Exploring SIMD Vectorization in Aggregation Pipelines for Encoded IoT Data

Rui Kang
Tsinghua University
kr20@mails.tsinghua.edu.cn

Shaoxu Song*
Tsinghua University
sxsong@tsinghua.edu.cn

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

***Abstract*—Time-series databases have been critical for collecting and analyzing data in industries where sensors send large amounts of IoT data by network devices. Both data received from networks and data collected in database storage are sufficiently encoded to reduce I/O occupation and latency. The IoT encoders successively combine the Delta, Repeat, and Packing operators, yielding a higher compression ratio than simply adopting each. However, efficient compression makes query execution even harder, requiring serial decoding before processing queries. Among them, selective aggregations, such as down-sampling, are the core of time series analytical queries. This paper identifies operators to process and accelerate IoT aggregation queries based on encoded data arrays, extensible to integrate thread-level and instruction-level designs. In addition, encoded data could aggregate directly in parallel without decoding, and encoding statistics can help to reduce unnecessary computation. Identified operators construct a pipeline query engine to integrate into an existing open source database, the Apache IoTDB. Remarkably, our systemic evaluations show vast improvements in the efficiency of selective aggregation over existing works.**

***Index Terms*—IoT Data, Vectorization, Query Optimization**

I. Introduction

IoT data are time-ordered sequences composed of timestamps and sensor readings. IoT databases encode each sequence for fast delivery through networks [1], [2], [3] and space-efficient storage on the disk [1], [4]. The encoding formats leverage combined encoders that can flush and recover incrementally when new IoT data points are generated. The Delta encoder calculates the often slight differences in consecutive values [2], [3]. It is used together with other encoders, such as Run-lengths [2] and Packing [2], [5], [6], to reduce unnecessary messages, as shown in Table I. While databases try to compress data effectively, combined encoders challenge the query performance because query engines need to decode data serially in the reverse order of applied encoders. Existing work like FastLanes [7], [8], [9] proposes new encoding formats to accelerate decoding speeds, but they do not apply to IoT data, as shown below.

Example 1 (IoT encoding formats)**.** *In IoT scenarios, an industrial device like a Raspberry Pi connects multiple sensors, such as temperature and velocity, to detect real-time system status. The embedded clock triggers the generation of time-series records, as shown in Figure 1(a). Devices and servers encode/compress the records to save I/O and storage.*

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

TABLE I: Combined encoders for IoT data, categorized into Delta, Repeat, and Packing by encoder semantics.

Method	Encoding operators		
	Delta	Repeat	Packing
RLBE [2]	$\pm$	Run-length	Fibonacci
TS_2DIFF [10]	$\pm^2$	None	Bitpack
Sprintz [11]	$\pm$	None	ZigZag,Bitpack
Chimp [5]	XOR	None	Pattern
Gorilla [6]	$\pm$, XOR	Flag	Pattern
Elf [12]	XOR	None	Pattern

The IoT databases deployed in cloud servers buffer received data packets of millions of sensors from networks [13]. The databases encode data incrementally to save the receiving buffers. When the buffers are filled, each sensor may only accumulate a short time series, which databases will flush to disk files. Thus, IoT encoders should be ***space-efficient*** *and* ***flexible*** *to apply to real-time IoT data.*

- ***Space efficiency.*** *IoT Encoders, listed in Table I, combine different encoding semantics and use Delta-Repeat-Packing successively to store with fewer bits.*
- ***Flexibility.*** *Encoders should work incrementally and apply to various input data sizes to save buffer spaces and flushing time.*

Space efficiency: *Figure 1(b) introduces the widely applied encoding format TS2DIFF for IoT data [4], [14], [15]. Since velocity changes closely over time, a Delta encoder finds differences between adjacent reads, e.g.,* $velocity[2] - velocity[1] = 4$ *(kilo-meters per minute). For the acceleration process, we use* $base$ *to reduce the velocity absolute difference, e.g.,* $D_2 = (velocity[2] - velocity[1]) - base = 2$*. Then, Bit-packing removes useless leading zeros of* D_i *to combine with the two Deltas, e.g., 10 bits for each* D_i*.* ***Flexibility:*** *Figure 1(b) shows that the TS2DIFF encoder writes new bits incrementally when receiving new data points. It keeps the latest record, like* $velocity[1]$*. When receiving* $velocity[2]$*, it writes new bits for* D_2 *and updates the latest as* $velocity[2]$*. Once the receiving buffer is filled, the storage engine flushes encoded blocks in Big-Endian.*

Instead, the FastLanes FLMM1024 *layout [7] applies to a fixed-sized data block each time, as shown in Figure 1(c). Compared to TS2DIFF, it provides a virtual 1024-bit register abstraction to recover more velocity records with*

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00040

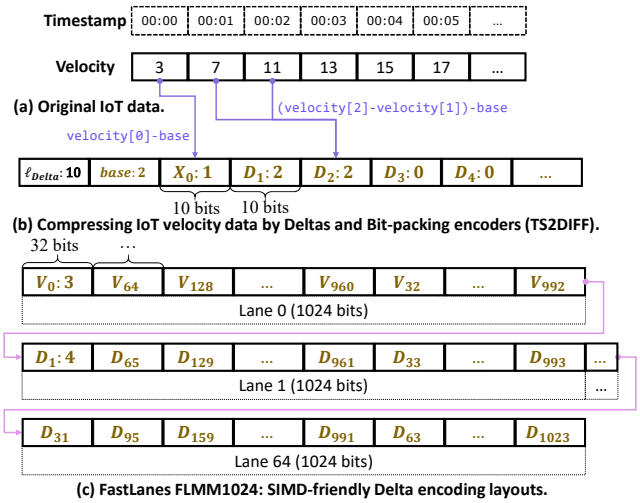


Fig. 1: IoT encoder schemes combine multiple encoders for storage efficiency and incremental encoding.

a single addition instruction (SIMD-friendly). It groups the original data and deltas into lanes separated by 1024 bits. The original data are kept in Lane 0, containing 32 elements of 32-bit width, $velocity[0, 64, 128, \ldots, 992]$*, and delta values are stored in other Lanes like* $D[1, 65, 129, \ldots]$*. It recovers* $velocity[1, 65, 129, \ldots, 993]$ *by a simple addition after loading Lane 0 and Lane 1 into SIMD registers. However, it takes every 1024 velocity record, which asks IoT servers to buffer 1024 data points for each time series. Each data point can consume 32 bits in the buffer.*

Thus, our proposal applies IoT encoders and accelerates IoT query performance. Compared to IoT encoders, FastLanes FLMM1024 [7] is problematic in IoT scenarios due to its lower compression ratio, buffer pressure, and inefficiency in handling short time series. Our approach also combines aggregation operators with decoders and explores the design beyond improving the decoders. Specifically, FastLanes stores more original data to achieve better SIMD performance. The buffer will be filled quickly when accumulating data points for each time series. Meanwhile, the number of records may not be sufficient to form a block, leading to empty values.

We introduce pipeline designs to integrate **core/instruction-level parallelism** and **operator fusion**. For **parallelism**, analytical query plans [16], [17] and GPU-based Delta decoding [18], [19] show effective practices for using multiple threads, indicating the necessity to design parallel pipelines. However, while GPU pipelines [18], [19] improve the decoder by an input sequence of 32-bit-width deltas, we split encoded data evenly to CPU cores under variable packing widths for deltas. Also, unlike GPU threads controlled by flexible CUDA programs, we accelerate each core performance according to limited instructions for IoT databases on industrial servers.

Operator fusion: Besides decoding IoT data, our pipelines merge query operators with decoders. The pipeline aims to avoid applying Delta and Repeat decoders and compute query

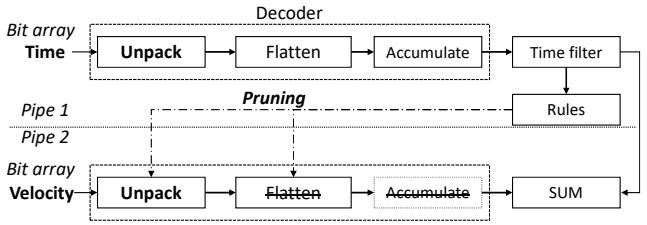


Fig. 2: Query pipelines on encoded IoT data for aggregations.

results directly on unpacked sequences. Thus, our pipelines will merge more than one decoder, different from the analytic query engines on fusing a single decoder, such as [20] to join records on Run-length encoded data and [21] on aggregation.

Example 2 (Pipeline exploration)**.** *Figure 2 shows our pipeline design, including decoders and vectorized query operators. Consider the following aggregation query.*

```
SELECT AVG(Velocity) FROM Velocity
WHERE  Time >= 00:03 AND Time <= 00:05;
```

The above IoT query filters the velocity series by a time range. Thus, we can use two decoding pipelines for timestamps and velocity, as shown in Figure 2, where decoders named unpack, flatten, and accumulate decode Packing, Repeat, and Delta encoding semantics, respectively. The pipeline computes the sum of valid velocities based on valid timestamps.

For core-level parallelism, we could execute the above pipeline in each core to save I/O. However, when data are encoded by Repeat or Fibonacci/Pattern Packing, splitting data to each core could result in a different number of decoded timestamps and velocities, causing data to broadcast I/O over cores. We refer to CPU SIMD instructions for instruction-level parallelism to design decoding layouts to improve efficiency and share SIMD registers among pipeline operators to reduce data movement. However, SIMD instructions are ineffective in unpacking and dealing with dependent Delta accumulation. For operator fusion, we notice that the sum of velocity between 00:03 and 00:05 can be calculated without decoding Repeat and Delta (or two Deltas). For the TS2DIFF format in Figure 1(b), the sum is equal to $3X_0+3D_1+3D_2+2D_3+D_4+12base$*. Moreover, it is possible to prune unnecessary data loading and decoding by filters. For example, we can prune all data decoding after observing* $timestamp$ *=00:05 and* $velocity$ *=17 because the encoded IoT data are ordered by timestamps.*

Contributions and Overviews

We organize this paper with the following contributions.

(1) Section III explores the design space of query pipelines, including data splitting (core-level), vectorized decoders and query operators (instruction-level), and discussions for just-in-time pipeline generation.

