Scalable Convergence Queries on Time Series Compounds

Noura Alghamdi*
nsalghamdi@uj.edu.sa
University of Jeddah
Jeddah, Saudi Arabia

Khalid Alnuaim*
kalnuaim@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

Xiaoshuai Li
xli3@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

Elke A. Rundensteiner
rundenst@wpi.edu
Worcester Polytechnic Institute
Worcester, MA, USA

Mohamed Y. Eltabakh
meltabakh@hbku.edu.qa
Qatar Computing Research Institute
Doha, Qatar

Abstract

The Time Series Compound (TSC) is a data model designed to represent sequences of interrelated time series objects, incorporating interleaved time-gap semantics into a unified structure. This enables a holistic representation of complex temporal phenomena, such as a patient's longitudinal medical history across visits. In this work, we formalize a novel class of queries over the TSC model with broad applicability, called **convergence queries**. They model the similarity of TSC objects increasing or decreasing over time based on rich time semantics. Convergence queries, unsupported by existing systems, pose significant processing challenges due to their high dimensionality, alignment complexity, and complex progression semantics intrinsic to TSC data. To address these challenges, we extend the TSC infrastructure to support convergence queries as first-class citizens. For this, we propose a new similarity progression trend scheme and define two types of convergence: *Strict* and *Statistical*. We design a fully distributed execution pipeline for processing these convergence queries at scale. To enhance usability, the system allows for expressive query predicates on convergence characteristics. We introduce query optimization strategies based on convergence semantics for efficient query execution. To improve system throughput, we develop batch-aware optimizations that enable shared access to overlapping data partitions across queries. Extensive experiments on terabyte-scale datasets show that our approach achieves up to 10× speedup over baseline methods and up to 95% accuracy, far surpassing kNN-based solutions for implementing convergence, which at best attain only 20% accuracy.

Keywords

Time Series, Time Series Compound, Convergence Queries, Distributed Query Processing, Query Optimization

1 Introduction

Over the last few decades, advances in science and engineering as well as the explosion of the Internet of Things (IoT) have led to the generation of huge volumes of digital data traces, referred to as *time series data* [18, 37, 46]. While existing techniques have explored time series data through various types of analytics, e.g., prediction [17, 27, 62], clustering [8, 25, 38], and classification [4, 11, 29], only a handful of classical query types have been investigated on time series data, in particular, kNN queries [63, 64, 69] and ϵ−range queries [23, 28, 60]. Moreover, these previous techniques adopt the traditional time series data model that treats each time series object, e.g., the heart rate measurements for a patient at a doctor's visit, as an independent standalone object in the database.

Yet the recent observation that applications often generate a *sequence of interconnected and intermittent time series objects separated by large time gaps* has led us to introduce a novel data model, referred to as *Time Series Compound (TSC)* [6]. *TSC* data arise in numerous areas from a patient's series of visits in healthcare systems [42] to machine condition monitors [19]. For example, the time series measurements of a patient across different visits can be viewed as a single TSC object that holistically captures the patient's medical history and recovery progression. Thus, the TSC model can naturally support important end-user queries such as: *"find the k patients whose medical history (i.e., their TSCs) are the most similar to a patient of interest (i.e., a given query-TSC)"*. TSC match semantics take the entire TSC structure as a unified object into account including the order among its time series object components and the sizes of the interleaved time gaps corresponding to times between hospital visits [6].

While our prior work has focused on extending classical similarity queries to the TSC model, we propose and study a new class of time-oriented queries: *convergence queries*. These queries aim to identify TSCs whose similarity to a given query-TSC is not just high in general, but *increasing over time*—that is, converging toward the query. This concept is critical for real-world applications that care about directional evolution, not just snapshot similarity. It is not captured by any existing time series query operators [15, 21, 28, 35, 54], and builds upon foundational ideas of convergence in real analysis [1] and statistical trend detection [33, 41].

Example 1. TSC Model and Convergence Query Semantics. *Many chronological diseases, e.g., arrhythmia and coronary artery heart diseases, require close periodic monitoring of patients to track their measurements, e.g., heart rate, over follow-up visits. These measurements across all visits of a patient can logically be viewed as a TSC containing time series objects in their chronological order along with time gaps in between. These time gaps may correspond to weeks or even months (see Figure 1). Now, assume two patients Alice and Tom each with three previous visits, i.e., three TS objects in each TSC, yet the time gaps between the visits for Alice are regular check-ups spaced apart by one year each, while Tom had an intensive observation period of one-week sessions all bunched into a short timeframe. Otherwise, the readings of their visits are identical, respectively. Now, consider we want to find patients who exhibit similar medical progression as Alice. If we were to apply a classical kNN similarity search query as in the traditional TS model adopted by the state-of-the-art (SOTA) techniques, e.g., [5, 63, 69], this would ignore gaps and chronological relationships among the time series*

*Both authors contributed equally to this research.

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.23

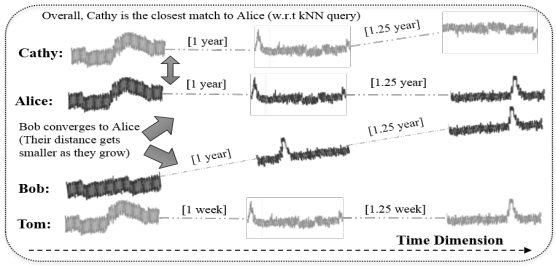

Figure 1: Example of TSCs: Convergence vs. Similarity.

objects. Thus, they would wrongly conclude that Alice and Tom are very similar and thus should receive a similar treatment. This may result in a faulty diagnosis. In contrast, the TSC model inherently captures the order and gap semantics. This is essential but not sufficient for allowing us to reason about temporal progression, namely, to discern that Tom's recorded sessions and observed symptoms are developing much faster than Alice's. Identifying such progression trends requires custom-query semantics—such as convergence queries—built on the TSC structure.

In this work, we propose a novel query class beyond the above classical queries like kNN and ϵ-range applicable to time-oriented data models such as time series and TSC models, called *convergence query*. Next, we motivate this prevalent but overlooked query type.

Example 2. Convergence Motivating Example. *As continuation to the medical scenario in Example 1, assume two additional patients Bob and Cathy, each with three previous visits separated by year-long time gaps (similar to Alice's case). Their TSC objects are depicted in Figure 1. For the purpose of predictive diagnostics and treatment prescription, classical similarity queries such as kNN and $\epsilon-$range that "find the k most similar TSCs to Alice's TSC" can be sometimes misleading, or at least may not capture the complete picture. In our example, Cathy's TSC would be the best match for Alice's TSC with respect to kNN distance. However, this result does not capture the fact that the patterns of these two patients are increasingly deviating from each other. In contrast, while Bob's TSC might not be among the kNN best matches for Alice, the fact that Bob's medical progression (TSC) is progressively getting more similar to Alice's health behavior over time could be a critical piece of information useful for the diagnosis of Alice's medical case. Such "convergence trend" may imply that Alice and Bob are responding similarly to their treatments; and thus that we may utilize Bob's medical responses as a better prediction for Alice's future measurements. Further, early detection that Cathy's measurements are diverging from Alice's could be a vital sign for re-examining Cathy's medical conditions–especially if many other patients show convergence to Alice's case–e.g., Alice's TSC may represent a hub or trend within the dataset.*

The TSC model, together with the proposed convergence semantics, is applicable across a wide range of domains. For instance, in manufacturing and IoT applications, sensors are typically attached to machines to record measurements during operation (time series readings), while remaining inactive when machines are idle (gap intervals). The early identification of machines whose TSCs are *converging* toward faulty patterns can play a critical role in preventive maintenance. In security applications, potential threat and attack attempts can be separated by days or even weeks, and only when viewed holistically and interconnected together (through the TSC model) can be analyzed for potential similarity and convergence to previous attack patterns. Another example from urban planning, where taxi trajectories can be categorized into active ride trajectories and passive empty-trip trajectories. Within the TSC model semantics, one trajectory class can be selectively masked (i.e., treated as gap intervals) to enable more focused analysis (e.g., TSC kNN or convergence) of the other class. This type of analysis is not feasible under the classical time series data model.

