TUCKET: A Tensor Time Series Data Structure for Efficient and Accurate Factor Analysis over Time Ranges

Ruizhong Qiu* University of Illinois Urbana–Champaign rq5@illinois.edu

Jun-Gi Jang* University of Illinois Urbana–Champaign jungi@illinois.edu

Xiao Lin University of Illinois Urbana–Champaign xiaol13@illinois.edu

Lihui Liu University of Illinois Urbana–Champaign lihuil2@illinois.edu

Hanghang Tong University of Illinois Urbana–Champaign htong@illinois.edu

ABSTRACT

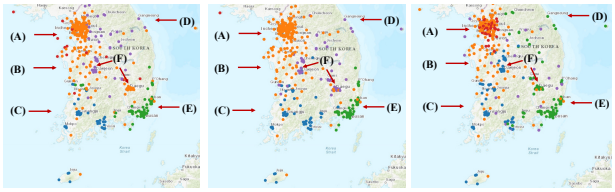
Given an evolving tensor time series and multiple time ranges, how can we compute Tucker decomposition for each time range efficiently and accurately? Tucker decomposition has been widely used in a variety of applications to obtain latent factors of tensor data. For example, Tucker decomposition on air pollution data allows us to analyze and compare air pollution patterns between different locations during different periods of time. In these applications, a common need is to compute Tucker decomposition for a given time range. Furthermore, real-world tensor time series are typically evolving in the time dimension. Such needs call for a data structure that can efficiently and accurately support range queries of Tucker decomposition and stream updates. Unfortunately, existing methods do not support either range queries or stream updates. For methods that do not support range queries, they have to re-compute from scratch for each query. Not until 2021 has a data structure called Zoom-Tucker been proposed to support range queries via block-wise preprocessing. However, Zoom-Tucker does not support stream updates and, more critically, suffers from a reluctant efficiency–accuracy tradeoff — a large block size causes inaccuracy, while a small block size leads to inefficiency. This challenging problem has remained open for years prior to our work. To solve this challenging problem, we propose TUCKET, a data structure that can efficiently and accurately handle both range queries and stream updates. Our key idea is to design a new data structure that we call a *stream segment tree* by generalizing the *segment tree*, a data structure that was originally invented for computational geometry. For a range query of length L, our TUCKET can find $O(\log L)$ nodes (called the *hit set*) from the tree and efficiently stitch their preprocessed decompositions to answer the range query. We also propose an algorithm to optimally prune the hit set via an approximation of subtensor decomposition. For the T-th stream update, our TUCKET modifies only amortized $O(1)$ nodes and only $O(\log T)$ nodes in the worst case. Extensive evaluation demonstrates that our TUCKET consistently achieves the highest efficiency and accuracy across four large-scale datasets. Our TUCKET achieves at least 3 times lower latency and at least 1.4 times smaller reconstruction error than Zoom-Tucker on all datasets. The full version can be found at https://github.com/q-rz/TUCKET/blob/main/TUCKET-Full.pdf.

PVLDB Reference Format:
Ruizhong Qiu, Jun-Gi Jang, Xiao Lin, Lihui Liu, and Hanghang Tong. TUCKET: A Tensor Time Series Data Structure for Efficient and Accurate Factor Analysis over Time Ranges. PVLDB, 17(13): 4746 - 4759, 2024.
doi:10.14778/3704965.3704980

*Equal contribution.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704980



(a) March 2015 (b) March 2016 (c) March 2017

Figure 1: Case study on Air Quality data (see Section 8.6)

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/q-rz/TUCKET.

1 INTRODUCTION

Tensor time series are ubiquitous in the real world, ranging from multimedia data such as videos and music to time series data such as stock prices, traffic volumes, climate, agriculture, environmental monitoring, and physical systems. Tensor decomposition, such as CANDECOMP [13] / PARAFAC [24] (CP), PARAFAC2 [36], and Tucker [62] decompositions, is a fundamental approach to tensor data analysis and performs an essential role in various applications including clustering [12, 26, 71], dimension reduction [37, 65], anomaly detection [19, 39], concept discovery [4, 32, 33], and so on [17, 40, 59, 69]. As a generalization of singular value decomposition, Tucker decomposition [62] seeks to approximately factorize a tensor into *factor matrices* for each mode of the tensor and a small *core tensor* characterizing the relations of the factor matrices. Factor matrices and the core tensor can serve as the input for downstream data mining algorithms such as clustering [71] and anomaly detection [39].

In the analysis of tensor time series, a common situation is to discover latent patterns in given time ranges [28]. For example, given air quality data (a 3-way tensor time series $\mathcal{X}$ where $\mathcal{X}_{t,i,j}$ represents the concentration value of air pollutant j in location i at time t), environmental scientists can find out which locations share similarity pollution patterns in March of each year by analyzing the Tucker decomposition of each month. (See Figure 1 for illustration and the case study in Section 8.6 for detail.) Range queries are necessary here because Tucker decompositions vary across different time ranges due to the evolving nature of tensor time series, as shown in Figure 1. Such needs give rise to an interesting research question: given an evolving tensor time series and

Table 1: Comparison in functionalities, time complexities per query, and overall space complexities. See Table 2 for the definitions of L, D, T, r, p. Empirically, our TUCKET is faster than all other methods on a GPU because it is more parallelizable. Zoom-Tucker cannot use a large block size b, or otherwise it would incur high error.

Method	Range Query	Stream Update	Time Complexity per Query	Space Complexity
Tucker-ALS [62]	✗	✗	$O(rD^{p-1}L)$	$O(D^{p-1}T)$
D-Tucker [29]	✗	✓	$O(r^2D^{p-2}L)$	$O((D^{p-1}+rD^{p-2})T)$
Zoom-Tucker [30]	✓	✗	$O\left(r^2D\frac{L}{b}+r^2L+r^{p+1}\frac{L}{b}\right)$	$O\left((rD+r^p)\frac{T}{b}+rT\right)$
TUCKET (ours)	✓	✓	$O(r^pD\log L+r^{2p-2}(D+L)+\log T)$	$O((rD+r^p)T+rT\log T)$

multiple time range queries, how can we design a data structure that can efficiently and accurately compute Tucker decomposition for each time range?

Unfortunately, existing Tucker decomposition methods do not support either range queries or stream updates (see Table 1). For methods that do not support range queries, they have to re-compute from scratch for each range query. For example, D-Tucker [29] handles stream updates via slice-wise preprocessing, but it supports only full Tucker decomposition and cannot answer range queries efficiently. Not until 2021 has a method called Zoom-Tucker [30] been proposed to support range queries. Zoom-Tucker consists of two phases: a preprocessing phase and a query phase. First, the preprocessing phase divides the timespan into blocks and preprocesses the Tucker decomposition of each block. Next, the query phase answers time range queries by stitching the preprocessed blocks included in the query range. However, Zoom-Tucker does not support efficient stream updates (i.e., appending a new tensor slice) due to its block structure.

Moreover, Zoom-Tucker suffers from a critical limitation: a large block size causes low accuracy for short ranges due to a high approximation error, while a small block size leads to inefficiency for long ranges that require to stitch many blocks. It means that Zoom-Tucker suffers from a reluctant tradeoff between accuracy and efficiency. How to avoid this tradeoff has been an open problem for years. Prior to our work, no existing method achieves both efficiency and accuracy for Tucker decomposition range queries. A crucial challenge here is how to design a more sophisticated data structure and organize preprocessed results to avoid the efficiency–accuracy tradeoff. What makes it even more challenging is that the data structure needs to efficiently support stream updates to the tensor time series.

To solve this challenging problem, we propose a new data structure called *Tucker Tree* (TUCKET) that can efficiently and accurately handle both range queries of Tucker decomposition and stream updates. The key idea of our TUCKET is to design a new data structure that we call a *stream segment tree* by generalizing the *segment tree* [10], a data structure that was originally invented for computational geometry. For a range query of length L, our TUCKET can find $O(\log L)$ nodes (called the *hit set*) from the tree via our optimal pruning algorithm and efficiently stitch their preprocessed decompositions to answer the range query. Besides that, for the T-th stream update, our TUCKET modifies only amortized $O(1)$ nodes and only $O(\log T)$ nodes in the worst case.

The main contributions of this paper are summarized as follows:

- **Data structure.** We design a new data structure *stream segment tree* to efficiently handle stream updates. It is much faster here than the interval tree [54] and the R-tree [23].

Table 2: Nomenclature.

Symbol	Description
vec	vectorization
mat_n	mode-n matricization
$\times_n$	mode-n tensor-matrix product
$\|\cdot\|_F$	Frobenius norm
$\otimes$	matrix Kronecker product
$^\top$	matrix transpose
$\mathcal{X}$	a tensor time series
p	number of modes of $\mathcal{X}$
T	size of the temporal mode of $\mathcal{X}$
$D_2, \dots, D_p$	sizes of non-temporal modes of $\mathcal{X}$
$D := \max\{D_2, \dots, D_p\}$	maximum size of non-temporal modes
$r_1, \dots, r_p$	target sizes of Tucker decomposition
$r := \max\{r_1, \dots, r_p\}$	maximum target size
$\mathcal{G}$	core tensor in Tucker decomposition
$\boldsymbol{U}^{(1)}, \dots, \boldsymbol{U}^{(p)}$	factor matrices in Tucker decomposition
$[T_s, T_e)$	time range of a query
$L := T_e - T_s$	length of the query range
θ	threshold for hit set pruning
$\sqcup$	disjoint union

- **Hit set pruning algorithm.** We derive an efficient scheme to approximate subtensor decompositions and employ it to further reduce the size of the hit set for each query compared with the standard segment tree.
- **Stitching algorithm.** We propose a new algorithm for stitching subtensor decompositions. Our stitching algorithm is more GPU-parallelizable and more numerically stable than Zoom-Tucker's stitching algorithm.
- **Theoretical guarantees.** We provide detailed theoretical guarantees for our proposed method in terms of time complexity, space complexity, and error analysis.
- **Empirical evaluation.** We conduct extensive experiments to evaluate our TUCKET against state-of-the-art methods for Tucker decomposition on large-scale real-world tensor time series datasets. Our TUCKET consistently achieves both the highest efficiency and the highest accuracy across all datasets. For example, our TUCKET achieves at least 3 times lower latency and at least 1.4 times smaller reconstruction error on all datasets.

2 PRELIMINARIES

In this section, we present the preliminaries on Tucker decomposition and Tucker-ALS. Main symbols used in this paper are summarized in Table 2. Due to the space limit, preliminaries on basic tensor operations are deferred to the full version.

Given a p-way tensor $\mathcal{X} \in \mathbb{R}^{D_1 \times \cdots \times D_p}$ and target sizes $r_1, \ldots, r_p$, *Tucker decomposition* [62] aims to find a *core tensor* $\mathcal{G} \in \mathbb{R}^{r_1 \times \cdots \times r_p}$ and column-orthonormal *factor matrices* $\boldsymbol{U}^{(n)} \in \mathbb{R}^{D_n \times r_n}$ $(n = 1, \ldots, p)$ that minimize

$$\|\mathcal{G} \times_1 \boldsymbol{U}^{(1)} \cdots \times_p \boldsymbol{U}^{(p)} - \mathcal{X}\|_{\mathrm{F}}^2. \tag{1}$$

Tucker decomposition is a generalization of the singular value decomposition (SVD) of matrices. Similarly with SVD, a real-world tensor $\mathcal{X}$ typically has $\mathcal{X} \approx \mathcal{G} \times_1 \boldsymbol{U}^{(1)} \cdots \times_p \boldsymbol{U}^{(p)}$ even for small target sizes $r_1, \ldots, r_p$ [44]. Hence, Tucker decomposition can serve as a compressed representation of a large tensor. Factor matrices and the core tensor can serve as the input for downstream data mining algorithms. For example, we can apply clustering [71] or anomaly detection [39] to the row vectors of the factor matrices $\boldsymbol{U}^{(n)}$ to discover similarity and dissimilarity patterns in the data.

A classic approach to Tucker decomposition is Tucker's alternating least squares method (Tucker-ALS) [62]. At each iteration, Tucker-ALS optimizes the factor matrix of only one mode while fixing all other factor matrices. Tucker [62] shows that the optimal factor matrix $\boldsymbol{U}^{(n)}$ for each mode n is the r_n leading left singular vectors of the matrix

$$\mathrm{mat}_n(\mathcal{X} \times_1 \boldsymbol{U}^{(1)\mathsf{T}} \cdots \times_{n-1} \boldsymbol{U}^{(n-1)\mathsf{T}} \times_{n+1} \boldsymbol{U}^{(n+1)\mathsf{T}} \cdots \times_p \boldsymbol{U}^{(p)\mathsf{T}}), \tag{2}$$

and that the optimal core tensor $\mathcal{G}$ is

$$\mathcal{X} \times_1 \boldsymbol{U}^{(1)\mathsf{T}} \cdots \times_p \boldsymbol{U}^{(p)\mathsf{T}}. \tag{3}$$

3 PROBLEM DEFINITION

In this section, we first introduce the problem definition and then present the design goals of TUCKET.

A tensor time series is a tensor where one of the modes represents time. Without loss of generality, let the first mode be the *temporal* mode. Let $\mathcal{X} \in \mathbb{R}^{T \times D_2 \times \cdots \times D_p}$ be a p-way tensor time series, where the size of the temporal mode is $D_1 := T$, the sizes of non-temporal modes are $D_2, \ldots, D_p$, and the number of modes is $p \geq 2$. We call T the *timespan*. For tensors $\mathcal{Y}_1, \ldots, \mathcal{Y}_s$ of the same shape except for the temporal mode, let $\begin{bmatrix} \mathcal{Y}_1 \\ \vdots \\ \mathcal{Y}_s \end{bmatrix}$ denote concatenation along the temporal mode. A *tensor stream* $\mathcal{X} \in \mathbb{R}^{* \times D_2 \times \cdots \times D_p}$ is a tensor time series with a growing temporal mode: at each time T, a tensor slice $\mathcal{X}_T \in \mathbb{R}^{D_2 \times \cdots \times D_p}$ is observed and appended to the tensor stream.

Given a tensor stream $\mathcal{X} \in \mathbb{R}^{* \times D_2 \times \cdots \times D_p}$ and the target sizes $(r_1, \ldots, r_p)$ for Tucker decomposition, we aim to design a data structure that supports the following two operations. T denotes the timespan before the operation.

