations handling large, high-frequency datasets, even with built-in compression. Effective lossless compression remains essential for improving scalability, performance, and cost-efficiency—especially in domains requiring exact value preservation, such as billing or scientific computing.

TSXor [2] is a recently proposed lossless compression algorithm tailored for time-series data. It exploits the predictability of sequential readings by encoding each new measurement as an XOR difference relative to a small window of prior values, significantly reducing storage requirements for datasets with repeated or slowly changing patterns. Moreover, TSXor's design enables fast decompression, making it well suited for real-time query workloads.

This paper integrates the TSXor compression algorithm into TimescaleDB, targeting regularly sampled time-series data. A key challenge in this implementation is that standard SQL lacks native support for bitwise operations on floating-point values or other low-level binary manipulation. We overcome this limitation by leveraging PostgreSQL's procedural languages (PL/pgSQL and PL/Python) to perform the XOR computations and to detect leading and trailing zero bits in 64-bit IEEE 754 double-precision values.

We propose a custom relational schema to efficiently encode both time and value fields. Instead of storing full time-stamps, our approach uses chunk-level delta encoding to record time intervals

36th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 29 - October 01, 2025, Regensburg, Germany

*Corresponding author.
alexander.stiwi@whz.de (A. Stiwi); manoel.rueffer@whz.de (M. Rüffer); koepcke@hs-mittweida.de (H. Köpcke); tobias.teich@whz.de (T. Teich)
0000-0003-2501-2609 (H. Köpcke)

published 2025-12-03

implicitly, greatly reducing metadata overhead. Compressed measurements are stored using a compact header in a SMALLINT column alongside a binary payload in a BYTEA column, thereby enabling selective per-chunk or per-value decompression and even parallel processing through standard SQL queries. This design remains fully compatible with TimescaleDB's chunked architecture and is implemented entirely using native database capabilities.

We evaluate our implementation on multiple real-world datasets to assess its compression effectiveness and runtime performance. TSXor achieves a 15–25% reduction in storage size with acceptable decompression times. While the compression procedure incurs additional CPU overhead, the resulting format is well suited for analytical scenarios in which data are compressed once but queried repeatedly.

We make the following contributions:

- We adapt and implement the TSXor compression algorithm entirely inside a SQL-based relational database, overcoming PostgreSQL's lack of native bitwise operations on floating-point data by using server-side procedural extensions. To support reproducibility, we have made the full implementation publicly available in a GitLab repository at https://gitlab.com/toobi7007/tsxor-sql.
- We design a relational storage schema for compressed time-series data that supports efficient storage of regular-interval measurements and enables selective per-chunk or per-value decompression through standard SQL queries.
- We evaluate our integrated solution on real-world datasets in TimescaleDB, reporting compression ratios (up to ~23% size reduction) and query performance that demonstrate the practicality of our approach for large-scale time-series analytics.

The remainder of the paper is structured as follows: Section 2 reviews related work in time-series compression and database integration. Section 3 provides background on the TSXor algorithm. Section 4 details our implementation of TSXor inside TimescaleDB. Section 5 presents the experimental evaluation results, and Section 6 concludes with insights and future directions.

2. Related Work

Time-series compression has been widely studied in the context of ever-growing IoT, financial, and sensor data volumes. A recent survey by Chiarot and Silvestri [3] emphasizes that effective approaches exploit temporal redundancy and predictable patterns. Among lossless methods, Gorilla [4] pioneered delta-of-delta encoding for timestamps and XOR encoding for 64-bit floats, achieving high compression and very fast decompression by producing long sequences of zero bits. FPC [5] similarly compresses floating-point series by storing XOR differences of consecutive values with variable-length codes, balancing compression ratio and CPU cost. TSXor [2] builds on these foundations using a sliding window of up to 127 prior readings (instead of only the last value) so that repeated or similar past values can be referenced. This extended window allows superior compression on periodic or repeating time series at a slight overhead in metadata; Bruno et al. [2] report that TSXor significantly outperforms Gorilla's per-point XOR scheme on regular-interval datasets.