Convergence Queries and Their Opportunities. The above examples call for this new class of queries, which we refer to as *"Convergence Query"*. The notion of *convergence* as highlighted above has been investigated in the literature from both theoretical and statistical perspectives by the mathematical community [22, 34, 52]. However, unfortunately, despite its potential utility for real-world applications, the data management community has to-date not yet explored *convergence* as a first-class query operator. As highlighted in Example 2, convergence queries could be leveraged as a building block operation in various types of analytics, e.g., prediction, top trends, and divergence.

In this paper, we focus on scalable processing of this new *Convergence Query* in the context of the TSC model. Beyond that, convergence queries could also introduce value-added semantics to other data types as long as they either involve data series and/or time attributes. This effort may thus opening new directions for future research beyond our initial first work.

Technical Challenges of Supporting Convergence Queries. This new query type leads to several technical challenges, including:

• *Lack of convergence query semantics*: As a new query type, TSC convergence query semantics must first be designed. We need to address research questions such as: What does it mean for two TSCs to be "converging"? What are the query input parameters? What is the expected output from this new query type? And how to measure the inputs similarity progression over time? This is especially important given our observation that existing similarity measures proposed for kNN and range queries produce a single global distance value [23, 28, 60, 69], which clearly is not sufficient for capturing convergence.

• *Higher-order and rich TSC model* : While attractive for studying convergence due to the long life span of TSC objects, the TSC data model is characterized by being composed of an irregularly-spaced sequence of time series objects separated by variable-length and often exceedingly long time gaps. Contrary to the traditional issue of missing values that might arise within a single time series [14, 31, 37], time gaps in *TSCs* not only can be much longer than the actual time series component but they also tend to be rich in time-semantic meaning, e.g., the time intervals between a patient's visits. Convergence queries must take these TSC properties into account by treating the composed TSC data as holistic objects.

• *Need for convergence-aware TSC-based distributed indexing*: The terabyte-scale of TSC data continuously generated by modern applications [2, 19, 57] warrants the need for large-scale distributed *TSC* processing systems. Index structures are needed as they play a critical role in speeding up queries. Unfortunately, the *TSC*-aware index we proposed in [6] is tailored for similarity-based operators, e.g., kNN and range queries. As we will show experimentally, the convergence operator cannot be developed as a post-processing stage on similarity operators's output. Hence, we instead need to design TSC index internals to directly support the convergence operator semantics.

• *Need for efficient query processing strategies for convergence queries*: As a new query type, there are no algorithms in the literature to implement and optimize the execution of convergence queries. Thus, we need to devise an entirely new execution pipeline that integrates different stages ranging from index-based retrieval, efficient and early filtering (whenever possible), similarity measures, and convergence tests, among others, all to be accomplished cohesively in a highly distributed fashion.

State-of-The-Art Techniques and Their Limitations: Most queries proposed in the literature on time series data are similarity-based queries and use either metric distances, e.g., [28, 58], or elastic distances, e.g., [12, 36, 58]. These queries can be classified into kNN queries [5, 6, 63, 69] and ϵ−range queries [23, 60]. The similarity measures used in these techniques, e.g., [43, 48, 58] typically return a single distance value for a pair of input objects. Compared to these query types, our targeted *convergence* query is a new distinct query type that requires re-thinking critical query processing-related aspects as highlighted in the aforementioned challenges. Moreover, our recent TSC data model proposed in [6] focuses on kNN queries and basic query types. The TSC-aware similarity semantics introduced in [6], while effective in capturing TSC whole match, falls short in capturing the progression of similarity of TSC over time.

Our Approach and Contributions. We propose the first end-to-end solution for convergence queries on large-scale TSC datasets. Our key contributions include:

1) Convergence as a Query Type. We are the first to investigate *convergence* as a first-class query type and to identify its potential utility as a building block operator in several types of analytics. Although convergence queries are broadly applicable to other time-oriented data types, e.g., time series applications [23, 57, 63, 69], we focus this first work on the TSC data model.

2) Formal Semantics for TSC Convergence Query. We introduce a new similarity measure, called *similarity progression sequence*, for measuring the distance progression between a pair of TSC objects. This measure, which is internal to the system (i.e., not part of the query answer), is designed to encode the characteristics of the TSC objects under comparison, e.g., their structural similarity, time alignments, and gap sizes. We then define the formal semantics of the convergence query on top of the similarity sequence (see Sections 4.2 and 4.3.)

3) Scalable Optimized Convergence Processing Strategies. We design a highly distributed execution strategy for scalable convergence queries over TB-scale datasets. This pipeline integrates the TSC index with plugable statistical tests for determining convergence. We also propose two types of optimization for improved performance, specifically the *predicate-driven partition prioritization* strategy for queries involving convergence-related predicates, and the *multi-query batch* strategy that exploits opportunities for execution sharing among multiple queries.

4) Extensive Experimental Study. Our results demonstrate that traditional baselines, e.g., full scan processing, are impractical due to their prohibitively high execution times¯our method achieves up to a 10× improvement in performance. Furthermore, the incorporation of the described optimization strategies contributes an additional 2× to 3× speedup, while maintaining high query recall levels of up to 95%. We provide empirical evidence that current state-of-the-art kNN operator for TSCs fails to support the semantics required by the proposed convergence operator, yielding below 20% accuracy.

Outline. In Section 3, we overview the TSC data model and its infrastructure. In Section 4, we propose the building blocks for the convergence query including the formal convergence definition between two TSC objects, the similarity progression measure, and characteristics and requirements of the convergence test. Section 5 introduces our convergence query engine and associated optimizations. Finally, the experimental study and conclusion are described in Sections 6 and 7, respectively.

2 Related Work

TSC Systems. The most closely related work to the proposed research is Sloth [6], the only infrastructure in the literature that supports TSC-type data. However, Sloth is limited to classical query types, such as KNN. As demonstrated in our comparative study, Sloth fails to produce correct results for convergence queries, particularly in terms of recall. Therefore, the innovations introduced in our work—including novel query semantics, efficient convergence execution strategies, and convergence-specific optimizations—are essential for effectively addressing this new problem.

Time Series and Trajectory Similarity Search. Another relevant line of work is similarity search over time series and trajectory data [16]. Trajectories can be viewed as a specialized form of multivariate time series that capture the movement of objects through space. A variety of distance metrics have been proposed in the literature, including Euclidean [24, 44], SAX-based [40, 69], DTW [30], LCSS [53], Hausdorff [45], and Fréchet distances [47]. Each metric typically necessitates dedicated index structures and query processing algorithms to efficiently support its specific semantics. In our system, the alignment and comparison between pairs of TSCs adhere to the classical Euclidean distance, which is commonly adopted in literature [5, 6, 24, 44, 69]. The exploration of alternative metrics is left as a direction for future work.

Learning-Based Similarity Methods: In contrast to the aforementioned similarity measures, which are all non-learning-based methods, recent research has introduced learning-based approaches such as SEANet [56], TS2Vec [65], TNC [51], TS-TCC [26], and Traj2SimVec [68]. These methods first encode raw objects into a learned embedding space, after which similarity is estimated based on the distance between the resulting embeddings. However, due to the inherent complexity of the TSC data model–such as multi time series objects and extended gap semantics–combined with the novel semantics required for convergence queries, current learning-based methods are not directly applicable to our context.