- **Range query of Tucker decomposition:** Given a time range $[T_\mathrm{s}, T_\mathrm{e}) \subseteq [0, T)$, we need to efficiently compute the Tucker decomposition of the subtensor $\mathcal{X}_{[T_\mathrm{s}, T_\mathrm{e})}$ using the data structure. The output is a core tensor $\mathcal{G} \in \mathbb{R}^{r_1 \times \cdots \times r_p}$ and factor matrices $\boldsymbol{U}^{(1)} \in \mathbb{R}^{(T_\mathrm{e} - T_\mathrm{s}) \times r_1}$, $\boldsymbol{U}^{(2)} \in \mathbb{R}^{D_2 \times r_2}$, …, $\boldsymbol{U}^{(p)} \in \mathbb{R}^{D_p \times r_p}$.
- **Stream update:** Given a tensor slice $\mathcal{X}_T \in \mathbb{R}^{D_2 \times \cdots \times D_p}$, we need to append the tensor slice to the tensor stream and update the data structure accordingly.

The problem definition is illustrated in Figure 2 with Air Quality data as an example. Air Quality data is a 3-way tensor time series $\mathcal{X} \in \mathbb{R}^{T \times D_2 \times D_3}$ where $\mathcal{X}_{t,i,j}$ represents the concentration value of

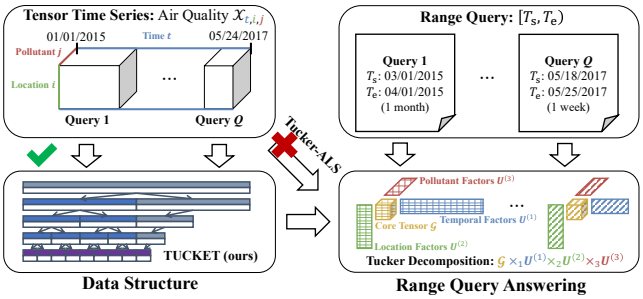


Figure 2: Illustration of range queries of Tucker decomposition. It is inefficient to directly apply Tucker-ALS for each range query from scratch. Instead, we aim to design a data structure that can efficiently and accurately answer range queries of Tucker decomposition without re-computing from scratch for each query.

air pollutant j in location i at time t. Consider a case study where we want to analyze air pollution patterns in March of each year. Here, each range query is a month (March 2015, March 2016, or March 2017; see Figure 1). With the help of Tucker decomposition range queries, we can find out which locations share similarity pollution patterns in each month by clustering the row vectors of the location factor matrix $\boldsymbol{U}^{(2)}$ of Tucker decomposition of each month. Results of the case study are shown in Figure 1. See Section 8.6 for detail.

We design our TUCKET with the following three design goals for a tensor time series data structure.

G1: Frequent arbitrary range queries. We focus on the situation where queries are frequent, and we only consider online algorithms (i.e., the algorithm has to process each operation sequentially as soon as it arrives). Thus, we need to optimize the worst-case complexity of answering each single range query. Besides that, we do not assume any extra prior knowledge about the distribution of possible range queries. Hence, we focus on the worst-case complexity parameterized by: (i) the maximum size of non-temporal modes, $D := \max\{D_2, \ldots, D_p\}$; (ii) the timespan, T; (iii) the maximum target size, $r := \max\{r_1, \ldots, r_p\}$; and (iv) the length of the query range, $L := T_\mathrm{e} - T_\mathrm{s}$.

G2: Periodic stream updates. In real-world use cases, stream updates to the tensor stream are typically periodic but may not be as frequent as range queries. For instance, in the stock example in Figure 2, the tensor stream is updated in a daily basis. Hence, we allow the stream update operation to be a little more expensive than range queries. Nonetheless, we still aim to optimize the time complexity of stream updates so that it scales at most sublinearly w.r.t. the total size $TD_2 \cdots D_p$ of the current tensor time series.

G3: Nearly linear space. Every preprocessing-based data structure is essentially a space–time tradeoff [16, 25] in that more preprocessing leads to higher efficiency. On the one hand, if no preprocessing were allowed, it would be impossible to outperform the naïve algorithm that simply computes from scratch for each query. On the other hand, if unlimited preprocessing were allowed, then a trivial algorithm would be to preprocess the answers for all possible $O(T^2)$ query ranges. To rule out such trivial algorithms, we require that the space used by the preprocessing phase should be nearly

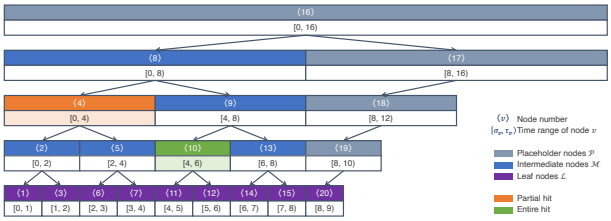


Figure 3: Illustration of our TUCKET over timespan $[0, 9)$. It has height $5 = \lceil \log_2 9 \rceil + 1$. When answering a range query $[T_s, T_e) = [1, 6)$ with pruning threshold $\theta = 0.7$, node $\langle 4 \rangle$ is a partial hit because $|[1, 6) \cap [0, 4)| = |[1, 4)| \geq 0.7|[0, 4)|$, and node $\langle 10 \rangle$ is an entire hit because $[4, 6) \subseteq [1, 6)$.

linear w.r.t. the timespan T, i.e., $\widetilde{O}(T)$. As a remark, we assume that $p = O(1)$ and $r = o(D)$ in our complexity analysis.

4 TUCKET: DATA STRUCTURE DESIGN

In this section, we detail the design of our proposed data structure Tucker Tree (TUCKET). We first introduce the challenges of the problem and our key ideas in Section 4.1 and then present the design of our stream segment tree in Section 4.2. Due to the space limit, detailed proofs are deferred to the full version.

4.1 Challenges & Key Ideas

Our TUCKET is designed for efficient Tucker decomposition over time ranges. An essential challenge of this problem is that the Tucker decomposition operation does not form either an Abelian group or a semigroup w.r.t. tensor concatenation. This means that we cannot use classic data structures such as prefix sum tables, Fenwick trees [20], or Cartesian trees [9] to compute Tucker decomposition over time ranges. The state-of-the-art idea (proposed in [30]) is to divide the time range $[0, T)$ into disjoint blocks $[0, b), [b, 2b), \ldots$ of equal size b and preprocess the Tucker decomposition for each block. However, this idea suffers from an inevitable tradeoff between efficiency and accuracy: a small block size b leads to inefficiency for long time ranges; a large block size causes inaccuracy for time ranges shorter than b. This is because a larger block size b corresponds to a coarser-grained preprocessing, which may fail to preserve finer-grained patterns that exist temporarily in time ranges shorter than b. For instance, if the block size b corresponds to a month, then Zoom-Tucker will be likely to yield inaccurate results when querying for a week. To the best of our knowledge, our work is the first data structure that addresses this challenge without the efficiency–accuracy tradeoff.

Our first key idea, aiming to address this challenge, is to divide the time range $[0, T)$ into carefully designed uneven, overlapping blocks such that every range query can be expressed as a disjoint union of a small number of "blocks." Our idea leads us to the segment tree [10], a data structure from computational geometry. Crucially, in a segment tree over a timespan $[0, T)$, we can associate each node v with a subrange $[\sigma_v, \tau_v)$ such that every range $[T_s, T_e) \subseteq [0, T)$ can be expressed as a disjoint union of at most $O(\log T)$ nodes. We call these nodes the *hit set* of $[T_s, T_e)$. Then, we can answer each range query efficiently by stitching the preprocessed subtensor decompositions of the hit set.

However, the original segment tree has a static structure and thus does not support stream data. Hence, we cannot simply apply the segment tree. As our second key idea, we propose a new data structure called the *stream segment tree* to support stream updates in our setting. Our key insight regarding why segment trees are static is that it is required to be a full binary tree in order to maintain a depth of $O(\log T)$. Instead, we propose to relax this requirement and allow our stream segment tree to be incomplete. To maintain a depth of $O(\log T)$, we propose extending the root instead of only extending leaf nodes like typical balanced search trees [22]. Furthermore, each stream update operation only involves amortized $O(1)$ nodes to be updated. We will describe the detailed design of our stream segment tree in Section 4.2.

4.2 Stream Segment Tree

The basic structure of our TUCKET is a *stream segment tree*, which is a generalization of the segment tree [10] from computational geometry. Here, we detail the design of our stream segment tree.

As we have discussed, the original segment tree does not support stream updates. To enable stream updates, our key idea is to employ an expanded segment tree with an incomplete structure that reserves the position of future nodes to support efficient updates but does not construct them explicitly.

Specifically, a stream segment tree is a binary tree where each node v is associated with a time range $[\sigma_v, \tau_v)$. There are three types of nodes in a stream segment tree: *leaf* nodes $\mathcal{L}$, *intermediate* nodes $\mathcal{M}$, and *placeholder* nodes $\mathcal{P}$.

Leaf nodes. Each leaf node $v \in \mathcal{L}$ represents a tensor slice $\mathcal{X}_t$, so we let the leaf range be $[\sigma_v, \tau_v) := [t, t+1)$. After T updates, we require time ranges of leaf nodes to be *contiguous*, i.e., the whole range $[0, T)$ is a disjoint union of the time ranges of leaf nodes:

$$\bigsqcup_{v \in \mathcal{L}} [\sigma_v, \tau_v) = [0, T). \tag{4}$$

Besides that, we preprocess the Tucker decomposition of $\mathcal{X}_{[t,t+1)}$ and store it as $\mathcal{Y}_v := \mathcal{H}_v \times_1 \boldsymbol{V}_v^{(1)} \cdots \times_p \boldsymbol{V}_v^{(p)}$. Note that we only compute the core tensor $\mathcal{H}_v$ and factor matrices $\boldsymbol{V}_v^{(1)}, \ldots, \boldsymbol{V}_v^{(p)}$ but never actually compute $\mathcal{Y}_v$, i.e., $\mathcal{Y}_v$ is only a symbol to refer to the product.

Intermediate nodes. For each intermediate node $v \in \mathcal{M}$ with time range $[\sigma_v, \tau_v)$, it represents a subtensor $\mathcal{X}_{[\sigma_v, \tau_v)}$. An intermediate node has exactly two children nodes $u_1, u_2 \in \mathcal{L} \cup \mathcal{M}$ that have *adjoint* time ranges:

$$[\sigma_v, \tau_v) := [\sigma_{u_1}, \tau_{u_1}) \sqcup [\sigma_{u_2}, \tau_{u_2}). \tag{5}$$

Besides that, similarly with leaf nodes, we preprocess the Tucker decomposition of $\mathcal{X}_{[\sigma_v, \tau_v)}$ and store it as $\mathcal{Y}_v := \mathcal{H}_v \times_1 \boldsymbol{V}_v^{(1)} \cdots \times_p \boldsymbol{V}_v^{(p)}$. Here, $\mathcal{Y}_v$ is still only a symbol to refer to the product.

Placeholder nodes. A placeholder node $v \in \mathcal{P}$ represents a future subtensor where some of its slices have not been observed yet. Formally, if the current observed timespan is $[0, T)$. then the time range $[\sigma_v, \tau_v)$ of the placeholder node has $\sigma_v < T$ and $\tau_v \geq T$. A placeholder node has either one or two children. If v has only one child, then it has a left child $u_1 \in \mathcal{L} \cup \mathcal{M} \cup \mathcal{P}$; otherwise, v has a left child $u_1 \in \mathcal{L} \cup \mathcal{M}$ and a right child $u_2 \in \mathcal{P}$. Similarly with intermediate nodes, we require its children to have *adjoint* time

ranges, i.e., $[\sigma_v, \tau_v) := [\sigma_{u_1}, \tau_{u_1}) \sqcup [\sigma_{u_2}, \tau_{u_2})$. Meanwhile, unlike leaf and intermediate nodes, since the subtensor of the time range $[\sigma_v, \tau_v)$ has not been completely observed yet, we do not preprocess the Tucker decomposition of a placeholder node and do not allow placeholder nodes to be in the hit set.

Logarithmic height. To efficiently answer range queries, we want that every range query $[T_s, T_e) \subseteq [0, T)$ can be divided into a disjoint union of a small number of nodes (called the *hit set*) in the stream segment tree. We make a key observation on the relation between the size of the hit set and the height of the stream segment tree, as formally stated in Lemma 4.1.

Lemma 4.1 (Hit set v.s. height). *Given a stream segment tree of height $h \geq 1$, for every range query, there exists a hit set of size $\leq \max\{2(h-1), 1\}$.*

Proof sketch. First, if the query range $[T_s, T_e)$ is a prefix or a suffix of the time range $[\sigma_v, \tau_v)$ of a node v (i.e., $T_s = \sigma_v$ or $T_e = \tau_v$), then an induction shows that there exists a hit set of size $\leq h$. Next, if the query range is neither a prefix nor a suffix of the time range of any node, then we can show that there exists two non-root nodes v_1, v_2 such that $\tau_{v_1} = \sigma_{v_2}$ and that $[T_s, T_e) = [\sigma_{v_1}, \tau_{v_1}) \sqcup [\sigma_{v_2}, \tau_{v_2})$. In this case, $[T_s, T_e) \cap [\sigma_{v_1}, \tau_{v_1})$ is a suffix of $[\sigma_{v_1}, \tau_{v_1})$, and $[T_s, T_e) \cap [\sigma_{v_2}, \tau_{v_2})$ is a prefix of $[\sigma_{v_2}, \tau_{v_2})$. Since v_1, v_2 are not the root, then $[T_s, T_e)$ has a hit set of size $\leq 2(h-1)$. Together, the size of the hit set is $\leq \max\{2(h-1), h\} = \max\{2(h-1), 1\}$. □

Lemma 4.1 motivates us to require the stream segment tree to have a small height. We show that the stream segment tree can indeed have a logarithmic height w.r.t. the time range T, as formally stated in Theorem 4.2.

Theorem 4.2 (Logarithmic height). *There exists a stream segment tree structure over range $[0, T)$ of height $\leq \lceil \log_2 T \rceil + 1$.*

Proof sketch. Using the algorithm in Section 6.2 to append the tensor slices $\mathcal{X}_0, \ldots, \mathcal{X}_{T-1}$ one by one, we can build a stream segment tree over $[0, T)$. By Theorem 6.2, this stream segment tree has height $\lceil \log_2 T \rceil + 1$. □

The tree structure in Theorem 4.2 not only exists in theory but can also be maintained efficiently. We will present an efficient algorithm to maintain the logarithmic height in a stream update in Section 6.2. From now on, we will refer to the tree structure in Theorem 4.2 simply as the "stream segment tree."