In recent years, researchers have proposed new compression algorithms designed for integration into database systems. Liakos et al. [6] introduced Chimp, which generalizes Gorilla by selecting a previous value that maximizes leading and trailing zero bits in the XOR (roughly halving Gorilla's storage), and Patas, which aligns encoded bits to byte boundaries for faster decompression with only a minor compression penalty. Both were implemented as built-in DuckDB codecs. Elf [7] pushes XOR compression further by deliberately zeroing a few least-significant bits of each value to produce longer zero runs, achieving the best compression among XOR-based methods at the cost of extra computation. Adaptive schemes have also been explored: Chen et al. [8] switch between multiple encoding strategies per chunk (AFC), and Afroozeh et al. [9] combine fixed-point conversion with bit packing for floats (ALP). These techniques can surpass Gorilla, FPC, and TSXor in compression ratio while remaining efficient; notably, ALP's vectorized implementation in DuckDB demonstrates that even

Reference	01100100
XOR	1101110101000001 + n Bytes
Exception	11111111 + 8 Bytes

Table 1
Stored data per encoding case (colors mark fields): case flag (green), window offset (blue), trailing-zero count TZ (brown), and length of differing bytes (magenta).

complex compression algorithms can be integrated into a columnar DBMS without harming query performance.

In contrast to prior work—which typically implements compression at the engine level or in low-level languages—our approach integrates TSXor entirely within an unmodified relational DBMS (TimescaleDB on PostgreSQL) using only SQL and built-in stored procedures. We overcome the lack of native bitwise operations on `DOUBLE PRECISION` values by invoking server-side Python (PL/Python) to XOR 64-bit IEEE 754 representations and detect zero runs, and we fit TSXor's format into standard SQL types. In particular, we pack each value's encoding metadata into a 16-bit `SMALLINT` column by repurposing unused bits of the original TSXor header (expanding its reference window from 7 to 11 bits). This design eliminates per-row binary blob overhead and enables efficient bit manipulation via SQL. The result is a fully database-native, chunk-based compression scheme integrated into TimescaleDB's hypertable architecture, accessible through standard SQL queries. This demonstrates that an advanced time-series compressor like TSXor can be realized inside a relational system, achieving significant space savings (15–25% in our experiments) without sacrificing queryability or requiring engine modifications.

3. Time Series Compression with TSXor

A time series $TS = [(t_1, x_1), \ldots, (t_n, x_n)]$ with $x_i \in \mathbb{R}$ at time t_i can be stored as a table A. TSXor compresses this raw table A into a smaller table A_c by scanning the data with a sliding window of up to 127 prior values. For each new value, TSXor selects one of three encoding cases. If the value exactly repeats a previous reading in the window, a 1-byte back-reference to that earlier entry is stored (Reference case). If not, TSXor computes the XOR against the closest prior value and stores only the significant (non-zero) portion of the XOR result, along with counts of leading and trailing zero bits (XOR case). If the value still cannot be efficiently encoded by the above methods, TSXor falls back to storing the full 64-bit value (Exception case).

Table 1 summarizes the bit format for each case. A 1-bit flag indicates the case type, followed by a 7-bit offset identifying the referenced index in the window. For an XOR encoding, two additional 4-bit fields record the number of trailing zero bits and the length in bytes of the remaining XOR difference. The 7-bit offset allows a window size of $W = 127$, which provides a good compression balance without requiring more bits. An Exception is indicated by a special offset value (all 7 offset bits set to 1, i.e., 127) and is followed by the full 8-byte value. Decompression simply reverses this process, using the stored offset and XOR metadata to recover each original value.

4. Tailoring TSXor in TimescaleDB

To integrate TSXor into TimescaleDB, we designed a specialized relational schema for the compressed table A_c. We create a table named `tsxor_comp` with columns `id` (SMALLSERIAL), `header` (SMALLINT), and `val` (BYTEA). Each tuple's `id` is an auto-incremented index within its chunk, ensuring the insertion order is preserved. The 16-bit `header` stores TSXor's encoding metadata (case flag, reference offset, trailing-zero count), and `val` holds the variable-length payload bytes (if any) for that entry. Since we focus on regular-interval time series, we avoid storing full timestamps for every data point. Instead, we maintain a separate metadata table `tsxor_meta` with columns `chunk_id`, `begin` (timestamp of the chunk's start), `delta` (time interval), and `chunkname`. For each chunk, `begin` and `delta` define the timeline so that an original timestamp for a row with index i can be reconstructed as $\texttt{begin} + i \times \Delta$. In

Reference	0110010011111111	
XOR	1101110101000001	11111111
Exception	1111111111111111	

Table 2
Original TSXor header bits with PostgreSQL overhead (red) and the 1-bit case flag (green), 7-bit offset (blue), 4-bit length of differing bytes (magenta) and 4-bit TZ (brown) fields; gray highlights the 2-byte boundary.