Distributed Infrastructures for Big Time Series. Several distributed systems have been developed to process large-scale TS datasets, e.g., [5, 23, 57, 63, 64, 69]. However, they only limited to basic similarity search queries. Some focus on whole time series matching and kNN, e.g., [63, 64, 69], while others focus on subsequence matching [5, 57, 60]. KV-Match [23, 60] supports distributed range queries and [5, 57] support kNN queries. Similarly, [20, 39, 61] address similarity search over trajectory data in distributed settings, but remain limited to classical kNN and spatial range queries. In contrast, our proposed distributed system uniquely supports both a novel TSC data model and advanced convergence query semantics.

Time Series Databases: Popular time series database engines, such as Apache IoTDB [55], InfluxDB [10], and TrajStore [59], provide highly efficient storage and compression mechanisms, high-throughput data ingestion and processing, and support for

online aggregations and analytical tasks. These systems are designed with real-time streaming data ingestion in mind, which is reflected on the design of their custom data storage and indexing mechanisms such as LSM [55] and TSM [10]. These indexes leverage both memory tables and compressed disk-based files for efficient data movement. However, these systems are designed exclusively for the classical time series data model and cannot handle the TSC model. Moreover, our proposed system is currently designed for batch processing, and thus its storage and indexing mechanisms are fully disk-based distributed files.

Although full native integration into existing systems is beyond the scope of this work—since it would require substantial modifications across all layers—we demonstrate a proof of concept by integrating our system into IoTDB [55] as an external library. In this setup, the two systems coexist, with the advantage of having the TSC operations callable directly from within the IoTDB environment. This enables applications to, for example, construct a TSC dataset from an existing TS tables in the database, and execute TSC-specific queries from within the IoTDB system (See Appendix F for more details).

3 TSC Model and Infrastructure: An Overview

A traditional **time series** $TS=\langle x_1,x_2,\cdots,x_m\rangle$, $x_j \in R$ *where* $1 \leq j \leq m$, corresponds to an ordered sequence of m real-valued readings, i.e., the cardinality $|TS| = m$. Each reading x_i has an associated timestamp t_i. Accordingly, the time interval that a time series spans is denoted as $span(TS) = [t_1, t_m]$. Without loss of generality, we assume that the readings arrive at fixed time granularities. Hence, we only store the timestamp of the first reading.

TSC Data Model. In our prior work [6], we proposed a new model, called the *Time Series Compound (TSC) model*, where a TSC object is represented as a sequence of interrelated time series objects along with their chronological order and time gap semantics. A TSC object and dataset are defined as follows.

Definition 1. [Time Series Compound (TSC)]. A time series compound $TSC = \langle t_0, (t_1,TS_1), (t_2,TS_2), \cdots, (t_n,TS_n)\rangle$ with identifier *oid(TSC)* and starting at timestamp t_0 corresponds to an ordered sequence of n time series (TS_i), where t_i is the start timestamp of TS_i, $t_0 \leq t_1 < t_2 < \cdots < t_n$, and $span(TS_1) < span(TS_2) < \cdots span(TS_n)$. This array of n+1 time stamps is denoted by *TIM(TSC)*, while the number of time series objects within the *TSC* is denoted by cardinality *car(TSC)=n.* [1]

Definition 2. [TSC Dataset (D_{TSC})]. A **time series compound dataset** $D_{TSC}=\{TSC_1, TSC_2, \cdots, TSC_N\}$ corresponds to a set of *N TSC objects*, each with different cardinalities and time gaps.

TSC-Based Infrastructure. We previously demonstrated in [6] that existing time series systems cannot efficiently support the new TSC abstraction and semantics. They investigated leveraging existing time series technology, augmented with a middle-tier layer for post-processing, to stitch together TSC advanced semantics. For example, to encode the equivalent of a TSC object in existing systems, they distinguish between three representations: one could store individual time series associated with a TSC as individual instances (called DECOMPOSE), or concatenate the time series objects together by either ignoring the time gaps (called CONCAT), or by filling in the whole gap durations with imputed values to construct one very long contiguous big TS object (called IMPUTE). Our experiments in [6] showed that none of these three middle-tier inspired approaches are sufficient, as they not only lack efficiency, and scalability, but also generate very poor-accuracy results.

[1]The notations of the interval relationships follow the Allen's Intervals Algebra [7]. For example, $span(TS_1) < span(TS_2)$ indicates that TS_1 *precedes* TS_2

```
1  FOR (singleQueryTSC | EACH TSC IN batchQueryTSC)
2  SELECT * FROM TSCDB
3  USING (Strict | Statistical) CONVERGENCE
4  [WHERE ConvPred]
5  ConvPred ::= (StructurePred | SimProgressionPred | ConvPred) AND ConvPred
6  StrucurePred ::= STRUCTURAL SIMILARITY op constant
7  SimProgressionPred ::= (TimeBasedPred | ValueBasedPred)
8  TimeBasedPred ::= PROGRESSION WINDOW BETWEEN startTime AND endTime
9  ValueBasedPred ::= measurements(PROGRESSION SIMILARITY[startIndex : endIndex])
   op Threshold[0..1]
10 measurements = min | max | median
11 Op = > | < | <= | = | >=
```

Figure 2: Convergence Query Syntax.

To overcome these limitations, in [6] we introduced a distributed infrastructure *Sloth* to manage and query large-scale TSC datasets, with its source code available in [3]. In Section 5.1, we briefly cover the components of *Sloth*, which relate the closest to the technologies we adopt as substrate for our infrastructure. To realize the proposed convergence operator, we propose and develop a set of novel query execution strategies, optimization techniques, and an end-to-end convergence-based framework integrating these components.

4 Convergence Evaluation strategy

Here, we present syntax and formal definition of the convergence query (Section 4.1), followed by the evaluation methodology along with characteristics of the convergence test (Sections 4.2-4.3).

4.1 Convergence Query

The query syntax of the convergence query is presented in Figure 2. The **FOR** clause indicates the *query object*, which is either a single query TSC (**singleQueryTSC**) or a batch of query objects (**batchQueryTSC**). The system retrieves the converging TSC objects from a target large-scale and disk-persistent TSC dataset (**TSCDB**) while satisfying optional convergence-related predicates (**ConvPred**). As further elaborated in Section 4, we support several types of predicates that allow users to limit the result set based on structural properties (**structurePred**) and on similarity progression properties (**simProgressionPred**). The **USING** clause enables the selection from two different convergence types, namely `Strict` and `statistical`, explained in Section 4.3. The formal definition of a convergence query is given below.

Definition 3. [TSC Convergence Query] Given a query TSC object $Q\text{-}TSC = \langle Qt_0, (Qt_1, QTS_1), (Qt_2, QTS_2), \cdots, (Qt_n, QTS_n)\rangle$, a time series compound dataset $D_{TSC} = \{TSC_1, TSC_2, \cdots, TSC_l\}$ and optionally a list of convergence predicates *ConvergencePred*, the query finds the set R = { $TSC_i \in D_{TSC}$} such that $\forall$ $TSC_i \in$ R, TSC_i ***converges*** to Q-TSC and satisfies the predicates in *ConvergencePred* as described in Fig. 2.

The convergence (either `Strict` or `statistical`) will be formally defined in Section 4.3. Informally, we assume that for two

given TSCs, the distance between each pair of time series gets smaller as time progresses (see Alice and Bob TSCs in Figure 1). Def. 3 can be naturally extended to support a batch of TSC queries as input.

Next, we present our proposed convergence evaluation methodology. In a nutshell, given a query time series compound (Q-TSC) and a target time series compound (TSC), the convergence query operator determines if the TSC converges to Q-TSC. This query operator adopts two main steps: (1) Compute the *similarity progression sequence* between Q-TSC & TSC, which corresponds to a vector of time-oriented TS-level distances. This is explained in Section 4.2, and (2) Determine whether the computed similarity progression sequence indicates convergence.