Proposition 4.3 (Space complexity). *The space complexity of a stream segment tree over range $[0, T)$ is $O((rD + r^p)T + rT \log T)$.*

Proof sketch. In a stream segment tree over the range $[0, T)$, there are $O(T)$ nodes each of which has $p - 1$ non temporal factor matrices of the size $O(rD)$ and a core tensor of the size $O(r^p)$. In addition, at each level of the tree, the sum of the sizes of temporal factor matrices is $O(rT)$. Therefore, the space complexity of the stream segment tree is $O((prD + r^p)T + rT \log T)$. □

Example. An example of our stream segment tree over timespan $[0, 9)$ is illustrated in Figure 3. It has height $5 = \lceil \log_2 9 \rceil + 1$. Leaf nodes $\langle 1 \rangle, \langle 3 \rangle, \langle 6 \rangle, \langle 7 \rangle, \langle 11 \rangle, \langle 12 \rangle, \langle 14 \rangle, \langle 15 \rangle, \langle 20 \rangle$ represent tensor slices $\mathcal{X}_0, \ldots, \mathcal{X}_8$, respectively. Intermediate nodes store preprocessed results. We will also use this example in the subsequent sections to illustrate other operations.

5 TUCKET: CORE ALGORITHMS

In this section, we present two core algorithms that will be used in the operations of TUCKET in Section 6. We describe how to optimally prune the hit set in Section 5.1 and how to stitch subtensor decompositions in the hit set in Section 5.2.

5.1 Optimally Pruning the Hit Set

The first core algorithm of TUCKET is finding a small hit set for each range query. By Lemma 4.1 & Theorem 4.2, we have shown that the hit set has a small size $O(\log T)$. Although this size cannot be improved for general operations, here we present a key observation about the Tucker decomposition of a subtensor and leverage this observation to further prune the hit set.

Approximating a subtensor decomposition. Here we present our key observation about the Tucker decomposition of a subtensor. Suppose that a tensor $\mathcal{X}_{[\sigma_v, \tau_v)}$ has Tucker decomposition $\mathcal{H}_v \times_1 \boldsymbol{V}_v^{(1)} \cdots \times_p \boldsymbol{V}_v^{(p)}$. Due to the low-rank nature of real-world tensors [44], they can typically be well approximated by Tucker decomposition:

$$\mathcal{X}_{[\sigma_v, \tau_v)} \approx \mathcal{H}_v \times_1 \boldsymbol{V}_v^{(1)} \cdots \times_p \boldsymbol{V}_v^{(p)}. \tag{6}$$

We observe that for real-world tensors, there typically exists a threshold $0 < \theta < 1$ (see Section 8.5) such that for a sub-range $[T_s', T_e') \subseteq [\sigma_v, \tau_v)$ with $\left|[T_s', T_e')\right| \geq \theta \left|[\sigma_v, \tau_v)\right|$ (i.e., the sub-range $[T_s', T_e')$ is not too small compared with $[\sigma_v, \tau_v)$), the subtensor $\mathcal{X}_{[T_s', T_e')}$ can be well approximated by

$$\mathcal{X}_{[T_s', T_e')} = (\mathcal{X}_{[\sigma_v, \tau_v)})_{[T_s' - \sigma_v, T_e' - \sigma_v)} \tag{7}$$

$$\approx (\mathcal{H}_v \times_1 \boldsymbol{V}_v^{(1)} \times_2 \boldsymbol{V}_v^{(2)} \cdots \times_p \boldsymbol{V}_v^{(p)})_{[T_s' - \sigma_v, T_e' - \sigma_v)} \tag{8}$$

$$= \mathcal{H}_v \times_1 (\boldsymbol{V}_v^{(1)})_{[T_s' - \sigma_v, T_e' - \sigma_v)} \times_2 \boldsymbol{V}_v^{(2)} \cdots \times_p \boldsymbol{V}_v^{(p)}. \tag{9}$$

This almost yields an approximate Tucker decomposition of $\mathcal{X}_{[T_s', T_e')}$, except that the temporal factor matrix $(\boldsymbol{V}_v^{(1)})_{[T_s' - \sigma_v, T_e' - \sigma_v)}$ is not necessarily column-orthonormal. To make it column-orthonormal, we can first compute a QR decomposition [21] $(\boldsymbol{V}_v^{(1)})_{[T_s' - \sigma_v, T_e' - \sigma_v)} =: \widetilde{\boldsymbol{Q}}\widetilde{\boldsymbol{R}}$ (where $\widetilde{\boldsymbol{Q}}$ is column-orthonormal) and then use the reverse associativity[1] [38] of $\times_1$ to give a Tucker decomposition:

$$\mathcal{X}_{[T_s', T_e')} \approx \mathcal{H}_v \times_1 (\boldsymbol{V}_v^{(1)})_{[T_s' - \sigma_v, T_e' - \sigma_v)} \times_2 \boldsymbol{V}_v^{(2)} \cdots \times_p \boldsymbol{V}_v^{(p)} \tag{10}$$

$$= \mathcal{H}_v \times_1 (\widetilde{\boldsymbol{Q}}\widetilde{\boldsymbol{R}}) \times_2 \boldsymbol{V}_v^{(2)} \cdots \times_p \boldsymbol{V}_v^{(p)} \tag{11}$$

$$= (\mathcal{H}_v \times_1 \widetilde{\boldsymbol{R}}) \times_1 \widetilde{\boldsymbol{Q}} \times_2 \boldsymbol{V}_v^{(2)} \cdots \times_p \boldsymbol{V}_v^{(p)}, \tag{12}$$

where the core tensor is $\mathcal{H}_v \times_1 \widetilde{\boldsymbol{R}}$, and the temporal factor matrix is $\widetilde{\boldsymbol{Q}}$. In this way, we can efficiently compute an approximate Tucker decomposition of a subtensor $\mathcal{X}_{[T_s', T_e')}$ using only a QR decomposition and a mode-1 product and do not need to further divide $[T_s', T_e')$ into smaller sub-ranges. This helps to reduce the size of the hit set.

Formulation of hit set pruning. This key observation motivates us to consider *partial hits*. Let $0 < \theta < 1$ denote the threshold above. We call a node $v \in \mathcal{M}$ a *partial hit* of a range query $[T_s, T_e)$ if $\left|[T_s, T_e) \cap [\sigma_v, \tau_v)\right| \geq \theta \left|[\sigma_v, \tau_v)\right|$ and $[\sigma_v, \tau_v) \not\subseteq [T_s, T_e)$; we call it an

[1] The reverse associativity means that $\mathcal{Z} \times_n \boldsymbol{A} \times_n \boldsymbol{B} = \mathcal{Z} \times_n (\boldsymbol{B}\boldsymbol{A})$.

Algorithm 1 (RECALL): Finding a pruned hit set

Input: current node v; query range $[T_s, T_e)$
Output: a pruned hit set of $[T_s, T_e)$ in the subtree rooted at v

```
if v ∈ L ∪ M and |[T_s, T_e) ∩ [σ_v, τ_v)| ≥ θ|[σ_v, τ_v)| then
    return {v}
end if
let u_1, u_2 be the left and right children of v, respectively
if T_e ≤ τ_{u_1} then
    return RECALL(u_1, [T_s, T_e))
else if T_s ≥ σ_{u_2} then
    return RECALL(u_2, [T_s, T_e))
else
    return RECALL(u_1, [T_s, T_e)) ∪ RECALL(u_2, [T_s, T_e))
end if
```

entire hit of $[T_s, T_e)$ if $[\sigma_v, \tau_v) \subseteq [T_s, T_e)$. Using Eq. (12), we can efficiently approximate the Tucker decomposition of $\mathcal{X}_{[T_s,T_e)\cap[\sigma_v,\tau_v)}$. Hence, we can reduce the size of the hit set by allowing partial hits in the hit set. Formally, minimizing the size of the hit set $\mathcal{S}$ can be formulated as the following optimization problem:

$$\min_{\mathcal{S}\subseteq\mathcal{L}\cup\mathcal{M}} |\mathcal{S}|, \tag{13}$$

$$\text{s.t.} \quad [T_s, T_e) = \bigsqcup_{v\in\mathcal{S}} [T_s, T_e) \cap [\sigma_v, \tau_v), \tag{14}$$

$$\big|[T_s, T_e) \cap [\sigma_v, \tau_v)\big| \geq \theta\big|[\sigma_v, \tau_v)\big|, \quad \forall v \in \mathcal{S}. \tag{15}$$

An optimal algorithm for pruning. To solve the formulation above for hit set pruning, we propose an efficient recursive algorithm that runs in $O(\log T)$ time. The basic idea is as follows. We start from the root node. If the root node is a partial hit, then we stop and return the root node as the hit set. Otherwise, we consider its two children and repeat the procedure above. The overall procedure is presented in Algorithm 1. Since the height of the stream segment tree is $O(\log T)$, and Algorithm 1 visits at most two nodes at each height, then the total running time of Algorithm 1 is $O(\log T)$.

Furthermore, our Theorem 5.1 shows that our Algorithm 1 is indeed optimal — it can find the smallest hit set that satisfies the constraints Eqs. (14) & (15).

THEOREM 5.1 (OPTIMALITY & COMPLEXITY OF ALGORITHM 1). *Given a range query $[T_s, T_e)$, Algorithm 1 minimizes the formulation in Eq. (13) within $O(\log T)$ running time and finds a hit set with $O(1)$ partial hits and $O(\log L)$ entire hits, where $L := T_e - T_s$.*

PROOF SKETCH. *Optimality.* Note that if the two children of a node v are both in the hit set, then replacing the two children with the node v gives a smaller, valid hit set. Using this fact, it can be shown that every node in the optimal hit set should not have a parent node that is also a valid hit. Finally, by analyzing the top-down procedure of Algorithm 1, we can show that Algorithm 1 can indeed find such a hit set, and that the hit set cannot be replaced with a smaller hit set.

Complexity. An induction similar with the proof of Lemma 4.1 shows that the hit set found by Algorithm 1 has $O(1)$ partial hits and $O(\log L)$ entire hits. Since the height of the stream segment tree is $O(\log T)$, the number of nodes traversed in the process of finding the hit set is at most $O(\log T + \log L) = O(\log T)$. Finally, as Algorithm 1 performs $O(1)$ operations per node traversed, its time complexity is $O(\log T)$. □

Example. Algorithm 1 is exemplified in Figure 3. When answering a range query $[1, 6)$ with $\theta = 0.7$, $[0, 4)$ is a partial hit because $|[1, 6) \cap [0, 4)| \geq 0.7|[0, 4)|$, and $[4, 6)$ is an entire hit because $[4, 6) \subseteq [1, 6)$. For the partial hit $[0, 4)$, we use Eq. (12) to approximate the Tucker decomposition of the sub-range $[1, 6) \cap [0, 4) = [1, 4)$.

5.2 Stitching Subtensor Decompositions

Another core algorithm of TUCKET is stitching subtensor decompositions in the hit set. Given a range query $[T_s, T_e)$, suppose that the (pruned) hit set is $\mathcal{S} = \{v_1, \ldots, v_s\}$, where $s := |\mathcal{S}|$ denotes the size of the hit set. For each partial hit, we use Eq. (12) to compute its approximate Tucker decomposition $\widetilde{\mathcal{Y}}_i$ of the subtensor $\mathcal{X}_{[T_s,T_e)\cap[\sigma_{v_i},\tau_{v_i})}$; for each entire hit v_i, we retrieve its preprocessed Tucker decomposition $\widetilde{\mathcal{Y}}_i := \mathcal{Y}_{v_i}$. Same as before, here $\widetilde{\mathcal{Y}}_1, \ldots, \widetilde{\mathcal{Y}}_s$ are just symbols to refer to the Tucker decomposition products. We aim to efficiently compute an approximate Tucker decomposition of $\mathcal{X}_{[T_s,T_e)}$ using these preprocessed subtensor decompositions $\widetilde{\mathcal{Y}}_1, \ldots, \widetilde{\mathcal{Y}}_s$.

A key observation is that $\mathcal{X}_{[T_s,T_e)\cap[\sigma_{v_i},\tau_{v_i})} \approx \widetilde{\mathcal{Y}}_i$ due to the low-rank nature of real-world tensors [44]. This motivates us to express $\mathcal{X}_{[T_s,T_e)}$ as a concatenation of the hit set along the temporal mode:

$$\mathcal{X}_{[T_s,T_e)} = \begin{bmatrix} \mathcal{X}_{[T_s,T_e)\cap[\sigma_{v_1},\tau_{v_1})} \\ \vdots \\ \mathcal{X}_{[T_s,T_e)\cap[\sigma_{v_s},\tau_{v_s})} \end{bmatrix} \approx \begin{bmatrix} \widetilde{\mathcal{Y}}_1 \\ \vdots \\ \widetilde{\mathcal{Y}}_s \end{bmatrix}. \tag{16}$$

Next, we design an efficient algorithm to compute the Tucker decomposition of $\mathcal{X}_{[T_s,T_e)}$ by stitching the subtensor decompositions $\widetilde{\mathcal{Y}}_1, \ldots, \widetilde{\mathcal{Y}}_s$. The key idea here is to leverage the concatenation form of $\widetilde{\mathcal{Y}} := \begin{bmatrix} \widetilde{\mathcal{Y}}_1 \\ \vdots \\ \widetilde{\mathcal{Y}}_s \end{bmatrix}$ and again utilize the reverse associativity[1] [38] of the tensor–matrix product so as to efficiently compute the matricizations in Tucker-ALS.

Let $\widetilde{\mathcal{H}}_i$ and $\widetilde{\boldsymbol{V}}_i^{(1)}, \ldots, \widetilde{\boldsymbol{V}}_i^{(p)}$ denote the core tensor and the factor matrices in $\widetilde{\mathcal{Y}}_i$, respectively, and let $\mathcal{G}$ and $\boldsymbol{U}^{(1)}, \ldots, \boldsymbol{U}^{(p)}$ denote the core tensor and the factor matrices of $\mathcal{X}_{[T_s,T_e)}$ to be computed, respectively. Since the optimal update of the factor matrix $\boldsymbol{U}^{(n)}$ is the r_n leading left singular vectors of the matricization in Eq. (2), we need to compute this matricization efficiently. Since the concatenation is along the temporal mode, we will describe how to efficiently compute the matricization for the temporal mode and the non-temporal modes separately. The overall procedure of stitching subtensor decompositions is presented in Algorithm 2.