Reference	011001001101
XOR	1101110110010100
Exception	111111111111

Table 3
Final TSXor header format: 1-bit case flag (green), 11-bit offset (blue), 4-bit TZ (brown); gray indicates 2-byte alignment.

other words, we replace each 8-byte timestamp [10] with a 2-byte index [11], saving 6 bytes per reading. This delta-based timestamp reconstruction is exact (no information loss) and takes advantage of the fixed-interval assumption.

Another challenge in a pure SQL implementation arises from PostgreSQL's storage and data type limitations. Storing binary data in a `BYTEA` field adds an additional length prefix of one or four bytes per entry, depending on the internal memory structure [12]. Furthermore, standard SQL does not allow bitwise operations on `DOUBLE PRECISION` values [13]. We address these issues by redesigning the TSXor header format and selecting suitable data types. The TSXor encoding header consists of a case flag, a reference offset, and a trailing-zero (TZ) count. These fields are packed into a 16-bit integer that fits into a `SMALLINT` column. By storing the header in a `SMALLINT` instead of as binary data, we eliminate the per-row `BYTEA` overhead and enable efficient bit manipulation in SQL through the use of bit shifts and masks. To achieve this 16-bit header, we slightly modified the original TSXor format. In the original design, a 4-bit field encoded the length of the differing bytes for the XOR case (needed to parse a linear binary stream). In our relational schema, the length of the XOR payload can be derived from the stored `val` field itself, making that field redundant. We therefore repurposed those 4 bits to extend the reference offset field from 7 bits to 11 bits. This allows a larger window size ($W = 2^{11} - 1 = 2047$) for looking up past values, compared to the original $W = 127$, potentially improving compression for long periodic patterns. Crucially, all three encoding cases now fit into a two-byte header. Tables 2 and 3 illustrate the header bit layout before and after this optimization. In the final format, the header contains a 1-bit case flag (green), an 11-bit offset (blue), and a 4-bit TZ count (brown), aligned exactly to 16 bits. These schema adaptations solve multiple implementation issues: the smallint header allows fast bit operations in SQL, removing the need for external bit parsing; avoiding a large `BYTEA` for the header saves storage overhead; and the expanded offset range slightly enhances compression effectiveness. Overall, we achieve a compact, lossless representation tailored to TimescaleDB's chunked architecture and suitable for in-database processing.

4.1. Compression Function ($A \rightarrow A_c$)

We implemented the TSXor compression algorithm as a stored procedure in PL/pgSQL [14] (with a helper in PL/Python). Each chunk is compressed independently, which means chunks can be processed in parallel. For a given hypertable of raw data A, the compression function scans through the values in chronological order, maintaining an in-memory sliding window of up to W recent values (initially empty). For each new data point x_i, the algorithm determines how to encode it based on the previous values in the window. First, we check if x_i exactly matches any value in the window. If so, we choose the *Reference* case, storing an offset to that previous occurrence and no additional data bytes. If not, we compute the XOR of x_i with each candidate in the window to find the one that maximizes the number of leading and trailing zero bits in the XOR result (this indicates the most compressible difference). To

perform this bit-level comparison on 64-bit floats, we call a PL/Python helper function that converts the DOUBLE PRECISION values to their 8-byte binary representation. After that, we calculate the XOR and zero-bit counts. Once the "best match" value is identified, we decide between the remaining cases: if the XOR difference can be stored more compactly than the 64-bit raw value, we use the *XOR* case (flag = 1 with an offset to the best-match value, plus the XOR difference bytes); otherwise, we resort to the *Exception* case (flag = 1 with a special offset and the full 8-byte value stored). After classifying the case, we construct the 16-bit header by packing the fields. For example, we place the trailing-zero count (TZ) into the header by shifting it into position and masking:

```
tz_bits := (best_match.tz << TZ_OFFSET) & TZ_MASK;
```

We combine the case flag and the reference offset into the header using predefined bit masks and shifts. The constants such as `TZ_OFFSET` and `TZ_MASK` define the bit positions of the fields. The new compressed tuple is then appended to `tsxor_comp` by inserting the assembled `header` together with the payload `val`. Depending on the encoding case, the payload may be empty in the Reference case, contain a few bytes in the XOR case, or consist of the full 8-byte value in the Exception case. The `id` column is assigned automatically in increasing order. We structured the compression procedure into distinct phases: maintaining the sliding window, comparing values at the bit level to find the best match, constructing the header, and inserting the output. Each phase can be understood and optimized separately, which made it easier to work within PL/pgSQL's constraints and to integrate the required Python calls for type conversion. Compressing each chunk in isolation aligns with TimescaleDB's architecture and also allows the operation to scale across multiple chunks, for example by running parallel jobs for different time partitions.

