OM^3: An Ordered Multi-level Min-Max Representation for Interactive Progressive Visualization of Time Series

YUNHAI WANG*, YUCHUN WANG*, and XIN CHEN*, Shandong University, China
YUE ZHAO, Shandong University, China
FAN ZHANG, Shandong Technology and Business University, China
EUGENE WU, Columbia University, United States
CHI-WING FU, Chinese University of Hong Kong, China
XIAOHUI YU, York University, Canada

We present a novel multi-level representation of time series called OM^3 that facilitates efficient interactive progressive visualization of large data stored in a database and supports various interactions such as resizing, panning, zooming, and visual query. Based on our proposed line-segment aggregation, this representation can produce error-free line visualizations that preserve the shape of a time series in windows of arbitrary sizes. To reduce the interaction latency, we develop an incremental tree-based query strategy to support progressive visualizations, allowing a finer control on the accuracy-time tradeoff. We quantitatively compare OM^3 with state-of-the-art methods, including a method implemented on a leading time-series database InfluxDB, in two settings with databases residing either in the local area network or on the cloud. Results show that OM^3 maintains a low latency within 300 ms on the web browser and a high data reduction ratio regardless of the data size (ranging from millions to billions of records), achieving around 1,000 times faster than the state-of-the-art methods on the largest dataset experimented with.

CCS Concepts: • **Information systems → Query optimization**; • **Human-centered computing → Information visualization**.

Additional Key Words and Phrases: Time series, interactive progressive visualization

ACM Reference Format:
Yunhai Wang, Yuchun Wang, Xin Chen, Yue Zhao, Fan Zhang, Eugene Wu, Chi-Wing Fu, and Xiaohui Yu. 2023. OM^3: An Ordered Multi-level Min-Max Representation for Interactive Progressive Visualization of Time Series. *Proc. ACM Manag. Data* 1, 2, Article 145 (June 2023), 24 pages. https://doi.org/10.1145/3589290

1 INTRODUCTION

The past decades have witnessed an explosion of time-series data in many applications, from financial engineering to manufacturing. A large amount of time-series data is collected by measuring variables over time at regular intervals, and usually stored in remote databases on cloud servers for

*The authors contributed equally to this research.

Authors' addresses: Yunhai Wang, cloudseawang@gmail.com; Yuchun Wang, iwangyuchun@gmail.com; Xin Chen, chenxin199634@gmail.com, Shandong University, Qingdao, Shandong, China, 266237; Yue Zhao, jack.zhao9802@gmail.com, Shandong University, Qingdao, Shandong, China, 266237; Fan Zhang, zhangfan@sdtbu.edu.cn, Shandong Technology and Business University, Yantai, Shandong, China, 265600; Eugene Wu, ewu@cs.columbia.edu, Columbia University, New York City, New York, United States, 10027; Chi-Wing Fu, cwfu@cse.cuhk.edu.hk, Chinese University of Hong Kong, Sha Tin, Hong Kong, China; Xiaohui Yu, xhyu@yorku.ca, York University, Toronto, Ontario, Canada, M3J1P3.

2836-6573/2023/6-ART145 $15.00
https://doi.org/10.1145/3589290

subsequent analysis via interactive visualizations on the client side. By interacting with time series displayed as line charts, users can conduct analytical tasks [1], such as peak identification, trend analysis, and pattern search.

To improve the user exploration efficiency, a variety of interaction techniques [1] have been developed, e.g., SignalLens [16], multi-focus zooming [12, 29], Zenvisage [27], and a few visual query tools [10, 21]. Recently, Siddiqui et al. [28] proposed an expressive shape search algebra, enabling users to interactively search for various desired patterns. However, almost all existing techniques require the whole data to be quickly loaded onto the client side for efficient rendering. Yet, the latency is often high for large time-series data stored on remote databases. This hinders smooth interactive exploration of temporal patterns at various resolutions.

A common approach to reducing the latency is data reduction [4], by aggregating the data to reduce its size before visualization. Various strategies have been proposed to preserve salient features in the reduction. However, most of them do not account for the perceptual effects of, e.g., resizing the display, and thus often produce erroneous visualizations that can significantly distort the shape of the rendered data. This issue is addressed by the visualization-oriented time-series reduction method M4 [13], which finds essential records per pixel column (where each pixel column consists of all pixels with the same x coordinate) in the display window to preserve the exact rendering of an input time series, referred as *error-free* line visualizations. Yet, M4 cannot efficiently sustain smooth interactions for large time-series data. First, the time complexity of executing a query is $O(n)$, where n is the time-series length. So, processing data with millions of records could easily take more than a second, which is beyond the latency limit [17] for interactive visual analysis. Second, M4 independently issues and fully executes a new query for each user action on the visualization. For these reasons, it does not support continuous interactions like panning and zooming, and precludes most interaction techniques that are commonly used for time-series exploration.

Progressive visualization [33] is a promising direction, where, instead of waiting for slow queries, the visualization immediately renders intermediate results that the user can potentially interact with. A prominent approach is based on IncVisage [23], which uses online sampling-based techniques to progressively reveal salient features. The approach quickly renders approximate visualizations (on the order of seconds) that update over time and eventually converge to the exact visualization (though this may take a minute or more). Each update sends a new result set to visualize, so network costs are linear in the number of updates—on the order of hundreds before convergence. These characteristics are ill-suited and not widely adopted for interactive visualization of large datasets.

In this paper, we present a new approach to *interactive progressive visualization of arbitrarily-sized time-series*. Our approach is designed to satisfy the following desiderata for interactive visualization of large time-series stored in a remote database: (i) ensuring error-free line visualizations at any scale; (ii) minimizing query latency and amount of data transfer to the visualization clients; (iii) supporting progressive refinement of intermediate visualization with a good trade-off between interaction latency and visualization quality; and (iv) offering rich interactions for fluid data exploration.

Our proposal centers around OM^3, an ordered multi-level min-max representation of a time-series dataset, which enables visualizations that preserve the shape of a time series displayed in any window size. This representation has $\lceil \log n \rceil$ levels and maintains minimum and maximum values at every time interval that is used to rasterize a pixel column in the display window with the width 2^i at the ith ($0 < i < \lceil \log n \rceil$) level. In addition, it tracks the temporal ordering of the paired minimum and maximum values at the leaf level. To do so, we formulate the *forward OM^3 transform* to recursively aggregate the time series for different time intervals at different resolutions, constructing a hierarchy of coefficients obtained by aggregating and differing between

Technique	Vis. Qual.	Data Tran.	Prog. Vis.	Int. Sup.
M4 [13]	✓	✓	×	×
IncVisage [23]	×	×	✓	×
OM^3	✓	✓	✓	✓

Table 1. Comparing the design objective of OM^3 with prior approximate visualization techniques based on four considerations: visualization quality, size of data transferred, progressive visualization and interaction support, where ✓and × indicate whether the technique accounts for a given consideration or not.

the aggregate values of two paired aggregated samples. We store these coefficients in a database table, whose size is only three-quarters of the original time series. By default, the procedure is limited to time-series lengths that are powers of two, and we extend it to support missing data and arbitrary lengths with the same storage space.

With OM^3 interactions issue queries over the hierarchy of coefficients and render the charts on the client using the data fetched from the server. A naive implementation would involve traversing all the coefficients falling into the specific time range for each interaction. This, however, could be prohibitively expensive for large data. To address this issue, we propose a visualization-aware incremental query algorithm of time complexity $O(w \log(n))$, where w is the display-window width and n is the time-series length. For every time interval corresponding to a pixel column in the visualization, it finds four M4 samples [13] and reuses the query results from the previous round of interaction to attain a substantial speed-up. Since the query evaluation proceeds incrementally level by level through the hierarchy of coefficients, the rasterization is progressively performed with a dynamically changed number of pixels rendered on screen and rapidly converges to exact visualizations. For example, it takes less than 300 ms to reach the convergence for the dataset with 1B records. As a further optimization on the cost of traversing OM^3, we develop an *efficient pruning strategy* in a breadth-first search, which can achieve a pruning ratio of around 45% on most datasets. In doing so, interactive, progressive visualization of large time series can become feasible, even for data with billions of records. Table 1 compares the design objectives of OM^3 and existing techniques along four considerations.

OM^3 can support a rich variety of interaction techniques such as panning, zooming, resizing, and TimeBox Search [11]. We implement OM^3 using JavaScript with PostgreSQL [20], and quantitatively compare it with the baseline approaches: M4 on PostgreSQL and a leading time-series database InfluxDB, and Haar wavelet on PostgreSQL, in two settings, with the database residing either in the local area network or on the cloud, on several real-world datasets. Evaluation results show that our method ensures error-free visualizations and offers latency under 300 ms (the upper limit of interactive latency is 500 ms [17]) and high data reduction ratio (99.5%), showing its capability to support progressive visualization of large time-series data. Besides, we conduct a case study to demonstrate the effectiveness of our provided interaction techniques in the exploration of large data.

In summary, we make the following main contributions.

- We develop OM^3, an ordered multi-level min-max representation that forms the basis for delivering interactive visualization of large time-series data with one billion records;
- We propose an incremental query algorithm with an efficient pruning strategy to enable progressive visualization and rich fluid interactions on the client; and
- We quantitatively compare OM^3 with state-of-the-art methods, manifesting the effectiveness of OM^3 in supporting interactive exploration of large time-series data.

The rest of this paper is organized as follows. In Section 2, we provide a review of related work. In Section 3, we revisit min-max aggregation and present an extension to support error-free visualization of time series. Section 4 presents our new solution based on OM^3. Section 5 reports the experimental results, and Section 6 concludes this paper.

2 RELATED WORK

Our work is related to the literature on interaction for line visualizations and data reduction of large time-series data. Below, we discuss these areas one by one.

2.1 Interaction for Line Visualizations

Time-series visualization [1] has been extensively explored for facilitating users to discover trends and patterns at varying time scales. Line chart is still the most popular vehicle for presenting time series. However, displaying many time series as line charts on a limited screen inevitably produces heavy overplot. To address this issue, various lens-based interaction techniques [16] and overview+detail interaction techniques [12, 29] have been developed for simultaneously showing the regions of interest in detail with an overview of the whole data. All these techniques are built on basic interaction operations: resizing, panning, and zooming, with the assumption that the latency of the data query is low. Yet, very few of them have been used for exploration of very large time-series data stored in remote databases. In contrast, the proposed OM^3 and its associated incremental query algorithm focus on enabling efficient interactive visualization of remote time-series data with billions of records.

As a separate line of research that is concerned with efficiently identifying patterns of interest, visual query tools help search for line chart visualizations matching a query pattern specified through the user interface. Among them, sketch-based queries [18, 27, 28, 32] allow users to sketch a temporal pattern to select and show the line charts that match the sketch. Yet, this approach does not provide an overview of the whole data. Instead, Timeboxes [11] and KD-Box [34] act as filters, in which users can specify query constraints by manipulating rectangular boxes in the charts with all time series and then filter out the time series that do not pass through the boxes. Our multi-level representation OM^3 further extends such queries to explore the detail of lines of interest at finer levels.

2.2 Time-series Data Reduction

To reduce the latency for large data query during interactive visualization, a few data reduction techniques have been proposed for time-series data, which can be grouped into two categories: point aggregation and line simplification.