Matricization of the temporal mode. Our goal is to compute the matricization in Eq. (2) without explicitly computing the large tensor $\widetilde{\mathcal{Y}}$. First, we rewrite the matricizations of $\widetilde{\mathcal{Y}}_i$ via the matrix Kronecker product $\otimes$:

$$\text{mat}_1(\widetilde{\mathcal{Y}}_i) = \text{mat}_1(\widetilde{\mathcal{H}}_i \times_1 \widetilde{\boldsymbol{V}}_i^{(1)} \cdots \times_p \widetilde{\boldsymbol{V}}_i^{(p)}) \tag{17}$$

$$= \widetilde{\boldsymbol{V}}_i^{(1)} \, \text{mat}_1(\widetilde{\mathcal{H}}_i) \bigotimes_{m=2}^{p} \widetilde{\boldsymbol{V}}_i^{(m)\mathsf{T}}. \tag{18}$$

Similarly, we can rewrite the matricization in Eq. (2) as

$$\text{mat}_1(\widetilde{\mathcal{Y}} \times_2 \boldsymbol{U}^{(2)\mathsf{T}} \cdots\cdots \times_p \boldsymbol{U}^{(p)\mathsf{T}}) = \text{mat}_1(\widetilde{\mathcal{Y}}) \bigotimes_{m=2}^{p} \boldsymbol{U}^{(m)}. \tag{19}$$

Algorithm 2 (Stitch): Stitching subtensor decompositions

Input: subtensor decompositions $\widetilde{\mathcal{Y}}_i := \widetilde{\mathcal{H}}_i \times_1 \widetilde{\boldsymbol{V}}_i^{(1)} \cdots \times_p \widetilde{\boldsymbol{V}}_i^{(p)}$ of the hit set $\{v_1, \ldots, v_s\}$

Output: stitched Tucker decomposition of $\begin{bmatrix} \widetilde{\mathcal{Y}}_1 \\ \vdots \\ \widetilde{\mathcal{Y}}_s \end{bmatrix}$

randomly initialize $\boldsymbol{U}^{(2)}, \ldots, \boldsymbol{U}^{(p)}$
repeat
  obtain $\boldsymbol{Z}^{(1)}$ using Eq. (22)
  let $\boldsymbol{U}^{(1)}$ be the r_1 leading left singular vectors of $\boldsymbol{Z}^{(1)}$
  $t_0 \leftarrow 0$
  for $i \leftarrow 1, \ldots, s$ **do**
    $t_i \leftarrow t_{i-1} + (\tau_{v_i} - \sigma_{v_i})$
  end for
  for $n \leftarrow 2, \ldots, p$ **do**
    obtain $\boldsymbol{Z}^{(n)}$ using Eq. (27)
    let $\boldsymbol{U}^{(n)}$ be the r_n leading left singular vectors of $\boldsymbol{Z}^{(n)}$
  end for
  reshape $\boldsymbol{Z}^{(p)}$ into a tensor $\mathcal{Z}^{(p)} \in \mathbb{R}^{r_1 \times \cdots \times r_{p-1} \times D_p}$
  $\mathcal{G} \leftarrow \mathcal{Z}^{(p)} \times_p \boldsymbol{U}^{(p)\mathsf{T}}$
until converged
return $(\mathcal{G}, \boldsymbol{U}^{(1)}, \ldots, \boldsymbol{U}^{(p)})$

Since the matricization of the concatenation $\widetilde{\mathcal{Y}}$ is equal to the concatenation of $\mathrm{mat}_1(\widetilde{\mathcal{Y}}_i)$'s, then by the mixed-product property[2] [11] of the Kronecker product, Eq. (19) can be further rewritten as:

$$\begin{bmatrix} \widetilde{\boldsymbol{V}}_1^{(1)} \mathrm{mat}_1(\widetilde{\mathcal{H}}_1)(\bigotimes_{m=2}^{p} \widetilde{\boldsymbol{V}}_1^{(m)\mathsf{T}})(\bigotimes_{m=2}^{p} \boldsymbol{U}^{(m)}) \\ \vdots \\ \widetilde{\boldsymbol{V}}_s^{(1)} \mathrm{mat}_1(\widetilde{\mathcal{H}}_s)(\bigotimes_{m=2}^{p} \widetilde{\boldsymbol{V}}_s^{(m)\mathsf{T}})(\bigotimes_{m=2}^{p} \boldsymbol{U}^{(m)}) \end{bmatrix} \tag{20}$$

$$= \begin{bmatrix} \widetilde{\boldsymbol{V}}_1^{(1)} \mathrm{mat}_1(\widetilde{\mathcal{H}}_1) \bigotimes_{m=2}^{p} (\widetilde{\boldsymbol{V}}_1^{(m)\mathsf{T}} \boldsymbol{U}^{(m)}) \\ \vdots \\ \widetilde{\boldsymbol{V}}_s^{(1)} \mathrm{mat}_1(\widetilde{\mathcal{H}}_s) \bigotimes_{m=2}^{p} (\widetilde{\boldsymbol{V}}_s^{(m)\mathsf{T}} \boldsymbol{U}^{(m)}) \end{bmatrix} \tag{21}$$

$$= \begin{bmatrix} \mathrm{mat}_1(\widetilde{\mathcal{H}}_1 \times_1 \widetilde{\boldsymbol{V}}_1^{(1)} \times_2 (\boldsymbol{U}^{(2)\mathsf{T}} \widetilde{\boldsymbol{V}}_1^{(2)}) \cdots \times_p (\boldsymbol{U}^{(p)\mathsf{T}} \widetilde{\boldsymbol{V}}_1^{(p)})) \\ \vdots \\ \mathrm{mat}_1(\widetilde{\mathcal{H}}_s \times_1 \widetilde{\boldsymbol{V}}_s^{(1)} \times_2 (\boldsymbol{U}^{(2)\mathsf{T}} \widetilde{\boldsymbol{V}}_s^{(2)}) \cdots \times_p (\boldsymbol{U}^{(p)\mathsf{T}} \widetilde{\boldsymbol{V}}_s^{(p)})) \end{bmatrix}. \tag{22}$$

Computintg Eq. (22) only involves small matrices for non-temporal modes and avoids explicitly computing the large tensor $\widetilde{\mathcal{Y}}$ which requires $O(rD^{p-1}(T_e - T_s))$.

Lemma 5.2 (Time complexity of the temporal mode). *Computing the matricization of the temporal mode in Eq.* (22) *takes* $O((r^2D + r^{p+1})s + r^pL)$ *time where* $L = T_e - T_s$.

Proof sketch. Eq. (22) consists of three computations whose costs are as follows: for $n = 2, ..., p$ and $i = 1, ..., s$, (1) matrix multiplications $\boldsymbol{U}^{(n)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(n)}$, (2) tensor-matrix products between $\widetilde{\mathcal{H}}_i$ and the preceding results $\boldsymbol{U}^{(n)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(n)}$, and (3) tensor-matrix products between the preceding results and matrices $\widetilde{\boldsymbol{V}}_i^{(1)}$ take $O(r^2Ds)$, $O(r^{p+1}s)$, and $O(r^pL)$ time, respectively. Therefore, the complexity for computing Eq. (22) is $O((r^2D + r^{p+1})s + r^pL)$. □

[2]The mixed-product property means that $(\boldsymbol{A} \otimes \boldsymbol{B})(\boldsymbol{C} \otimes \boldsymbol{D}) = (\boldsymbol{AC}) \otimes (\boldsymbol{BD})$.

Matricization of the non-temporal modes. For a non-temporal mode $n = 2, \ldots, p$, the matricization is different from Eq. (22) because the concatenation is along the temporal mode. Nevertheless, we can still consider using the Kronecker product to rewrite the matricization in Eq. (2) as:

$$\mathrm{mat}_n(\widetilde{\mathcal{Y}}) \bigotimes_{m \neq n} \boldsymbol{U}^{(m)} = [\mathrm{mat}_n(\widetilde{\mathcal{Y}}_1), \ldots, \mathrm{mat}_n(\widetilde{\mathcal{Y}}_s)] \bigotimes_{m \neq n} \boldsymbol{U}^{(m)}. \tag{23}$$

Let $t_0 := 0$, and $t_i := t_{i-1} + (\tau_{v_i} - \sigma_{v_i})$ for $i = 1, \ldots, s$. Then, each hit node v_i corresponds to the subtensor $\widetilde{\mathcal{Y}}_{[t_{i-1}, t_i)}$. By splitting the temporal factor matrix as $\boldsymbol{U}^{(1)} = \begin{bmatrix} \boldsymbol{U}^{(1)}_{[t_0, t_1)} \\ \vdots \\ \boldsymbol{U}^{(1)}_{[t_{s-1}, t_s)} \end{bmatrix}$ and using the mixed-product property of the Kronecker product again, we can further rewrite Eq. (23) as

$$\sum_{i=1}^{s} \mathrm{mat}_n(\widetilde{\mathcal{Y}}_i) \Big(\boldsymbol{U}^{(n)}_{[t_{i-1}, t_i)} \otimes \bigotimes_{m \neq 1, n} \boldsymbol{U}^{(m)} \Big) \tag{24}$$

$$= \sum_{i=1}^{s} \widetilde{\boldsymbol{V}}_i^{(n)} \mathrm{mat}_n(\widetilde{\mathcal{H}}_i) \Big(\bigotimes_{m \neq n} \widetilde{\boldsymbol{V}}_i^{(m)\mathsf{T}} \Big) \Big(\boldsymbol{U}^{(1)}_{[t_{i-1}, t_i)} \otimes \bigotimes_{m \neq 1, n} \boldsymbol{U}^{(m)} \Big) \tag{25}$$

$$= \sum_{i=1}^{s} \widetilde{\boldsymbol{V}}_i^{(n)} \mathrm{mat}_n(\widetilde{\mathcal{H}}_i) \Big((\widetilde{\boldsymbol{V}}_i^{(1)\mathsf{T}} \boldsymbol{U}^{(1)}_{[t_{i-1}, t_i)}) \otimes \bigotimes_{m \neq 1, n} (\widetilde{\boldsymbol{V}}_i^{(m)\mathsf{T}} \boldsymbol{U}^{(m)}) \Big). \tag{26}$$

Finally, we rewrite the Kronecker product form in Eq. (26) back to the matricization form:

$$\begin{aligned} \sum_{i=1}^{s} \mathrm{mat}_n(\widetilde{\mathcal{H}}_i \times_1 (\boldsymbol{U}^{(1)\mathsf{T}}_{[t_{i-1}, t_i)} \widetilde{\boldsymbol{V}}_i^{(1)}) \\ \times_2 (\boldsymbol{U}^{(2)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(2)}) \cdots \times_{n-1} (\boldsymbol{U}^{(n-1)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(n-1)}) \times_n \widetilde{\boldsymbol{V}}_i^{(n)} \\ \times_{n+1} (\boldsymbol{U}^{(n+1)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(n+1)}) \cdots \times_p (\boldsymbol{U}^{(p)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(p)})). \end{aligned} \tag{27}$$

Computing Eq. (27) only involves small matrices for non-temporal modes and avoids explicitly computing the large tensor $\widetilde{\mathcal{Y}}$.

Lemma 5.3 (Time complexity of the non-temporal modes). *Computing the matricization of the non-temporal modes in Eq.* (27) *takes* $O(r^pDs + r^2L)$ *time where* $L := T_e - T_s$.

Proof sketch. Eq. (27) needs four computations: for $m = 2, ..., n-1, n+1, ..., p$ and $i = 1, ..., s$, (1) matrix multiplications $\boldsymbol{U}^{(1)\mathsf{T}}_{[t_{i-1}, t_i)} \widetilde{\boldsymbol{V}}_i^{(1)}$, (2) matrix multiplications $\boldsymbol{U}^{(m)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(m)}$, (3) tensor-matrix products between $\widetilde{\mathcal{H}}_i$ and the current results $\boldsymbol{U}^{(m)\mathsf{T}} \widetilde{\boldsymbol{V}}_i^{(m)}$, and (4) tensor-matrix products between the current results and matrices $\widetilde{\boldsymbol{V}}_i^{(n)}$ take $O(r^2L)$, $O(r^2Ds)$, $O(r^{p+1}s)$, and $O(r^pDs)$ time, respectively. Hence, the total complexity for computing Eq. (27) is $O(r^pDs + r^2L)$. □

Error analysis. We provide an error analysis of our Stitch algorithm in the following Proposition 5.4.

Proposition 5.4 (Error bound). *Let* $\mathcal{X}$ *be the concatenation of subtensors* $\mathcal{X}^{(i)}$ $(i = 1, \ldots, s)$*, and let* $\mathcal{Y}^{(i)}$ *denote the rank-r Tucker decomposition of* $\mathcal{X}^{(i)}$*. Suppose that alternating least squares are solved optimally, and that* $\mathcal{X}$ *is approximately low-rank (i.e.,* $\mathcal{X} = \mathcal{W} + \mathcal{E}$ *where* $\mathcal{W}$ *has Tucker rank* $\leq r$*, and* $\frac{\|\mathcal{E}\|_F}{\|\mathcal{X}\|_F} \leq \epsilon$ *for small* $\epsilon > 0$*). Then, the stitching algorithm finds a rank-r Tucker decomposition* $\mathcal{Y}$ *of* $\mathcal{X}$ *with reconstruction error* $\frac{\|\mathcal{X} - \mathcal{Y}\|_F}{\|\mathcal{X}\|_F} \leq O(\epsilon)$.

Proof sketch. Let $\mathcal{W}^{(i)}$ and $\mathcal{E}^{(i)}$ denote the part of $\mathcal{W}$ and $\mathcal{E}$ corresponding to the time range of $\mathcal{X}^{(i)}$, respectively. Then, $\|\mathcal{X}^{(i)} - \mathcal{Y}^{(i)}\|_{\mathrm{F}} \le \|\mathcal{X}^{(i)} - \mathcal{W}^{(i)}\|_{\mathrm{F}} = \|\mathcal{E}^{(i)}\|_{\mathrm{F}}$ for all i. Let $\widetilde{\mathcal{Y}}$ denote the concatenation of $\mathcal{Y}^{(i)}$. Thus,

$$\|\mathcal{X} - \widetilde{\mathcal{Y}}\|_{\mathrm{F}} \le \sqrt{\sum_{i=1}^{s} \|\mathcal{E}^{(i)}\|_{\mathrm{F}}^2} = \|\mathcal{E}\|_{\mathrm{F}} \le \epsilon \|\mathcal{X}\|_{\mathrm{F}}. \tag{28}$$

Since $\mathcal{Y}$ is the Tucker decomposition of $\widetilde{\mathcal{Y}}$, then

$$\begin{aligned}\|\mathcal{X} - \mathcal{Y}\|_{\mathrm{F}} &\le \|\mathcal{X} - \widetilde{\mathcal{Y}}\|_{\mathrm{F}} + \|\widetilde{\mathcal{Y}} - \mathcal{Y}\|_{\mathrm{F}} \le \epsilon\|\mathcal{X}\|_{\mathrm{F}} + \|\widetilde{\mathcal{Y}} - \mathcal{W}\|_{\mathrm{F}} \\ &\le \epsilon\|\mathcal{X}\|_{\mathrm{F}} + \|\widetilde{\mathcal{Y}} - \mathcal{X}\|_{\mathrm{F}} + \|\mathcal{X} - \mathcal{W}\|_{\mathrm{F}} \le 3\epsilon\|\mathcal{X}\|_{\mathrm{F}}. \quad \square\end{aligned}$$

Proposition 5.4 implies that the reconstruction error of our Stitch algorithm is very close to the error of computing Tucker decomposition from scratch via TuckerALS and does not depend on the number s of subtensors to be stitched.