4.2 TSC Structural and Similarity Progression

The TSC convergence operator requires a new notion of *similarity progression over time between TSCs*. That is, for a pair of TSCs as depicted in Fig. 3, our goal is to create a sequence of time-oriented distances based on which we can determine convergence. We tackle this similarity progression modeling between *TSCs* using a three-pronged strategy consisting of *structure-similarity check*, *alignment*, and *similarity sequence generation*.

First, given two TSC objects, namely, a query object (Q-TSC) and a target object (TSC), we determine whether Q-TSC and TSC are structurally similar. Namely, two TSCs objects are said to be structurally similar if their overall structure is similar in terms of number and positions of time series objects and gaps, otherwise we assume they are not comparable. This similarity is captured by the *Structural Similarity Score* defined below.

Definition 4. [Time Series Overlaping in TSCs]. Given a query *TSC Q-TSC* = $\langle\ Qt_0, (Qt_1, QTS_1), (Qt_2, QTS_2), \cdots, (Qt_n, QTS_n)\rangle$ and a target *TSC* object *TSC* = $\langle\ t_0, (t_1, TS_1), (t_2, TS_2), \cdots, (t_l, TS_l)\rangle$, a time series from *Q-TSC* (say QTS_i) is said to *overlap* with a time series from *TSC* (say TS_j) iff: (1) $\text{span}(\text{QTS}_i) \bigotimes \text{span}(\text{TS}_j)$, where $\bigotimes$ is any of Allen's interval relations $\{o, oi, s, si, d, di, f, fi, =\}$, and (2) the size of the overlapping segment exceeds a system-defined threshold , i.e., $[2 * (\#\ overlapping\ points)/(|QTS_i| + |TS_j|)] > \beta$.

For example, referring to Figure 3, the pair of time series objects from left are considered overlapping (they have *o* & *oi* relationship), and also the pair from right are considered overlapping (they have = relationship).

Definition 5. [Structural Similarity Score]. Given a query *TSC Q-TSC* = $\langle\ Qt_0, (Qt_1, QTS_1), (Qt_2, QTS_2), \cdots, (Qt_n, QTS_n)\rangle$ and a target *TSC* object *TSC* = $\langle\ t_0, (t_1, TS_1), (t_2, TS_2), \cdots, (t_l, TS_l)\rangle$, their structural similarity score *structuralSim* is defined as:

$$structuralSim = 2 * \left\lfloor \frac{\#\ Overlapped\ TSs\ from\ QTSC}{car(\text{Q-TSC}) + car(TSC)} \right\rfloor \quad (1)$$

where $car(.)$ is defined in Def. 1. StructuralSim $\in [0, 1]$, where 0 means the pair does not have any overlapping TS component objects and 1 means that all TS objects of the pair overlap.

For example, the query object (QTSC) and the target object (TSC) in Figure 3 have six TS objects combined. Among them, two are considered overlapping, yielding a structure-similarity score of $4/6 \approx 67\%$. Users can express a constraint on such score (Line 5, Fig. 2), which is leveraged for early filtering on candidate pairs before applying more expensive evaluation.

Thereafter, a structurally similar pair of TSCs is aligned through a process that involves *time alignment* and *weighted point-to-point matching* based on different cases as detailed in Example 3. During the alignment process, the query object remains *immutable* to ensure that scores from comparing different TSCs (from the dataset) to the query object are comparable relative to each other.

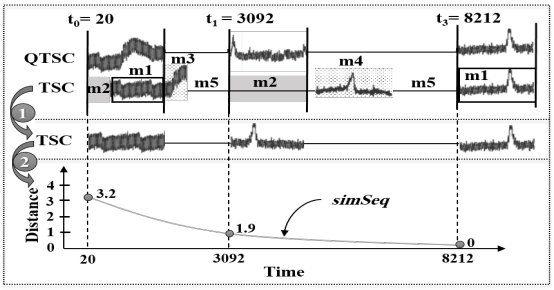

Figure 3: TSC Similarity Progression Semantics.

Example 3. *Referring to Fig. 3, since QTSC and TSC are structurally similar, they now go through the alignment process as follows. Both TSCs are first left-aligned at a common starting time (e.g., t_0 in our example). This creates four possible point-to-point match types: (1) value-to-value (e.g., the m_1 segments in Fig. 3), where there are time-aligned values in both objects, (2) value-to-gap (e.g., the m_2 segments), where the QTSC has a value while the target TSC has missing value, (3) gap-to-value (e.g., the m_3 and m_4 segments), where QTSC does not have a value but the target TSC does, and finally (4) gap-to-gap (e.g., the m_5 segment in Fig. 3), where both objects encounter gaps. To align with the immutable QTSC object, the m_1 and m_5 segments in TSC remain intact, while m_2 segments are imputed, and m_3 and m_4 segments are masked. This results in a modified TSC object as illustrated in Fig. 3, the middle TSC object.*

Thereafter, we compute the distances between value-to-value points in the QTSC and the altered TSC objects using a weight function. In this function, users can optionally define penalty terms (weights) on the imputed or masked values during the score calculations. We skip the details of the function as it can be found in [6], while we focus here on the generation of the novel similarity sequence. Specifically, we do not aggregate the value-to-value distances to one single aggregate score. Instead, we locally aggregate the *value-to-value* distances only at the TS-level. That is, we maintain a distance score for each aligned pair of the TS objects within the TSCs. This results in a sequence of similarity scores referred to as *similarity progression sequence* (*"simSeq"*, in short). This sequence is generated using the *similarity progression metric* defined below.

Definition 6. [*TSC* Similarity Progression Measure]. Given a query *TSC* object *Q-TSC* = $\langle\ Qt_0, (Qt_1, QTS_1), (Qt_2, QTS_2), \cdots, (Qt_n, QTS_n)\rangle$ and a target *TSC* object *TSC* = $\langle\ t_0, (t_1, TS_1), (t_2, TS_2), \cdots, (t_l, TS_l)\rangle$, a distance measure (d), then the sequence of the directional distances at the TS-level from *Q-TSC* to *TSC* is defined as:

$$\begin{aligned} simSeq(\text{Q-TSC} \rightarrow \text{TSC}) = \langle s_1 : [Qt_1, d(QTS_1, TS_x)], \\ s_2 : [Qt_2, d(QTS_2, TS_y)],\ \cdots, s_n : [Qt_n, d(QTS_n, TS_z)]\rangle \end{aligned} \quad (2)$$

where each s_i is a TS-level local similarity score consisting of two components; time (Qt_i) and distance value (d), $1 \le i \le n$, $n = car(Q - TSC)$ and TS_x, TS_y, and TS_z are TS objects in the candidate TSC that overlap with the corresponding QTS objects in Q-TSC, i.e., QTS_1, QTS_2 and QTS_n respectively. *simSeq()* function is asymmetric, i.e., *simSeq(Q-TSC → TSC) != simSeq(TSC → Q-TSC)*.

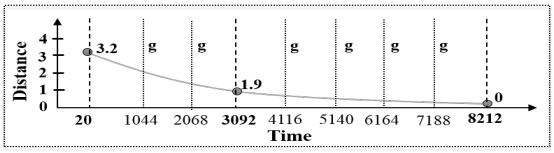

Figure 4: Regularizing *simSeq* generating *rSimSeq*

Example 4. *Continuing with Example 3 and Figure 3, we now have three aligned TS objects starting at times $t_0 = 20$, $t_1 = 3092$, and $t_3 = 8212$. We apply a TS-level similarity distance calculations using a given weighted Euclidean distance function [6], and let us assume the distance scores are 3.2, 1.9, and 0, respectively. This sequence produces the illustrated simSeq as a two dimensional distance vector per Eq. 2, i.e., $simSeq = \langle s_1 : (20, 3.2), s_2 : (3092, 1.9), s_3 : (8212, 0)\rangle$. Users can apply predicates on either or both of these dimensions as highlighted in the query syntax in Figure 2 (Lines 7 and 8).*

Characteristics of Similarity Progression Sequence. We discern the main characteristics of *simSeq* from Example 4. First, *simSeq* is an irregular time series consisting of car(Q-TSC) values, i.e. $|simSeq|$ = car(Q-TSC), i.e. 3 in Example 4. The irregularity arises due to TSC-specific characteristics such as different gap lengths in between TS objects within any given TSC object. Second, the *simSeq* values are distant from each other. This holds because of the existence of exceedingly long gaps of the underlying TSC objects (refer to Section 1).