(2) Section IV shows our operator fusion techniques for merging multiple decoders with aggregations.

(3) Section V concludes pruning rules based on (time/value) filtering results and encoder statistics.

(4) Section VI integrates our proposed techniques to execute complex queries in the Apache IoTDB [22]. Remarkably, in Section VII, we evaluate our proposed querying pipelines and compare them to existing approaches, which shows the advances and efficiency in considering combined encoders by pipelines instead of optimizing each.

We show the preliminaries, related works, and conclusions of our work in Section II, VIII, and IX, respectively.

II. Preliminaries

This section introduces the semantics of the IoT time series, concluded from encoding formats and queries. These basic operations construct the pipeline for decoding and aggregation queries. We also introduce useful CPU SIMD instructions for vectorizing decoders and query operators in each pipeline.

A. Aggregation query expressions

Expressions formalize a query pipeline from encoded IoT data (bit arrays) to an aggregation result. Thus, an expression should support operations on sequences of bits and byte-aligned values like time series.

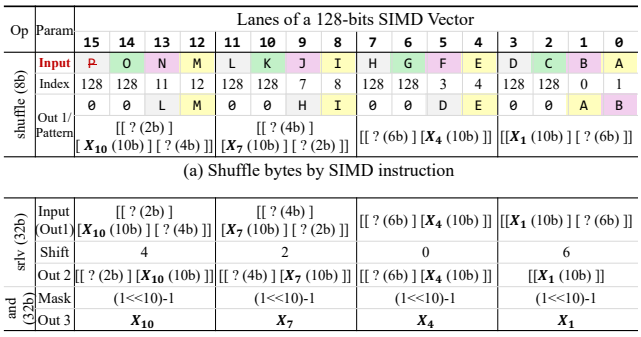
Definition 1. *An IoT time series of schema* $S(T, A, \dots)$ *consists of ordered sequences for its columns with* $t[0] < t[1] < \dots$ *and* $\{[\![t[0], t[1], \dots t[n]]\!], [\![a[0], a[1], \dots a[n]]\!], \dots\}$.

Unlike pipelining decoders, query operators work on each time-series tuple, consisting of a timestamp and values at the same position. Timestamps and values may have different packing widths. The schema also records constant packing widths for reading bits from input sequences like $S(Time, A \mid \omega_T, \omega_A)$, where ω is the minimum bit length of reading elements. The IoT expression e is a sequence or,

$$
\begin{aligned}
e ::= & \Gamma_{\omega \to \omega'}(e) && \text{(Bit extension)}\\
& \mid e_1 \ op \ e_2 \mid op(e) && \text{(Arithmetic operations)}\\
& \mid e_1 \bowtie e_2 \mid e_1 \circ e_2 && \text{(Natural join, concatenation)}\\
& \mid e[pos_1 : pos_2] && \text{(Position-based fraction)}\\
& \mid \sigma_\theta(e) \mid \pi_X(e) \mid \rho_{S'}(e) && \text{(Filter, projection, rename)}\\
& \mid f(e, mask) && \text{(Valid value aggregation)}\\
& \mid G_{sw(T_{min}, \Delta T):f}(e) && \text{(Sliding window Aggregation)}
\end{aligned}
$$

The expression includes operations for a physical query plan. Unlike relational query plans, it involves sequential operations like position-based fractions and bit/vector-level operations, such as bit extension and masked value aggregation.

Definition 2 (Operation semantics)**.** ***Bit extension:*** *Given an input sequence* s *with value width* ω*, the operation extends every* ω *bits into* ω' *bits.* ***Arithmetic operations:*** *The expression supports unary or binary element-wise operations, such as addition* $e_1 + e_2$*.* ***Natural join:*** *Given timestamp sequences* T_1 *and* T_2*, natural join produces a mask for valid tuples of each time series, such as* $mask_1 = [\![-1 \text{ if } t_1[i] = t_2[j] \text{ else } 0 \mid \forall i \exists j]\!]$*.* ***Concatenation:*** *Given time series of the same schema, this operation concatenates the sequences of the same attribute. We sort the data before applying any operations depending on time orders.* ***Fraction and filter:*** *Given a time series, the position-based fraction finds slices of each column, and the filter uses attribute/inter-column predicates to generate a mask sequence.* ***Valid value aggregation*** *applies an associative/algebraic function* f *to the valid values of an input sequence, such as* $sum(velocity, 1)$*, which returns the sum of all velocities.* ***Sliding window aggregation:*** *Given a time series, sliding window description* $sw(T_{min}, \Delta T)$*, and aggregation function* f*, this operation applies multiple time-range filters to the series for aggregation. Each time range is defined by window instance* $w(T_{min} + k\Delta T, \Delta T)$*, standing for the predicate conjunction* $T \geq T_{min} + k\Delta T \wedge T < T_{min} + (k+1)\Delta T$ *with* $k \geq 0$ *and* $T_{min} + k\Delta T < T_{max}$*.*



(a) Shuffle bytes by SIMD instruction

(b) Shift bits by SIMD instruction (SRLV) for unpacking data

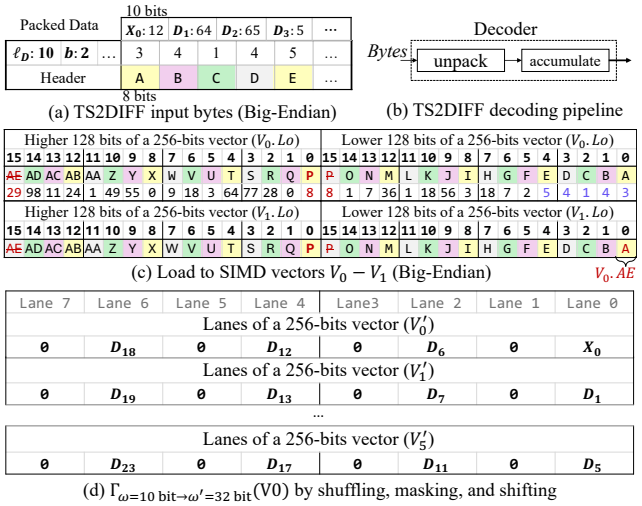
Fig. 3: Useful SIMD instructions for unpacking.

Example 3 (Expressions)**.** *Consider the query in Example 2. When the input bit arrays of timestamps and velocities are encoded by TS2DIFF, we first unpack values by extending bits* $\Gamma_{10 \to 32}(velocity)$*. When velocities and timestamps are loaded in SIMD registers, the time-range filter generates a mask* $mask = \sigma_\theta(dec_{Delta}(\Gamma_{10 \to 32}(timestamp) + base))$*, applicable to velocities to find the sum/average,* $sum(dec_{Delta}(\Gamma_{10 \to 32}(velocity) + base), mask)$*. We can pipeline most operations, but some decoders require a pipeline design to connect to existing operations.*

B. Overview of CPU SIMD Instructions

We show practical SIMD instructions to implement decoders and operators to make our design not specific to a single quantity or one instruction set, like 32 bits/AVX2. Our decoding and query processing use instructions to shuffle bytes, shift bits, and perform lane-wise operations, which can extend to other quantities and instruction sets. As industrial IoT databases are typically deployed on x86 servers, we consider SSE, AVX/AVX2, and AVX-512.

First, the instructions for shuffling bytes across lanes can unpack values in a SIMD register. We assume that data are encoded by 10-bit packing. Figure 3(a) shows the byte-level movement in a 128-bit register by a shuffling operation, such as _mm_shuffle_epi8, based on two input vectors, *Input* and *Index*. In *Input* vector, each lane of $0 - 15$ contains one byte from encoded data in Big-Endian format. The *Index* vector

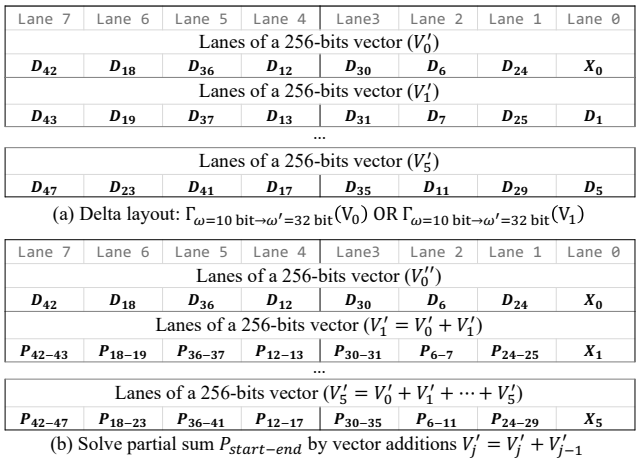


(a) TS2DIFF input bytes (Big-Endian) (b) TS2DIFF decoding pipeline

(c) Load to SIMD vectors $V_0 - V_1$ (Big-Endian)

(d) $\Gamma_{\omega=10\text{ bit}\to\omega'=32\text{ bit}}(V0)$ by shuffling, masking, and shifting

Fig. 4: Unpack bit arrays into SIMD vectors for fast decoding.



(a) Delta layout: $\Gamma_{\omega=10\text{ bit}\to\omega'=32\text{ bit}}(V_0)$ OR $\Gamma_{\omega=10\text{ bit}\to\omega'=32\text{ bit}}(V_1)$

(b) Solve partial sum $P_{start-end}$ by vector additions $V_j' = V_j' + V_{j-1}'$

Fig. 5: Construct Delta decoding layout for the partial sum.

shuffles bytes from Input lanes to the output vector, where index value 128 leads to an output value of 0. The last row shows the pattern of the output vector *Out 1*, e.g., the right-most 32-bit lane contains 10 bits of encoded value X_1 and 6 uninterested bits. Since there are variable numbers of uninterested bits in each 32-bit lane of *Out 1*, Figure 3(b) applies an operation (srlv) to shift bits according to different uninterested bits, as shown in the *Shift* vector. The bit-shifting output is *Out 2*. Finally, in the last two rows, a mask vector selects the 10 bits containing X_1, X_4, X_7, X_{10} from *Out 2*. The above decoding algorithm is easy to extend to other encoding-widths and instruction sets.