Point Aggregation. To ensure a manageable size for the query data, point aggregation methods often first group the entire time series into a few time spans and then compute an aggregated value and an aggregated timestamp for each interval using aggregation functions such as *minimum*, *maximum*, *mean*, and *median*. A common method is the piece-wise aggregate approximation (PAA) [14], which selects the minimum (first) timestamp and computes an average value. Based on these aggregation functions, a few methods have been proposed to offer fast approximate query [2, 6]. Yet, regardless of the aggregation function being used, the line chart of the aggregated data may distort the original shape of the time series. To address the issue, M4 [13] aims to preserve the shape by selecting two data points with the *minimum* and *maximum* values, and another two with the *minimum* and *maximum* timestamps in each pixel column, connecting the four data points in time order with the rest of the time series, and rasterizing the line segments in the pixel column.

With M4, the number of data points needed to be transferred from the database to the visualization client can be significantly reduced. However, it cannot facilitate smooth interaction of large time-series data due to the following two major drawbacks. First, its query execution has high interaction latency. The most costly operation in the whole query process is the join operation, which has a computational complexity of $O(n + 4w)$ for matching n data points with $4w$ aggregates. Second, M4 cannot reuse previous query results to support incremental update, which is essential to supporting interactive operations such as zooming and panning. As a result, it requires very frequent data query and transfer from the remote database during user exploration, thus hindering effective exploration of large data. Although SampleAction [8] and IFOCUS [15] support incremental visualizations by using online aggregation, they cannot provide error-free visualizations until converging to precise results. In contrast, OM^3 enables users to interactively explore large-scale time-series data with flexible interactions.

Line Simplification. Given a time-series curve, line simplification is another data-reduction approach that aims to preserve its rough shape with fewer curve segments. One widely-used method is the top-down Ramer-Douglas-Peucker [7, 9] algorithm, which joins the first and last points in the curve and divides the curve into two segments, if the distance from the farthest point to the line is larger than a threshold. Then, each curve segment is recursively divided until all points of the original curve are within the simplification's tolerance. Fu et al. [5] further attempts to preserve salient points in the simplification. However, these methods have a high computational complexity, $O(n^2)$ or $O(n \log n)$, (n is the number of data points in the time series), so interactive queries cannot be supported for large data. Instead, INCVISAGE [23] uses online sampling-based algorithms to incrementally generate curve segments, attaining 46x speedup relative to baselines. Yet, it provides only approximate visualizations and does not support interactive operations like zooming. Since the time complexity for constructing the OM^3 representation is only $O(n)$ and querying with OM^3 is only $O(w \log n)$ (w is display-window width), OM^3 is able to support progressive interactive visualization of large time series.

3 LINE-SEGMENT AGGREGATION

In this section, we revisit min-max aggregation and propose a line-segment aggregation of time series for generating error-free visualizations. As shown in Fig. 1(b), simply using the min-max aggregation may produce errors in line-chart visualizations with missing pixels (e.g., E_1) and false pixels (e.g., E_2). To address this issue, M4 [13] locates the two data points of maximum and minimum timestamps and the two data points of maximum and minimum data values in each pixel column and then connects these points to form line segments that represent the line chart. Fig. 1(c) shows its result, which exactly matches the visualization produced by directly rasterizing the whole data points shown in Fig. 1(a). However, some inter-column segments formed by M4 are redundant and unnecessary for yielding error-free visualizations.

Definition 1. A *time series* $T = \{(t_i, v_i))\}_{i=1}^{n}$ is an ordered list of uniformly-sampled data points, where i is the index, t_i is the i-th timestamp ($t_i < t_{i+1}$), and v_i is the data value at time t_i.

Definition 2. An *equidistant grouping* of a time series refers to a non-overlapping partition of T into w groups of same width in time domain: $G(T, w) = \{B_1, B_2, \cdots, B_w\}$, where each group B_i contains the data points in time interval $[(i - 1) * \delta + 1, i * \delta]$, δ is round(n/w), and w is the display-window width. For each group, its value range is $\mathbb{O}_i = [G_{\min}(T, w, i), G_{\max}(T, w, i)]$ defined by the minimal and maximal values $G_{\min}(T, w, i)$ and $G_{\max}(T, w, i)$, respectively.

Definition 3. A line visualization of time series T *error-free*, if the set of its rendered foreground pixels is the same as the set rendered by rasterizing all line segments over all data points in T.

Definition 4. For each group B_i, the *intra-column line segment* is defined by $G_{\min}(T, w, i)$ and $G_{\max}(T, w, i)$ of this group, while the *inter-column line segment* between B_i and the next group B_{i+1} is defined by connecting the last data record of B_i and the first data record of B_{i+1}. As shown in

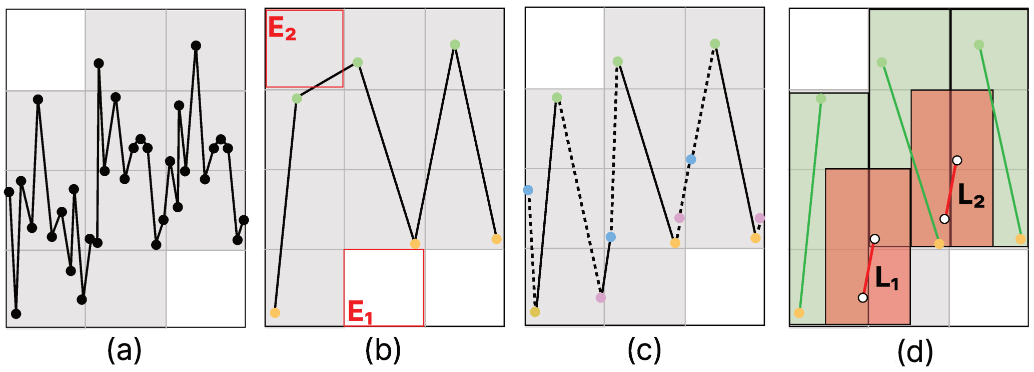


Fig. 1. Illustration of Theorem 1. (a) The visualization created by using all data points, where all rasterized pixels are shown in gray; (b) the visualization of the min-max aggregation yields errors of a missing pixel (E_1) and a false pixel (E_2); (c) the visualization of the M4 aggregation yields the same rasterized pixels as the ones in (a) but the dotted line segments are redundant and unnecessary; and (d) all pixels are correctly set by first rasterizing the lines defined by the min-max aggregation result in each pixel column (in green) and then rasterizing the indispensable inter-column lines (L_1) between adjacent pixel columns.

Fig. 1(b), only rasterizing all intra-column line segments cannot guarantee error-free visualization, but on the other hand, the dotted line segments defined by M4 samples in Fig. 1(c) are redundant and unnecessary.

Definition 5. The *line-segment aggregation* $G_l(T, w)$ of a given w equidistant grouping of T consists of an intra-column line segment in each group and any inter-column line segment whose contained pixels are completely covered in the ranges $\mathbb{O}_i$ and $\mathbb{O}_{i+1}$. For the example in Fig. 1(d), the time series is shown on a three-pixel display window; the line-segment aggregation consists of the three green intra-column lines and one inter-column line L_1.

Theorem 1. The line visualization of a time series T produced by rasterizing its line-segment aggregation $G_l(T, w)$ is error-free.

Proof. If the value range $\mathbb{Q} = [v_j, v_{j+1}]$ of an inter-column line segment is within $\mathbb{O}_i \cap \mathbb{O}_{i+1}$, the corresponding inter-column pixels will be covered by the set of intra-column pixels. As a result, the inter-column lines (see, e.g., line segment L_2 in Fig. 1(d)) are not required for ensuring error-free visualization. However, an inter-column line segment is indispensable, if its $\mathbb{Q}$ is not within $\mathbb{O}_i \cap \mathbb{O}_{i+1}$ (see, e.g., line segment L_1 in Fig. 1(d)), since L_1 can be rasterized into pixels out of the intra-column pixels. Combining all these pixels together, we can then ensure an error-free visualization. For time series with missing data values between columns B_i and $B_{i+\Delta}$, using the inter-column line segment between the two columns can help ensure the generation of error-free visualizations.

We can see that the data points used for defining the line-segment aggregations are the subset of the all line segments formed by connecting all four data points in each group: the data points with the minimal and maximal values and the data points with the first and last timestamps in M4 [13]. Since the min-max aggregation on a large time interval can be decomposed into an aggregation on a few sub-intervals, the pre-computed aggregation results of some time intervals can be used to accelerate the computation. This enables the reuse of the line-segment aggregation results from the previous round of interaction to support interactive visualization.

As such, we propose to build a hierarchical data representation, which brings two benefits: (i) avoid explicit identification of required inter-column line segments during interactive query, and (ii) reuse pre-computed aggregation results of different time intervals to speed up the computation. We detail the design of this new data representation in Section 4.

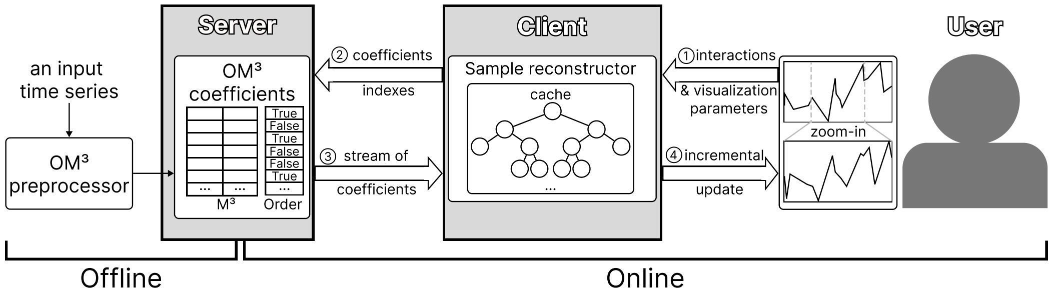


Fig. 2. Our OM^3-based architecture for interactive progressive error-free visualization of time series stored in a database.

4 OUR SOLUTION

Based on the line-segment aggregation described in Section 3, we propose the OM^3 representation. Particularly, we shall show how this representation facilitates interactive progressive visualization of large time series stored in remote databases with low latency. Fig. 2 shows the overall architecture, which consists of two stages: offline preprocessing and online query.

Offline Preprocessing. We apply our OM^3 preprocessor (see Section 4.1) to convert each time series stored in a relational database at a server into our OM^3 coefficients by a *forward transform*. This is a one-time pre-processing conducted locally on the server.

Online Query. Once the OM^3 coefficients are available, users can interactively explore the data on the client by retrieving just the necessary coefficients from the server to reconstruct the original data records via an *inverse transform*. To produce a visualization from scratch, the server can automatically produce query statements to retrieve coefficients based on the display-window width and the visible time range of the time series on the display window. Then, the client receives a stream of coefficients from the server to produce the visualization, while maintaining an OM^3 coefficient tree of the reconstructed data as cache to accelerate subsequent visualization generation during interactive exploration. To support smooth interactions, the server performs incremental queries to request data on demand, enabling the client to smoothly pan, zoom, resize, and query the time series by exploiting the tree hierarchy.