4.2. Decompression Function ($A_c \rightarrow A$)

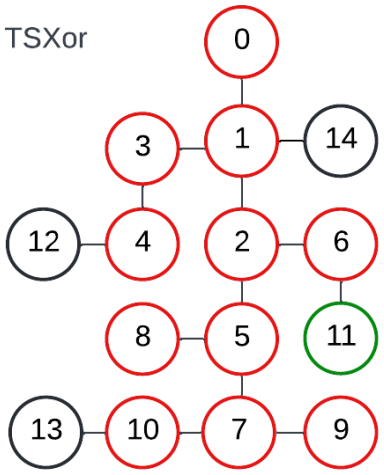
The decompression process reverses the compression, reconstructing the original series A from its compressed form A_c. We implement this as a PL/pgSQL function that can restore either entire chunks or individual values on demand. Given the set of compressed tuples in table `tsxor_comp`, we retrieve each tuple's header and value bytes (`val`) and invert the encoding. Importantly, because each compressed row carries all information needed to recover that data point, decompression does not require scanning the entire dataset sequentially.

For a full-chunk decompression, we simply iterate over the compressed rows in increasing `id` order. Decoding each compressed entry works as follows:

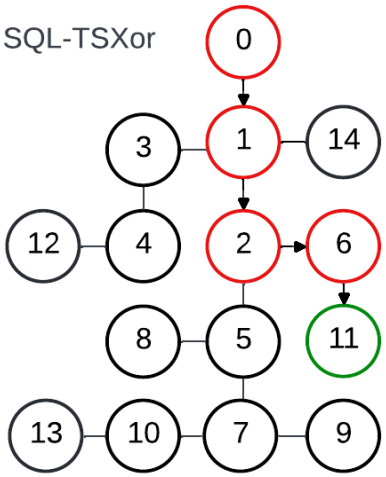
- **Reference copy (flag = 0):** The header indicates the value is a duplicate of a recent reading. We read the offset from the header and copy the value from the prior entry at `id` minus the offset. Because we process in order, that referenced value will already have been decompressed.
- **XOR difference (flag = 1, offset ≠ 2047):** The header indicates an XOR-encoded delta. The offset tells us which earlier reading serves as the reference. We retrieve the previously decoded value at `id` − offset and XOR it with the stored `val` bytes. Before XORing, we pad these bytes with the appropriate number of zero-bytes (as recorded in the header) to reassemble the full 64-bit IEEE 754 representation. Applying this XOR yields the original value.
- **Exception value (flag = 1, offset = 2047):** This is a special case indicating no shorter XOR encoding was possible. In this case, the `val` field contains the complete 8-byte binary representation of the original value. We therefore read those bytes directly (or cast them to DOUBLE PRECISION) to obtain the original value.

In this way, we reconstruct the sequence of original readings $x_1, x_2, \ldots, x_n$ exactly. Typically, the first reading x_1 in each chunk is stored as an Exception (since there is no prior value to reference), providing a starting point for decoding subsequent values.

A key advantage of our relational TSXor format is its support for *targeted* decompression of specific points or ranges without having to decompress an entire chunk. Every compressed tuple is identified



(a) Original TSXor decompression illustrated as a graph: the target value (node 11, green) can only be recovered by traversing a chain of intermediate values (red nodes).



(b) Relational (SQL-based) TSXor decompression: using direct references, the target value (node 11, green) is reached via a shorter path that bypasses most intermediate nodes (red).