III. ETSQP: Encoded Time-series Query Pipelines

Based on basic time-series query operations and existing advances in decoding data [7], [23], [24], [9], the decoding and aggregating pipeline can adopt core- and instruction-level accelerations. Section III-A discusses the techniques for improving decoders. Sections III-B-III-C further introduce the Just-in-time compilation and core-level parallelism. We analyze our pipeline designs in Section III-D.

A. Accelerated Decoding and Query Operators

Motivated by Section II-B, we can vectorize decoding pipelines and use the decoded vectors in query executions. We classify our pipeline designs according to the complexity of decoder SIMD implementations.

1) Vectorized unpacking for constant packing widths: Vectorizing the pipeline of constant-width unpacking can speed up IoT decoders like TS2DIFF with an expression pattern $dec_{Delta}(\Gamma_{10\to32}(velocity) + base)$. Motivated by FastLanes-Delta layout [7] in Figure 1(c), besides improving data unpacking, we focus on generating unpacked vectors that are SIMD-friendly for Delta decoding. Algorithm 1 describes data unpacking and uses the unpacked layout for further Delta decoding, illustrated by Figure 4-5, where Figure 4(a) shows the input byte sequence s and (c) loads 25 Deltas.

Lines 1-5 decide the number of used vectors and load bytes from memory to SIMD registers. We first assume $n_{ld} = 2$ and $n_\nu = 6$, which are decided later in Proposition 1. Line 3 loads bit arrays into $n_{ld} = 2$ vectors V_{0-1}(Figure 4(c)), where the last byte of every 128 bits stays unused and loaded again in the next 128 bits because $\lfloor 128/10 \rfloor = 12$. For every 128 bits, Figure 3 indicates the decoded 12 elements can be shuffled to 3 vectors (with four in each) or 6 vectors (with two in each), as Figure 4(d) shows. FastLanes-Delta motivates us to put as many dependent deltas $X_0, D_1, \ldots$ into the same Lane of as many different vectors n_ν. Line 8 shuffles n_{ld} loaded vectors V0-V1 based on Figure 3(a) and fills n_ν vectors, as shown in Figure 4(d)-5(a). To reduce time costs, instead of shifting bits and masking each lane of shuffled vectors $V_i^{'}$, we use Line 9 to shift and mask after shuffling all loaded vectors (Figure 5(a)). Similar to FastLanes-Delta, Lines 10-12 solve partial sum $P_{a-b} = \sum_{k=a}^{b} D_k$, while $X_b = P_{0-b} = \sum_{k=0}^{b} D_k$. The Line 13 finds the prefix vector $v'_{prefsum}$ for P_{0-a}, defined by $[0, P_{0-23}, P_{0-5}, P_{0-29}, P_{0-11}, P_{0-35}, P_{0-17}, P_{0-41}]$ for Lane 0-7, which is a common vector for all partial sums $V'_{[1,2,\ldots n_\nu]}$. Based on $v'[6]$, consider $L0$ as $Lane0$ and $L0246$ as $L0+L2+L4+L6$, we have, $v'_{prefsum} = [0, L0246, L0, L0246+L1, L0+L2, L0246+L1+L3, L0+L2+L4, L0246+L1+L3+L5]$. We use three pairs ($[\log(\frac{\omega_{SIMD}}{\omega'})]$) of permutevar8x32 and addition instructions in AVX2 or AVX-512 to solve $v'_{prefsum}$ based on $V_6^{''}$. Finally, Line 15 adds prefix sum to partial sum vector as the decoded data.

We discuss our decisions in detail based on the algorithmic structures. Firstly, for Lines 1-2, we derive the number of vectors n_ν used in Figure 4(d). When unpacking width $\omega' = 32$ bits, each unpacked vector holds ω_{SIMD}/ω' values, where ω_{SIMD} is 256 bits in AVX2 devices and 512 bits under AVX-512. Since we cannot unpack more values than that we load, we have $n_\nu \cdot \frac{\omega_{SIMD}}{\omega'} \leq n_{ld} \cdot \frac{\omega_{SIMD}}{\omega}$. For x86 servers, we have $n_\nu \leq 16$ under AVX2 machines and $n_\nu \leq 32$ under AVX-512 devices. We estimate the CPU clocks of Algorithm 1. That is, data loading costs $t_{load} \cdot n_{ld}$ (Line 3), Endian conversion costs $t_{shuffle}$ for each loaded vector

Algorithm 1 Dynamic layout unpacking and Delta recovery

Input: Byte sequence s, constant packing width ω, extended value width ω', base for Delta values $base$, SIMD vector width ω_{SIMD}.
Output: Decoded vectors $\mathbf{dec}(S)$.

1: Decide the number of loaded vectors n_{ld}
2: Decide the number of unpacked vectors n_ν
3: Loaded vectors $v[1, \dots n_{ld}]$
4: Shuffle loaded vectors $v[i]$ in Little-Endian
5: Prepare namespace for unpacked vectors $v'[1, \dots n_\nu]$
6: **for** each loaded data to unpack, $i = 1, 2, \dots, n_{ld}$ **do**
7: **for** each unpacked vector $j = 1, 2, \dots, n_\nu$ **do**
8: $v'[j] = v'[j] \mid shuffle(v[i], j)$
9: Shift and mask $v'[1, 2, \dots, n_\nu]$ by Figure 3
10: Start Delta decoding on unpacked vectors $v'[i]$:
11: **for** unpacked vector index $j = 2, 3, \dots, n_\nu$ **do**
12: $v'[j] = v'[j-1] + v'[j]$ (Partial sum)
13: Permute bytes to solve prefix sum $v'_{prefsum}$
14: **for** unpacked vector index $j = 1, 2, \dots, n_\nu$ **do**
15: $v'[j] = v'[j-1] + v'_{prefsum}$
16: **return** $v'[1, 2, \dots, n_\nu]$

(Line 4), unpacking consumes $t_{unpack} n_\nu \cdot n_{ld} + (t_{and} + t_{shift}) n_\nu$ with $t_{unpack} : t_{shuffle} + t_{or}$(Figure 3, Line 8-9). For Delta decoding, Line 12 costs $(n_\nu - 1)t_{add}$, Line 13 costs t_{prefix}, and Line 15 costs $n_\nu t_{add}$. To optimize the CPU performance, the average time costs $T_{AVG} = \frac{(t_{load}+t_{shuffle})n_{ld}+t_{unpack}n_\nu \cdot n_{ld}+n_\nu(2t_{add}+t_{and})+t_{prefix}-t_{add}}{n_\nu \omega_{SIMD}/\omega'}$.

Proposition 1. *The average time T_{AVG} represents the decoding time for each data point and achieves its optimal value when the number of vectors n_ν is $[\sqrt{\frac{\omega'}{\omega}\frac{t_{prefix}-t_{add}}{t_{unpack}}}]$, where ω and ω' are bit widths of packed and unpacked data, respectively, and t_{unpack}, t_{prefix}, and t_{add} are time costs of Lines 8, 13, and vector addition. The minimum average time is $C + \frac{\sqrt{\omega'\omega}}{\omega_{SIMD}}\sqrt{2t_{unpack} \cdot (t_{prefix} - t_{add})}$, where $C = \frac{\omega}{\omega_{SIMD}}(t_{load} + t_{shuffle}) + \frac{(2t_{add}+t_{and}+t_{shift})\omega'}{\omega_{SIMD}}$.*

Proof. We compute the average time by estimating the time costs of each round, divided by the number of decoded values $n_\nu \omega_{SIMD}/\omega'$. Since $n_\nu \cdot \frac{\omega_{SIMD}}{\omega'} \le n_{ld} \cdot \frac{\omega_{SIMD}}{\omega}$, we have

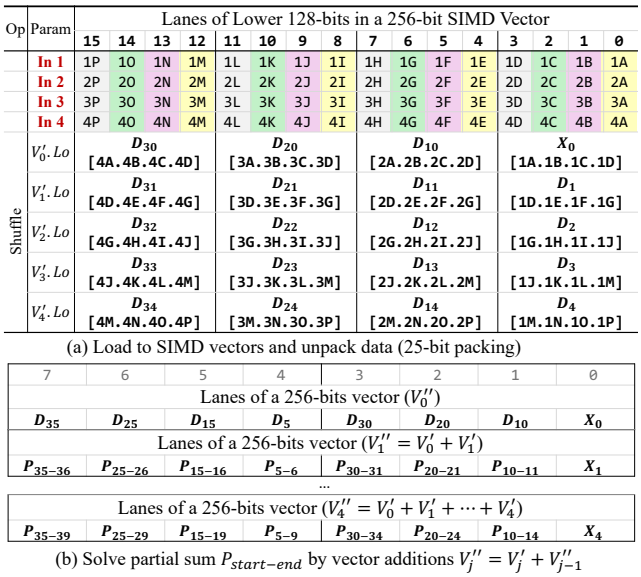
$$T_{AVG} \ge \frac{\omega}{\omega_{SIMD}}(t_{load} + t_{shuffle} + t_{unpack} n_\nu) + \frac{\omega'}{\omega_{SIMD}}(2t_{add} + t_{and} + \frac{t_{prefix} - t_{add}}{n_\nu}).$$

Thus, for an optimal minimum T_{AVG}, since n_ν is an integer, we have the decision in Proposition 1. □

Figure 4(a) indicates the vector number n_ν in (d) for optimal time is $\sqrt{\frac{32}{10} \cdot \frac{11}{2}} \approx 4$ according to [25]. We choose 6 vectors from $\{[\frac{\omega_{SIMD}}{\omega\alpha}] \mid \alpha = 2, 4, 8 \dots, \frac{\omega_{SIMD}}{\omega'}\}$ in Figure 4(d) to use as many loaded data, where α is the number of lanes in V'_i filled by each loaded vector.

Secondly, Line 3 loads vectors from memory. Since the shuffle instruction for unpacking in SSE and AVX2 moves bytes within 128-bit lanes, we read one more byte at the header to shuffle bytes within every 128-bit lane (1 byte+(10 bit)*12) in Figure 4(c), and there are insufficient bits for the 25th item, D_{24}.Our algorithm ensures that no register (lane) is left empty and D_{24} will be reloaded in the next unpacking vector V_1. Figure 5 shows reloaded vector fills empty lanes for Delta decoding together with aligned values of V_0 in lanes.