6 TUCKET: OPERATIONS

Having described our design of the data structure in Section 4 and two core algorithms in Section 5, we next introduce how to answer Tucker decomposition range queries in Section 6.1 and how to maintain the tree after appending a tensor slice in Section 6.2.

6.1 Querying over a Time Range

A query over time range $[T_{\mathrm{s}}, T_{\mathrm{e}})$ asks to find the Tucker decomposition of the subtensor $\mathcal{X}_{[T_{\mathrm{s}},T_{\mathrm{e}})}$. Equipped with the two core algorithms in Section 5, we are ready to present the algorithm for answering the range query. First, we use Algorithm 1 w.r.t. the root of the stream segment tree to find an optimally pruned hit set $\mathcal{S} \subseteq \mathcal{L} \cup \mathcal{M}$. For each partial hit, we use Eq. (12) to compute its approximate Tucker decomposition $\widetilde{\mathcal{Y}}_i$ of the subtensor $\mathcal{X}_{[T_{\mathrm{s}},T_{\mathrm{e}})\cap[\sigma_{v_i},\tau_{v_i})}$; for each entire hit v_i, we retrieve its preprocessed Tucker decomposition $\widetilde{\mathcal{Y}}_i := \mathcal{Y}_{v_i}$. Same as before, here $\widetilde{\mathcal{Y}}_1, \ldots, \widetilde{\mathcal{Y}}_s$ are just symbols to refer to the Tucker decomposition products. Finally, we use Algorithm 2 to stitch the subtensor decompositions $\widetilde{\mathcal{Y}}_1, \ldots, \widetilde{\mathcal{Y}}_s$ into the Tucker decomposition $\mathcal{G} \times_1 \boldsymbol{U}^{(1)} \cdots \times_p \boldsymbol{U}^{(p)}$ of the queried subtensor $\mathcal{X}_{[T_{\mathrm{s}},T_{\mathrm{e}})}$.

The overall procedure can be illustrated using the example in Figure 3. When answering a range query $[T_{\mathrm{s}}, T_{\mathrm{e}}) = [1, 6)$ with $\theta = 0.7$, first we use Algorithm 1 to divide $[1, 6)$ into two subranges $[1, 4)$ (a partial hit of $[0, 4)$) and $[4, 6)$ (an entire hit). Since $[1, 4)$ is a partial hit of $[0, 4)$, then we use Eq. (12) to approximate the Tucker decomposition of $[1, 4)$. Finally, we use Algorithm 2 to stitch the decompositions of $\mathcal{X}_{[1,4)}$ and $\mathcal{X}_{[4,6)}$ into an approximate Tucker decomposition of $[1, 6)$.

Proposition 6.1 (Time complexity). *Given a query $[T_{\mathrm{s}}, T_{\mathrm{e}})$, TUCKET performs Recall and Stitch operations and takes $O(r^p Ds + r^{2p-2}(D+L) + \log T)$ time overall, where the query length $L := T_{\mathrm{e}} - T_{\mathrm{s}}$, and the hit set size $s = O(\log L)$.*

Proof sketch. For each iteration, there are five computations: (1) the Recall algorithm, (2) the matricization of the temporal mode, (3) the matricization of $p - 1 = O(1)$ non-temporal modes, (4) Singular value decomposition p times, and (5) the computation for updating core tensor. Therefore, the overall time complexity is $O(r^p Ds + r^{2p-2}D + r^{2p-2}L + \log T)$. $\square$

Algorithm 3 (Insert): Inserting a leaf node

Input: current node v; current time T; tensor slice $\mathcal{X}_T$

if $\sigma_v = T$ and $\tau_v = T + 1$ **then**
  preprocess the Tucker decomposition $\mathcal{Y}_v$ of $\mathcal{X}_T$
  $\mathcal{P} \leftarrow \mathcal{P} \setminus \{v\}$
  $\mathcal{L} \leftarrow \mathcal{L} \cup \{v\}$
else
  $\mu \leftarrow \left\lfloor \frac{\sigma_v + \tau_v}{2} \right\rfloor$
  if $T < \mu$ **then**
    if v does not have a left child u_1 **then**
      create a left child $u_1 \in \mathcal{P}$ with $[\sigma_{u_1}, \tau_{u_1}) \leftarrow [\sigma_v, \mu)$
    end if
    Insert$(u_1, T, \mathcal{X}_T)$
  else
    if v does not have a right child u_2 **then**
      create a right child $u_2 \in \mathcal{P}$ with $[\sigma_{u_2}, \tau_{u_2}) \leftarrow [\mu, \tau_v)$
    end if
    Insert$(u_2, T, \mathcal{X}_T)$
    if $\tau_v = T + 1$ **then**
      $\mathcal{Y}_v \leftarrow$ Stitch$(\{\mathcal{Y}_{u_1}, \mathcal{Y}_{u_2}\})$ via Algorithm 2
      $\mathcal{P} \leftarrow \mathcal{P} \setminus \{v\}$
      $\mathcal{M} \leftarrow \mathcal{M} \cup \{v\}$
    end if
  end if
end if

Algorithm 4 (Append): Appending a tensor slice

Input: current root node r; current time T; tensor slice $\mathcal{X}_T$
Output: the root after appending

if $T = 0$ **then**
  create a node $r' \in \mathcal{P}$ with $[\sigma_{r'}, \tau_{r'}) \leftarrow [0, 1)$
  $r \leftarrow r'$
else if $T = \tau_r$ **then**
  create a node $r' \in \mathcal{P}$ with $[\sigma_{r'}, \tau_{r'}) \leftarrow [0, 2\tau_r)$
  let r be the left child of r'
  $r \leftarrow r'$
end if
Insert$(r, T, \mathcal{X}_T)$ via Algorithm 3
return r

6.2 Appending a Tensor Slice

Appending a tensor slice $\mathcal{X}_T$ extends the timespan from $[0, T)$ to $[0, T + 1)$. To process this operation, we need to (i) maintain the stream segment tree structure and (ii) update the Tucker decomposition of nodes in the tree. Due to the special structure of the stream segment tree, we cannot maintain a logarithmic height via rotation operations like typical balanced search trees [22]. To address this issue, we leverage the fact that we have only the *appending* operation but no arbitrary insertion operations. Our key idea here is to insert not only a leaf node but also possibly a root node so as to maintain the logarithmic height. In the following, we will first describe the case where we do not need to insert a root node and then discuss the case where we need to insert a root node.

If the root node r is a placeholder node, then its time range $[0, \tau_r)$ includes $\mathcal{X}_T$. Thus, we can insert $\mathcal{X}_T$ into the tree. The insertion procedure is a recursive algorithm starting from the root r. Suppose that we are currently at a node v. Let u_1, u_2 denote the left and

right children of v, respectively. If $T < \tau_{u_1}$, then we insert $\mathcal{X}_T$ into the subtree rooted at u_1. Otherwise, we need to insert $\mathcal{X}_T$ into the subtree rooted at u_2. If either u_1 or u_2 does not exist yet, we create that node before insertion. After insertion, we revise the type of the node v. If $T = \tau_v - 1$, then the range $[\sigma_v, \tau_v)$ has been fully observed, so we let the node v become an intermediate node. The overall insertion procedure is formally presented in Algorithm 3.

Meanwhile, if the root node r is already an intermediate node, then its time range $[0, \tau_r)$ has been fully observed. In this case, we create a new placeholder node r' with time range $[0, \tau_{r'}) := [0, 2\tau_{r'})$, let node r be the left child of r, and let r' be the new root. Since the new root r' is now a placeholder node, then we use Algorithm 3 to insert $\mathcal{X}_T$ into the tree. The overall appending procedure is formally presented in Algorithm 4.

The overall procedure can be exemplified using Figure 3. Suppose that the current timespan is $[0, 8)$ (i.e., the current root is the node $\langle 8 \rangle$), and that we want to append the slice $\mathcal{X}_8$. Since the root node $\langle 8 \rangle$ is an intermediate node, then we create a new root $\langle 16 \rangle$ as an intermediate node and let $\langle 8 \rangle$ be its left child. Then, we insert $\mathcal{X}_8$ into the new root. As $T = 8$, we need to insert $\mathcal{X}_8$ into the right child of $\langle 16 \rangle$. Since $\langle 16 \rangle$ does not have a right child yet, we create a new intermediate node $\langle 17 \rangle$ as its right child and insert $\mathcal{X}_8$ into $\langle 17 \rangle$. We repeat this procedure until it reaches the leaf node $\langle 20 \rangle$. We preprocess the Tucker decomposition of $\mathcal{X}_{[8,9)}$ via Tucker-ALS and store it at node $\langle 20 \rangle$.

Theorem 6.2 (Logarithmic height). *If we use the algorithm above for timespan $[0, T)$, then it can construct a stream segment tree of height $\lceil \log_2 T \rceil + 1$.*

Proof sketch. We can use an induction on T to show that at time T, the number of new leaf nodes that can be inserted into the sub-tree rooted at each node v is $\max\{\tau_v - T, 0\}$. This implies that the number of leaf nodes in a stream segment tree of height h is greater than the number of leaf nodes of a perfect binary tree of height $h - 1$ and at most that of a perfect binary tree of height h. It follows that the height of a stream segment tree is $\lceil \log_2 T \rceil + 1$. □

Proposition 6.3 (Time complexity). *When appending a tensor slice $\mathcal{X}_T$, the amortized and worst-case time complexities are $O(rD^{p-1} + r^{2p-2}(D + \log T))$ and $O(rD^{p-1} + r^{2p-2}(D \log T + T))$, respectively.*

Proof sketch. We analyze the worst-case and amortized complexities of a stream segment tree. The amortized complexity is the time complexity of creating a stream segment tree divided by T. We preprocess T tensor slices of leaf nodes with the complexity $O(rD^{p-1}T)$. For intermediate nodes, we perform $O(T)$ stitch operations with the complexity $O(r^{2p-2}DT + r^{2p-2}T \log T)$. Hence, the amortized complexity is $O(rD^{p-1} + r^{2p-2}D + r^{2p-2} \log T)$. Appending a tensor slice $\mathcal{X}_T$ requires three computations: (1) performing Tucker-ALS of the tensor slice, (2) updating the non-temporal factor matrices, and (3) updating the temporal factor matrices. Hence, the worst-case complexity of appending a tensor slice $\mathcal{X}_T$ is $O(rD^{p-1} + r^{2p-2}D \log T + r^{2p-2}T)$ which is the sum of the complexities of the three computations. □

Table 3: Summary of real-world tensor time series datasets.

Dataset	#Entries	Shape	Modes
Air Quality	47M	$21000 \times 376 \times 6$	(time, location, air pollutant)
Traffic	212M	$2033 \times 1084 \times 96$	(time, frequency, sensor)
US Stock	739M	$2000 \times 4347 \times 85$	(time, company, stock feature)
KR Stock	875M	$3000 \times 3432 \times 85$	(time, company, stock feature)

7 IMPLEMENTATION

Since the bottleneck of Tucker decomposition is the tensor numerical operations, we speed up these computations using a graphical processing unit (GPU). Though originally designed to accelerate computer graphics and image processing, modern GPUs are powerful in parallelizing dense numerical operations in general scientific computing. To develop a prototype of our TUCKET that is compatible across various platforms, we choose the PyTorch [53] CUDA [45] library to set up and run GPU operations. Although Python execution is relatively slow compared with many other programming languages, it does not affect the overall performance much because the tensor operations are typically much more expensive than Python execution.

8 EXPERIMENTAL EVALUATION

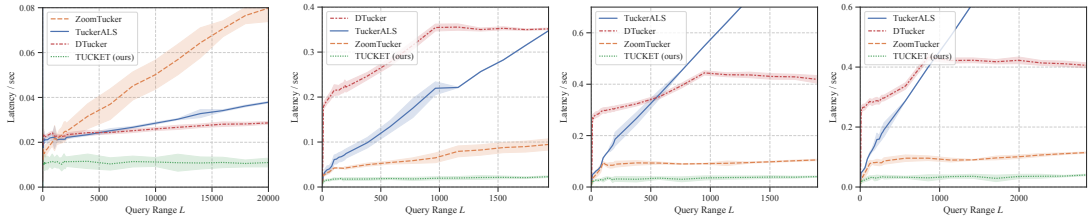
In this section, we evaluate our TUCKET by comparing it with state-of-the-art methods on four large-scale real-world tensor time series datasets. We summarize our evaluation results from our experiments as follows:

(i) TUCKET consistently achieves the lowest latency over all query ranges on real-world tensor time series data **(Section 8.3)**.
(ii) We empirically demonstrate that TUCKET constructs the whole tree in nearly linear time and consumes nearly linear space in total **(Section 8.3)**.
(iii) The reconstruction error of TUCKET is much smaller than D-Tucker and Zoom-Tucker and is comparable with the brute-force method Tucker-ALS **(Section 8.4)**.
(iv) Our new stitching algorithm is more GPU-parallelizable and more numerically stable than that of Zoom-Tucker **(Section 8.5)**.
(v) The pruning threshold $\theta = 0.7$ can achieve both high efficiency and accuracy in our experiments **(Section 8.5)**.

8.1 Experimental Settings

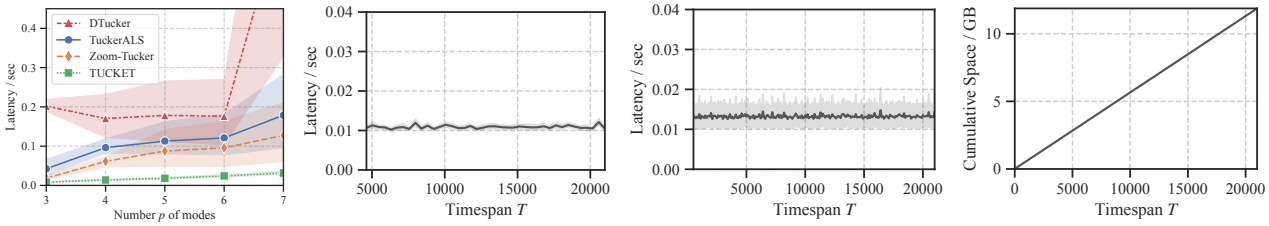
Datasets. We use four large-scale real-world tensor time series datasets, which are summarized in Table 3. **(D1)** Air Quality data is represented as 3-way tensor time series (time, location, air pollutant). It is collected from the Air Korea[3] website. **(D2)** Traffic data[4] [56] is 3-way tensor time series (time, frequency, sensor) representing a collection of traffic volume measurements around Melbourne. **(D3 & D4)** We use daily stock features (e.g., prices, volumes, and technical indicators) on the U.S. and Korean stock markets, respectively, to build 3-way tensor time series (time, company, stock feature). **(D5)** To evaluate the scalability w.r.t. the number p of modes, we generate synthetic tensors with the following sizes: (1)

[3] https://www.airkorea.or.kr/web/
[4] https://github.com/florinsch/BigTrafficData



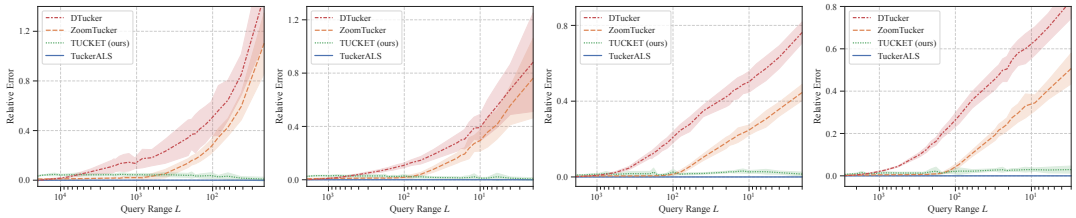
(a) Air Quality (b) Traffic (c) US Stock (d) KR Stock

Figure 4: Comparison in the latency of range queries. Our TUCKET (green dotted line) consistently achieves the lowest latency for all query ranges while the performance of other methods varies drastically across datasets.