Figure 1: TSXor decompression

by its chunk and a row index `id`, enabling direct lookups. For example, given a timestamp t within a known chunk, we can determine its row index $i = (t - begin)/\Delta$ via the metadata table and directly retrieve that compressed row (along with any others it depends on) for decoding. In the original TSXor design (as with many streaming compressors), decoding an arbitrary point could require materializing a long chain of prior values because each value references a previous one and so on. This scenario is illustrated in Figure 1a, where to obtain the target value (shown as green node 11) the decoder must traverse through many intermediate nodes (red). By contrast, in our adapted scheme the stored offset in each header lets us jump directly to the referenced entry without sequentially processing all others. Figure 1b shows how the path to decode the target is dramatically shortened: the green node 11 can be reached via a direct path that bypasses most of the red intermediate nodes. In practice, this means that to retrieve a particular reading, we can fetch only the needed reference entries (via index lookups in the table) and apply the XOR operations, rather than unpacking every value along the way. This direct-addressing capability not only reduces work for point queries, but also enables parallel decompression of multiple points or sub-intervals of the series. Overall, our decompression function leverages the relational model for flexible retrieval—one can reconstruct entire chunks efficiently or isolate and decode only specific readings as needed. The combination of delta-based time offsets and compact `SMALLINT` headers ensures that the original time series is recovered exactly (i.e., losslessly) with minimal overhead, all within the PostgreSQL/TimescaleDB environment.

Decompression Complexity: Each compressed reading in TSXor encodes a reference offset to one of the last W values in a sliding window and a bitmask of differing bits. During decompression, the decoder maintains a buffer of the last W decompressed values in memory, which allows each new value to be reconstructed with only a constant amount of work: a direct array access and a single XOR operation. This yields $O(1)$ time per value and overall $O(N)$ time for a series of length N, with a very small constant factor. In typical time-series data, many readings repeat or differ in only a few bits, which makes the XOR operation extremely fast. Empirical results show that decompressing a complete chunk is an order of magnitude faster than compressing it, so the decoding step adds very little latency to query execution. Memory overhead is also modest: aside from the output array of length N, the algorithm only needs to keep the recent W values and a small buffer for the XOR bitmasks. Space usage is therefore $O(N)$ for the output and essentially $O(1)$ extra working space, since W is fixed and small. These properties make TSXor decompression both time-efficient and memory-efficient for typical workloads.

5. Evaluation

5.1. Experimental Setup

We evaluated our implementation using five real-world time series datasets, each stored as a TimescaleDB hypertable (compression on plain tables was much slower). The first dataset, MPI, contains weather observations from the Max Planck Institute for Biogeochemistry in Jena [15]. We used the readings from July–December 2023 (timestamp and barometric pressure only) and partitioned the hypertable into 4-day time chunks. The second dataset, OPS [16], comes from the Open Power System Data project. It provides Great Britain's solar generation (using attributes timestamp and GB_UKM_solar_generation_actual); we truncated this series to end at 2016-11-13 13:00:00 and used 8-day chunks. The third dataset, ETTh, is the Electricity Transformer Temperature (ETT) collection [17], from which we used the hourly HUFL field (High Useful Load) with its timestamps. The fourth dataset, IHEPC [18], is the Individual Household Electric Power Consumption dataset. We combined its separate date and time fields into a single timestamp and kept only the Global_active_power measurement; to ensure a clean time series, missing entries (marked with "?" in the source data) were replaced with 0. We limited IHEPC to the first 32,767 records and organized it into a hypertable with 2-day chunks. The final dataset, JPNA [19], is the monthly balance sheet of Japan's central bank, which we used as provided (monthly frequency, no preprocessing beyond date parsing). All datasets were ingested into a TimescaleDB hypertable and compressed using our TSXor functions.

5.2. Results

Table 4 reports the execution times for compressing each dataset ($A \rightarrow A_c$) and for decompressing ($A_c \rightarrow A$). As expected, decompression is significantly faster than compression in all cases. For instance, compressing the MPI series (390 seconds) took roughly 15× longer than decompressing it (25 seconds); similar speedups (10–20×) hold for the other datasets. The largest dataset (IHEPC) required about 9 minutes to compress 32k points, while its decompression finished in only 32 seconds. These runtimes are acceptable for offline compression jobs, but they are much higher than the original TSXor implementation's performance (which was in C++). The overhead stems from our use of SQL and PL/pgSQL, which are interpreted and not optimized for intensive bitwise operations. We observed that the compression runtime grows rapidly with larger window sizes – in fact, using the maximum 11-bit window (2048 reference points) caused the SQL implementation to run for several hours. For practicality, we limited the window to 7 bits (127 points) in our experiments, which kept compression times to the order of minutes. Future optimizations, such as leveraging a lower-level language extension or using temporary tables to break up the task, could further improve the compression speed.