OM^3 inherently requires the time series to be complete (i.e., without missing values) and have a power-of-two length. It also requires the window width to be a power of two. However, none of these can be ensured for general time-series data during the interactive data exploration. Hence, we further extend OM^3 to address these issues. In this section, we will first describe the forward and inverse OM^3 transforms and our extensions for the time series with missing data and arbitrary lengths, incremental tree-based query, acceleration strategies on OM^3, and families of interactions supported by OM^3.

4.1 OM^3 Transforms

Given a time series T of length $n = 2^l$, we obtain its hierarchical line-segment aggregation via two operations: (i) hierarchical min-max aggregation, and (ii) the ordering of the min-max aggregation results at the finest level. Operation (i) helps build the min-max lines, while operation (ii) facilitates the construction of the inter-column lines. We refer to such a line-segment aggregation result as the $\underline{o}$rdered $\underline{m}$ulti-level $\underline{m}$in-$\underline{m}$ax representation of the time series, i.e., OM^3. So, the OM^3 transform converts time series T from an ordered list of data records to a set of coefficients that describe T in different levels of detail. Using these coefficients, the original data records at any level can be reconstructed by an inverse transform.

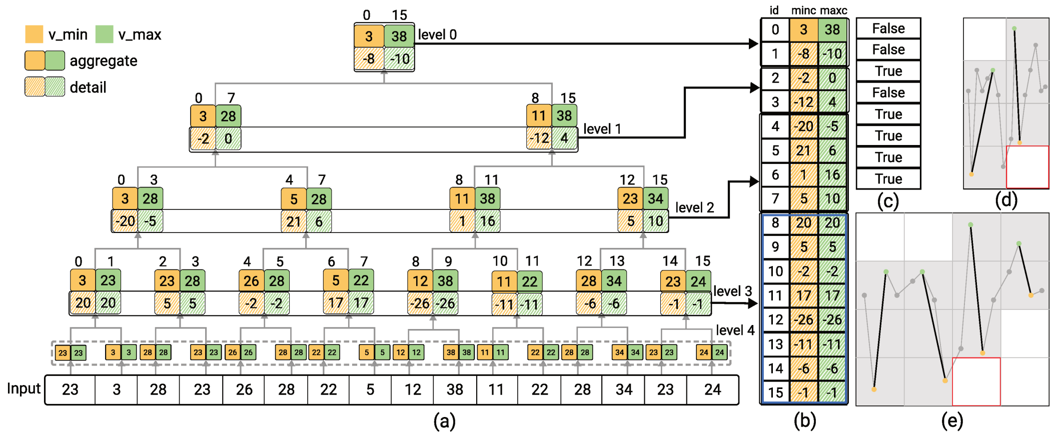


Fig. 3. Illustrating the OM^3 forward transform and the coefficients stored in the database. (a) The input data (bottom) with 16 samples is recursively transformed to build the four-level coefficient tree. Each tree node has two aggregate coefficients and two associated detail coefficients. To start, we replicate the input data twice to construct the coefficients at level 4, marked by the dotted box. (b) The initial database table stores the final two aggregate coefficients at the top (level 0) and all detail coefficients from the forward transforms; detail coefficients that are redundant for reconstructing the original data are marked by the blue box. (c) The additional database table stores all ordering coefficients. (d,e) Line visualizations (in black) reconstructed by the inverse transform from the aggregate coefficients on a two-pixel-wide window (d) and a four-pixel-wide window (e), where the pixels with the red boxes are missed.

Forward Transform. We hierarchically decompose the input time series to transform it into a hierarchy of aggregate and difference coefficients, and a set of ordering coefficients applied to the leaf level. Here, we employ the min-max aggregation functions for preserving the shape of the time series at different levels as well as the ordering relationship between each pair of consecutive data records. However, directly storing min-max aggregates at every node in the hierarchy requires more storage overhead than the original data, resulting in high latency (see Section 5.1). Inspired by Haar wavelet [3], we further apply the difference operations to aggregate values for providing a compact representation (smaller than the original data) yet with sufficient information to reconstruct the original data. In doing so, uniformly-sampled time series with 2^l values becomes a complete binary tree with l levels of coefficients. Each node in the hierarchy has two pairs of aggregate and detail coefficients at level j ($0 \leq j \leq l-1$).

We denote the minimal and maximal aggregate coefficients of the i-th node at j-th level as A^j_{2i}, A^j_{2i+1}, and the corresponding detail coefficients as D^j_{2i} and D^j_{2i+1}. Given an input of 2^l data points, we first replicate each sample twice to form A^l, where $A^l_{2i} = A^l_{2i+1} = v_i$. Then, we can compute two aggregate coefficients A^{j-1} and two detail coefficients D^{j-1} for the i-th ($i \in \{0, 1, \cdots, 2^j - 1\}$) node at level $j-1$ based on the four aggregate coefficients $\{A^j_{4i}, A^j_{4i+1}, A^j_{4i+2}, A^j_{4i+3}\}$ of their two children nodes at level j:

$$A^{j-1}_{2i} = \min(A^j_{4i}, A^j_{4i+2}), \qquad D^{j-1}_{2i} = A^j_{4i} - A^j_{4i+2}, \tag{1}$$

$$A^{j-1}_{2i+1} = \max(A^j_{4i+1}, A^j_{4i+3}), \qquad D^{j-1}_{2i+1} = A^j_{4i+1} - A^j_{4i+3}, \tag{2}$$

where A^{j-1}_{2i} is $\min(v)$; A^{j-1}_{2i+1} is $\max(v)$; and D^{j-1}'s are the differences between corresponding data points. By successively and recursively repeating this process, we can form a hierarchy of aggregate and detail coefficients. The intuition in our formulation is that every chunk of two aggregate coefficients represent $\min(v)$, and $\max(v)$ specifically for a certain time span in the input time series, i.e., one timestamp at level l, two timestamps at level $l-1$, four timestamps at level $l-2$, etc.

Fig. 3(a) shows an example four-level hierarchy produced from 16 data points. We can see that the eight coefficient tuples at level 3 are obtained by aggregating and differing between two adjacent aggregate coefficients at level 4.

Note also that in our implementation, we directly compute A^{l-1} from input data without explicitly replicating the input data points to form A^l. As shown in Fig. 3(a), the coefficients at level 3 can be obtained by operating on the two adjacent data points. Since A^{l-1} encodes only the set of original sample values in the input without information on their order, we further store the ordering relationship between A^{l-1}_{2i} and A^{l-1}_{2i+1} as the ordering coefficient O to facilitate the reconstruction of the input and set O_i to true, if A^{l-1}_{2i} is less than A^{l-1}_{2i+1}, and false, otherwise. Meanwhile, the detail coefficients at level $l-1$ (e.g., D^{l-1}_{2i} and D^{l-1}_{2i+1}) are redundant, as we do not need to reconstruct A^l_{2i} and A^l_{2i+1} during the inverse transform. In Fig. 3(b), the cells highlighted by the blue box mark the redundant detail coefficients. For short, we refer these aggregate and detail coefficients to be stored as the M^3 coefficients, and the combination of M^3 coefficients and ordering coefficients as the OM^3 coefficients. So, only n M^3 coefficients and half of the n Boolean ordering coefficients are indispensable. For efficient storage of the M^3 coefficients in database, we use an ID column to maintain the ordering of the records, see the left column of Fig. 3(b). Since n Booleans (ordering coefficients) take small space, we store them separately. For the input time series of length n, each data point has two values t_i and v_i, and thus the total number of required coefficients is approximately three-quarters of the input data size.

After removing the redundant detail coefficients, we can pack the aggregate and detail coefficients into a database table with two columns (see the example shown in Fig. 3(b)) and the ordering coefficients into a table (see Fig. 3(c)), which stores all the information necessary for reconstructing the input data points.

Inverse Transform. As the forward transform employs min and max aggregations, we need different mechanisms to reconstruct the aggregate coefficients. Given the aggregate and detail coefficients at level $j-1$, the aggregate coefficients at level j can be found by reversing Eqs. (1) & (2). For example, we can reverse Eq. (1) to find A^j_{4i} and A^j_{4i+2} based on the sign of D^{j-1}_{2i}:

$$\begin{aligned} & A^j_{4i} = A^{j-1}_{2i},\ A^j_{4i+2} = A^{j-1}_{2i} - D^{j-1}_{2i} \quad \text{if } D^{j-1}_{2i} < 0 \\ \text{or}\quad & A^j_{4i+2} = A^{j-1}_{2i},\ A^j_{4i} = A^{j-1}_{2i} - D^{j-1}_{2i} \quad \text{otherwise.} \end{aligned} \tag{3}$$

Similarly, we can reverse Eq. (2) to find A^j_{4i+1} and A^j_{4i+3}.

For a display window with power-of-two width $w = 2^j$, the inverse transform requires fetching coefficients whose ID ranges $[0, 2w-1]$ from the M^3 coefficient table and then performing the inverse transform from level 0 to level $j-1$ to reconstruct the original data points. Figs. 3(d,e) show a 16-samples time series reconstructed to fit two- and four-pixel-wide windows, respectively. Yet, the visualization rendered by the reconstructed min-max aggregate in each pixel column misses some pixels; see the ones marked by the red boxes. The reason is that some inter-column pixels cannot be determined using only the min-max aggregation. We will address this issue with our tree-base query in Section 4.2.

Transformation of Non-canonical Data. To handle time series with missing values and/or of non-power-of-two length, which we collectively call non-canonical time series, we may simply fill the missing values with an estimated value (e.g., simple average) or pad it with zeros to extend its length to a power of two. However, value filling and zero padding will influence the calculation of the minimum and maximum values, hindering the OM^3 transforms.

Instead, we represent a non-canonical time series as the one with a power-of-two length (see the bottom in Fig. 4(a)) and pad the null value to the samples without data values. Then we construct a

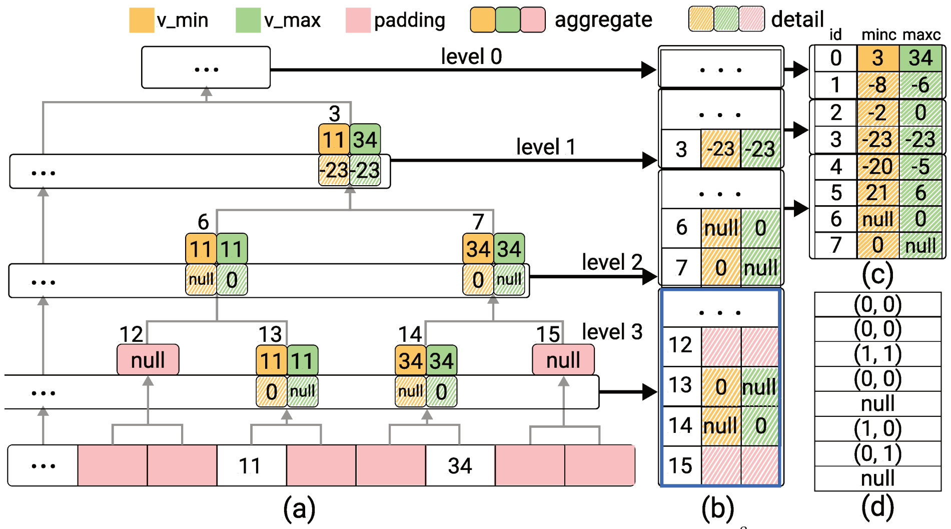


Fig. 4. Handling a ten-samples time series with missing data values. (a) An OM^3 coefficients tree with null nodes (in pink). (b) Coefficients in the three-column format, where the blue box and pink background mark the coefficients that do not need to be stored in the database. (c) The three-column M^3 coefficient table stores all required aggregate and detail coefficients in the database. (d) The ordering coefficient table stores the 2-bit ordering coefficients.

complete binary tree and assign an ID to each node based on its location in the tree (see the number on top of each node in Fig. 4(a)). In the forward transform, we take the nodes corresponding to the samples with missing data values as null. When there is one null child node, we assign its detail coefficients as [0, null] or [null, 0], where the position of the null indicates which (left or right) child node is null. If both children are null, we assume this node is also null and do not store it in the database. So, the M^3 coefficient table may have discontinuous ID column for storing only the coefficients of the data points actually from the input time series.