4.3 Trend Analysis & Identifying Convergence

Our system supports two convergence types, namely *strict* and *statistical*. Strict convergence requires the similarity progression sequence to be monotonically decreasing without exceptions. Although it is a quick and low-cost test, it might be rarely observed in real-world applications due to the potential presence of fluctuations and noise. In contrast, statistical convergence relaxes the monotonicity requirement by applying a statistical trend analysis test to determine whether the overall sequence exhibits a significant downward trend. This allows detecting convergence even in the presence of minor fluctuations.

As a proof-of-concept, we integrate the well-established Mann-Kendall (MK) Test [41], a non-parametric test widely used for detecting monotonic trends in time series data. MK is particularly well-suited to our use case, as it handles missing values and does not assume a specific data distribution.

Mann-Kendall Test. It is a statistical non-parametric test that assesses if a monotonic trend holds in a time series. We opt for this test because it fulfills the TSC-specific characteristics of *simSeq* identified in Section 4.2. The original MK-test [34, 41] requires the data values to be independent [32, 33]. Meaning, it requires that the time between readings be sufficiently large so that no correlation between measurements can arise. Despite that, different versions of the test have been proposed to handle serial correlation, e.g., [32]. Finally and most importantly, the MK test can be computed even if there are gaps (also called *non-detected* values) in the time series. This is important because our TSC data by definition has gaps. The notion of gaps in the MK test is due to the assumption that time series are regular, i.e., generated or collected at regularly-spaced intervals of time. Thus, for a time series with values that do not happen at a regular time interval, it would be injected with gap (non-detect) values to convert it to become regular.

Regularizing *simSeq*. As mentioned earlier under *characteristics of simSeq* in Section 4.2, *simSeq* is an irregular TS. Thus in order to study its trend using the MK-test, we first regularize it (Fig. 4). We achieve this by transforming the gaps in the sequence into non-detect points (or gap points) in the m^{th} dimensional space, where $m = |TS|$. We chose to transform *simSeq* into the m dimensional space. Given the similarity progression sequence *(simSeq)*:

$$simSeq(\text{Q-TSC} \rightarrow \text{TSC}) = \langle s_1 : [Qt_1, d(QTS_1, TS_x)], \\ s_2 : [Qt_2, d(QTS_2, TS_y)], \ \cdots, s_n : [Qt_n, d(QTS_n, TS_z)]\rangle \quad (3)$$

We start by regularizing the *simSeq* readings over the time dimension as follows. $\forall$ pairs of consecutive values s_i and s_{i+1} in *simSeq*, where $0 \leq i \leq n$ and $n = |simSeq|$, we compute the number of gap points in the m^{th} dimensional space between them by first computing the length of the gap between each two consecutive points, and then divide the result by m.

$$\#g_i : numberOfGapPoints(s_i, s_{i+1}) = \frac{Qt_{i+1} - (Qt_i + |TS|)}{m} \quad (4)$$

Thus, the result would be a regular similarity sequence *rSimSeq* of values (i.e., TS-level distances) separated by a variable number of gap points as follows.

$$rSimSeq(Q - TSC \rightarrow TSC) = \langle s_1,\ \langle g * \#g_1\rangle,\ s_2, \\ \langle g * \#g_2\rangle,\ s_3,\ \cdots\ \langle g * \#g_{n-1}\rangle,\ s_n\rangle \quad (5)$$

where $\langle g * \#g_i\rangle$ corresponds to a sequence of gap points (g) between s_i ans s_{i+1}, with the number of these gap points equal to $\#g_i$ (Eq. 4).

Example 5. *Continuing with Example 4, given simSeq, i.e., $simSeq = \langle s_1 : (20, 3.2), s_2 : (3092, 1.9), s_3 : (8212, 0)\rangle$, let us assume the length of TS objects contained in Q-TSC and TSC is $|TS| = m = 1024$. As per the description above and Eq. 4 & Eq. 5, the regularized simSeq, i.e., rSimSeq= ⟨(20, 3.2), (1044, g), (2068, g), (3092, 1.9), (4116, g), (5140, g), (6164, g), (7188, g), (8212, 0)⟩, as depicted in Fig. 4. Since it is a regular sequence, we can ignore the time dimension and only keep the values. Thus, rSimSeq=⟨3.2, g, g, 1.9, g, g, g, g, 0⟩.*

Brief Review of Mann-Kendall Test. MK-test computes the MK-statistics (S_{MK}) of a sequence $rSimSeq(Q - TSC \rightarrow TSC) = \langle s_1,\ \langle g * \#g_1\rangle,\ s_2,\ \langle g * \#g_2\rangle,\ s_3,\ \cdots\ \langle g * \#g_{n-1}\rangle,\ s_n\rangle$ using Eq. 6. This compares each data value in the sequence to *all subsequent data values*. The initial value of S_{MK} is assumed to be 0 (indicating no trend). If a data value from a later time period is higher than a data value from an earlier time period, S_{MK} is incremented by 1, and if it is smaller, then S_{MK} is decremented by 1. Otherwise, S_{MK} stays the same. The net result of all such increments and decrements yields the final value of S_{MK}.

$$S_{MK} = \sum_{k=1}^{n-1} \sum_{j=k+1}^{n} sign(s_j - s_k) \quad (6)$$

where $j > k$, $0 < k < n$, $1 < j \leq n$, and

$$sign(s_j - s_k) = \begin{cases} 1 & \text{if } s_j - s_k > 0 \\ 0 & \text{if } s_j - s_k = 0 \text{ or if } s_j \text{ or } s_k \text{ are gaps} \\ -1 & \text{if } s_j - s_k < 0 \end{cases} \quad (7)$$

Example 6. *Continuing with Example 5, given the sequence rSimSeq= ⟨3.2, g, g, 1.9, g, g, g, g, 0⟩, $S_{MK}(rSimSeq) = -3$.*

Afterwards, the Mann-Kendall computes the variance of S_{MK}.

$$\text{VAR}(S_{MK}) = \frac{1}{18}\Big[n(n-1)(2n+5) - \sum_{p=1}^{G} t_p(t_p-1)(2t_p+5)\Big], \quad (8)$$

where n equals to the number of points in $rSimSeq$, G denotes the number of tied groups and t_p the number of values in the p^{th} group. The term "tied groups" refers to repeated identical values. If there are multiple gap values, they are represented as a group.

Example 7. *Continuing with Example 6, given rSimSeq = ⟨3.2, g, g, 1.9, g, g, g, g, 0⟩, we have a single tied group of 6 gap points, i.e., $G = 1$, $t_1 = 6$, and $n = 9$. Thus, $VAR(S_{MK}) = 47.67$.*

Lastly, the MK-test computes the normalized MK-test statistic, denoted by Z_{MK}, as follows.

$$Z_{MK} = \begin{cases} \frac{S_{MK}-1}{\sqrt{VAR(S_{MK})}} & \text{if } S_{MK} > 0 \\ 0 & \text{if } S_{MK} = 0 \\ \frac{S_{MK}+1}{\sqrt{VAR(S_{MK})}} & \text{if } S_{MK} < 0 \end{cases} \quad (9)$$

This statistic is then used to assess the presence of a monotonic trend, based on a predefined significance level (typically 95%), which corresponds to a two-tailed critical threshold of approximately 1.96.