Table 5 shows the compression factors achieved by TSXor on each dataset, measured as *original_size/compressed_size*. For example, the MPI data is reduced by a factor of 1.180× (meaning the compressed table A_c is about 85% of the original size A), and OPS achieves 1.228× (compressed to 81% of original). In practical terms, we realize about 15–23% storage savings on those datasets. The IHEPC dataset sees a smaller gain (factor 1.085× ≈ 8% reduction), and the Bank JPN data does not compress at all (factor 1×) due to its short, irregular series. However, when considering the total storage footprint including PostgreSQL's page overhead and TOAST storage for BYTEA columns, the effective compression is lower. The "Total Relation" factors in Table 5 drop to 0.70–0.75× for MPI, OPS, and ETTh, indicating that the compressed data plus overhead slightly exceeds the original size. PostgreSQL pads and aligns BYTEA data for efficient updates, incurring significant overhead that nullifies a portion of the space savings. In our SQL implementation, this overhead offsets the compression gains, resulting in little to no net space reduction for some datasets when all metadata is included. By contrast, the original TSXor (with a binary file storage) and other specialized compression methods like Gorilla or FPC achieve better compression ratios on raw data. Nonetheless, given the advantages of our approach – namely, fine-grained per-value decompression and direct relational querying on compressed data – we consider the trade-off acceptable. The pure relational storage still yields moderate size reduction, and the fast decompression performance makes the approach practical for real-world analytic workloads.

	$A \to A_c$	$A_c \to B$
MPI	390	25
OPS	488	30
ETTh	349	21
Bank JPN	4	1
IHEPC	540	32

Table 4
Execution time (seconds) for TSXor compression and decompression on each dataset.

	Compression factor	
	Relation	Total Relation
MPI	1.1803x	0.7273x
OPS	1.2276x	0.7521x
ETTh	1.2308x	0.7519x
Bank JPN	1x	1x
IHEPC	1.0854x	0.6973x

Table 5
Compression factors achieved by TSXor on each dataset. "Relation" is the factor for the compressed table alone; "Total Relation" includes all stored data (including overhead). A factor of 1× indicates no reduction.

5.3. Discussion

Strengths: TSXor successfully compresses real-world time series while remaining lossless and fully operable within SQL. A key advantage of our integration is the fast decompression – query workloads benefit from minimal latency, since retrieving data ($A_c \to A$) is an order of magnitude faster than the compression process. In practice, once a chunk of data is stored in compressed form, it can be read on-demand with modest overhead, enabling interactive queries over compressed time-series. Additionally, because each TimescaleDB chunk is compressed independently, the implementation can leverage parallelism (multiple chunks processed concurrently) and allows selective decompression: queries can target only the specific chunks or even individual records needed for a given time range, rather than scanning an entire dataset. This chunk-level granularity contains query costs and fits naturally with the database's partitioned architecture. Overall, the TSXor integration provides transparent space savings and maintains query efficiency, making it a promising technique for managing large regular-interval series in relational systems.

Compression effectiveness: The compression ratio achieved by TSXor in our SQL implementation is relatively modest. Unlike specialized compression tools (which might achieve 2–3× size reduction on similar data), TSXor yielded at most 1.23× (≈19% reduction) in our tests. For datasets where values do not repeat or follow predictable patterns (such as the volatile Bank JPN series), TSXor may provide little to no compression benefit.

Storage overhead: The space overhead of our relational encoding is significant. Each compressed value carries a 16-bit header (stored in a `SMALLINT` column to record leading and trailing zero counts), and additional metadata—such as chunk timestamp references or tuple headers—and index entries introduce extra bytes. In our experiments this overhead accounted for roughly 30–50% of the total storage, which in some cases outweighed the compression gains and resulted in a net increase in size (as reflected by the "Total Relation" factors below 1× in Table 5). Reducing this overhead would require packing more values into each row or using array/binary types to store headers and data more compactly. Such changes, however, complicate selective access and were not implemented in our prototype.

Execution time: Performing TSXor compression inside PostgreSQL incurs considerable runtime overhead. Our implementation uses PL/pgSQL with calls to PL/Python for bit-level manipulation on each data point. This approach, while convenient, is much slower than a native C/C++ implementation due to interpretation and context-switching overhead. Consequently, compressing large chunks was

time-consuming (several minutes for hundreds of thousands of points), which would be impractical for real-time or high-frequency data ingestion. In its current form, TSXor compression is best suited for offline or batch operation (e.g., compressing chunks of historical data periodically). We also note that our experiments processed chunks sequentially; although chunk-wise compression can be parallelized across multiple CPU cores, our integration did not yet exploit parallel execution. Leveraging TimescaleDB's parallel background workers or other multi-process techniques could significantly improve throughput and is left for future work. On the other hand, decompression is relatively fast (tens of seconds or less for the same data volumes), so query performance on compressed data is quite acceptable.