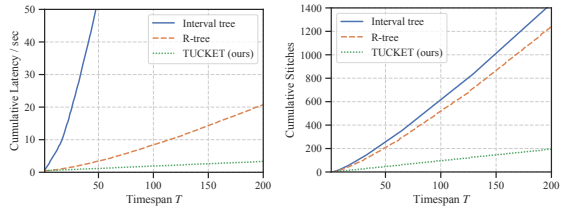
(a) Latency v.s. p (b) Latency v.s. T (c) Latency of Append (d) Cumulative space

Figure 5: Scalability tests



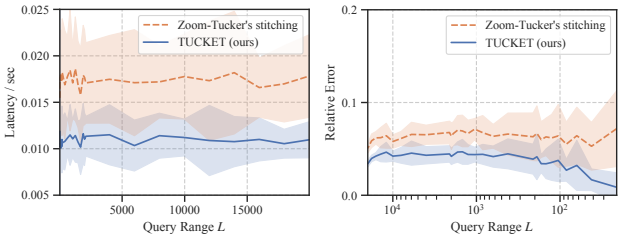
(a) Air Quality (b) Traffic (c) US Stock (d) KR Stock

Figure 6: Comparison in the relative error of range queries. In contrast to D-Tucker and Zoom-Tucker, our TUCKET (green dotted line) consistently achieves comparable error with that of Tucker-ALS for all query ranges.



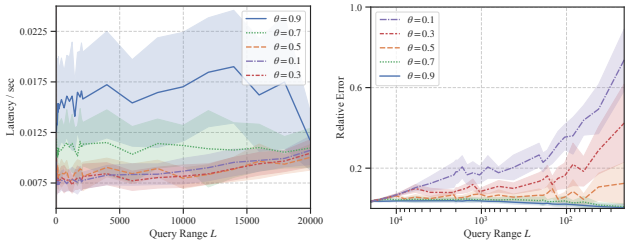
(a) Cumulative latency (b) Cumulative stitches

Figure 7: Comparison with the interval tree and the R-tree in terms of the Append operation.



(a) Latency v.s. stitching algorithm (b) Relative error v.s. stitching algorithm

Figure 8: Comparison with Zoom-Tucker's stitching



(a) Latency v.s. pruning threshold θ (b) Relative error v.s. pruning threshold θ

Figure 9: Latency and relative error v.s. pruning threshold

$1000\times1000\times1000$, (2) $1000\times100\times100\times100$, (3) $1000\times100\times100\times10\times10$, (4) $1000\times100\times10\times10\times10\times10$, (5) $1000\times10\times10\times10\times10\times10\times10$.

Evaluation metrics. Regarding efficiency, since no baseline methods support batch operations, we do not report the throughput. Instead, we report the *latency* (in seconds) of each operation.

Regarding accuracy, for each range query $[T_s, T_e)$, we report the *relative error* between each method A and Tucker-ALS:

$$\frac{\|\mathcal{X}_{[T_s,T_e)} - \mathcal{Y}_A\|_F}{\|\mathcal{X}_{[T_s,T_e)} - \mathcal{Y}_{\text{Tucker-ALS}}\|_F} - 1, \tag{29}$$

where $\mathcal{Y}_{\text{Tucker-ALS}}$ and $\mathcal{Y}_A$ denote the reconstructed tensors by Tucker-ALS and the method A, respectively.

Platform. We conduct all experiments in the Docker environment on an Ubuntu 20.04.6 LTS cloud server with an Intel Xeon CPU @ 2.00 GHz and an NVIDIA P100 16GB GPU.

Evaluation framework. When comparing the performance of TUCKET with baselines, since no baseline methods support range queries and stream updates simultaneously, we first construct the data structures of all methods and then run range queries. The appending operation is evaluated separately in Section 8.3.

Hyperparameters. For alternating least squares, we set the maximum number of iterations to 20 and the tolerance to 0.01. We set a target size to 10, except that we set the target size to 5 when the size of a non-temporal mode is smaller than 10. For TUCKET, we use the pruning threshold $\theta = 0.7$ for all datasets.

8.2 Baseline Methods

To evaluate the effectiveness of our proposed TUCKET, we compare it with state-of-the-art methods for Tucker decomposition. The baselines are described below.

- **Tucker-ALS** [62] utilizes alternating least squares optimization to compute Tucker decomposition, thus achieving the best reconstruction error. However, in our setting, Tucker-ALS is essentially a brute-force algorithm, unable to support range queries or streaming updates efficiently.
- **D-Tucker** [29] decomposes compressed matrices sliced from the input tensor and further updates the factors and cores iteratively, which enables a fast and memory-efficient decomposition of large and dense tensors. Notably, the iteration phase of D-Tucker facilitates its seamless adaptation to tasks involving stream updates. However, it does not support range queries. For a fair comparison, we extract the sub-tensor corresponding to the range query from the preprocessed slices of D-Tucker.
- **Zoom-Tucker** [30] supports Tucker decomposition range queries via block-wise preprocessing and by merging block results during the query pahse. While Zoom-Tucker demonstrates efficient performance on range queries, it faces limitations in supporting stream updates. For a fair comparison, we use block size $\frac{T}{2\lceil \log_2 T \rceil}$ in Zoom-Tucker, which ensures that the maximum number of blocks is comparable with the maximum size of the hit set of TUCKET. Besides that, since we have enhanced the stitching algorithm of subtensor decompositions in Section 5.2, we also use our stitching algorithm in Zoom-Tucker for a fair comparison.

8.3 Efficiency & Scalability Tests

In this subsection, we evaluate the time and space efficiencies of TUCKET in range query answering and appending tensor slices.

Efficiency of range query answering w.r.t. query length L. As shown in Figure 4, TUCKET consistently achieves the lowest latency compared to all baseline methods, with its latency remaining almost stable regardless of the query range. In contrast, the latency of baselines increases dramatically as the query range expands. On the Air Quality dataset, TUCKET exhibits an average latency of 0.011 seconds on a GPU, significantly smaller than that of all other baselines. In Traffic, US Stock, and KR Stock datasets, although Zoom-Tucker and TUCKET demonstrate similar trends, our TUCKET outperforms Zoom-Tucker by a considerable margin.

Scalability w.r.t. number p of modes. We measure latencies of TUCKET and baselines by varying the number p of modes on synthetic tensor time series under query lengths 98, 192, and 384 and target rank $r = 5$. As shown in Figure 5a, TUCKET is still the most efficient method under a higher number p of modes. TUCKET consistently achieves the lowest latency compared to all baselines across all query lengths and number of modes. TUCKET achieves 5.9 times lower latency than the second-fastest method, Zoom-Tucker, when the number of modes p is 7 and the query length is 384. This highlights the superiority of our recall and stitching algorithms in terms of the scalability w.r.t. the number p of modes.

Efficiency of query answering w.r.t. timespan T. We keep the query length L the same while appending new slices between queries to increase T. The results of latency v.s. timespan T on Air Quality are shown in Figure 5b. We can see that the latency of range queries almost has no notable change. This validates our time complexity where the dominant term $O(r^p D \log L)$ scales with only L and does not explicitly depend on T.

Efficiency of appending tensor slices. We plot the latency of Append on the Air Quality dataset v.s. the timespan T in Figure 5c. The results validate the amortized time complexity $O(rD^{p-1} + r^{2p-2}(D + \log T))$ of Append. Notably, Figure 5c shows that the time complexity is nearly constant w.r.t. T. This is because D is typically much greater than $\log T$ as long as T is not too large.

Space consumption of our TUCKET. We plot the cumulative space usage on the Air Quality dataset v.s. the timespan T in Figure 5d. The results validate the space complexity $O((rD + r^p)T + rT \log T)$ of our TUCKET. Notably, Figure 5d shows that the space complexity is nearly linear w.r.t. T. This is because D is typically much greater than $\log T$ as long as T is not too large.

8.4 Accuracy of Range Query Answering

We measure relative errors with respect to time range queries. Figure 6 shows the results. TUCKET consistently has comparable errors to Tucker-ALS which performs Tucker decomposition from scratch. As $T_e - T_s$ decreases, TUCKET and Tucker-ALS have little variation in errors, while the errors of D-Tucker and Zoom-Tucker increase dramatically. This is because TUCKET effectively preserves information for short time ranges using the proposed stream segment tree in the preprocessing phase, whereas D-Tucker and Zoom-Tucker compromise the accuracy for short time ranges in the preprocessing phase.

8.5 Ablation Studies

To further understand TUCKET, we conduct the following ablation studies: (i) comparing our stream segment tree with other data structures, (ii) comparing our new stitching algorithm with that of Zoom-Tucker, and (iii) analyzing the effect of pruning threshold θ.

Comparison of data structures. We compare our stream segment tree with the interval tree [54] (using AVL balancing [3]) and the (1-dimensional) R-tree [23] (using B-tree balancing [8]) in terms of the Append operation. We report in Figure 7 the cumulative latency and

the cumulative number of Stitch operations in Append because the Stitch operation is the bottleneck during Append. We see that our TUCKET achieves 83.5 times and 3.4 times speedup over the interval tree and the R-tree, respectively; they need $O(\log T)$ Stitch operations per Append because every node on the path from the inserted node to the root needs a Stitch. In contrast, our proposed *stream segment tree* needs only 1 Stitch (amortized) per Append because our placeholder nodes need no Stitch.

Comparison of stitching algorithms. We compare our stitching algorithm with Zoom-Tucker's stitching algorithm in terms of latency and relative error by replacing our Stitch (Algorithm 2) with Zoom-Tucker's stitching algorithm. In Figure 8a, our stitching algorithm achieves lower latency than the stitching algorithm of Zoom-Tucker. This result implies that our stitching algorithm is more GPU-parallelizable than Zoom-Tucker's stitching. Figure 8b shows that our stitching algorithm has lower relative errors than Zoom-Tucker's stitching. The error gap widens as the query range decreases since Zoom-Tucker's stitching needs to compute the inverse for rank-deficient matrices in short query ranges.

Effect of the pruning threshold θ. We test the model efficiency and the relative error of TUCKET with respect to the pruning threshold. We conduct the experiment on the Air Quality dataset. Figure 9b shows the relative error of TUCKET with respect to the query range. As we can see, as the θ value increases, the relative error consistently decreases for all query ranges. Meanwhile, the difference between $\theta = 0.7$ and $\theta = 0.9$ is tiny. When comparing $\theta = 0.7$ and $\theta = 0.9$, $\theta = 0.7$ is better since it has lower latency than $\theta = 0.9$ over all the query ranges as shown in Figure 9a. Therefore, $\theta = 0.7$ is the best choice for balancing efficiency and accuracy as it allows TUCKET to avoid pruning-induced accuracy loss and handle a similar number of blocks.

8.6 Case Study

To demonstrate the application of TUCKET, we present a case study with Air Quality data. We run TUCKET for three time ranges (March 2015, March 2016, and March 2017) to obtain the location factor matrices $\boldsymbol{U}^{(2)} \in \mathbb{R}^{376 \times r}$ of each time range. Here, the i-th row vector of $\boldsymbol{U}^{(2)}$ represents the air pollution patterns of the i-th location. Then, we perform K-means clustering w.r.t. the row vectors of $\boldsymbol{U}^{(2)}$ to find 5 clusters of the locations.

Clustering results are shown in Figure 1. We can see that some regions consistently exhibit similar clustering patterns across all years while some other regions have varying clustering patterns depending on the year. Regions (A) and (D) had similar patterns in March 2015 and 2016, but their patterns changed in March 2017. Meanwhile, regions (B), (C), and (E) had similar clustering patterns across all years. Region (F) has slightly different clustering patterns across the years. Air pollution patterns can be attributed to changes in weather conditions, the occurrence of yellow dust and fine dust, industrial activity, and traffic volume. TUCKET enables researchers and practitioners to explore diverse time ranges on Air Quality data efficiently and accurately.

9 RELATED WORK

Tensor decomposition. Tensor decompositions have been widely used for analyzing real-world tensors. Due to their large sizes, developing efficient and scalable CP methods [6, 7, 15, 27, 42, 70] and Tucker methods [29, 31, 46, 47, 61] have attracted considerable interest. The vast majority of these works focus on decomposing the entire tensor from scratch and thus cannot efficiently answer range queries. Although Zoom-Tucker [30] supports efficient range queries, it has an unwilling tradeoff between accuracy and efficiency due to a fixed block size. In addition to the above methods for dense tensors, there are plenty of tensor decomposition methods for sparse tensors where only a few entries are nonzeros. Many works [34, 35, 41, 48–50, 52, 57, 58, 67] have developed scalable tensor decomposition for sparse tensors in parallel and distributed systems. Numerous tensor decomposition methods [14, 18, 43, 60, 66] utilize neural networks for predicting unobserved entries of sparse tensors. However, they do not support range queries either.

Time series databases. Time series databases utilize various data structures to handle time-series data. Many time series databases, including KairosDB [2], Apache IoTDB [64], and LittleTable [55], are designed based on log-structured merge tree (LSM-tree) [51] for managing time-series data. Other time series databases, including InfluxDB [1], BTrDB [5], and EdgeDB [68], utilize their own tree structures to manage massive time series data. However, none of these works considers tensor time series or supports tensor decomposition range queries, which is harder than the simple queries supported by existing time series databases.

10 CONCLUSION & FUTURE WORK

In this paper, we have proposed a tensor time series data structure called TUCKET that can efficiently and accurately support range queries of Tucker decomposition and stream updates to the tensor time series. To the best of our knowledge, our TUCKET is a first-of-its-kind method that creatively generalizes the segment tree to the Tucker decomposition range query problem with stream updates. We provide (i) fine-grained theoretical guarantees and (ii) time and space complexities for our proposed method. We also experimentally show that TUCKET consistently achieves the best efficiency and accuracy in time range query answering.