After performing the forward transform, we store the node index and detail coefficients into the M^3 coefficient table with three columns. As shown in Fig. 4(b), we do not save the redundant detail coefficients shown in the blue box. For the ordering coefficients, we store them as a two-bit ordering coefficient table to handle all missing values related cases. When neither of the paired samples is missing, we set the corresponding ordering coefficient O_i to $(0, 0)$ if A^{l-1}_{2i} is less than A^{l-1}_{2i+1} and $(1, 1)$ if A^{l-1}_{2i} is greater. Otherwise, we set O_i to $(0, 1)$ if A^{l-1}_{2i} is missing and $(1, 0)$ if A^{l-1}_{2i+1} is missing, and null for both are missing. For the example in Fig. 4(a), such coefficients are shown in Fig. 4(d). Since this table takes little space, we store the indices and detail coefficients contiguously in the M^3 coefficient table, which takes only around three-quarters of the input data size. Although this storage overhead is smaller than the input, OM^3 is able to faithfully recover the original input data.

During the inverse transform, we can still visit the children nodes of the i-th node from the coefficients tree using the indices $2i$ and $2i + 1$. When the detail coefficients of the i-th node have a null value, we directly assign its min-max aggregate coefficients to its non-null child node and do not reconstruct the other node. For ensuring an accurate reconstruction of the input samples, we still need to use the coefficients from the ordering coefficient table to identify the samples with missing values at the finest level.

Time Complexity. For a time series of n samples, the time complexity of the forward transform is $O(n)$, while using $O(n)$ space. The inverse transform shares the same complexity.

Algorithm 1 Incremental Tree-based Query

```
Input: Coefficient tree $T$, ordering coefficients $O$, window width $w$, data size $n$
Output: The rasterized result
 1: function BFS($T$, $O$, $w$, $n$)
 2:     $\mathbb{V}$ = nodes of the current level in $T$
 3:     $\mathbb{M}$ = initialize $w$ groups for recording min-max values
 4:     $p$ = the number of current level in $T$, $l$ = ceil($log_2(n)$), $j = p$
 5:     while $j < l$ do
 6:         $\mathbb{B} = \emptyset$    // store boundary nodes at level $j$
 7:         for $v$ in $\mathbb{V}$ do
 8:             $t_{min}, t_{max}$ = ComputeNodeTimeRange($v$.index, $j$, $n$)
 9:             $col_{min}$ = floor($w * t_{min}/n$), $col_{max}$ = floor($w * t_{max}/n$)
10:             if $col_{min} == col_{max}$ then
11:                 update the values in $\mathbb{M}[col_{min}]$
12:             else
13:                 $\mathbb{B}$.append($v$)
14:             end if
15:         end for
16:         if $j == l - 1$ then return Rasterization($\mathbb{M}$, $\mathbb{B}$, $O$)
17:         else if $j > p$ then
18:             for $v$ in $\mathbb{B}$ do
19:                 $t_{min}, t_{max}$ = ComputeNodeTimeRange($v$.index, $j$, $n$)
20:                 $col_{min}$ = floor($w * t_{min}/n$), $col_{max}$ = floor($w * t_{max}/n$)
21:                 if $v$.min ≥ $\mathbb{M}[col_{min}]$.min && $v$.max ≤ $\mathbb{M}[col_{min}]$.max && $v$.min ≥ $\mathbb{M}[col_{max}]$.min && $v$.max ≤ $\mathbb{M}[col_{max}]$.max then    // pruning
22:                     $\mathbb{B}$.remove($v$)
23:                 end if
24:             end for
25:         end if
26:         $\mathbb{C}$ = LoadDetailCoefficients($\mathbb{B}$)    // query from the server
27:         $\mathbb{V}$ = InverseTransform($\mathbb{C}$, $T$)    // reconstruct child nodes
28:         $j$++
29:         yield Rasterization($\mathbb{M}$)
30:     end while
31: end function
```

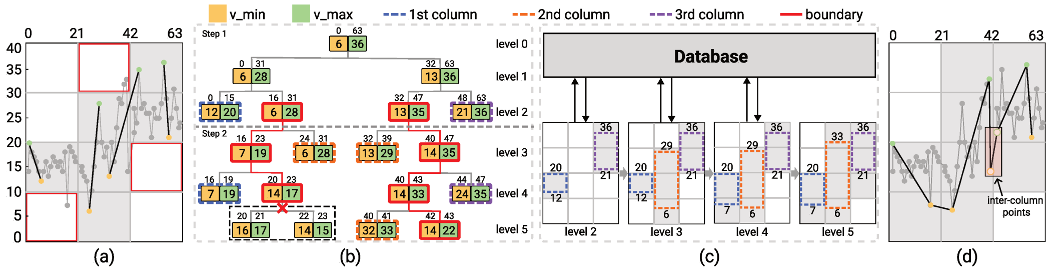


Fig. 5. Illustrating how incremental tree-based query supports for accurately visualizing a time series on a three-pixel-width window. (a) Simply generating the visualization by using four groups of the aggregate values at the level 2 could easily miss pixels (red boxes). (b,c) Our query method with pruning can efficiently visit necessary nodes in the OM^3 coefficient tree to locate the minimum and maximum aggregates for each pixel column in the target window level by level and uses these value ranges to rasterize the pixel columns in (c). (d) The final visualization produced by our method (grey pixels) is error-free; its rasterization precisely matches the data points in the original line chart. Here, the detail coefficients of the node [14,17] do not need to be loaded, since its value range is covered by the value ranges of the adjacent two pixel columns [7,20] and [6,29], while the inter-column line highlighted in (d) corresponds to the node [14,22].

4.2 Incremental Tree-based Query

To adapt OM^3 to arbitrary-sized windows, one simple way is to first reconstruct the min-max aggregate at a level finer than the display window and then render the reconstructed min-max lines to the given window. This approach, however, could miss some pixels in the generated visualizations. Fig. 5(a) illustrates an example three-pixel-width target window, in which we reconstruct four groups of the min-max aggregate values at level 2 and render these min-max lines on this window; we show also the original line chart (grey line) as a reference. Comparing the two charts, we can see that three pixels (red boxes) are missed in the generated visualization, as the four-pixel-width line chart does not provide appropriate data samples precisely for the three-pixel-width window, especially near the pixel-column boundary. Also, when using the correct min-max aggregate coefficients, only some pixels are missed (see Figs. 3(d,e)); yet, we can recover these pixels by some inter-column lines, as explained and shown in Theorem 1 (see Section 3).

To address the issue, we propose a two-step tree-based incremental query scheme for efficiently finding appropriate data points in each pixel column of the target window. Assuming that the width of the display window w is in range $[2^{p-1}, 2^p]$ for some positive integer p, we first retrieve the coefficients for the window of width 2^p from the server database and reconstruct 2^p data points via an inverse transform. Then, the client can successively retrieve detail coefficients of *boundary nodes* that contain the first/last timestamp of each pixel column from the server to reconstruct the original data points at the finest level. Before performing this two-step query, we first send the whole ordering coefficients from the server to the client. By finding the minimum and maximum data values in every pixel column of the target window and the indispensable inter-column samples within a single query, we can then obtain accurate aggregates for producing error-free visualizations. Yet, extensively visiting the coefficient tree all the way to the leaf nodes and transferring all information from server to client would lead to extra costs in computing time and network bandwidth. Also, some inter-column lines are unnecessary as shown by Theorem 1. These observations form the basis for reducing the tree traversal cost.

To avoid exhaustive tree traversal, we introduce a breadth-first search method with an efficient pruning strategy to avoid unnecessary traversal, as outlined in Algorithm 1. When traversing each level, we initialize an empty list $\mathbb{B}$ to store those boundary nodes (line 6) that contain data points near the boundary between adjacent pixel columns. Then, for each node, we examine its time range against the time range of the nearby pixel columns in the target window. If the time range of a node falls entirely inside the time range of a pixel column, the node itself can provide appropriate information for updating the data value range (i.e., [v_min,v_max]) of the pixel column (lines 7-11), so we do not need to further visit its child nodes to explore the finer-level time ranges. Otherwise, we append it to the list $\mathbb{B}$ (line 13). For the boundary node at a level greater than p, we remove it from $\mathbb{B}$, if its data value range is already covered by the current data ranges of the two associated pixel columns (lines 17-21). Once all nodes are checked, we load the coefficients for all remaining nodes in $\mathbb{B}$ and reconstruct the corresponding data points at the next level (lines 25-26). By recursively visiting only necessary child nodes with our pruning strategy level by level, we can efficiently find the data value range of each pixel column in the target window and rasterize all inner-column pixels by the associated pixel column's data value range (line 29). At the last level, we rasterize all inner-column pixels and inter-column pixels defined by the ordered boundary samples collected from $\mathbb{B}$ and O (line 16). In doing so, progressive visualization is achieved, where the visualization can be incrementally refined during the level-by-level traversal (see Fig. 5(d)).

Taking node $[12, 20]$ in Fig. 5(c) as an example. Its time range $[0, 15]$ falls entirely in the first pixel column's time range $[0, 21]$ in the target three-pixel-width window, so we do not need to visit its child nodes and can initialize the data value range (minimum and maximum) of the first pixel column as $[12, 20]$; see the left column of the level 2 result in Fig. 5(d). For node $[6, 28]$, its time range $[16, 31]$ overlaps the first two pixel columns in the target window, so we append it to the boundary node list and visit its child nodes $[7, 19]$ (time range $[16, 23]$) and $[6, 28]$ (time range $[24, 31]$) at level 3. Later, for node $[14, 17]$ at level 4, its value range is covered by the value ranges of the adjacent two pixel columns $[7, 20]$ and $[6, 29]$, so we do not need to visit its child nodes. For some pixel columns, considering only the minimum and maximum values within each pixel column may not be sufficient to accurately rasterize its pixels. For example, the leaf node $[14, 22]$ (time range $[42, 43]$) crosses the second pixel column (time range $[22, 42]$) and the third pixel column (time range $[43, 63]$), forming an indispensable inter-column line (see the highlighted one in Fig. 5(b)). Hence, we put it into the boundary node list and connect its data points to rasterize the bottom gray pixel in the third column of the level 5 result in Fig. 5(d).

Fig. 5(b) shows the exhaustive tree traversal case that loads all relevant coefficients of boundary nodes at one time. Fig. 5(c) shows the nodes in the coefficient tree that we need to visit with our pruning strategy, and Fig. 5(d) shows how the data value ranges and rasterization (grey pixels) are updated progressively for each pixel column. Note that the pruning strategy only searches necessary values, as we described in Theorem 1, and skips inter-column samples whose value ranges covered by both corresponding consecutive groups automatically. Since the value ranges of adjacent pixel columns depend on each other, we sequentially prune nodes and leave the parallel processing as future work.