Data type constraints: The present TSXor integration supports only a single numeric (double-precision) value column with a fixed time interval. The algorithm relies on the IEEE 754 binary representation of floating-point values and on uniform time steps (so that timestamps can be delta-encoded implicitly). It cannot directly compress other data types (e.g., integers, strings) or time series with irregular timestamps without modification. Adapting TSXor to irregularly spaced data would require storing each timestamp or using a more complex reference scheme, likely increasing overhead. Similarly, multi-variate time series (with multiple measured fields) would need to be compressed either column-by-column or with an extension of the algorithm to vector data; we have not explored such extensions. These constraints limit the applicability of our current implementation to regularly-sampled, uni-variate numerical series, which fortunately covers many IoT and sensor data scenarios, but it may not generalize to all time-series use cases.

Portability: In principle, the TSXor compression pipeline can be transferred to other relational database systems, but this requires specific technical features and considerable re-engineering. The hypertable abstraction in TimescaleDB, which organizes data into time-based chunks for partitioning and parallel execution, is comparable to partitioning mechanisms that also exist in other systems such as table partitioning in PostgreSQL [20]. More decisive for portability, however, are the low-level capabilities of the target system. These include support for suitable data types and binary storage, since our design relies on 16-bit `SMALLINT` headers combined with `BYTEA` or BLOB payloads. Furthermore, the system must provide a procedural extension or scripting interface that enables bit-level operations on 64-bit floating-point values as well as the execution of dynamic SQL to construct queries. TSXor is therefore conceptually transferable to any platform that supports numeric and binary data handling together with table partitioning, yet the actual feasibility and efficiency of an integration strongly depend on the degree of support for in-database programming and extensible compression frameworks.

In summary, while our TSXor integration demonstrates clear benefits in space reduction and read performance, it also faces important limitations. The compression gains are moderate and can be negated by storage overhead in some cases; the compression process is computationally heavy due to the use of interpreted languages; and the method is specialized to a particular data type and time interval pattern. Nonetheless, our results confirm that a tailored algorithm like TSXor can be successfully deployed inside a relational database, yielding tangible storage savings and acceptable performance for many time-series workloads. Addressing the above limitations (for example, through more efficient in-database implementations or support for a broader range of data) will further improve its practicality, but even in its current form TSXor provides a useful proof-of-concept for integrating domain-specific compression directly into a SQL environment.

6. Conclusion and Future Work

This work presented a relational integration of the TSXor compression algorithm for regularly sampled time-series data. Our implementation within TimescaleDB enables lossless compression with efficient, per-value decompression and direct SQL access. While compression ratios were moderate (up to 23%), query performance remained strong. The approach demonstrated that domain-specific compression can be successfully embedded into a SQL-based DBMS without external tooling.

Future work includes extending the prototype to support parallel, multi-chunk compression, enabling automated segmentation and concurrent processing of large datasets. We also plan to explore a

consolidated storage layout where all compressed payloads reside in a single column, and rows reference offsets—reducing overhead at the cost of increased access complexity. Beyond relational systems, a graph database implementation may better reflect TSXor's internal structure, while leaner engines like SQLite could be benchmarked for storage efficiency. A full departure from the DBMS layer—e.g., a succinct binary file format using compact tree representations—offers another promising path. Finally, combining TSXor with built-in compression features in TimescaleDB may yield further improvements in space and performance.

Declaration on Generative AI

Portions of this manuscript were prepared with the assistance of OpenAI's ChatGPT (GPT-5) to support language editing, phrasing, and stylistic refinement. All conceptual contributions, analyses, data interpretations, and conclusions were developed by the authors, who verified and approved the final version of the manuscript and take full responsibility for its content.

References

[1] P. Asghari, A. M. Rahmani, H. H. S. Javadi, Internet of things applications: A systematic review, Computer Networks 148 (2019) 241–261. URL: https://www.sciencedirect.com/science/article/pii/S1389128618305127. doi:https://doi.org/10.1016/j.comnet.2018.12.008.