Future work includes extending TUCKET to sparse tensors and to other tensor decompositions such as CANDECOMP/PARAFAC and PARAFAC2 decompositions, and supporting multi-mode range queries via nested segment trees [63].

ACKNOWLEDGMENTS

This work was supported by NSF (2316233), NIFA (2020-67021-32799), and IBM–Illinois Discovery Accelerator Institute. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. Jun-Gi Jang was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (RS-2023-00238596).

REFERENCES

[1] [n.d.]. InfluxDB. https://www.influxdata.com/ Accessed: 2024-02-22.

[2] [n.d.]. KairosDB. https://kairosdb.github.io/ Accessed: 2024-02-26.

[3] Georgii M. Adelson-Velskii and Evgenii Landis. 1962. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences* 146 (1962), 263–266.

[4] Dawon Ahn, Sangjun Son, and U Kang. 2020. Gtensor: Fast and accurate tensor analysis system using GPUs. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 3361–3364.

[5] Michael P Andersen and David E Culler. 2016. {BTrDB}: Optimizing Storage System Design for Timeseries Processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 39–52.

[6] Grey Ballard, Koby Hayashi, and Kannan Ramakrishnan. 2018. Parallel nonnegative CP decomposition of dense tensors. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 22–31.

[7] Casey Battaglino, Grey Ballard, and Tamara G Kolda. 2018. A practical randomized CP tensor decomposition. *SIAM J. Matrix Anal. Appl.* 39, 2 (2018), 876–901.

[8] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 107–141.

[9] Michael A. Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Proceedings 4*. Springer, 88–94.

[10] Jon Louis Bentley. 1977. Algorithms for Klee's rectangle problems. (1977).

[11] Bobbi Jo Broxson. 2006. The Kronecker product.

[12] Xiaochun Cao, Xingxing Wei, Yahong Han, and Dongdai Lin. 2014. Robust face clustering via tensor decomposition. *IEEE Transactions on Cybernetics* 45, 11 (2014), 2546–2557.

[13] J Douglas Carroll and Jih-Jie Chang. 1970. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart–Young" decomposition. *Psychometrika* 35, 3 (1970), 283–319.

[14] Huiyuan Chen and Jing Li. 2020. Neural tensor model for learning multi-aspect factors in recommender systems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 2020.

[15] Dehua Cheng, Richard Peng, Yan Liu, and Ioakeim Perros. 2016. SPALS: Fast alternating least squares via implicit leverage scores sampling. *Advances in neural information processing systems* 29 (2016).

[16] Alan Cobham. 1966. The recognition problem for the set of perfect squares. In *7th Annual Symposium on Switching and Automata Theory*. IEEE, 78–87.

[17] Miguel Ramos de Araujo, Pedro Manuel Pinto Ribeiro, and Christos Faloutsos. 2017. Tensorcast: Forecasting with context using coupled tensors (best paper award). In *ICDM*. IEEE, 71–80.

[18] Jicong Fan. 2021. Multi-mode deep matrix and tensor factorization. In *international conference on learning representations*.

[19] Hadi Fanaee-T and João Gama. 2016. Tensor-based anomaly detection: An interdisciplinary survey. *Knowledge-Based Systems* 98 (2016), 130–147.

[20] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* 24, 3 (1994), 327–336.

[21] Walter Gander. 1980. *Algorithms for the QR decomposition.* Technical Report 02. Eidgenoessische Technische Hochschule, Zuerich. 1251–1268 pages.

[22] Leonidas J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. IEEE, 8–21.

[23] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[24] Richard A Harshman. 1970. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multimodal factor analysis. (1970).

[25] Martin Hellman. 1980. A cryptanalytic time–memory trade-off. *IEEE Transactions on Information Theory* 26, 4 (1980), 401–406.

[26] Heng Huang, Chris Ding, Dijun Luo, and Tao Li. 2008. Simultaneous tensor subspace selection and clustering: The equivalence of high order SVD and k-means clustering. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 327–335.

[27] Shengyu Huang, K Selçuk Candan, and Maria Luisa Sapino. 2016. BICP: block-incremental CP decomposition with update sensitive refinement. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1221–1230.

[28] Jun-Gi Jang, Dongjin Choi, Jinhong Jung, and U Kang. 2018. Zoom-SVD: Fast and memory efficient method for extracting key patterns in an arbitrary time range. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 1083–1092.

[29] Jun-Gi Jang and U Kang. 2020. D-Tucker: Fast and memory-efficient tucker decomposition for dense tensors. In *2020 IEEE 36th International Conference on Data Engineering*. IEEE, 1850–1853.

[30] Jun-Gi Jang and U Kang. 2021. Fast and memory-efficient Tucker decomposition for answering diverse time range queries. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 725–735.

[31] Jun-Gi Jang and U Kang. 2023. Static and streaming tucker decomposition for dense tensors. *ACM Transactions on Knowledge Discovery from Data* 17, 5 (2023), 1–34.

[32] ByungSoo Jeon, Inah Jeon, Lee Sael, and U Kang. 2016. SCouT: Scalable coupled matrix-tensor factorization—Algorithm and discoveries. In *2016 IEEE 32nd International Conference on Data Engineering*. IEEE, 811–822.

[33] Inah Jeon, Evangelos E Papalexakis, Uksong Kang, and Christos Faloutsos. 2015. HaTen2: Billion-scale tensor decompositions. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1047–1058.

[34] Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[35] Oguz Kaya and Bora Uçar. 2016. High performance parallel algorithms for the tucker decomposition of sparse tensors. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 103–112.

[36] Henk A. L. Kiers, Jos MF Ten Berge, and Rasmus Bro. 1999. PARAFAC2—Part I. A direct fitting algorithm for the PARAFAC2 model. *Journal of Chemometrics: A Journal of the Chemometrics Society* 13, 3-4 (1999), 275–294.

[37] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv:1511.06530* (2015).

[38] Tamara G. Kolda and Brett W. Bader. 2009. Tensor decompositions and applications. *SIAM Rev.* 51, 3 (2009), 455–500.

[39] Danai Koutra, Evangelos E Papalexakis, and Christos Faloutsos. 2012. Tensorsplat: Spotting latent anomalies in time. In *2012 16th Panhellenic Conference on Informatics*. IEEE, 144–149.

[40] Timothée Lacroix, Guillaume Obozinski, and Nicolas Usunier. 2019. Tensor decompositions for temporal knowledge base completion. In *International Conference on Learning Representations*.

[41] Hao Li, Zixuan Li, Kenli Li, Jan S. Rellermeyer, Lydia Y. Chen, and Keqin Li. 2021. SGD_Tucker: A novel stochastic optimization strategy for parallel sparse Tucker decomposition. *IEEE Trans. Parallel Distributed Syst.* 32, 7 (2021), 1828–1841.

[42] Xinsheng Li, Shengyu Huang, K Selçuk Candan, and Maria Luisa Sapino. 2016. 2PCP: Two-phase CP decomposition for billion-scale dense tensors. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 835–846.

[43] Hanpeng Liu, Yaguang Li, Michael Tsang, and Yan Liu. 2019. Costco: A neural tensor completion model for sparse tensors. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 324–334.

[44] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. 2012. Tensor completion for estimating missing values in visual data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 1 (2012), 208–220.

[45] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *Proceedings of the 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. IEEE, 836–838.

[46] Linjian Ma and Edgar Solomonik. 2021. Fast and accurate randomized algorithms for low-rank tensor decompositions. *Advances in Neural Information Processing Systems* 34 (2021), 24299–24312.

[47] Osman Asif Malik and Stephen Becker. 2018. Low-rank tucker decomposition of large tensors using tensorsketch. *Advances in Neural Information Processing Systems* 31.

[48] Jinoh Oh, Kijung Shin, Evangelos E Papalexakis, Christos Faloutsos, and Hwanjo Yu. 2017. S-hot: Scalable high-order tucker decomposition. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. 761–770.

[49] Sejoon Oh, Namyong Park, Jun-Gi Jang, Lee Sael, and U Kang. 2019. High-performance tucker factorization on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2237–2248.

[50] Sejoon Oh, Namyong Park, Sael Lee, and Uksong Kang. 2018. Scalable tucker factorization for sparse tensors-algorithms and discoveries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1120–1131.

[51] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[52] Evangelos E Papalexakis, Christos Faloutsos, and Nicholas D Sidiropoulos. 2012. Parcube: Sparse parallelizable tensor decompositions. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part I 23*. Springer, 521–536.

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, Vol. 32.

[54] Franco P. Preparata and Michael Ian Shamos. 1985. *Computational Geometry: An Introduction.* Springer-Verlag.

[55] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 125–138.

[56] Florin Schimbinschi, Xuan Vinh Nguyen, James Bailey, Chris Leckie, Hai Vu, and Rao Kotagiri. 2015. Traffic forecasting in complex urban networks: Leveraging big data and machine learning. In *Proceedings of the 2015 IEEE International Conference on Big Data*. IEEE, 1019–1024.

[57] Tianyi Shi, Maximilian Ruth, and Alex Townsend. 2023. Parallel algorithms for computing the tensor-train decomposition. *SIAM Journal on Scientific Computing* 45, 3 (2023), C101–C130.

[58] Shaden Smith and George Karypis. 2017. Accelerating the tucker decomposition with compressed sparse tensors. In *European Conference on Parallel Processing*. Springer, 653–668.

[59] Alessandro Spelta. 2017. Financial market predictability with tensor decomposition and links forecast. *Applied Network Science* 2, 1 (2017), 1–15.

[60] Conor Tillinghast, Shikai Fang, Kai Zhang, and Shandian Zhe. 2020. Probabilistic neural-kernel tensor decomposition. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 531–540.

[61] Charalampos E Tsourakakis. 2010. Mach: Fast randomized tensor decompositions. In *Proceedings of the 2010 SIAM international conference on data mining*. SIAM, 689–700.

[62] Ledyard R Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

[63] V. K. Vaishnavi. 1982. Computing point enclosures. *IEEE Trans. Comput.* 100, 1 (1982), 22–29.

[64] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[65] Hongcheng Wang and Narendra Ahuja. 2008. A tensor approximation approach to dimensionality reduction. *International Journal of Computer Vision* 76 (2008), 217–229.

[66] Xian Wu, Baoxu Shi, Yuxiao Dong, Chao Huang, and Nitesh V Chawla. 2019. Neural tensor factorization for temporal interaction learning. In *Proceedings of the Twelfth ACM international conference on web search and data mining*. 537–545.

[67] Fan Yang, Fanhua Shang, Yuzhen Huang, James Cheng, Jinfeng Li, Yunjian Zhao, and Ruihao Zhao. 2017. Lftf: A framework for efficient tensor analytics at scale. *Proceedings of the VLDB Endowment* 10, 7 (2017), 745–756.

[68] Yang Yang, Qiang Cao, and Hong Jiang. 2019. EdgeDB: An efficient time-series database for edge computing. *IEEE Access* 7 (2019), 142295–142307.

[69] Lina Yao, Quan Z Sheng, Yongrui Qin, Xianzhi Wang, Ali Shemshadi, and Qi He. 2015. Context-aware point-of-interest recommendation using tensor factorization with social regularization. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1007–1010.

[70] Zeliang Zhang, Zhuo Liu, Susan Liang, Zhiyuan Wang, Yifan Zhu, Chen Ding, and Chenliang Xu. 2023. Scalable CP Decomposition for Tensor Learning using GPU Tensor Cores. *arXiv preprint arXiv:2311.13693* (2023).

[71] Lizhuang Zhao and Mohammed J Zaki. 2005. Tricluster: An effective algorithm for mining coherent clusters in 3D microarray data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. 694–705.

A PRELIMINARIES ON BASIC TENSOR OPERATIONS

A *p-way tensor* can be viewed as a p-dimensional array. Each dimension of a tensor is called a *mode*. To distinguish tensors from matrices and scalars, we use a bold calligraphic font for tensors, a bold italic font for matrices, and non-bold fonts for scalars. The indices of vectors, matrices, and tensors start from 0.

The *vectorization* of a tensor $\mathcal{X} \in \mathbb{R}^{D_1 \times \cdots \times D_p}$ is a column vector where each entry $\mathcal{X}_{i_1,\ldots,i_p}$ goes to $(\text{vec}(\mathcal{X}))_j$ at

$$j = \sum_{n=1}^{p} i_n \prod_{m=n+1}^{p} D_m. \tag{30}$$

The *mode-n matricization* of a tensor $\mathcal{X} \in \mathbb{R}^{D_1 \times \cdots \times D_n \times \cdots \times D_p}$ is a matrix $\text{mat}_n(\mathcal{X}) \in \mathbb{R}^{D_n \times (D_1 \cdots D_{n-1} D_{n+1} \cdots D_p)}$ defined by stacking the mode-n slices of $\mathcal{X}$ into a matrix:

$$\text{mat}_n(\mathcal{X}) := \begin{bmatrix} (\text{vec}(\mathcal{X}_{:,\ldots,:,0,:,\ldots,:}))^\top \\ \vdots \\ (\text{vec}(\mathcal{X}_{:,\ldots,:,D_n-1,:,\ldots,:}))^\top \end{bmatrix}. \tag{31}$$

The *mode-n product* of a tensor $\mathcal{G} \in \mathbb{R}^{r_1 \times \cdots \times r_n \times \cdots \times r_p}$ with a matrix $\boldsymbol{U} \in \mathbb{R}^{D \times r_n}$ is a tensor $\mathcal{G} \times_n \boldsymbol{U} \in \mathbb{R}^{r_1 \times \cdots \times r_{n-1} \times D \times r_{n+1} \times \cdots \times r_p}$ defined by

$$\text{mat}_n(\mathcal{G} \times_n \boldsymbol{U}) := \boldsymbol{U} \cdot \text{mat}_n(\mathcal{G}), \tag{32}$$

where $\cdot$ denotes the matrix multiplication. Besides that, the *Frobenius norm* $\|\cdot\|_\text{F}$ of a tensor $\mathcal{X} \in \mathbb{R}^{D_1 \times \cdots \times D_p}$ is the square root of the sum of the squares of all its entries:

$$\|\mathcal{X}\|_\text{F} := \sqrt{\sum_{i_1=0}^{D_1-1} \cdots \sum_{i_p=0}^{D_p-1} \mathcal{X}_{i_1,\ldots,i_p}^2}. \tag{33}$$

We refer readers to [38] for further details on tensor operations.