Time Complexity. Given a window of width w, in the worst case, we need to traverse all the boundary nodes, so the time complexity is $O(w \log(n))$. With our method, a large fraction of queries can be terminated early, irrespective of the length of the time series, so the time complexity is close to $O(w)$. For the case shown in Fig. 5(b), there are 12 nodes from level 3 to level 5. Yet our method only needs to visit 10 of the nodes. Experimentally, we found that the pruning ratio is around or larger than 45% for all tested data (see Section 5.4). For the three-pixel-width window shown in Fig. 5, directly applying M4 would fetch 12 samples for the three pixel columns, whereas our tree-based search would issue three database queries to fetch 20 coefficients from five nodes. Note that as the data gets larger, our method would become faster. It is because our query can be done with a time complexity of $O(w \log(n))$. In contrast, M4 has a time complexity of $O(n)$, and n is typically much larger than w, as a time series could contain millions or even billions of samples.

4.3 Additional Acceleration Strategies

The coefficients are stored in the database table with the schema (id, minc, maxc) as depicted in Fig. 3(b). For a window of width $w \in [2^{p-1}, 2^p]$, where p is a positive integer, we only need to load coefficients up to level $p - 1$ by executing the following query:

```
select minc, maxc where [id>= start_id and index<= end_id].
```

To further reduce the query latency, we introduce the following three additional strategies to accelerate the query.

Coefficients Prefetching. To better support drill-down operations, we can pre-load the coefficients at levels p and $p + 1$. As coefficients are stored contiguously, retrieving them from the database incurs minimal cost. On the other hand, the number of coefficients 2^{p+2} is small compared to n (thanks to the OM^3 transform), so transferring all these coefficients can be done quickly.

Tree Caching. We can cache the reconstructed coefficient tree in the past interactions on the client side at runtime. By storing the tree as a list in the memory, we employ the least recently used (LRU) strategy for the cache replacement. During the interaction, the coefficients at the coarser levels are frequently used but they typically consume a small amount of memory. Hence, a small cache is enough for supporting fluid interaction (see Section 5.2).

Query Merging. During the interaction based on incremental tree-based query, we need to fetch also the information on multiple boundary nodes from the database with the following query:

```
select minc, maxc where [condition1] or ... or [conditionN]
```

where each condition corresponds to the starting and ending indices of a boundary node. To improve the query speed, we merge multiple conditions with continuous records together, e.g., the queries of nodes [6, 28] and [13, 35] at level 2 in Fig. 5(b) can be merged. In doing so, we can reduce the overall query time by 40%.

Furthermore, we use two threads: one for analyzing the coefficients and the other for issuing queries to the database. Once the analysis is done, the queried coefficients at the finer level can be reused for the analysis at the next level efficiently.

4.4 Supported Interactions

OM^3 enables dynamic data fetching on demand for generating error-free visualizations. By storing the coefficients of multiple time series in the same database, we can allow interactive exploration of multiple large-scale time-series data with smooth interactions by concurrent query processing. Below, we introduce two families of common interactions that can be supported effectively by OM^3.

Resizing, Panning, and Zooming. One common feature of these interactions is that they all involve a new mapping between the time range of the data and the width of the display window. For instance, resizing shows the same time series on a window of changing sizes (see the example in Fig. 5), whereas panning and zooming show a different time range of the time series on the same window.

For a time series shown in a window of width $w \in (2^{p-1}, 2^p]$, we can efficiently zoom into a time interval $[t_a, t_b]$ in four steps:

(1) compute node indices μ_a and μ_b associated with time stamps t_a and t_b, respectively, at level p;
(2) find minimal extra level ϵ such that $|\mu_b - \mu_a + 1|2^\epsilon \geq w$;
(3) fetch the coefficients of all nodes within index range $[\mu_a, \mu_b]$ and their children from level $p+1$ up to level $p+\epsilon$; and
(4) perform the tree-based search to find the boundary data points.

With our incremental tree-based search, we can perform this process efficiently. For zooming in and out with a scaling factor of two, some boundary data points in the current view can be reused for the next view, thus further reducing the query cost. For a time series shown on a window of width w in $[2^{p-1}, 2^p]$ and current time interval $[t_a, t_b]$, panning into a potentially overlapping interval $[t_c, t_d](t_b - t_a = t_d - t_c)$ amounts to steps (1) and (4) for zooming.

Timebox Query. Another family of common window query interactions is timebox query [11], in which users can select lines covered by a user-specified range or by a given rectangle. For multiple time series shown on the client visualization, timebox query retrieves a subset of lines that satisfy the user-specified constraints and highlights them in the current display window. However, to handle a large number of lines, the retrieval process proposed in [11] is inefficient. Also, users cannot explore details of the queried lines. Thanks to the OM^3 coefficient tree, we can further accelerate the retrieval process and support zooming in and out of the queried lines.

Given a box constraint with a certain value range and time range, users want to find the set of time series whose data values are entirely covered by the box in the box's time range. As each node in the coefficient tree stores the minimum and maximum data values for the time range covered by the node, we can efficiently perform a BFS search to find the nodes whose time range is associated with the box's time range. There are three cases in the BFS search:

- Case (i): no intersection between the value range of the node and the value range of the box;
- Case (ii): the value range of the node is fully covered by the value range of the box; and
- Case (iii): the value range of the node is partially covered by the value range of the box, so we need to further explore the child nodes to check if it is Case (i) or Case (ii).

Since this query is based on the given visualization, its processing does not require communication with the server, so it can run very fast solely on the client side. Once the queried lines are available, users are allowed to further interact with the queried lines by using the panning, zooming and resizing operations.

5 EVALUATION

We developed our system in a client-server architecture. The client was implemented in JavaScript running on an 8-core 3.2GHz MacBook Pro with 32GB RAM, whereas the server was implemented using the PostgreSQL database (v14.2) and InfluxDB (v2.0) running on an 8-core Intel Xeon Platinum 8269CY CPU with 2.5 GHz, 32GB RAM, and 250GB SSD with Ubuntu 22.04 in the Alibaba Cloud. We chose PostgreSQL because it was used for implementing the original M4 [13]. As shown in Fig. 2, the system can be configured with the coefficients of any reversible multi-level transform, and thus we compare the OM^3-based system with the one configured by Haar wavelet [3], which applies the average function to compute the aggregate coefficients. Since the OM^3 representation is slightly different for time series with missing values, our implementation maintains different functions for preprocessing and reconstructing the time series without and with missing values. We performed the corresponding forward transforms in the server and stored the result coefficients in the PostgreSQL database. Besides comparing with the PostgreSQL-based M4 implementation that can generate error-free time-series visualizations, we further developed a version of M4 in InfluxDB, which is a leading time-series database [22] optimized for storing and querying time-series data. Since InfluxDB requires the data with the column of timestamps, it cannot be used for storing Haar wavelet and OM^3 coefficients.

To show that OM^3 enables error-free time-series visualizations with low latency, we performed three quantitative comparisons with state-of-the-art methods in static, interactive, and progressive settings (Sections 5.1, 5.2, and 5.3) and a qualitative case study (Section 5.5). Since OM^3 needs an index while M4 runs immediately, the static setting reports the comparison between M4 and OM3's first queries (with index cost), while the interactive setting focuses on the comparison on the subsequent queries. To study the influence of the network transmission cost, we further conducted the same evaluation in an alternative setting with databases residing in the same local area network as the client, and found that the only difference from the cloud setting is the round-trip time in the network (2 ms vs. 12 ms). Finally, we conducted an ablation study on OM^3 to learn the effect of different acceleration strategies on tree-based query and a case study to demonstrate its effectiveness for interactive exploration. Hence, we only show the evaluation results for the cloud setting here; the full evaluation results, including the screenshots and the corresponding scores of different measures on each dataset, can be found in the supplemental material.

5.1 Static Visualization

We compare line charts generated by various methods on the same data in terms of visual quality, data reduction ratio, and runtime.

Procedure. For each dataset, we store it (or its coefficients) in the server database. Then, the client requests the server to send the data for generating line charts of six different window widths (200, 400, 600, 800, 1000, 1200) and same window height (600).

Methods. Six methods are included for comparison: two versions of M4 [13], Haar wavelet [30], our method based on the multi-level min-max representation, and our OM^3 with and without the pruning strategy shown in Algorithm 1. Note that we implemented two versions of M4, namely M4-P based on PostgreSQL and M4-I based on InfluxDB. Also, we refer OM^3 based on the multi-level min-max representation as M^3 and OM^3 without pruning as OM^3-NP for short. M^3 and OM^3 use the same pruning-based incremental query algorithm with the only difference being the coefficients stored at each node. Since the query results of InfluxDB always contain the timestamp column, we implement M4 by only using the four extreme operators without the join operation required by the

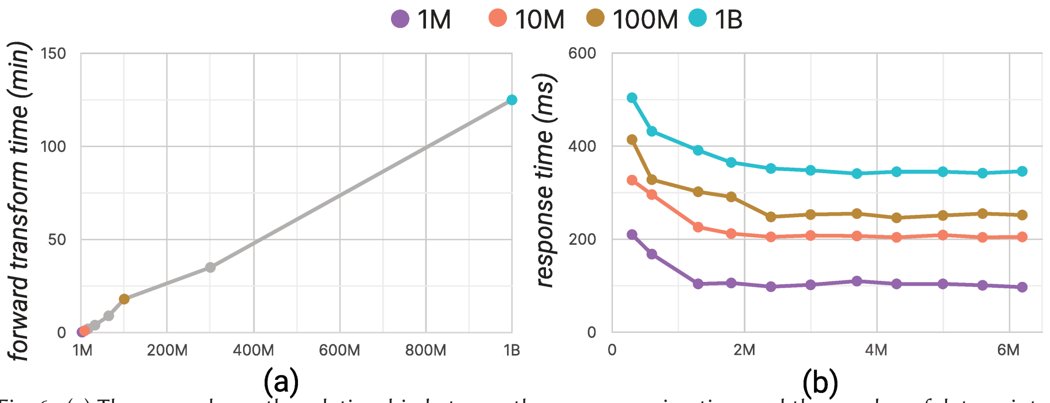


Fig. 6. (a) The curve shows the relationship between the pre-processing time and the number of data points. (b) The curves show how different cache sizes influence the response time for four datasets of different sizes.

M4 in PostgreSQL. The detailed database query statements for both methods can be found in the supplemental material. For Haar wavelet, as it only works for display windows of power-of-two widths, we fetch coefficients at a level finer than the window width, reconstruct the chart, and then compress it to fit the window.

Datasets. For a comprehensive evaluation, we employed 13 datasets of varied sizes and distributions, having 1 million to 1.1 billion data records. Among them, three are real datasets (stock price, ball speed, and electrical power, with 8M, 7M, and 32M data records, respectively) employed in the evaluation of M4 [13]. The other 10 are synthetic datasets generated with three components: $f_t = \mathbf{T}_t + \alpha \mathbf{S}_t + \beta \mathbf{W}_t$, where f_t is the data value at time step t, $\mathbf{T}_t$ is the major trend component, $\mathbf{S}_t$ is defined by the sine function, $\mathbf{W}_t$ is white noise, and α, β are parameters. The trend component $\mathbf{T}_t$ randomly selects a trend distribution from a given list of three trend distributions, i.e., constant, linear, and quadratic functions, and white noise $\mathbf{W}_t$ is randomly generated.