Example 8. *Continuing with Examples 6 & 7, given that $S_{MK} = -3$ & $VAR(S_{MK}) = 47.67$, then $Z_{MK}(rSimSeq) = -0.29$.*

Because $Z_{MK} < 0$, this indicates a decreasing trend. However, since $|Z_{MK}| = 0.29$ is much smaller than the critical threshold (1.96 for 95% confidence), the trend is not statistically significant.

Thus, we conclude that the TSC does *not* converge to the Q-TSC in this case. This reflects a limitation of the MK-test when applied to *short sequences* (e.g., card(Q-TSC) < 5). We intentionally used a short query sequence in this example for clarity of explanation.

5 Convergence Query Engine

To support scalable convergence query evaluation, we develop an execution engine that integrates with the TSC indexing framework and enables efficient retrieval, filtering, and trend analysis. This section presents the core components of the engine, strategies for single processing, and optimizations for batch query processing, and predicate-aware early filtering.

5.1 The Underlying TSC Infrastructure

As mentioned in Section 3, our proposed convergence framework leverages components of Sloth, our prior distributed infrastructure introduced in [6]. Sloth, built on top of Apache Spark [66], supports various types of operators and queries on TSC datasets, e.g., manipulation and transformation operators for creating new datasets, SQL-like queries to select TSC objects matching certain criteria, and kNN queries for similarity search. Since our convergence query engine builds on top of Sloth's storage and indexing layers, we describe the TSC index framework next.

The TSC index is a distributed indexing structure composed of two layers: a top layer and a bottom layer (see Figure 5). The top layer, referred to as the *time-aware layer*, resides on the cluster's master node and serves as a centralized component. Its main features are illustrated in Figure 5. During the index construction phase, the framework decomposes each TSC object into its constituent TS objects. These TS objects are then distributed to the

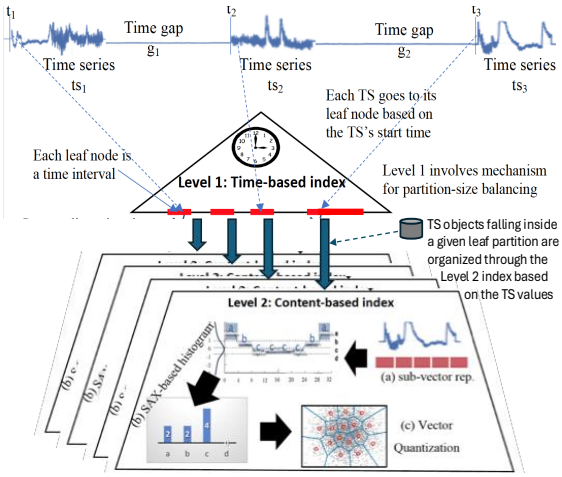

Figure 5: Overview of TSC Indexing Framework.

appropriate leaf partitions of the top layer based on their start times. The time intervals defining the boundaries of these leaf partitions (indicated in red in Figure 5) are dynamically determined by the indexing framework to ensure balanced partition sizes. As outlined in [6], this is accomplished by sampling complete TSC objects from the dataset, extracting the start timestamps from their TS components (e.g., t_1, t_2, t_3 in Figure 5), modeling the distribution of these timestamps, and selecting partition boundaries accordingly to achieve load balance. During query processing, the top layer facilitates the efficient identification of relevant data partitions that contain structurally matching candidates.

The bottom layer is build in a fully-distributed fashion over each leaf partition independent of the other partitions. This layer is based on three major steps as depicted in Fig. 5. First, each TS object is divided into equal-size subvectors. Such subvectorization helps preserve the locality of the features to be extracted and also helps mitigating misalignments at query time (e.g., when comparing two TSC objects whose individual TS objects do not perfectly align). Second, for each sub-vector, features are extracted in a compact structure referred to as *SAX-based histograms [6]*. This structure provides a very compact representation of the data. It also helps to mitigate misalignments at query time. The third and final step is called *vector quantization*, where the objects are further clustered for more compression. This bottom layer also stores metadata information essential for re-constructing *TSC* objects at query time.

In this work, we preserve the overall index structure, thereby enabling both operators—kNN and convergence—to utilize the same framework without incurring additional storage or maintenance overhead. However, several algorithmic enhancements are introduced to optimize the index lookup and management: (1) A modified index lookup mechanism, where for the kNN query operator [6], the bottom layer applies early local kNN filtering (i.e., each partition returns its top-K results), while for convergence queries all structurally matching TSCs are returned (Section 5.2); (2) Predicate pushdown techniques to avoid accessing irrelevant index partitions (Section 5.4); and (3) An incremental maintenance strategy to efficiently update the index during data appends (Appendix E).

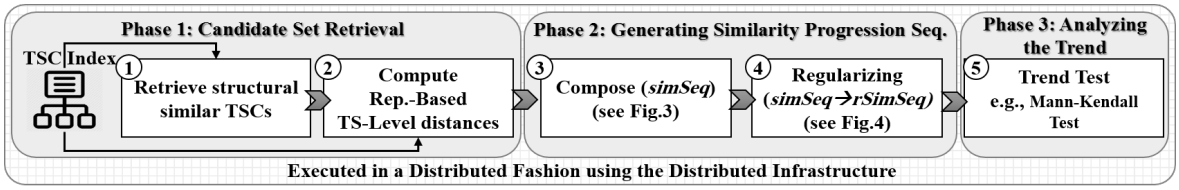

Figure 6: Convergence Analytics using Different Components of our TSC-Holistic System

5.2 Single Query Strategy (SQ-Strategy)

Phase 1: Candidate Set Retrieval. Given a query *Q-TSC*, the master node decomposes it into its QTS objects. Then, it extracts the start times t of all *QTS* objects of the query *Q-TSC* to determine which partitions to load. This is done by computing all the time overlaps for the *QTS objects* using the following equation:

$$timeOverlapRange(QTS) = [Qt - |TS| + odp, Qt + |TS| - odp] \quad (10)$$

where Qt is the start time of *QTS*, |TS| is TS's length, and odp is the maximum overlapped data points (user-parameter).

Then, the master node uses these time ranges to query the top time-based layer of the index to get their corresponding partition IDs. Thereafter, the master node loads these identified candidate partitions into the workers' memory. It then broadcasts the query *Q-TSC* with its *TSC-aware feature representation [6]* to the workers.

Now that the candidate partitions are loaded into the workers' memory, we start computing the TS-level distances. This is achieved by performing two level filtering at the index's bottom layer as follows. *First*, the workers fetch the TS objects that overlap with the *QTS objects* from the local structure, i.e., using the *HashMaps*. *Second*, the workers perform the approximate distance calculation between the feature representation of *Q-TSC* and the *TSC*-aware feature representation [6] of these candidates.

Phase 2: Generating Similarity Progression Sequences. Now, we have the feature-based distances at the TS-level between all *QTSs* in the given *Q-TSC* and the respective *TSs* of the candidate *TSCs* (i.e., the TSC database). The distances would be in the form: (*oid(TSC), Qt_i, dist*), where *oid(TSC)* is the ID of the candidate database TSC's, Qt_i refers to the timestamp of QTS_i, and *dist* denotes the feature-based distances between QTS_i and the overlapped TS_x in *TSC*.

Thus, to compose the full similarity progression sequence, we perform *groupByKey* action. This results in a collection of candidates similarity sequences in the form: (*oid(TSC), List<Qt_i, dist>*). After this, we sort the list of the distances for each sequence (i.e., List<Qt_i, dist>) based on time, i.e., Qt_i. Thus, the result is a collection of similarity progression sequences of all candidate matches.