B PROOFS

B.1 Proof of Lemma 4.1

Given a stream segment tree of height h and any range query $[T_s, T_e)$, let $[\sigma_v, \tau_v)$ be the time range of the root node v. Then, when we consider how to find a hit set of range query, there are only three different cases:

(1) The range query $[T_s, T_e)$ is a prefix of the time range $[\sigma_v, \tau_v)$, i.e., $T_s = \sigma_v$.
(2) The range query $[T_s, T_e)$ is a suffix of the time range $[\sigma_v, \tau_v)$, i.e., $T_e = \tau_v$.
(3) The range query $[T_s, T_e)$ is neither a prefix nor a suffix of the time range $[\sigma_v, \tau_v)$, i.e., $T_s \neq \sigma_v$ and $T_e \neq \tau_v$.

For case (1), let $f_1(h)$ be the size of a hit set on a stream segment tree of height h. It is evident that $f_1(1) = 1$ since the root node constitutes an entire hit under this circumstance. When $h > 1$, we need to consider whether the range query $[T_s, T_e)$ can be entirely contained within the range of the left children. Let u represent the left child of the node v. If the query range can be entirely entained, i.e., $T_e \leq \tau_u$, then we should recursively search the hit set within the sub-tree with u as the new root node, implying $f_1(h) \leq f_1(h-1)$. Otherwise, we simply put the left child u into the hit set and recursively search within the right sub-tree, which implies $f_1(h) \leq f_1(h-1) + 1$. To sum up, we can easily have the following expression

$$f_1(h) \leq f_1(h-1) + 1 \tag{34}$$

Since we have the base case that $f_1(1) = 1$, it is easy to prove that

$$f_1(h) \leq h. \tag{35}$$

Similarly, we can also prove that $f_2(h) \leq h$, where $f_2(h)$ represents the size of a hit set for case (2).

Now consider case (3) where $f_3(h)$ is denoted as the size of the hit set. Suppose u is the left child of the node v with the time range of $[\sigma_u, \tau_u)$, if the range query spans the time range of both the left child and the right child, i.e., $T_s < \tau_u < T_e$, then we are supposed to search for the hit set in both the left sub-tree and the right sub-tree. Please note that in this situation, the new range query will be either the prefix or the suffix for those sub-trees, i.e., $f_3(h) \leq f_1(h-1) + f_2(h-1)$. Conversely, if the range query are limited within the time range of one sub-tree, then the hit set should be searched within the sub-tree, i.e., $f_3(h) \leq f_3(h-1)$. To summarize, we have the following expression

$$f_3(h) \leq f_1(h-1) + f_2(h-1) \leq 2(h-1). \tag{36}$$

It follows from the three cases that there exists a hit set of size

$$\leq \max\{f_1(h), f_2(h), f_3(h)\} = \max\{2(h-1), h\} = \max\{2(h-1), 1\}. \tag{37}$$

B.2 Proof of Theorem 4.2

Using the algorithm in Section 6.2 to append the tensor slices $\mathcal{X}_0, \ldots, \mathcal{X}_{T-1}$ one by one, we can build a stream segment tree over $[0, T)$. By Theorem 6.2, this stream segment tree has height $\lceil \log_2 T \rceil + 1$.

B.3 Proof of Proposition 4.3

In a stream segment tree over the range $[0, T)$, there are $O(T)$ nodes each of which has $p-1$ non temporal factor matrices of the size $O(rD)$ and a core tensor of the size $O(r^p)$. In addition, at each level of the tree, the sum of the sizes of temporal factor matrices is $O(rT)$. Therefore, the space complexity of the stream segment tree is $O((prD + r^p)T + rT \log T)$.

B.4 Proof of Theorem 5.1

Optimal hit sets. We aim to prove that Algorithm 1 successfully finds an optimal solution for Eq. (13). Mathematically, we want to show that, if there exists a hit set that satisfies the conditions specified in Eq. (14) and (15), then the size of this hit set must be greater than or equal to the size of the hit set generated by Algorithm 1.

First, due to the top-down nature of Algorithm 1, for any node v in the hit set generated by Algorithm 1, its parent node is guaranteed to be unable to satisfy Eq. (15). Therefore, if there exists a hit set $\mathcal{S}'$ that satisfies Eq. (14) and Eq. (15), then there is no node in $\mathcal{S}'$ could be the parent node of any node in the hit set $\mathcal{S}^*$ generated by Algorithm 1. Furthermore, for each node v' in the hit set $\mathcal{S}'$, we iteratively search for its parent node along its path to the root node. If there is no node on this path which belongs to $\mathcal{S}^*$, then the node v' has no contribution to covering the query range. The reason is that Algorithm 1 ensures that the hit set $\mathcal{S}^*$ completely covers the range query. As a result, we can remove the node v' from the hit set $\mathcal{S}'$ to make it smaller.

So far, we have successfully proven two properties of an optimal hit set $\mathcal{S}_{opt}$: (1) for any node v^* from the hit set $\mathcal{S}^*$ of Algorithm 1, no node from $\mathcal{S}_{opt}$ could exist on the path between the node v^* and

the root node; (2) for any node v in an optimal hit set $\mathcal{S}_{opt}$, there must exist a node $v^* \in \mathcal{S}^*$ on the path between the root node and the node v. Therefore, the size of this optimal hit set is at least as large as the size of $\mathcal{S}^*$, which indicating $\mathcal{S} = \mathcal{S}_{opt}$.

Logarithmic running time. We want to further prove that the running time of Algorithm 1 is $O(\log T)$. Since Algorithm 1 performs only $O(1)$ operations at each node, we only need to calculate the number of nodes traversed in the process of finding any range query in the segment tree. The proof here is quite similar with B.1. Similarly, we reconsider the three cases in B.1.

(1) The range query $[T_s, T_e)$ is a prefix of the time range $[\sigma_v, \tau_v)$, i.e., $T_s = \sigma_v$.
(2) The range query $[T_s, T_e)$ is a suffix of the time range $[\sigma_v, \tau_v)$, i.e., $T_e = \tau_v$.
(3) The range query $[T_s, T_e)$ is neither a prefix nor a suffix of the time range $[\sigma_v, \tau_v)$, i.e., $T_s \neq \sigma_v$ and $T_e \neq \tau_v$.

Now, let $g_i(h)$ represent the number of traversed nodes in a stream segment tree with a height of h in case (i), v represent the root node, and u_1 and u_2 represent the left children and right children of v, respectively. Then, for case (1), if the range query can be entirely contained within the left sub-tree of u_1, then the traversed nodes will be the current root node v and all the nodes traversed within the left sub-tree, i.e., $g_1(h) \leq 1 + g_1(h-1)$. Conversely, if the range query spans the time range of both the left sub-tree and right sub-tree, then the set of traversed nodes will be the current node v, the left children u_1, and all the nodes which will be traversed within the right sub-tree, i.e., $g_1(h) \leq 2 + g_1(h-1)$. To sum up, we have the following expression

$$g_1(h) \leq 2 + g_1(h-1). \tag{38}$$

Therefore, considering the base case that $g_1(1) = 1$, we have

$$g_1(h) \leq 2h - 1. \tag{39}$$

We can also have the same conclusion for case (2), i.e., $g_2(h) \leq 2h-1$.

As for case (3), if the range query can be contained within the time range of any sub-tree, then similarly we count the current node v and further recursively search within the sub-tree, i.e., $g_3(h) \leq 1 + g_3(h-1)$. Otherwise, please note that in this situation, the new query range will be either a prefix or suffix of the new sub-tree, which implies $g_3(h) \leq g_1(h-1) + g_2(h-1)$. To sum up, we have

$$g_3(h) \leq 4h - 2. \tag{40}$$

Therefore, given any stream segment tree with a height of h, the number of traversed nodes will be $O(h)$. Since $h = \lceil \log_2 T \rceil + 1$ in Theorem 6.2, we finally prove that the running time is $O(\log T)$.

B.5 Proof of Lemma 5.2

Eq (22) consists of three computations whose costs are as follows: for $n = 2, ..., p$ and $i = 1, ..., s$, (1) matrix multiplications $\boldsymbol{U}^{(n)\mathsf{T}}\widetilde{\boldsymbol{V}}_i^{(n)}$, (2) tensor-matrix products between $\widetilde{\mathcal{H}}_i$ and the preceding results $\boldsymbol{U}^{(n)\mathsf{T}}\widetilde{\boldsymbol{V}}_i^{(n)}$, and (3) tensor-matrix products between the preceding results and matrices $\widetilde{\boldsymbol{V}}_i^{(1)}$ require $O(spr^2D)$, $O(spr^{p+1})$, and $O(r^p(T_e - T_s))$, respectively. Therefore, the complexity for computing Eq (22) is $O(spr^2D + r^p(T_e - T_s) + spr^{p+1})$.

B.6 Proof of Lemma 5.3

In Eq (27), there are four computations: for $m = 2, ..., n-1, n+1, ..., p$ and $i = 1, ..., s$, (1) matrix multiplications $\boldsymbol{U}^{(1)\mathsf{T}}_{[t_{i-1}, t_i)}\widetilde{\boldsymbol{V}}_i^{(1)}$, (2) matrix multiplications $\boldsymbol{U}^{(m)\mathsf{T}}\widetilde{\boldsymbol{V}}_i^{(m)}$, (3) tensor-matrix products between $\widetilde{\mathcal{H}}_i$ and the preceding results $\boldsymbol{U}^{(m)\mathsf{T}}\widetilde{\boldsymbol{V}}_i^{(m)}$, and (4) tensor-matrix products between the preceding results and matrices $\widetilde{\boldsymbol{V}}_i^{(n)}$ require $O(r^2(T_e - T_s))$, $O(spr^2D)$, $O(spr^{p+1})$, and $O(sr^pD)$, respectively. Hence, the complexity for computing Eq (27) is $O(spr^2D + sr^pD + r^2(T_e - T_s) + spr^{p+1})$.

B.7 Proof of Proposition 6.1

For each iteration, there are five computations: (1) the Recall algorithm, (2) the matricization of the temporal mode, (3) the matricization of $p-1$ non-temporal modes, (4) singular value decomposition p times, and (5) the computation for updating core tensor. Following the Lemmas 5.2 and 5.3, the first two computations require $O(sp^2r^2D + spr^pD + pr^2(T_e - T_s) + r^p(T_e - T_s) + sp^2r^{p+1})$. Since we perform SVD for $p-1$ matrices of the size $D \times r^{p-1}$ and the matrix of the size $(T_e - T_s) \times r^{p-1}$, this requires $O(p\min(r^{p-1}D^2, r^{2(p-1)}D) + \min(r^{p-1}(T_e - T_s)^2, r^{2(p-1)}(T_e - T_s))$. It can be expressed as $O(pr^{2(p-1)}D + r^{2(p-1)}(T_e - T_s))$ when $D > r^{p-1}$ and $(T_e - T_s) > r^{p-1}$. The last computation for a core tensor requires $O(r^pD)$. Therefore, the time complexity is $O((sp^2r^2 + spr^p + pr^{2(p-1)})D + (pr^2 + r^p + r^{2(p-1)})(T_e - T_s) + sp^2r^{p+1} + \log T)$ which is the sum of the complexities of these computations.

B.8 Proof of Theorem 6.2

In this subsection, we will first prove that every insertion of Algorithm 4 is valid, and further prove that the height of the constructed stream segment tree is $\lceil \log_2 T \rceil + 1$ with T representing the time span.

First, to prove a valid insertion is equivalent to prove that, when a new node is inserted into a segment tree, the segment tree is not full. Please note that the design of creating nodes and insertion in Algorithm 4 ensures that, when a new node at the time of T are inserted into the node v, the range query $[\sigma_v, \tau_v)$ always contains the time of T, i.e., $\sigma \leq T < \tau_v$. Therefore, we only need to prove that for any node v in a stream segment tree, the number of new nodes that can be inserted into the sub-tree rooted at v is $\max\{\tau_v - T, 0\}$.

Specifically, we choose to mathematical induction to prove the above statement. When $T = 1$, this statement obviously holds since it means creating the first node for a stream segment tree. Now, let us assume the statement holds true when $T = t_0$. In this situation, if a node of t_0 is inserted within a sub-tree rooted at the node v, Algorithm 4 ensures that $\tau_v > t_0$, which implies $\max\{\tau_v - t_0, 0\} > 0$. Then this insertion is valid since there is still space for a new node within the sub-tree of v. After this insertion, we can easily calculate that the number of new nodes that can still be inserted will be $\max\{\tau_v - t_0, 0\} - 1 = \max\{\tau_v - (t_0 + 1), 0\}$. Therefore, the statement continues to hold when $T = t_0 + 1$, which completes the proof.

Second, we want to calculate the height of a constructed stream segment tree. From the previous discussion, we can easily know that given a stream segment tree with a height of $h-1$, nodes can be continuously added until the stream segment tree is full. Once

the tree becomes full, the attempt to add another node results in creating a new root node based on Algorithm 4, with the original root node becoming the node of the left sub-tree. Consequently, the height of the new segment tree becomes h. Therefore, it is evident that the number of leaf nodes in a segment tree of height h is greater than that of a perfect binary tree with height $h-1$ and less than or equal to that of a perfect binary tree with height h. Then we successfully show that the height of a stream segment tree is $\lceil \log_2 T \rceil + 1$ since the number of leaf nodes is equal to T.

B.9 Proof of Proposition 6.3

We analyze the worst-case and amortized complexities of a stream segment tree. Note that s is equal to 2. We omit the number of iterations for the stitch operation and Tucker decomposition of the tensor slice for brevity.

First, performing Tucker-ALS of the tensor slice requires $O(prD^{p-1})$. When we append a tensor slice $\mathcal{X}_T$, the worst-case number of the stitch operations is $O(\log T)$. Then, updating the non-temporal factor matrices requires $O((p^2r^2+pr^p+pr^{2(p-1)})D \log T + p^2r^{p+1} \log T)$ derived from the first and third terms of the time complexity in Proposition 6.1. Similarly, updating the temporal factor matrices requires $O((pr^2 + r^p + r^{2(p-1)})T)$ where the sum of the row sizes of the temporal factor matrices is $O(T)$. Hence, the worst-case complexity of appending a tensor slice $\mathcal{X}_T$ is $O(prD^{p-1} + (p^2r^2 + pr^p + pr^{2(p-1)})D \log T + p^2r^{p+1} \log T + (pr^2 + r^p + r^{2(p-1)})T)$.

The amortized complexity is the time complexity of creating a stream segment tree divided by T. We preprocess T tensor slices of leaf nodes with the complexity $O(prD^{p-1}T)$. For intermediate nodes, we perform $O(T)$ stitch operations with the complexity $O((p^2r^2+pr^p+pr^{2(p-1)})DT+p^2r^{p+1}T+(pr^2+r^p+r^{2(p-1)})T \log T)$. Hence, the amortized complexity is $O(prD^{p-1} + (p^2r^2 + pr^p + pr^{2(p-1)})D + p^2r^{p+1} + (pr^2 + r^p + r^{2(p-1)}) \log T)$.