Measures. We quantitatively evaluate the quality and performance of visualization generation using the following four measures.

- *Visual quality.* We follow M4 [13] to use *structural similarity index* (SSIM) [31] to measure the similarity between line visualizations generated by the five tested methods and directly yielded from the original data. The final score is normalized to range [0, 1]; a larger value indicates better quality.
- *Data reduction ratio* is the percentage of data (over the size of the whole dataset) that is reduced by different methods, while the rest is transferred from server to client for generating the line chart. A higher value is better, indicating a smaller portion of data that has to be transferred.
- *Query time* is the total query execution time on the server.
- *Response time* measures the total time from issuing the query to rendering the line chart on the client.

Pre-processing. Fig. 6(a) shows the pre-processing time of OM^3 for all tested datasets. We can see that the runtime is almost proportional to the data size. The dataset with 1M records requires only 7s. For 1M, 100M, and 1B data records, the storage requirements for the original data (and corresponding OM^3 coefficients) are 43M (25M), 5.54G (3.26G), and 44.3G (26.1G), respectively. So, OM^3 coefficients require only 60% the storage of the input. In contrast, those of M^3 coefficients are 55.2M, 6.5G and 52.2G, almost twice those of OM^3.

Results. Since generating line charts of different window widths requires different number of data points, we analyze the results separately for different window widths. We found results of different

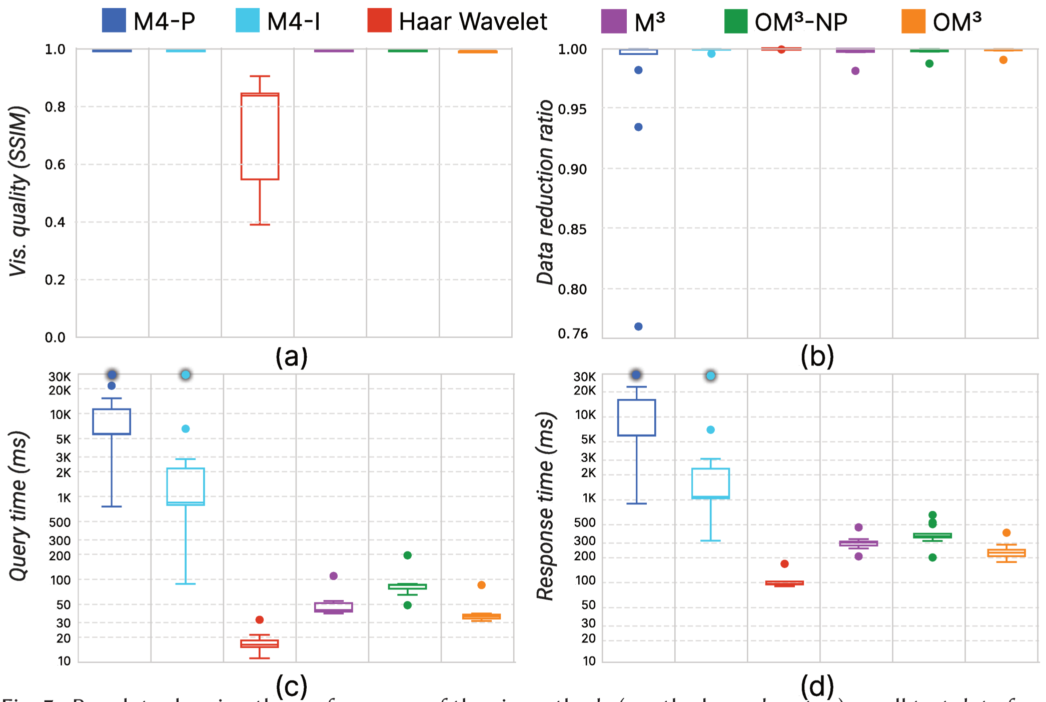


Fig. 7. Boxplots showing the performance of the six methods (see the legend on top) on all test data for a window of width 1000 in terms of the four measures: (a) SSIM; (b) data reduction ratio; (c) query time; and (d) response time. For the outliers with values out of the plot range (see, e.g., query time), we indicate them by a dark transparent shadow.

window widths have similar summary statistics on the four measures, so we only show those for window width 1000 in Fig. 7. For the complete results, please refer to the supplemental material.

We have four observations from Fig. 7. First, M4-P, M4-I, M^3, OM^3, and OM^3-NP all produce error-free visualizations, whereas Haar wavelet cannot preserve the original shape of the time series. Second, all methods have high data reduction ratios larger than 0.98 on average, while M4-P performs relatively poorly with a data reduction ratio of only around 0.77 for some data caused by the join operation. Third, Haar wavelet, M^3, and OM^3 are all able to support interactive exploration with the average response time around 100 ms, 280 ms, and 210 ms, respectively. In contrast, the average response time of M4-P is larger than 500 ms for all the test data, while the one of M4-I is more than 1 minute for 16M data and it quickly grows as the dataset size increases. For the query time, Haar wavelet, M^3, OM^3, and OM^3-NP all take less than 100 ms, whereas the other methods take more than 1 second for most data. Fourth, M^3 requires more coefficients to load and transfer, resulting in a slightly smaller data reduction ratio than OM^3 (0.996 vs. 0.998) and higher query time (45 ms vs. 30 ms) and response time (280 ms vs. 210 ms) on average. Last, our pruning strategy helps avoid around half of the data points in the OM^3 search, contributing to a reduction of query time by around 200 ms.

After carefully checking the outliers in Fig. 7 (b), we find that the lowest data reduction ratios of M^3, OM^3, and OM^3-NP are caused by the smallest synthetic data with 1M samples. The reason is that the number of data points required by M^3 and OM^3 is almost constant for the same window width, regardless of the data size, while the other datasets have at least 8M records and the number of coefficients loaded by OM^{3}is only around 20K due to the tree-based search. In contrast, the lowest data reduction ratios of M4-P are caused by the three real datasets. These datasets have

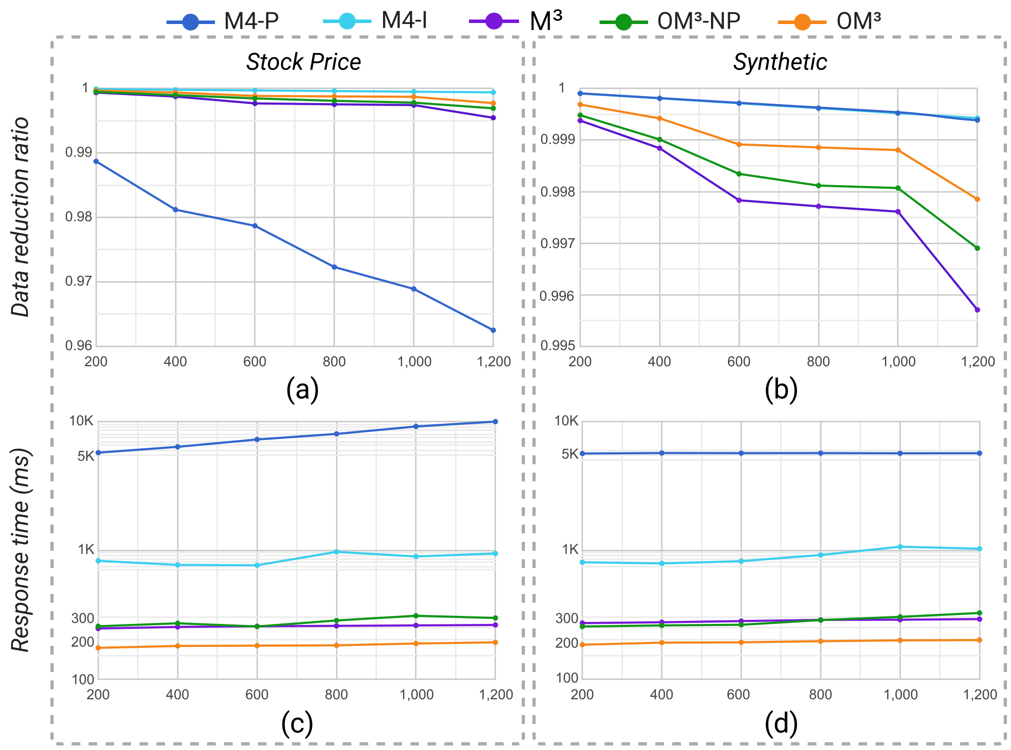


Fig. 8. Plots showing how data reduction ratio (a,b) and response time (c,d) of the five methods (M4-P, M4-I, M^3, OM^3, and OM^3-NP) vary with the window width (along the horizontal axis). (a,c) Results on a real dataset of non-power-of-two length; and (b,d) results on a synthetic dataset of power-of-two length. Both datasets have around 8M data records.

multiple copies of minima and maxima in each pixel column, where M4-P needs to find all these duplicates from the server. In all methods, the maximum query and response times are caused by the dataset with ~1.1 billion records. While our OM^3 takes only ~380 ms, OM^3-NP, M4-I, and M4-P require 800 ms, 9.1 min, and 13.2 min, respectively. So, OM^3 is around 1000 times faster than the state-of-the-art method M4 for error-free line plotting on the largest dataset.

To learn how the measures vary with window widths, we further explore two typical datasets, both with ~8M records. The dataset used in Figs. 8(a,c) is a real-world dataset with missing values, while the one in Figs. 8(b,d) is a complete synthetic dataset with a power-of-two length. Here, we exclude Haar wavelet, as it cannot produce error-free visualizations. We can see that the data reduction ratio of all methods decreases as the window width increases, as more data points are needed for rendering more pixel columns. Among these methods, the decreasing rate of M4-I is the smallest and the ones of M4-P and M^3 on the stock price and synthetic dataset are the largest, respectively, while the ones of OM^3 and OM^3-NP are in-between them. Yet, the data reduction ratio of OM^3 for both datasets is always larger than 0.99. Compared to OM^3-NP, OM^3 takes only half amount of data points for generating the same visualization.

Regarding the response time, the observations are consistent with those shown in Fig. 7. The latency of OM^3 is consistently less than 200 ms for both datasets as the window width increases, those of M^3 and OM^3-NP are both around 300 ms, whereas those of M4-I and M4-P are around 1 s and 5 s, respectively.