Thirdly, since IoT databases use Big-Endian to encode real-time IoT data, Line 4 shuffles bytes towards Little-Endian. The shuffle instruction in Figure 3(a) can alter a byte to the desired destination. We use the shuffle index $[0, 1, 2, 3]$ for Lane 0, which can similarly apply to other Lanes.



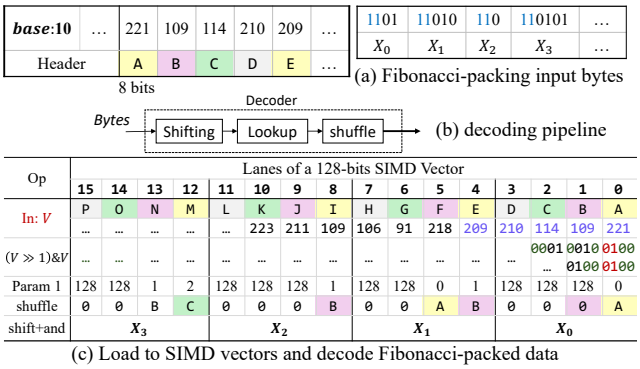
(a) Load to SIMD vectors and unpack data (25-bit packing)

(b) Solve partial sum $P_{start-end}$ by vector additions $V''_j = V'_j + V''_{j-1}$

Fig. 6: Unpacked vector layout for 25-bit inputs.

Example 4. *Figure 6 shows our unpacking layout when the packing width is 25 bits ($\Gamma_{\omega:25b \to \omega':32b}(s)$). Still, Lines 1-2 decide parallelism by Proposition 1 with $n_\nu : \sqrt{\frac{32}{25} \cdot \frac{11}{2}} \approx 3$. We choose $n_\nu : 5$ vectors from $\{[\frac{\omega_{SIMD}}{\omega\alpha}] \mid \alpha = 2, 4, 8 \dots, \frac{\omega_{SIMD}}{\omega'}\} = \{5, 2, 1\}$ to use as many loaded data ($\alpha = 2$) and $n_{ld} : \frac{n_\nu \omega_{SIMD}}{\omega' \alpha} = 4$. For the lower 128 bits, the loaded bytes of four inputs are shown in each lane of the shuffle result for better performance of decoding TS2DIFF. As shown in Figure 6, each loaded vector reserves $\alpha : 2$ lanes of registers in target width $\omega' : 32b$ for each unpacked vector, such as X_0, D_5 from V_0 in V'_0.*

2) Variable widths unpacking and repeat flatten: Unlike constant-width unpacking, variable-width unpacking and Repeat flattening are irregular decoders that generate variable-length decoded sequences. The decoding pipelines under variable-width unpacking require the same steps as Algorithm 1 shows, while they have different $unpack$ implementations at Line 8. Figure 7 unpacks a Fibonacci-encoded sequence having variable packing widths. We notice that each pair of 1s indicates an Fibonacci encoding termination. The results of $(V \gg 1)\&V$ in Figure 7(c) separate encoded elements.



(c) Load to SIMD vectors and decode Fibonacci-packed data

Fig. 7: Unpacking values from variable bit widths.

To unpack values, the shuffle indexes in vector *Param 1* is decided by a dictionary based on the occurrences of 1s in each byte. Then, we follow Algorithm 1 to unpack values. For RLE decoders, we refer to [26] that recovers by inserting 0's in Lanes according to run-lengths, taking Run-length encoding as a Delta encoder by XOR operations.

B. Just-in-Time Decoder Generator

Just-in-time compilation can avoid a runtime decision of SIMD parameters. Line 8 of Algorithm 1 suggests we use JIT to decide shuffling indexes, shown in Figure 3(a), according to the packing width of an input data page and the filling target lanes of the unpacked vectors, in Figure 4(d). When we have the decisions of vector numbers n_ν or loaded vectors n_{ld} (Line 1-2), we have only one layout to use as much loaded data (to save I/O). Thus, it is possible to look up shuffling index vectors for Line 8, bit-shifting/mask vectors for Line 9, and permute indexing vectors to solve prefix sum at Line 13.

C. Decoder Pipelines

In IoT databases like IoTDB, each time series is stored as multiple pages in the file system, each encoded separately with a private header and packing width. Besides vectorizing decoding pipelines, we distribute data to cores and run pipelines in each core to improve core-level parallelism. Figure 8 applies decoding pipelines to IoT data pages. Each page is encoded separately with a header containing the first value.

When we have more encoded pages than CPU cores, each core will execute a pipeline (Algorithm 1) on several pages; otherwise, we can split pages based on available cores p_c. For constant packing widths, each page slice has the same bits for the same number of unpacked elements, such as unpacked Deltas in TS2DIFF (Figure 5(b)). Page separation can utilize more cores, but computing prefix sums (Line 13) depends on decoded data from other CPU cores, as shown in Figure 8. To avoid separating pages into too many slices, each page will have at most $\lceil \#\text{Pages}/p_c \rceil$ slices given #Pages as the remaining pages. Figure 8 shows a split node to distribute pages or slices to pipelines. For variable packing width like Fibonacci packing, we separate each page evenly by bits and extend each page slice by the maximum packing width, like

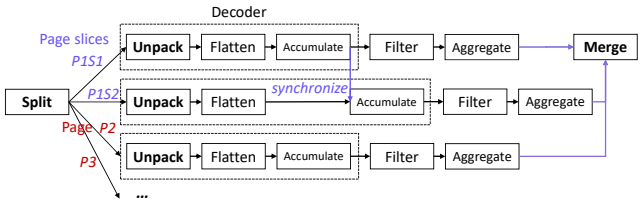


Fig. 8: Using multiple threads in a pipeline to decode pages.

32 bits. The decoder can unpack one more value from the end and drop the bits of an incomplete value in the front.

D. Performance Analysis

Based on extended operators in Section III-A and III-C, we analyze the costs of executing our pipelines and compare the costs to serial aggregating.

Theorem 2. *Let $T_{parallel}$ and T_{serial} denote the estimated execution time of ETSQP (Algorithm 1) and value-wised decoding that applies unpacking, flattening, and accumulation serially to decode a value at a time. Consider instruction- and core-level parallelism. The acceleration ratio is estimated as,*

$$\frac{T_{serial}}{T_{parallel}} \sim \mathcal{O}\left(\frac{1}{\frac{\omega}{2p_c \cdot \omega_{SIMD}} + \frac{2\omega' + \sqrt{\omega\omega'}\sqrt{4\lceil \log(\omega_{SIMD}/\omega') \rceil - 1}}{p_c(t_{visMem}/t_{op}+2)}}\right),$$

where ω and ω' represent the packing width of input bit arrays and the quantity of computation.

Proof. When decoding pipelines work parallelly in cores and share registers to execute query operations based on decoded vectors, serial decoding needs to unpack each value with a memory visit overhead, accumulate Deltas, and store decoded data as a basic data block, as traditional databases do. Unlike instructions on registers, memory visiting has various time costs according to cache hits. We use t_{visMem} for memory access and t_{op} for the latency of simple operations like additions. Thus, serial decoding costs $2t_{visMem} + t_{shift} + t_{mask} + t_{regSave}$, which represents loading bytes to a register and storing data, shifting a register to remove trailing garbage bits, masking to remove heading garbage bits, and saving current value temporarily. Based on Proposition 1, decoding $n_\nu \frac{\omega_{SIMD}}{\omega'}$ values costs $T_{AVG} \cdot n_\nu \frac{\omega_{SIMD}}{\omega'}$, reaching its optimum when n_ν is $\lceil \sqrt{\frac{\omega'}{\omega} \frac{t_{prefix} - t_{add}}{t_{unpack}}} \rceil$. Thus, we have the estimation of $\frac{T_{serial}}{T_{parallel}}$ as concluded in Theorem 2. □

Theorem 2 indicates the acceleration is variable under different memory access patterns t_{visMem}/t_{op}. Specifically, when decoding 10-bit packed TS2DIFF data stored in memory with 16 threads/AVX2 instructions, our decoding pipeline can achieve approximately 15.3 times improvements.

IV. Operator Fusion: Aggregation without Decoding

Motivated by Example 2, after unpacking bit arrays, it is possible to fuse aggregations with Repeat and Delta decoders.When Delta finds differences between adjacent series values, the fused aggregations can be either associative, like

SUM, or algebraic, depending on associative functions like AVG.

Based on the TS2DIFF format, we can compute the SUM aggregation by scanning Deltas. When dealing with Delta-Repeat formatted data, it is possible to scan the sequence and compute the result using a polynomial. This section extends the idea of sequential Delta-Repeat pair scan to other functions, such as $\sum_i A_i B_i$ and $\sum_i A_i^2$, to compute more aggregations like variances and correlations.

For $\sum A_i B_i$, the input contains two sequences of Delta-Repeat (run-length) pairs $\langle \Delta A, RLE_1 \rangle$, $\langle \Delta B, RLE_2 \rangle$, and the number of valid tuples $valid$ to aggregate, s.t. $valid \leq \min(RLE_1, RLE_2)$; otherwise, we aggregate $\min(RLE_1, RLE_2)$ valid tuples. We have, $\sum_{i=n+1}^{n+valid} a_i b_i = \sum_{i=1}^{valid} (a_n + i\Delta A)(b_n + i\Delta B)$, which is equal to a polynomial composed of four items, $valid \cdot A_n B_n + A_n \cdot \sum(i\Delta B) + B_n \cdot \sum(i\Delta A) + \sum(i^2 \Delta A * \Delta B)$. Here, a_n are decoded values, updated by $a_{n-RLE_{n-1}} + RLE_{n-1}\Delta A_{n-1}$ to avoid decoding each value. The above polynomial is general and can be transferred to a function that applies each Delta-Repeat pair and computes incrementally.

Proposition 3. *Let f and g denote an associative aggregation function and a unary function on an attribute A, such that $g(A_{h+1}) = g(A_h) + g(\Delta A_{h+1})$ formalizes a Delta decoding process. Then, the target $(f \cdot g)$ could be aggregated incrementally on unpacked IoT formatted files.*

V. ETSQP-prune: Pruned Pipeline by Statistics

Motivated by Example 2, we can avoid loading and decoding IoT bit arrays based on filters. This section constructs the rules concerning time and value filters.