Phase 3: Analyzing the Trend. Depending on the convergence mode in the query, the system applies one of two strategies:

USING `Strict` convergence, the system checks whether the similarity progression sequence is *monotonically decreasing*—i.e., each value is smaller than the one before it. This check is lightweight and efficient, requiring no statistical-based test.

USING `Statistical` convergence, the system applies the Mann-Kendall (MK) test [33] to determine whether the sequence shows a statistically significant downward trend. The MK test is run with a user-defined confidence level (e.g., 95%) and returns the trend (upward, downward, or no trend), a p-value, and supporting metrics, such as the Theil-Sen estimator [50]. The system passes only the TSCs exhibiting a downward trend for the Q-TSC, along with their metadata scores.

5.3 Batch Queries Strategies

Convergence queries are often issued in batches, e.g., given a set of TSCs representing a set of patients, find for each the top k converging TSCs from the dataset. To efficiently process multiple Q-TSCs, we propose two strategies that minimize redundant partition loads.

5.3.1 Partition-Sharing Strategy (ShareBQ). This strategy identifies overlapping partitions across all queries in a batch and loads them once. Each query then independently computes TS-level distances from the shared data. The execution flow is as follows:

- Extract start times of all QTS objects in the batch.
- Identify and load the union of relevant partitions.
- Perform TS-level distance comparisons and generate similarity progression sequences (`simSeq`).
- For each candidate, apply the Mann-Kendall trend test (if the statistical convergence is requested) or the monotonic decreasing test (if strict convergence is requested).
- Return ranked convergence matches per Q-TSC.

This sharing reduces I/O and improves parallelization efficiency.

5.3.2 Partition-Ranking Strategy (RankBQ). RankBQ extends ShareBQ by skipping low-impact partitions based on a scoring mechanism, trading a slight decrease in recall for substantial speedup. We estimate three scores per partition, namely:

- *AccessScore*: Frequency of access across batch queries.
- *QuantityScore*: Importance of each QTS relative to its parent Q-TSC.
- *PositionScore*: Location of QTS (head, middle, tail).

Partitions are ranked based on these metrics with tunable weights. A user-specified skip percentage determines how many lowest-ranked partitions to omit. This supports tunable accuracy-speed tradeoffs and complements ShareBQ for high throughput workloads. In Appendix A, we provide more details on both strategies.

5.4 Predicate-Guided Partition Prioritization

Users can apply value-based predicates on the similarity progression sequence (*simSeq*) to control the desired convergence degree (SQL syntax in Line 9, Figure 2), e.g., the maximum allowed distance between two converging TSCs must be less than a threshold. Such predicates, in a naive setting, can be applied at the very end of query processing, i.e., after Phase 3 in Section 4.2.

Here, we make the important observation that, semantically, the filtering of such predicates often tends to be correlated to certain regions in TSCs, opening an opportunity for optimization. For example, queries that use predicates such as $\min(simSeq[0,-1]) < \theta$ (e.g., $\theta = 0.1$) suggest a tail-filtering pattern (from

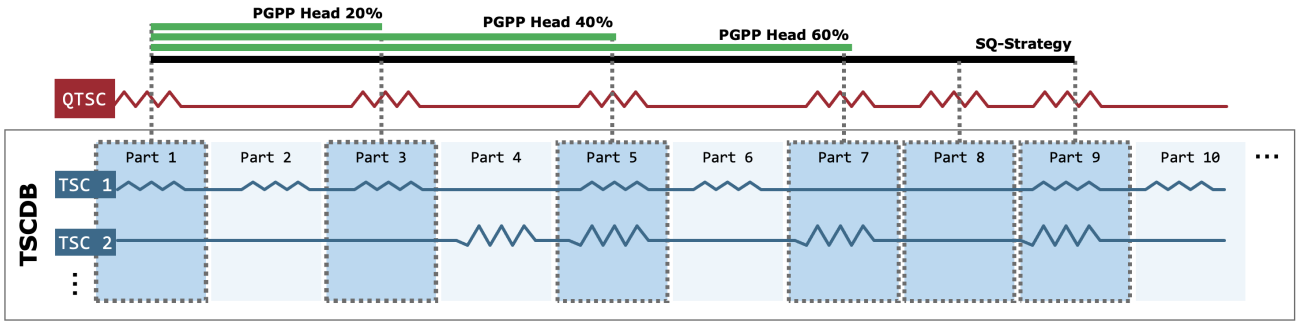

Figure 7: Comparison of SQ-Strategy and PGPP for head-filtering predicates. PGPP prioritizes early segments of the TSC based on the predicate $\max(\textit{simSeq}) > \theta$, reducing partition access compared to SQ-Strategy which loads all segments uniformly.

the right side) because the minimum distance between two converging TSCs is most likely at (or near) the tail (right) side. In contrast, a predicate like $\max(\textit{simSeq}[0,-1]) > \theta$ (e.g., $\theta = 0.9$) implies head-filtering (from the left side) because the maximum distance between two converging TSCs is most likely at (or near) the start (left) side.

Leveraging the above observations, we now propose a lightweight rule-based loading strategy which we call *Predicate-Guided Partition Prioritization (PGPP)*. PGPP dynamically selects which portions of the TSC to load based on the type of predicate. For *min* constraints, it employs a tail-first approach, loading TSC segments starting from the most recent time windows (the right side). For *max* constraints, it uses a head-first strategy, beginning from the earliest time windows on the left side (see Figure 7).

As we demonstrate in our experiments (Section 6.3), the PGPP strategy significantly reduces the number of partitions touched at query time while preserving a high recall. The reason is that the partitions first retrieved by PGPP allow identifying and eliminating most converging TSCs that do not satisfy the given predicates. As illustrated in Figure 7, PGPP can be applied in a more or less aggressive manner controlled by a system parameter, offering a tuning tradeoff between speed versus accuracy. For example, a 20% threshold is more aggressive than 40% and will result in a smaller number of partitions (Part 1 & Part 3) being loaded, and hence faster execution. However, TSC2, in this example, does not have presence in both Part 1 and Part 3. Hence if it is part of the ground truth, it will be missed under the 20% threshold. The PGPP optimization can be seamlessly integrated into both single and batch query execution pipelines.

5.5 System's Fault Tolerance

Fault tolerance in our system is inherently supported by the underlying Apache Spark infrastructure. The query pipeline operates as a series of Spark jobs, which are further distributed into individual tasks. These tasks are automatically monitored, and any failures are transparently handled through task re-execution. Additionally, all TSC datasets and index structures are stored in HDFS, enabling robust recovery from partition losses or node failures via HDFS's built-in replication and fault-tolerance mechanisms.

5.6 Convergence Query Complexity Analysis

In this section, we present the complexity analysis of the search algorithm for single query processing. Let N denote the total number of TS objects in the dataset, P the number of partitions created by the top-level index, and $|Q|$ the number of TS objects in the query TSC. The top-level index search to find the right partition executes in $O(\log P)$, and it ensures that each query TS only needs to be compared against approximately N/P candidates (instead of all N). Within each partition, the second-level quantized index enables sub-linear candidate pruning: rather than scanning all N/P candidates, only a fraction $\rho \cdot (N/P)$ are examined, where $\rho \ll 1$ reflects the selectivity of the histogram-based filtering. Finally, all TSC objects that pass the structure-similarity filters (say C) are tested for convergence using the MK-test. Hence, the overall complexity becomes:

$$O\big(|Q| \cdot (\log P + \rho \cdot \tfrac{N}{P}) + C\big) \rightarrow O\big(|Q| \cdot (\rho \cdot \tfrac{N}{P}) + C\big).$$

In contrast, a naive baseline that performs a full scan has complexity:

$$O(|Q| \cdot N) + C.$$

Thus, the two-level indexing strategy achieves up to a P/ρ-fold speedup over the baseline.