[2] A. Bruno, F. M. Nardini, G. E. Pibiri, R. Trani, R. Venturini, Tsxor: A simple time series compression algorithm, in: T. Lecroq, H. Touzet (Eds.), String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings, volume 12944 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 217–223. URL: https://doi.org/10.1007/978-3-030-86692-1_18. doi:10.1007/978-3-030-86692-1_18.

[3] G. Chiarot, C. Silvestri, Time series compression survey, ACM Comput. Surv. 55 (2023). URL: https://doi.org/10.1145/3560814. doi:10.1145/3560814.

[4] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, K. Veeraraghavan, Gorilla: a fast, scalable, in-memory time series database, Proc. VLDB Endow. 8 (2015) 1816–1827. URL: https://doi.org/10.14778/2824032.2824078. doi:10.14778/2824032.2824078.

[5] M. Burtscher, P. Ratanaworabhan, Fpc: A high-speed compressor for double-precision floating-point data, IEEE Trans. Comput. 58 (2009) 18–31. URL: https://doi.org/10.1109/TC.2008.131. doi:10.1109/TC.2008.131.

[6] P. Liakos, K. Papakonstantinopoulou, Y. Kotidis, Chimp: efficient lossless floating point compression for time series databases, Proc. VLDB Endow. 15 (2022) 3058–3070. URL: https://doi.org/10.14778/3551793.3551852. doi:10.14778/3551793.3551852.

[7] R. Li, Z. Li, Y. Wu, C. Chen, Y. Zheng, Elf: Erasing-based lossless floating-point compression, Proc. VLDB Endow. 16 (2023) 1763–1776. URL: https://doi.org/10.14778/3587136.3587149. doi:10.14778/3587136.3587149.

[8] H. Chen, L. Liu, J. Meng, W. Lu, Afc: An adaptive lossless floating-point compression algorithm in time series database, Information Sciences 654 (2024) 119847. URL: https://www.sciencedirect.com/science/article/pii/S0020025523014329. doi:https://doi.org/10.1016/j.ins.2023.119847.

[9] A. Afroozeh, L. X. Kuffo, P. Boncz, Alp: Adaptive lossless floating-point compression, Proc. ACM Manag. Data 1 (2023). URL: https://doi.org/10.1145/3626717. doi:10.1145/3626717.

[10] PostgreSQL Global Development Group, Date/Time Types, 2025. URL: https://www.postgresql.org/docs/current/datatype-datetime.html, accessed 25.08.2025.

[11] PostgreSQL Global Development Group, Numeric Types , 2025. URL: https://www.postgresql.org/docs/current/datatype-numeric.html, accessed 25.08.2025.

[12] PostgreSQL Global Development Group, inary Data Types, 2025. URL: https://www.postgresql.org/docs/current/datatype-binary.html, accessed 25.08.2025.

[13] PostgreSQL Global Development Group, Bit String Functions and Operators, 2025. URL: https://www.postgresql.org/docs/current/functions-bitstring.html, accessed 25.08.2025.
[14] PostgreSQL Global Development Group, PL/pgSQL — SQL Procedural Language, 2025. URL: https://www.postgresql.org/docs/current/plpgsql-overview.html, accessed 25.08.2025.
[15] Max Planck Institute for Biogeochemistry, Weather Data from Jena, WS Beutenberg, 2025. URL: https://www.bgc-jena.mpg.de/wetter/weather_data.html, accessed 25.08.2025.
[16] J. Muehlenpfordt, Time series, 2020. URL: https://data.open-power-system-data.org/time_series/2020-10-06. doi:10.25832/TIME_SERIES/2020-10-06.
[17] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, W. Zhang, Informer: Beyond efficient transformer for long sequence time-series forecasting, in: The Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Conference, volume 35, AAAI Press, 2021, pp. 11106–11115.
[18] G. Hebrail, A. Berard, Individual Household Electric Power Consumption, UCI Machine Learning Repository, 2006. DOI: https://doi.org/10.24432/C58K54.
[19] Federal Reserve Bank of St. Louis, FRED: Japan Total Assets, 2025. URL: https://fred.stlouisfed.org/series/JPNASSETS, accessed 25.08.2025.
[20] PostgreSQL Global Development Group, Table Partitioning, 2025. URL: https://www.postgresql.org/docs/current/ddl-partitioning.html, accessed 25.08.2025.