A. Pruning Rules on Time Filters

We construct the heuristic rules to skip decoding by a time range filter. Each data page contains the encoded timestamps and values of a part of the time series. As IoT records are generated evenly over time, timestamps can be encoded efficiently using the Delta-Repeat-Packing format. Assume that the filter is a conjunction. Each page header stores statistics in addition to the first element, including (1) the Packing parameters of Delta and Repeats, (2) the size of each column, and (3) the D-R tuple number encoded in the current file. Formally, we follow the heuristic rule to prune.

Proposition 4. *Consider the time series encoded by combining Packing, Repeat, and a range filter $T > t_1 \wedge T < t_2, t_1 < t_2$. Given $t[k]$ as a decoded value with position k and n as the size of the input sequence, we prune the remaining sequence when (1) $t[k] < t_1$ and $D_M < \frac{c_1 - a[k]}{R_M(n-k-1)}$; or (2) $t[k] > t_2$ and $D_m > \frac{c_2 - a[k]}{R_M(n-k-1)}$, where variables D_m and D_M represent the lower and upper bound of the time intervals, estimated by the packing width, $D_m \geq minBase$ and $D_M \leq minBase + 2^\omega - 1$. Also, the variable R_M is the upper bounds of Run-length values, estimated by packing widths ω_{RLE} and the value of $minBase_{RLE}$ or dictionary in the header.*

Specifically, when the interval D is constant, valid positions can be found directly based on t_1 and t_2. When $t[0] < t_1$, we can compute the prefix sum by SIMD and avoid decoding each value before reaching a position i with $t[i] = t_1$.

B. Extensive Pruning on Value Filters

When the projection attribute appears in the filter, pruning rules can also be applied directly to encoded data, thereby avoiding decoding. Proposition 5 supports pruning on Delta-encoded data, and Proposition 4 prunes on Delta-Repeat encoded data. We first estimate the lower and upper bounds of the remaining attribute values by packing widths or dictionary (when ZigZag or pattern-based Packing is used to encode data). These lower/upper bounds are then utilized to test the filter, helping to determine whether the decoding process should be terminated.

Proposition 5. *Consider the time series encoded by combining Packing and Delta encoders, and a range filter $A > c_1 \wedge A < c_2, c_1 < c_2$. Given $a[k]$ as a decoded value with position k and n as the size of the input sequence, we prune the remaining sequence when (1) $a[k] < c_1$ and $D_M < \frac{c_1 - a[k]}{n-k-1}$; or (2) $a[k] > c_2$ and $D_m > \frac{c_2 - a[k]}{n-k-1}$. The variables D_m and D_M are the lower and upper bounds of delta values, estimated by packing widths ω and the value of $minBase$ in the header. Specifically, $D_m \geq minBase$ and $D_M \leq minBase + 2^\omega - 1$.*

VI. System Integration and Implementation

This section details how the different proposed techniques are combined to execute time-series queries in an open-source IoT database, Apache IoTDB [22].

A. Technique Integration

Algorithm 2 summarizes how decoding pipelines integrate with other query operations like aggregation, join, and series merging [16]. It requires the storage information and logical query plan e to construct the parallel pipelines. The construction process is a top-down traversal of query operators. In the Apache IoTDB, the file system stores each time series into multiple pages [27], which are bit arrays encoded separately.

Consider data filtering $\sigma_\theta(e')$ at Line 1. When θ conjuncts single-column and inter-column predicates, we need to separate them to push down simple filters to prune decoding,

$$\mathcal{J} = [e_i, Pipe(P, \sigma_{\theta \setminus \theta'}(e_i)) \mid e_i \in Pipe(P, \sigma_{\theta'}(e'))], \quad (1)$$

where $Pipe$ is invoked recursively to construct series decoding before generating inter-column filtering pipelines. In addition, $[e_i, \sigma(e_i) \mid e_i \in Pipe]$ means to construct a pipeline composed of e_i and $\sigma(e_i)$, where e_i executes ahead of $\sigma(e_i)$. For simple filtering on time series (Line 5), we check if the remaining related pages are enough to apply to cores by $\sum_{ts \in P} |P(ts)| < p_c$; otherwise, we split pages down into slices according to Section III-C. We generate decoding pipelines in $\mathcal{J}$ for timestamps and values on each page with single-column filters to prune vector loading (Section V),

$$[dec(p_i.T), dec(p_i.A), \sigma_\theta(p_i) \mid p_i \in P(e'), \exists t, a \models \theta]. \quad (2)$$

Algorithm 2 Pipe: Combine decoding with operations

Input: Query plan e and disk pages to decode for related time series $P : \{ts_i \rightarrow [p_1, p_2 \dots]\}$.
Output: Pipeline jobs for threads $\mathcal{J} : [job_i : e_i]$.

```
1: if e = σ_θ(e') or e is a time series then
2:    if θ contains single-column predicates θ' then
3:       Separate different filters to get J by Eq.1
4:    else if e or e' is a time series then
5:       if There are more cores than rest pages then
6:          Slice the rest pages by Section III-C
7:       Decide decoding pipeline instances J by Eq.2
8:    else  Apply Inter-column filters by Eq.3
9: else if e = G_{sw(T_min,ΔT):f}(e'), f is associative then
10:   Build aggregation jobs by Eq.4
11: else if e = e_1 ∘ e_2 is a series concatenation node then
12:   Prepare pipeline J by Eq.5.
13: else if e = e_l ⋈ e_r is a natural join node then
14:   Generate natural join pipeline by Eq.6
15: return J
```

After assigning the pages/slices to a pipeline instance, we remove consumed pages from information P. For inter-column filters, we can apply them to decoded vectors,

$$\mathcal{J} = [e_i, \sigma_\theta(e_i) \mid e_i \in Pipe(P, e')]. \quad (3)$$

Line 9 splits aggregation jobs by windows $w_i(T_{min} + i * \Delta T, \Delta T)$, representing a time range $T_{min} + i * \Delta T \leq T < T_{min} + (i+1) * \Delta T$. We construct pipelines $\mathcal{J}$ to apply f to the time range query pipeline $\sigma_{w_i}(e')$ and collect all results,

$$[e_i, f(e_i) \mid e_i \in Pipe(P, \sigma_{w_i}(e'))] \cup Merge(\mathcal{J}). \quad (4)$$

Finally, series merging and natural join are binary operators. The series concatenation $e_l \circ e_r$ constructs $\mathcal{J}$ by,

$$[e_{l,i}, e_{r,i}, e_{l,i} \circ e_{r,i} \mid e_{l,i}.T \cap e_{r,i}.T] \cup Merge(\mathcal{J}), \quad (5)$$

where $e_{l,i} \circ e_{r,i}$ applies to overlapped pages $e_{l,i}.T \cap e_{r,i}.T$, $e_{l,i} \in Pipe(P, e_l)$, and $e_{r,i} \in Pipe(P, e_r)$. Unlike series merging, the natural join $e_l \bowtie e_r$ (Line 13) only finds a subset of tuples sharing equal timestamps. It needs to integrate the decoding pipelines of multiple attributes, like different series values. $\mathcal{J}$ is composed of

$$[e_{l,i}, e_{r,i}, e_{l,i}.T \bowtie e_{r,i}.T, (e_{l,i}.A, e_{r,i}.A)mask \mid e_{l,i}, e_{r,i}] \quad (6)$$

together with a merge node $Merge(\mathcal{J})$, where $mask$ is timestamp natural join result $e_{l,i}.T \bowtie e_{r,i}.T$ to apply to vectors.

While filters integrate with decoding pipelines directly, other operations append new expressions (pipeline nodes) to the tails of dependent pipelines. To avoid materializing decoded data to memory, the decoding of timestamps and values is executed in a round-robin fashion, one series at a time, similar to vectorized engines like Monet/X100 [28]. In each period, we load/decode a part of bit arrays into timestamp/value vectors based on Algorithm 1 and execute pipelines generated by Algorithm 2. Unlike Monet/X100, for binary operators on multiple columns, we separate and group decoding pipelines by time ranges, controlled by a merge node to combine multiple attributes. Figure 9 illustrates how query operations integrate multiple decoding pipelines. In Figure 9(a), each merge node is responsible for a time range to integrate all related decoding pipelines, where Thread 1 and 3 decode time-series *ts1* and share the same time range with *ts2* in Thread 2. Figure 9(b) shows the join pipeline, where the first three threads are in the time range [0:00, 1:00) and the mask vectors are generated within the shared time range. The variable packing width input will be processed in parallel by cores, with each core utilizing an equally sized page slice along with extra bits (Section III-C) instead of randomly seeking offsets.

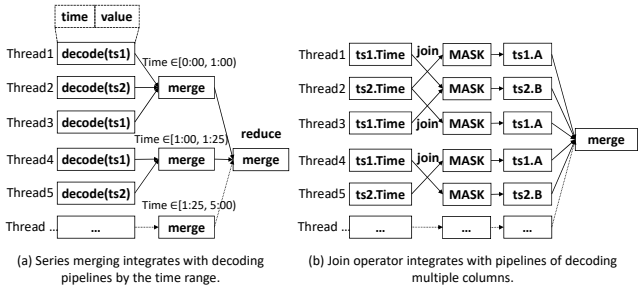


Fig. 9: Integration of decoding pipelines with query operations.

B. System Setups

The Apache IoTDB executes pipelines by the Java Virtual Machine, version 17.0. Our vectorized query processing engine consists of a SQL parser for converting SQL queries to logical plans, a pipeline generator (Algorithm 2) to transfer plans to pipeline jobs, and a job scheduler to bind pipeline expressions with CPU threads. We implement decoders and query operators by the Java Vector API [29], such as unpacking and filtering. The pipeline of each thread is compiled Just-in-Time (JIT) to decode each page by specific encoding parameters (Section III-B) and inline vectorized operators. Since decoding layouts are decided at runtime, we adjust vectorized operators to execute on correspondent values. For instance, the mask vectors from time filtering have the same layout as timestamp vectors and may differ from series value layouts, requiring a shuffling operation to adjust masks.

C. Implementation Details

This section discusses the fundamental challenges and details of implementing our integrations.