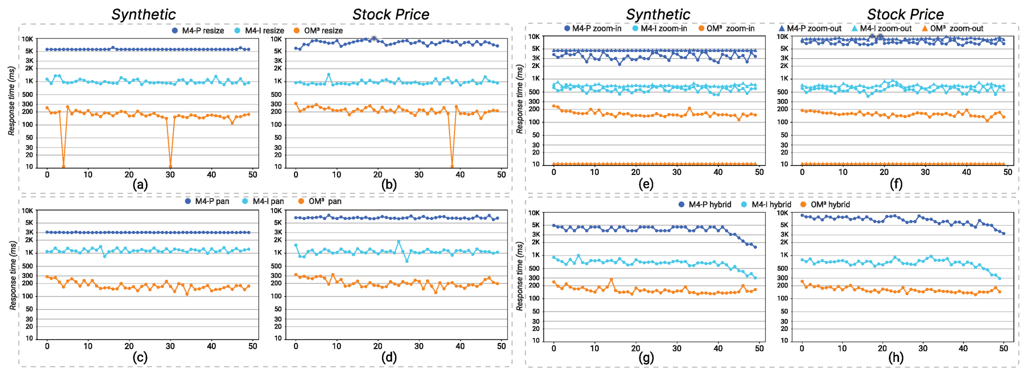


Fig. 9. The curves show how the response time varies within 50 trials of applying four types of interactions: resizing (a,b), panning (c,d), zooming (e,f), and hybrid of resizing, panning, and zooming (g,h) to two datasets. (a,c,e,g) Results on a synthetic data and (b,d,f,h) results on a real data.

Overall, these quantitative results show that our OM^3 is able to produce error-free static visualizations in less than 300 ms for datasets ranged from millions to a billion of records, which is clearly not achievable by the state-of-the-art method, M4.

5.2 Interactive Visualization

Since OM^3 and M4 are error-free (above), we now study latency.

Data. As OM^3 and M4 show different performance on synthetic and real data, we further consider the following two datasets (see also Fig. 8): a real stock price dataset with 8M records and a synthetic dataset with 8M records, which have very different characteristics. We render both data on the same window width 1000.

Procedure. The 4 interactions below are run 50× each:

- *Resize* the display width to $w \in [500, 1200]$, uniformly drawn.
- *Pan* the time range to random offset. Fixed range size $\lfloor n/2 \rfloor$ and display width of 1000.
- *Zoom* from a random initial scale to random zoomed-in scale. After each zoom-in, we zoom-out to return to the initial scale.
- *Hybrid* makes 50 sequential interactions: first zoom-in, then randomly pick from resize, pan, zoom-in/out as above. Zoom-out returns to the previous rather than initial scale.

Each interaction is immediately applied after the previous finishes.

Measures. We measure each interaction's latency from initiation to the client obtaining the updated chart. We don't include time to render the initial static chart since it was studied in Section 5.1.

Cache Size. To explore how the cache size influences the response time so that we can set it properly, we applied OM^3 to four synthetic datasets (1M, 10M, 100M, and 1 billion points) with varying cache sizes. For each setting, we performed 100 interactions (resizing, panning, and zooming) with random parameters and the average response time. From Fig. 6(b), we can see that the response time rapidly decreases as the cache size increases and then plateaus after a certain cache size. So, we empirically set the cache size to 3MB.

Results. Due to the space limit, we only show the response time of 50 trials of the resizing, panning and zooming-in/out interactions in Fig. 9 and refer readers to the supplemental material for the query time and data reduction ratio. From Fig. 9, we have three observations. First, OM^3 is always significantly faster than M4. Its latency for all interactions is always <300 ms on average for both datasets. In contrast, the response times of M4-I and M4-P are at least 500 ms and 2 seconds for both datasets, respectively. Second, the response time of OM^3 for all interactions is slightly longer at the beginning and then remains stable. The reason is that OM^3 dynamically updates the coefficient tree on the client side and reuses the stored coefficients for the follow-up interactions, rather than performing each query from scratch, like in M4. Thus, even with a small cache, resizing can be done in 10 ms (see the three extreme points in Figs. 9(a,b)) for the changed window widths whose corresponding coefficients have been cached, and our zooming-out interaction takes only 12 ms on average (see Figs. 9(e,f)). Most panning, resizing, zooming-in, and the hybrid interactions take <250 ms on average; thanks to our efficient query strategy. In contrast, the zooming-out and zooming-in operations with M4-P take around 4 seconds and over 6 seconds for the synthetic and stock price datasets, respectively, while both operations with M4-I take similar time (1 second) for both datasets. This is as expected, since M4-P requires querying more records in the database, while InfluxDB is optimized for range queries over the time-series data [26]. Last, M4-P is significantly slower on the stock price data than the synthetic data because this data has more duplicate values and the joint operation in M4-P causes more redundant data points. In contrast, our method and M4-I perform similarly on both datasets.

5.3 Interactive Progressive Visualization

The last experiment focuses on exploring the effectiveness of OM^3-empowered progressive visualization of very large data. We conducted the evaluation on two large datasets: electrical power with 30M records and a synthetic data with 1B (2^{30}) records. After visualizing the data on a 1000×600 window, we change this view by applying each of the three types of interactions (resizing, panning, and zooming-in) 20 times with different parameters. We did not test zooming-out, as it only takes around 15 ms for any data (see Figs. 9(e,f)) by reusing the cached coefficients. Once an interaction begins, we take the final visualization as the reference to measure the SSIM value and the response time for each intermediate result.

Parameters. For resizing and zooming-in, the starting view shows the whole time range. The widths of the resized windows after 20 successive operations are 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 600, 1200, 1800, 2400, 3000, 3600, 4200, 4800, 5400, and 6000, while the zoomed relative time range is generated by $[s\%, s\% + 100\% - 4\% * r]$, where s is a random start position and r is a unique integer in [1,20]. For panning, we only show the data within the first 50% time range in the display window and then pan the view to time range $[2r\%, (50 + 2r)\%]$, where r is a unique integer in [1,20].

Results. To show that OM^3 can generate meaningful intermediate results at interactive speeds, we run 20 trials for each interaction according to the above parameters. For each trial, we measured the response time and the quality of the visualization generated by the results of tree-based query at each level (in terms of SSIM). The boxplots in Figs. 10(a,b) show that the response time to achieve 95% SSIM accuracy is less than 100 ms for panning and zooming-in with all parameters. Regarding resizing, the response time varies significantly with different parameters, but the largest time duration is still under 300 ms with window width 6000 for both data. To explore how the visualization quality improves over time, the line charts in Figs. 10(c,d) plot the curves between the SSIM values and the response time of a randomly-selected trial. We can see that the SSIM value quickly increases and reaches 0.95 within 75 ms, and then grows slowly to generate error-free visualizations. Such results indicate that our approach makes a trade-off between interactivity (<70-100ms response) and progressive results (SSIM $\geq$ 0.95).

5.4 Ablation Study

We conduct an ablation study on OM^3 to examine the effect of the four major acceleration strategies: node pruning, coefficients prefetching, tree caching, and query merging. As such, we tested the

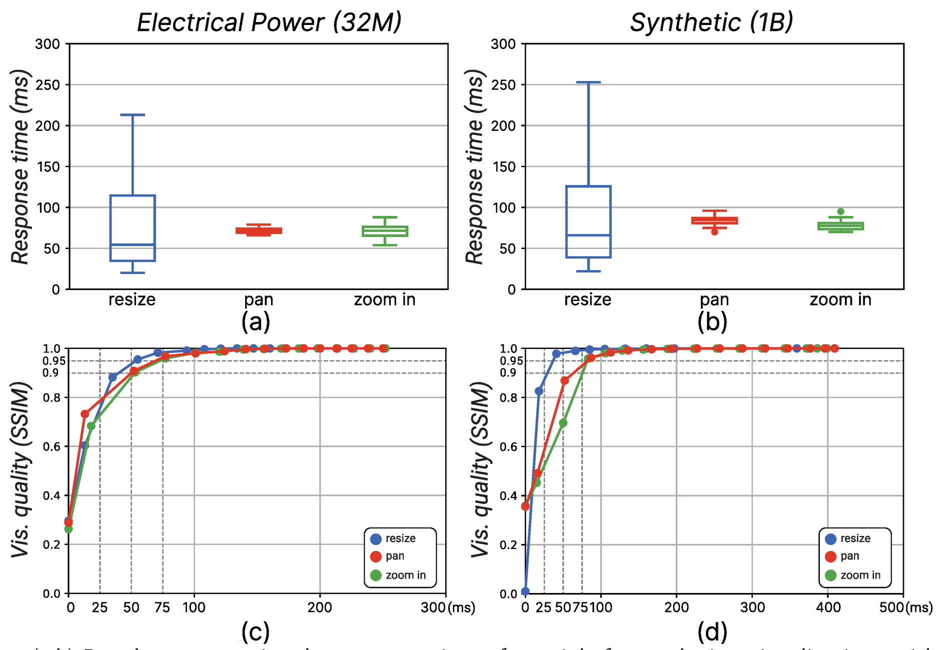
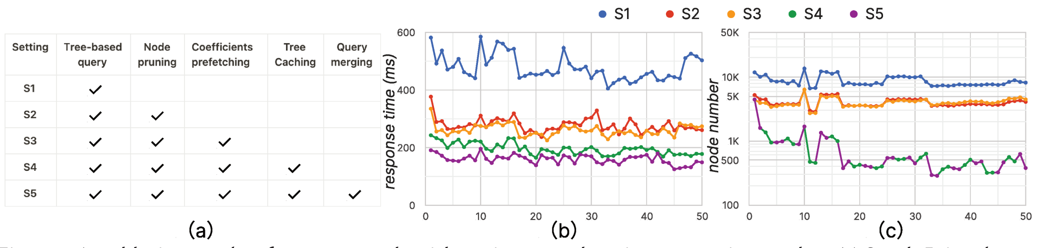


Fig. 10. (a,b) Boxplots summarize the response time of 20 trials for producing visualizations with SSIM values larger than 0.95 after performing three interactions on the two datasets. (c,d) The line charts show how SSIM-based visualization quality evolve over the response time for a randomly-selected trial for each interaction on the two datasets.

five settings shown in Fig. 11(a), where tree-based query without any acceleration is the baseline. For each setting, we perform hybrid interactions 50 times as described in Section 5.2. As all the strategies do not influence the visualization quality, we only measure the response time and number of nodes accessed for each interaction.



Setting	Tree-based query	Node pruning	Coefficients prefetching	Tree Caching	Query merging
S1	✓				
S2	✓	✓			
S3	✓	✓	✓		
S4	✓	✓	✓	✓	
S5	✓	✓	✓	✓	✓

Fig. 11. An ablation study of our approach with various acceleration strategies on the 8M Stock Price dataset by performing hybrid interactions 50 times. (a) Five studied settings, (b,c) the curves show the response time and the number of nodes accessed evolve over 50 interactions.

As Fig. 11(b) shows, directly performing tree-based queries requires at least around 450 ms for each interaction and accessing around 9K nodes. The node pruning strategy helps reduce the response time and number of accessed nodes to <300 ms and 5K for most interactions, bringing 1.71x speedup on average and pruning around 45% nodes in the query. Coefficients prefetching leads only to slight changes to the query performance (from 278 ms to 260 ms on average) and number of nodes accessed except for the first several interactions. This is reasonable, since prefetching only takes effect for the first query when constructing the static visualizations, so it might not be very helpful for the later interactions. Next, tree caching further helps reduce the response time and number of accessed nodes, from 260 ms and 4k nodes to 195 ms and 0.7k nodes on average, respectively. So, it brings 1.33x speedup and reduces the number of nodes accessed to 18%. Query merging barely changes the number of nodes accessed but it further helps reduce the response time from 195 ms to 160 ms on average, a 1.22x speedup. Combining all these acceleration strategies together brings around 3.0x speedup and reduces the number of nodes accessed to only 8%.