Memory management. Since IoT data can accumulate into long series, loading all queried pages in memory is impossible. Thereby, for a task with heavy processing time, the Apache IoTDB will load pages gradually based on memory consumption and pipeline execution. Thus, ETSQP has much lower memory overhead when loading pages, and the database can launch multiple pipelines by different cores simultaneously.

Behavior on failures. IoT databases have the risk of computing incorrectly due to overflows. While decoding efficiently

TABLE II: Dataset statistics

Name	Label	#Size	#Attr	Category
Atmosphere	Atm	132K	3	IoT
Climate	Clim	8.4M	4	IoT
Gas [30]	Gas	925k	19	IoT, Open
Timestamp	Time	1B	2	IoT
Sine-function	Sine	1B	6	Generated
TPC-H [31]	TPCH	24k	4	Generated

TABLE III: Benchmark queries

Query id	SQL expressions
Q1	SELECT SUM(A) FROM ts(T, A) SW($T_{min}, \Delta T$);
Q2	SELECT AVG(A) FROM ts(T, A) SW($T_{min}, \Delta T$);
Q3	SELECT SUM(A) FROM (SELECT * FROM ts WHERE A $>$ a);
Q4	SELECT ts1.A+ts2.A FROM ts1, ts2;
Q5	SELECT * FROM ts1 UNION ts2 ... ORDER BY TIME;
Q6	SELECT * FROM ts1, ts2 ...;

in each thread using vector quantities of fewer bits, query operators also apply to shared decoded registers, causing potential overflows. Most operators without value changes, such as filters and joins, have no overflow issues. We explicitly check the input and output vectors for valid value and sliding window aggregations. For sums of values, we check lane symbols and raise an overflow error when two corresponding lanes of the same symbol are different from the lane in the result vector. Thus, failure checking also works for negative values, such as temperatures. We apply a mask to save overflowed input lanes to aggregate with a larger quantity.

VII. Evaluation

We evaluate our integrated pipeline engine by the following questions. (1) Our performance on real-world IoT queries (Section VII-B-VII-C). (2) The contributions of each encoder on improving aggregation throughput (Section VII-D). (3) The efficiency of the integrated IoT database compared to other traditional analytical engines (Section VII-E). (4) The effectiveness of our implementation details (Section VII-F).

A. Benchmark Details and Baselines

Execution model: To simulate real-world IoT database deployment, we evaluate the formalized pipelines on a machine with 2×Intel Xeon(R) E5-2620 v4 CPUs, 128 GB RAM, 20 MB cache on-chip, and AVX2 instructions. We build our project by the Clang compiler v16.0 with the -O3 flag.

Datasets: Table II lists dataset statistics. Except for the synthetic datasets Sine and TPCH, other datasets are open-source or collected from real IoT scenarios.

Baselines: We compare different parallel implementations by integrating them as operators in our pipeline.

(1) **ETSQP** is our parallel pipeline without pruning rules.

(2) **ETSQP-prune** applies the pruning rules in Section V to improve I/O performance.

(3) **Serial** applies a serial pipeline to decode and aggregate.

(4) **FastLanes** [7] is a state-of-the-art SIMD accelerated decoding approach focusing mostly on a single decoder by storing data in a SIMD-friendly manner.

(5) **SBoost** [23] accelerates Delta decoding by SIMD instructions and threads without unpacking layout determination and operator fusion.

Benchmark queries: Based on real-world applications [32], [33], Table III shows 6 queries in SQL query language [34] that benchmark our proposed time-series query techniques. Aggregation queries with time or value range filters (**Q1-Q3**) test the performance of decoding and operator fusion.

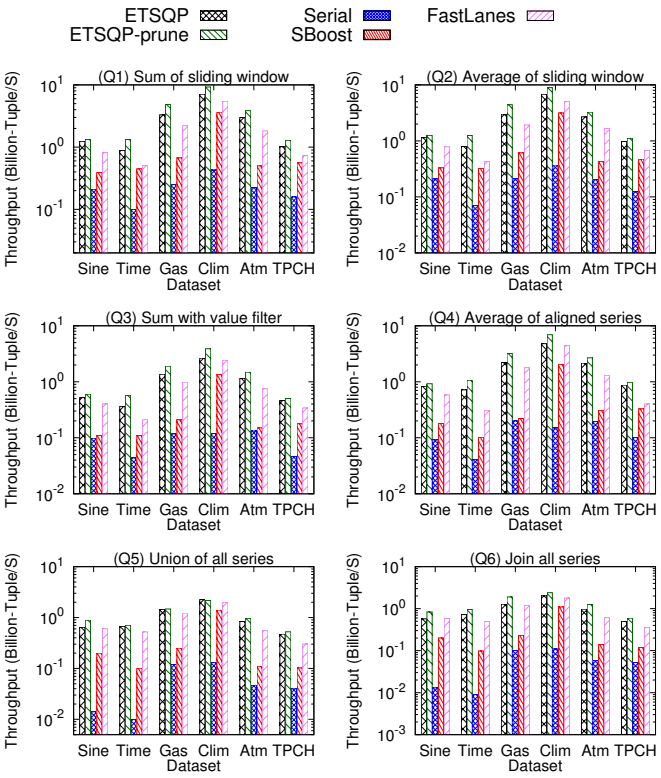


Fig. 10: Performance of SIMD approaches over IoT queries.

Based on Section VI-A, complex queries (**Q5-Q6**) integrate decoding pipelines with query operations and decode multiple columns.Specifically, series merge query **Q5** integrates decoding pipelines of overlapped time ranges, and natural join query **Q5** should decode multiple timestamp columns before exchanging and generating mask vectors. The default filter selectivity is 0.5. Each sliding window instance has 10^3 points.

B. Performance of IoT Queries

We use the throughput to compare the efficiency of proposals over different datasets. The throughput evaluates the number of tuples in loaded pages per second that counts tuples of pruned pages or page slices.

Figure 10 studies the decoding and querying efficiency over the widely applied IoT encoder, TS2DIFF. It shows up to one order-of-magnitude improvement in throughput compared to serial pipelines and 3-10 times acceleration over existing approaches (SBoost [23] and FastLanes [7]). It shows our

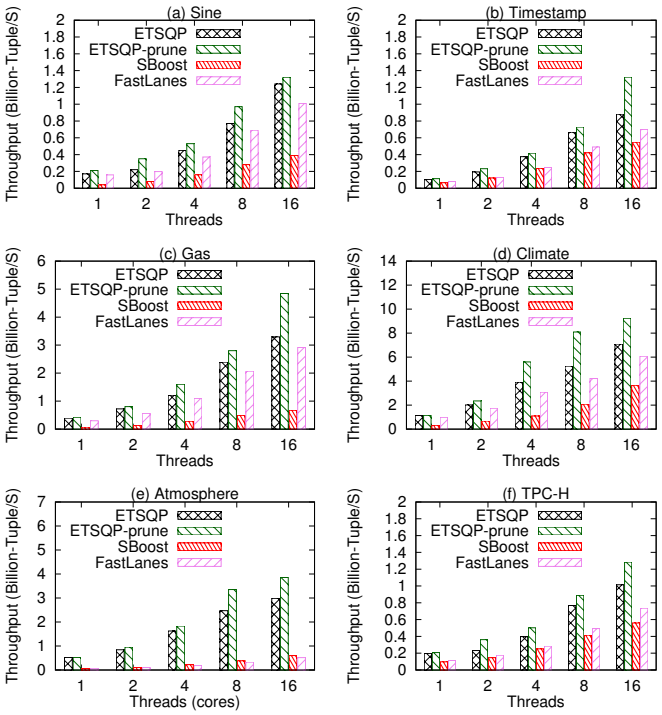


Fig. 11: Query performance over varied threads.

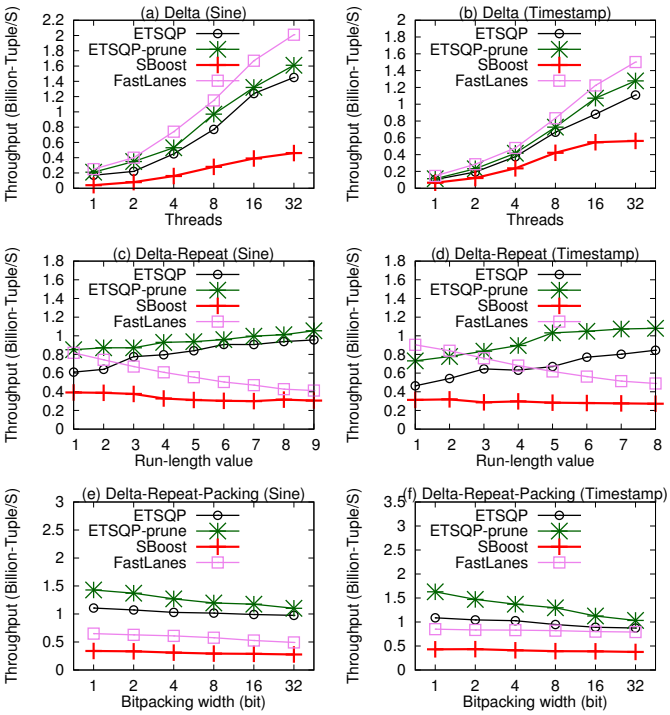


Fig. 12: Query performance on micro-benchmark.

efficiency by adopting core- and instruction-level parallelism. The pruning rules avoid considering unnecessary slices, examined by comparing ETSQP and ETSQP-prune. Theorem 2 is examined by Q3 in Figure 10, which requires decoding data before applying value filters. Figure 10(Q5-Q6) shows our proposed pipeline ETSQP can robustly schedule complex jobs on multi-column queries. Compared to other baselines like FastLanes, merging time series with two columns (timestamps and values) is observed with more significant performance gaps under larger datasets in Figure 10(Q5), indicating the effectiveness of time-based merge nodes. Moreover, for inter-column operations, our pipelines outperform other works on time-series natural join in Figure 10(Q6) because of the shared mask vectors to reduce I/O.

C. Micro-benchmark for Pipeline Threads

Our proposed system, ETSQP, and baseline SBoost [23] support multithreading. SBoost splits the data into slices, where a thread processes each slice. As shown in Figure 8, when splitting each page into slices, some threads should keep waiting for the end of the decoding process of a dependent slice. For example, page slice P1S2 should wait for P1S1. The CPU idle time and memory occupation are insignificant when we have few slices. Thus, our job scheduler splits pages only when there are insufficient pages to assign to free cores; otherwise, decoding and query pipelines consume data pages. This advance makes our work more efficient in deploying on more threads than SBoost. As shown in Figure 11, ETSQP can benefit more throughput gains from threads. Finally, we distribute FLMM1024 formatted pages or page slices of FLMM1024 blocks to ensure fair thread utilization. We notice that FastLanes can also benefit from slicing and parallel processing, but FastLanes performance gains grow slower, owing to its lower compression ratio and I/O bottleneck for IoT queries.

D. Micro-benchmark Evaluation of Operators

This section evaluates the parallel performance of each operator by a more careful micro-benchmark. SBoost [23] does not change the layouts and is easy to extend to accelerate combined decoders. We analyze the root causes of the performance differences between our proposals and SBoost [23] with data encoded by Delta, Repeat, and Packing by a full schema correlation on Sine and timestamp datasets. Specifically, the Delta-only encoding ensures the same data representation in both studies for comparison. Together with Repeat and Packing, we control for specific algorithmic insights (like pruning parameters) in ETSQP and ETSQP-prune. Figure 12 reports the throughput of different operator parameters. We benchmark the decoding pipelines by a simple time range query with a default selectivity of 0.5. ETSQP-prune adopts time-based pruning rules to prune pipeline processing, while ETSQP needs to decode and filter assigned pages, which have time ranges intersecting with the time-range filter.

(1) Delta-only encoding, the same data representation considered in SBoost [23], is utilized for a fair comparison. Compared to SBoost, we split pages into slices when there are not enough pages for cores to avoid thread idle and synchronization time. Thereby, with the increase of the thread number, our approaches utilize better CPU resources than

SBoost, i.e., more significant throughput improvement by more threads in Figure 12(a)-(b).

(2) The Delta-Repeat combination evaluates the algorithmic decision of directly counting the satisfied tuples without decoding into the original data. A larger run-length (Repeat) value means more saved decoding costs by our methods. In contrast, the larger run-length is, SBoost needs the more unpacking operations. Therefore, the differences between our proposals and SBoost are more significant for larger run lengths in Figure 12(c)-(d).

(3) For the Delta-Repeat-Packing combination, it controls for the cutoffs by our ETSQP-prune, i.e., the pruning parameters. Proposition 5 derives the lower and upper bounds of values, referring to the Bitpacking widths of Delta and RLE encoded sequences. The ETSQP-prune method prunes subsequences based on the derived bounds, which cannot satisfy the query ranges. A smaller Bitpacking width leads to tighter bounds and thus more effective pruning, as illustrated in Figure 12(e)-(f).

(4) To compare the new encoding format FastLanes, in Figure 12(a-b), based on a fair comparison of shared core-level parallelism, we notice better performance by FLMM1024 layout for SIMD decoding on Delta sequences. In Figure 12(c-d), more repeated deltas mean better compression and I/O performance of ETSQP based on IoT encoders, while FastLanes has an I/O bottleneck. In Figure 12(e-f), although FastLanes FLMM1024 also avoids leading zeros in IoT data, FastLanes-BitPacking is not space-efficient compared to IoT encoders and meets the I/O bottleneck when packing widths grow (meanwhile, data points stay unvaried).

E. Evaluations of System Deployment

Comparing IoTDB to IoTDB-SIMD, we notice a 10-40% acceleration of simple queries (Q1-Q4) among datasets and a 30-40% improvement of complex queries (Q5-Q6). Without vectorized decoders and query operations, IoTDB should unpack data by reading each byte sequentially and sharing data blocks with cache coherence. In contrast, IoTDB-SIMD adopts round-robin vectorized loading/decoding, which shares vectors to avoid materializing data in memory. Comparing IoTDB-SIMD to MonetDB, IoT encoders reduce the I/O latency, and merging nodes on shared time ranges avoids materializing and broadcasting decoded data across threads. Comparing IoTDB-SIMD to Spark/HDFS, we share common in generating query execution codes at query time (JIT). The HDFS compressor is not efficient enough to reduce I/O, which is a bottleneck in executing IoT queries.

F. Parameter Studies

Figure 14 shows our parameter studies on fused decoders, staged time consumptions, and the effect of synchronization when splitting pages. Figure 14(a) shows our advances in fusing as many decoders as possible to avoid dependency resolutions. Figure 14(b) shows memory I/O is one of the bottleneck stages in IoT databases with 40%-50% processing time, which includes the time for distributing pages to each

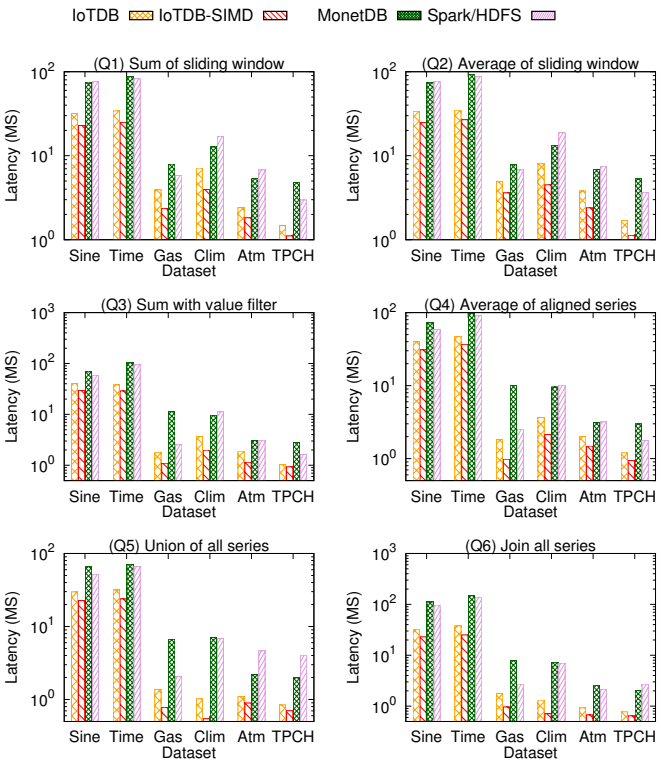


Fig. 13: Performance comparison of answering time and value range queries deployed on the Apache IoTDB.

core and collecting results. For page slice inputs, Figure 14(c-d) suggests avoiding idles will reduce execution time, whereas materializing decoded results can cost significantly more than idle time. Splitting the query pipeline into two tasks and letting the threads pick the tasks without waiting for the prefix sum reduces idle time. When generating more slices, it is possible to reduce execution time by avoiding idles. Nevertheless, splitting pipelines fails to share vectors and materializes more unpacked data in memory, leading to more I/O, which is the bottleneck of most IoT queries.

VIII. Related Work

In this section, we discuss the widely used time series encoding schemes and methods for improving decoding performance. Existing works also construct query plans to consider encoder schemes as query optimizations for selective aggregations.

A. Time Series Encoding

The introduction (Section I) indicates the SIMD-friendly layouts extended from existing encoders can not be combined to encode IoT data effectively, such as the FastLanes [7]. As surveyed in [1], existing mainstream databases have long supported time series encoding schemes like Delta-Repeat-Packing [15], [35], [22] to compress and save the I/O bandwidth [4]. Delta encoding subtracts the previous data from the current data to reduce the bit-widths and has motivated

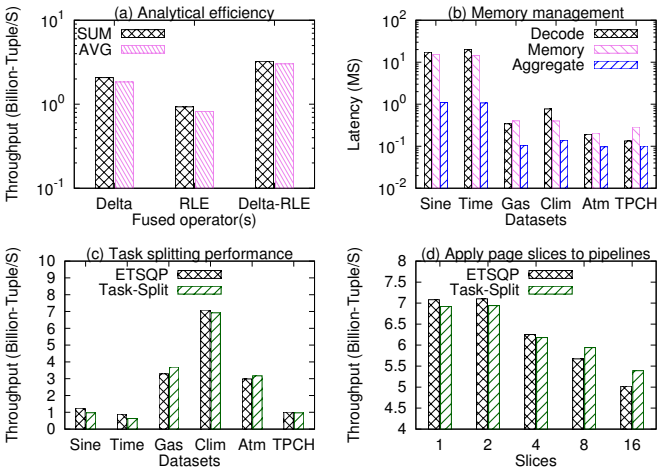


Fig. 14: Ablation study of our parallel pipeline designs.

the studies including TS2DIFF [22], [15] and Gorilla [6]. Gorilla also observes constant increases in IoT timestamps and applies two Deltas on the time dimension. When there are many repeat values in the time series, run-length encoder [36] uses one of the consecutive repeated values and its repeat times to reduce space usage. The Repeat encoder is applied in RLBE [2] and Gorilla [6] encoders. While RLBE applies a run-length encoder as described above, the Gorilla uses a bit (setting as 1) to indicate redundancy, which reduces the space needed to store run lengths. Since IoT data (differences) have limited precision and absolute value, the Packing encoder removes leading zeros. The constant-width Packing encoders can reduce the leading zeros. For example, the Packing method in Sprintz [11] uses ZigZag and BitPacking to convert negative values to positive codings. The pattern-based variable width Packing notices the trailing zeros of IoT data in floating number formats such as Elf [12]. The Fibonacci representation also packs data with variable widths but has the acceleration method to decode in parallel (Figure 7).

B. Time Series Decoders

Observing the hardness of decoding data, existing works propose multi-threading and instruction-based accelerations such as [37] for BitPacking, SBoost [23] and FastLanes [7] for Delta, run-length, and BitPacking. Using more threads requires splitting input data into slices, which relies on easy separation of elements, e.g., constant packing widths. The SIMD methods use instructions over vectors of 128/256/512 bits to decode multiple attribute values simultaneously.