5.5 Case Study

We ran a case study using the Kaggle [25] stock market data: per-minute prices of 44 stocks from 8 sectors, from 2015 to 2022. To analyze the relative change, we normalized each stock's time series to the starting day price. We forward OM^3 transform each stock and store its coefficients in a separate table. Then, we explore via a JavaScript-based browser application.

We load the data into a 700×600 display window and generate the visualization shown in Fig. 12(a). We can see that almost all stock prices decreased largely around 2020 and speculate that COVID-19 was the cause. However, it is difficult to see the details due to the visual clutter. By creating a timebox covering a few stocks with similar prices in the range [0, 70%], we can clearly observe the minimum and maximum values of a few stocks from the information technology, financial, and energy sectors. To explore the details, we zoom into the time interval from Jan. 2019 to Feb. 2022 and further apply a time box. The visualization in Fig. 12(c) shows that the stocks from the information technology sector show the best recovery and their prices even exceed the peaks before COVID-19, whereas those from the financial sector grow the slowest and fail to recover to the prices before COVID-19. Since the response time of the timebox query and the zoom-in operation (1 ms and 286 ms) is below the required limit of interactive latency 500 ms [17], we can smoothly explore how the stock prices develop for varying time intervals and compare them on demand at interactive speed.

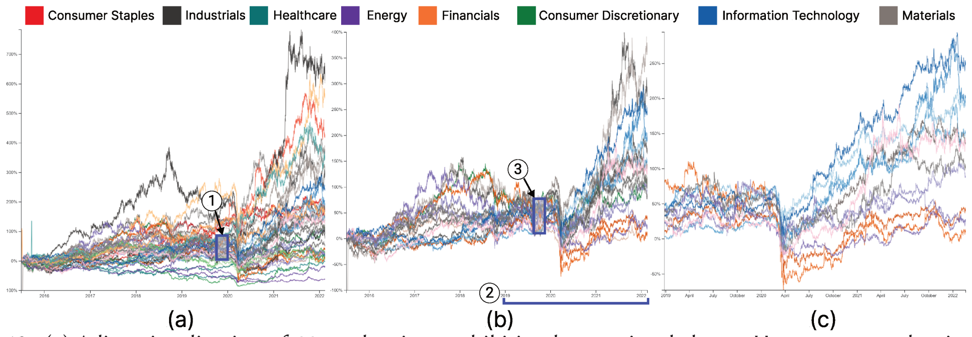


Fig. 12. (a) A line visualization of 44 stock prices, exhibiting heavy visual clutter. Hence, we put the timebox marked by the blue box to filter out the stocks with different behaviors and produce the visualization shown in (b). (c) Further, we zoom into a specific time interval between 2019 and 2022 and then use a timebox to further filter the stocks.

5.6 Limitations

OM^3 does not yet maintain coefficients under streaming data nor updates. Second, OM^3 focuses on data whose sampling rate (x-axis) is uniform. The coefficients could potentially encode timestamps to support non-uniform sampling. Third, error-free visualization might not be necessary nor desired for noisy data, particularly when exploring trends [23] rather than details. One possibility is super-sampling to anti-alias the line charts, studying their effects on users for different visual analysis tasks.

6 CONCLUSIONS AND FUTURE WORK

OM^3 is a novel multi-level encoding for progressive error-free interactive visualizations of large time-series. We revisit min-max time series aggregation and prove that a line-segment aggregation produces error-free line charts. We thus formulate the forward and inverse OM^3 transforms to convert between a time series and a hierarchy of coefficients. This supports progressive error-free interactive visualization at any scale. We then extend OM^3 to time-series with missing values or non-power-of-two lengths. OM^3 only needs $\approx 3/4$ space of the original time series.

To adopt OM^3 for smooth interactions, we design an incremental tree-based query method with time complexity $O(w \log(n))$ (for display width w and time-series length n), and an efficient pruning strategy. We combine prefetching, caching, and merging query ranges to further reduce latency. These efficiently support the common time-series interactions, including resize, pan, and zoom. Both quantitative and qualitative evaluations show how OM^3 efficiently helps users explore billion-record datasets on remote cloud databases with $\approx 300ms$ latencies.

Future work can further reduce query times by adding lossless compression [24] and using a predictive prefetching framework [19]. When applying OM^3 to charts with multivariate time series, we can leverage line-based density representations, e.g., [34], to manage heavy visual clutter. We can also incorporate other types of visualizations such as trendlines [23] to better support interactive exploration, since OM^3 faithfully recovers the original input data. We can also extend to streaming data via partial updates to the coefficient tree; most updates are likely local and do not change the min/max aggregates in higher levels of the tree (which would propagate to their descendants).

7 ACKNOWLEDGMENTS

The authors would like to thank the fruitful discussions with Dominik Moritz and André Kohn. This work was supported by the grants of the National Key R&D Program of China (2022ZD0160805), NSFC (No. 62132017, 62141217), NSERC, and NSF (No. 1845638, 2008295, 2106197, 2103794), as well as Amazon and Adobe.

REFERENCES

[1] Wolfgang Aigner, Silvia Miksch, Heidrun Schumann, and Christian Tominski. 2011. *Visualization of Time-Oriented Data.* Springer Science & Business Media.

[2] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate Query Processing Using Wavelets. *The VLDB Journal* 10, 2 (01 Sep 2001), 199–223.

[3] Chang Chiann and Pedro A. Morettin. 1998. A Wavelet Analysis for Time Series. *Journal of Nonparametric Statistics* 10, 1 (1998), 1–46.

[4] Tak chung Fu. 2011. A Review on Time Series Data Mining. *Engineering Applications of Artificial Intelligence* 24, 1 (2011), 164–181.

[5] Tak chung Fu, Fu lai Chung, Robert Luk, and Chak man Ng. 2008. Representing Financial Time Series Based on Data Point Importance. *Engineering Applications of Artificial Intelligence* 21, 2 (2008), 277–300.

[6] Graham Cormode, Minos Garofalakis, and Dimitris Sacharidis. 2006. Fast Approximate Wavelet Tracking on Streams. In *Advances in Database Technology - EDBT 2006.* Springer Berlin Heidelberg, Berlin, Heidelberg, 4–22.

[7] David H. Douglas and Thomas K. Peucker. 1973. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *Cartographica: the international journal for geographic information and geovisualization* 10, 2 (1973), 112–122.

[8] Danyel Fisher, Igor Popov, Steven Drucker, and MC Schraefel. 2012. Trust me, I'm partially right: incremental visualization lets analysts explore large datasets faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* 1673–1682.

[9] John E. Hershberger and Jack Snoeyink. 1992. *Speeding Up the Douglas-Peucker Line-Simplification Algorithm.* Technical Report. University of British Columbia, 1992.

[10] Harry Hochheiser and Ben Shneiderman. 2003. *Interactive Graphical Querying of Time Series and Linear Sequence Data Sets.* Ph. D. Dissertation. USA. AAI3094494.

[11] Harry Hochheiser and Ben Shneiderman. 2004. Dynamic Query Tools for Time Series Data Sets: Timebox Widgets for Interactive Exploration. *Information Visualization* 3, 1 (2004), 1–18.

[12] Waqas Javed and Niklas Elmqvist. 2010. Stack Zooming for Multi-Focus Interaction in Time-Series Data Visualization. In *2010 IEEE Pacific Visualization Symposium (PacificVis).* 33–40.

[13] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. 2014. M4: A Visualization-Oriented Time Series Data Aggregation. *Proc. VLDB Endow.* 7, 10, 797–808.

[14] Eamonn J. Keogh and Michael J. Pazzani. 2000. A Simple Dimensionality Reduction Technique for Fast Similarity Search in Large Time Series Databases. In *Knowledge Discovery and Data Mining. Current Issues and New Applications*, Takao Terano, Huan Liu, and Arbee L. P. Chen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–133.

[15] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. 2015. Rapid sampling for visualizations with ordering guarantees. In *Proceedings of the vldb endowment international conference on very large data bases*, Vol. 8. NIH Public Access, 521.

[16] Robert Kincaid. 2010. SignalLens: Focus+Context Applied to Electronic Time Series. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (2010), 900–907.

[17] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2122–2131.

[18] Miro Mannino and Azza Abouzied. 2018. Expressive Time Series Querying with Hand-Drawn Scale-Free Sketches. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems.* Association for Computing Machinery, New York, NY, USA, 1–13.

[19] Haneen Mohammed. 2020. Continuous prefetch for interactive data applications. In *SIGMOD.* 2841–2843.

[20] Bruce Momjian. 2001. *PostgreSQL: Introduction and Concepts.* Vol. 192. Addison-Wesley New York.

[21] Prithiviraj K. Muthumanickam, Katerina Vrotsou, Matthew Cooper, and Jimmy Johansson. 2016. Shape Grammar Extraction for Efficient Query-by-Sketch Pattern Matching in Long Time Series. IEEE, 121–130.

[22] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time Series Databases and InfluxDB. *Studienarbeit, Université Libre de Bruxelles* 12 (2017).

[23] Sajjadur Rahman, Maryam Aliakbarpour, Ha Kyung Kong, Eric Blais, Karrie Karahalios, Aditya Parameswaran, and Ronitt Rubinfield. 2017. I've seen "enough" incrementally improving visualizations to support rapid decision making. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1262–1273.

[24] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. 2006. Fast Lossless Compression of Scientific Floating-Point Data. In *Data Compression Conference (DCC'06).* IEEE, 133–142.

[25] Debashis Sahoo. 2022. Stock Market Data. https://www.kaggle.com/datasets/debashis74017/stock-market-data-nifty-50-stocks-1-min-data

[26] John Shahid. 2019. InfluxDB documentation.

[27] Tarique Siddiqui, Albert Kim, John Lee, Karrie Karahalios, and Aditya Parameswaran. 2016. Effortless Data Exploration with zenvisage: An Expressive and Interactive Visual Analytics System. *Proceedings of the VLDB Endowment* 10, 4 (2016).

[28] Tarique Siddiqui, Paul Luh, Zesheng Wang, Karrie Karahalios, and Aditya Parameswaran. 2020. ShapeSearch: A Flexible and Efficient System for Shape-based Exploration of Trendlines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 51–65.

[29] James Walker, Rita Borgo, and Mark W. Jones. 2016. TimeNotes: A Study on Effective Chart Visualization and Interaction Techniques for Time-Series Data. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 549–558.

[30] David F. Walnut. 2002. *An Introduction to Wavelet Analysis.* Springer Science & Business Media.

[31] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612.

[32] Martin Wattenberg. 2001. Sketching a Graph to Query a Time-Series Database. In *CHI '01 Extended Abstracts on Human Factors in Computing Systems* (Seattle, Washington) *(CHI EA '01).* Association for Computing Machinery, New York, NY, USA, 381–382.

[33] Emanuel Zgraggen, Alex Galakatos, Andrew Crotty, Jean-Daniel Fekete, and Tim Kraska. 2016. How Progressive Visualizations Affect Exploratory Analysis. *IEEE transactions on visualization and computer graphics* 23, 8 (2016), 1977–1987.

[34] Yue Zhao, Yunhai Wang, Jian Zhang, Chi-Wing Fu, Mingliang Xu, and Dominik Moritz. 2022. KD-Box: Line-segment-based KD-tree for Interactive Exploration of Large-scale Time-Series Data. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2022), 890–900.

Received October 2022; revised January 2023; accepted February 2023