Some of the decoding approaches also improve the layouts of encoding schemes to generate SIMD-friendly codings, such as FastLanes [7] and Lemire [8] on Frame-Of-Reference. While existing encoders compress data by elements, they keep a group of data to encode. For example, FastLanes-Delta keeps each frame i of four elements at positions $0{+}i, 4{+}i, 8{+}i, 12{+}i$ and finds the differences of consecutive frames (every four elements). They focus on better layouts for a single encoder, but IoT data apply combined encoders, asking for compatible layouts. For example, FastLanes-BitPacking keeps elements at positions 0 and 128 together to shift and construct unpacked vectors. Thus, the proposed Delta and Bitpacking layouts can not form combined encoders for IoT data.

C. Query Accelerations on Encoding Schemes

Our approach constructs query pipelines concerning existing encoding schemes. Existing works have exploited sufficiently in implementing relational SIMD operators [38] and generate query plans based on compression methods [21], [39], [20]. Instead of using the dictionary and run-length encoders on the raw data, IoT data is encoded by consecutive differences, leading to different implementations of operators. BitWeaving [40] also considers the Bitpacking schemes to reduce data loading, while our work focuses on the Delta-based encoding that has the dependency of decoding and filtering. IoT data pages comprise bit arrays encoded from time-ordered tuples, making it possible to prune a page or a slice directly on filters.

D. System Comparisons

Our proposal, ETSQP, improves IoT decoder performance by altering the layout of unpacking, which resolves Delta recovery dependency and shares data in registers between decoders. Instead, existing analytical engines encode data by single encoders and decompress data by blocks, causing higher I/O among disks, memory, and cores. Moreover, our pipelines support operator fusion on multiple decoders, such as Repeat flattening (RLE) and Delta accumulation. Finally, based on ordered time-series data, we create merge nodes on each time range to execute inter-column operations like natural joins, whereas other general systems do not make such assumptions.

IX. Conclusions

We observe the differences between IoT encoders and the encoders for relational data. That is, IoT data is an ordered sequence typically encoded by combined encoders, formatted as Delta-Repeat-Packing. This paper formalizes the pipeline models to execute selective aggregation queries over encoded data. We identify the basic operators to decode, filter, and aggregate for serial pipelines. The parallel pipeline contains new operators formally defined by SIMD instructions and data splitting to decode more efficiently. We also notice the chances of aggregation without decoding and pruning unnecessary aggregations. Remarkably, our systemic evaluations show vast improvements in the efficiency of selective aggregations over existing works. Additionally, our evaluations could help to choose better existing encoders for IoT data.

Acknowledgement: This work is supported in part by the National Key Research and Development Plan (2021YFB3300500), the National Natural Science Foundation of China (92267203, 62021002, 62072265, 62232005), State Grid 5700-202435261A-1-1-ZN, and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] G. Chiarot and C. Silvestri, "Time series compression survey," *ACM Comput. Surv.*, vol. 55, no. 10, pp. 198:1–198:32, 2023. [Online]. Available: https://doi.org/10.1145/3560814

[2] J. Spiegel, P. Wira, and G. Hermann, "A comparative experimental study of lossless compression algorithms for enhancing energy efficiency in smart meters," in *16th IEEE International Conference on Industrial Informatics, INDIN 2018, Porto, Portugal, July 18-20, 2018*. IEEE, 2018, pp. 447–452. [Online]. Available: https://doi.org/10.1109/INDIN.2018.8471921

[3] A. Kamilaris, Y. Tofis, C. Bekara, A. Pitsillides, and E. Kyriakides, "Integrating web-enabled energy-aware smart homes to the smart grid," *International Journal On Advances in Intelligent Systems*, vol. 5, no. 1, pp. 15–31, 2012.

[4] J. Xiao, Y. Huang, C. Hu, S. Song, X. Huang, and J. Wang, "Time series data encoding for efficient storage: A comparative analysis in apache iotdb," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2148–2160, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2148-song.pdf

[5] P. Liakos, K. Papakonstantinopoulou, and Y. Kotidis, "Chimp: Efficient lossless floating point compression for time series databases," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3058–3070, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3058-liakos.pdf

[6] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1816–1827, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1816-teller.pdf

[7] A. Afroozeh and P. A. Boncz, "The fastlanes compression layout: Decoding >100 billion integers per second with scalar code," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2132–2144, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p2132-afroozeh.pdf

[8] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Softw. Pract. Exp.*, vol. 45, no. 1, pp. 1–29, 2015. [Online]. Available: https://doi.org/10.1002/spe.2203

[9] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen, "A general simd-based approach to accelerating compression algorithms," *ACM Trans. Inf. Syst.*, vol. 33, no. 3, pp. 15:1–15:28, 2015. [Online]. Available: https://doi.org/10.1145/2735629

[10] https://iotdb.apache.org/UserGuide/Master/Data-Concept/Encoding.html.

[11] D. W. Blalock, S. Madden, and J. V. Guttag, "Sprintz: Time series compression for the internet of things," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 3, pp. 93:1–93:23, 2018. [Online]. Available: https://doi.org/10.1145/3264903

[12] R. Li, Z. Li, Y. Wu, C. Chen, and Y. Zheng, "Elf: Erasing-based lossless floating-point compression," *Proc. VLDB Endow.*, vol. 16, no. 7, pp. 1763–1776, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p1763-li.pdf

[13] R. Kang and S. Song, "Optimizing time series queries with versions," *Proc. ACM Manag. Data*, vol. 2, no. 3, p. 159, 2024. [Online]. Available: https://doi.org/10.1145/3654962

[14] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. Mcgrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun, "Apache iotdb: Time-series database for internet of things," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2901–2904, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2901-wang.pdf

[15] https://www.influxdata.com/.

[16] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, H. Garcia-Molina and H. V. Jagadish, Eds. ACM Press, 1990, pp. 102–111. [Online]. Available: https://doi.org/10.1145/93597.98720

[17] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 743–754. [Online]. Available: https://doi.org/10.1145/2588555.2610507

[18] "Nvidia official document on prefix sum," pp. part–vi–gpu–computing/chapter–39–parallel–prefix–sum–scan–cuda. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/

[19] D. Merrill and M. Garland, "Single-pass parallel prefix scan with decoupled look-back," *NVIDIA, Tech. Rep. NVR-2016-002*, 2016.

[20] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, S. Chaudhuri, V. Hristidis, and N. Polyzotis, Eds. ACM, 2006, pp. 671–682. [Online]. Available: https://doi.org/10.1145/1142473.1142548

[21] J. Li, D. Rotem, and J. Srivastava, "Aggregation algorithms for very large compressed data warehouses," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds. Morgan Kaufmann, 1999, pp. 651–662. [Online]. Available: http://www.vldb.org/conf/1999/P61.pdf

[22] C. Wang, J. Qiao, X. Huang, S. Song, H. Hou, T. Jiang, L. Rui, J. Wang, and J. Sun, "Apache iotdb: A time series database for iot applications," in *ACM SIGMOD International Conference on Management of Data, SIGMOD*, 2023.

[23] H. Jiang and A. J. Elmore, "Boosting data filtering on columnar encoding with SIMD," in *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, W. Lehner and K. Salem, Eds. ACM, 2018, pp. 6:1–6:10. [Online]. Available: https://doi.org/10.1145/3211922.3211932

[24] F. Zhang, W. Wan, C. Zhang, J. Zhai, Y. Chai, H. Li, and X. Du, "Compressdb: Enabling efficient compressed data direct processing for various databases," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 1655–1669. [Online]. Available: https://doi.org/10.1145/3514221.3526130

[25] Intel, "Intel intrinsic guide." pp. intrinsics–guide/index.html, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs

[26] F. Lemaitre, A. M. Hennequin, and L. Lacassagne, "How to speed connected component labeling up with SIMD RLE algorithms," in *WPMVP@PPoPP '20: Sixth Workshop on Programming Models for SIMD/Vector Processing lolocated with the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, USA, February 22, 2020*, J. Eitzinger and L. Lacassagne, Eds. ACM, 2020, pp. 2:1–2:8. [Online]. Available: https://doi.org/10.1145/3380479.3380481

[27] X. Zhao, J. Qiao, X. Huang, C. Wang, S. Song, and J. Wang, "Apache tsfile: An iot-native time series file format," *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 4064–4076, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p4064-song.pdf

[28] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 225–237. [Online]. Available: http://cidrdb.org/cidr2005/papers/P19.pdf

[29] "Java vector api." [Online]. Available: https://openjdk.org/jeps/338

[30] "Gas dataset (uci)," pp. Gas+sensors+for+home+activity+monitoring. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/

[31] TPC-H. [Online]. Available: https://www.tpc.org/tpch/

[32] A. Khelifati, M. Khayati, A. Dignös, D. Difallah, and P. Cudré-Mauroux, "Tsm-bench: Benchmarking time series database systems for monitoring applications," *Proc. VLDB Endow.*, vol. 16, no. 11, p. 3363–3376, aug 2023. [Online]. Available: https://doi.org/10.14778/3611479.3611532

[33] R. Liu and J. Yuan, "Benchmark time series database with iotdb-benchmark for iot scenarios," *CoRR*, vol. abs/1901.08304, 2019. [Online]. Available: http://arxiv.org/abs/1901.08304

[34] S. J. Cannan and G. A. M. Otten, *SQL - The Standard Handbook*. McGraw-Hill Book Company, 1993.

[35] https://www.timescale.com/.

[36] S. W. Golomb, "Run-length encodings (corresp.)," *IEEE Trans. Inf. Theory*, vol. 12, no. 3, pp. 399–401, 1966. [Online]. Available: https://doi.org/10.1109/TIT.1966.1053907

[37] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 385–394, 2009. [Online]. Available: http://www.vldb.org/pvldb/vol2/vldb09-327.pdf

[38] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, M. J. Franklin, B. Moon, and

A. Ailamaki, Eds. ACM, 2002, pp. 145–156. [Online]. Available: https://doi.org/10.1145/564691.564709

[39] Z. Chen, J. Gehrke, and F. Korn, "Query optimization in compressed database systems," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, S. Mehrotra and T. K. Sellis, Eds. ACM, 2001, pp. 271–282. [Online]. Available: https://doi.org/10.1145/375663.375692

[40] Y. Li and J. M. Patel, "Bitweaving: fast scans for main memory data processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 289–300. [Online]. Available: https://doi.org/10.1145/2463676.2465322