6 Experimental Evaluation

6.1 Implementation & Experimental Setup

Cluster Setup. Our cluster consists of 2 super nodes each composed of 56 Intel@Xeon CPU E5-2690 2.60GHz processors, 500GB RAM, 3.5 TB HDD (each CPU has 8 GBs of assigned dedicated memory). Each node is connected to a Gigabyte Ethernet switch and runs Ubuntu 16.04.3 LTS with Spark-2.0.2 and Hadoop-2.7.3. The memory in each machine is equally distributed across the machine's cores. Note that, while our prototype currently utilizes two supercomputer nodes–reflecting the resources currently available to our group–the system components are designed to be highly scalable on commodity hardware, without the need for high-capacity global memory or centralized processing.

6.1.1 ***Baseline Algorithms****. Single Query Strategy Baseline Algorithms.* We have two baselines:

(1) FullScan. This method harnesses the parallel processing power of the distributed infrastructure by a full scan over the *TSC* materialized data. We first group and materialize the time series data by their *TSC* id, i.e., *OID(TSC)* onto the same partition. Query processing then can use full scans over the partitions in parallel to answer the query input by computing the simSeqs with all database *TSC* objects. Afterwards, the MK-test executes in parallel across all cores to identify and report the converging TSCs.

(2) S-kNN. Here, we investigate how to effectively leverage existing search techniques to implement our target operator. Among existing approaches are traditional time series management systems enhanced with TSC semantics (Section 3), and the Sloth system [6]. Prior work in [6] has demonstrated that Sloth significantly outperforms other conventional methods, e.g., the time series systems enhanced with the strategies mentioned in Section 3 such as IMPUTE, DECOMPOSE, and CONCAT, i.e., this strategies suffered from very low accuracy and hence considered impractical. Accordingly, our evaluation focuses on a comparison with the Sloth system. We utilize its TSC kNN operator, which retrieves the *top k* most similar matches for a query object. To maximize the likelihood of identifying converging TSCs, we set a high value for k (default: K = 2000), 10 times the average number of converging results for a typical query TSC. On top of these retrieved candidates, our lightweight post-processing layer applies the MK-test to assess convergence.

Batch of Queries Strategy Baseline Algorithms. We compare with two baseline solutions:

(1) FullScan as described above with all query objects compared in the same scan. And **(2) repeated execution of the SQ-Strategy**, namely, executing the best single query strategy on each query in the batch independently.

6.1.2 ***Datasets.*** We work with the following four datasets.

(1) TSC-RandomWalk Dataset. First introduced in our prior work [6], it extends the RandomWalk benchmark widely used for time series evaluation, e.g., [23, 63, 69]. Briefly, generating D_{TSC} consists of: (1) designing the template of a typical *TSC*, and (2) generating TS objects to compose each *TSC*. We generate multiple datasets by varying the numbers of *TSC* objects N over ⟨5, 10, 15, 20⟩ millions with an average *car(TSC)*=15 where |TS|=1024, resulting in datasets of approximately ⟨260 GB, 520 GB, 790 GB, 1.1 TB⟩. We also create another series of datasets where we fix the N to 5 millions and vary the average cardinality over ⟨10, 15, 20, 25⟩ TS objects resulting in datasets of ⟨ 150 GB, 220 GB, 400 GB, 500 GB⟩.

(2) Chin Movement & (3) ECG Signal Datasets. We use two datasets from a sleep study [2]: (1) Chin Movement [67] and (2) ECG Signal [13]. We extracted the TSC objects following the method described in [6]. This results in around 5 million *TSCs* per dataset (approximately 260 GB), where each TSC on average consists of 15 TS objects. **(4) NYC Taxi Commission.** We use the yellow taxi trip records from 2009 to 2021 [49]. The TSC objects were extracted as described in [6]. This results in around 2.5x10^6 *TSCs* ≈ 110 GB.

6.1.3 ***Default Parameter Settings.*** *TSC-aware representation* has a set of parameters, including the number of sub-vectors (SV), codeBook size (CB), and SAX parameters *(SAX-Alphabet) and (SAX-Segments)*. The following default settings as in [6]: *SV = 8*, *CB=32*, *SAX-Alphabet=2* and *SAX-Segments=2*. Regarding dataset size = 5 millions TSC objects of cardinality *car(TSC)=15* TSs in each. The default batch size is 33 unless otherwise stated. For the probability level of significance for MK-test, we set β = 0.05 for the baseline, i.e., the exact solution *FullScan*, and for our approximate strategies (i.e., SQ-Strategy, ShareBQ-Strategy and RankBQ-Strategy), empirically β = 0.25 seems the right choice to achieve high accuracy. All experiments are conducted under failure-free conditions, as the fault tolerance capabilities are entirely provided by the underlying Spark infrastructure without modification.

6.1.4 ***Convergence Query Evaluation Metrics.*** In the following, we evaluate convergence query strategies by measuring the query performance of processing a bunch of 33 distinct queries. For *single query strategies*, we apply the queries individually (see Section 5.2), then we take the average of query response time. For *batch of queries strategies*, we exploit sharing opportunities while processing the queries as explained in Section 5.3. We then divide the total time of running the batch by the size of the batch. The recall of both strategies is measured by taking the average of each single query's recall – a standard metric for high-dimensional retrieval queries [5, 9, 63, 69]. Given a query Q-TSC, the set of exact converging TSCs $G(Q-TSC) = \{g_1, \cdots, g_{|G(Q-TSC)|}\}$, and the set of approximate converging *TSCs* as R(Q) = $\{r_1, \cdots, r_{|R(Q-TSC)|}\}$. Then *recall* is defined as:

$$recall = \frac{|G(Q-TSC) \cap R(Q-TSC)|}{|G(Q-TSC)|} \tag{11}$$

Notice that recall ∈ $[0, 1]$ where in the ideal case, the *recall* score is 1.0, i.e. 100%. This means all converging *TSC* are returned.

6.2 Evaluating SQ-Strategy & ShareBQ

In this section, we first evaluate the query response time of single query strategies including our proposed algorithm *SQ-Strategy* (refer to Fig. 8, 1^{st} row) under various settings. We then evaluate our optimized algorithm *ShareBQ* against the batch base method *FullScan* and our *SQ-Strategy* (refer to Fig. 8, 2^{nd} row). Finally, we report the recall of all these algorithms in Fig. 8, 3^{rd} row.

6.2.1 ***Varying Dataset Sizes By Changing Number of TSCs.*** *Single Query Strategies.* In Fig. 8 col.(a) row(1), we compare the base solution (*FullScan*) to our proposed solution(*SQ-Strategy*). *FullScan* pays the cost of returning exact answers with a slow response time. Worst yet, as the number of TSCs in the dataset increases, the response time increases exponentially. In contrast, our *SQ-Strategy* is orders of magnitude faster especially for the TB-scale dataset. This is because 1) *SQ-Strategy* only loads the partitions that are likely to contain a structural-similar TSCs, and 2) the TS-Level distance comparisons are feature-based not raw point-by-point based matches as in *FullScan*.

Comparing *SQ-Strategy* to *S-kNN*, it is consistently up to three times faster than *S-kNN* for different dataset sizes. Better yet, the recall of *SQ-Strategy* is consistently above 80% and approaching 90% as dataset sizes increase (Fig. 8 col.(a) row(3)). While *S-kNN* recall is bad, < 20% for different dataset sizes. Worse yet, *S-kNN* recall does not have a clear pattern. This reflects the fact that the *kNN* and *convergence* operators have conceptually unrelated semantics. Meaning, a close kNN match does not imply that it converges to a given query, and vice versa.

Batch of queries strategies. Refer to Fig. 8 col.(a) row(2), comparing *FullScan* to *ShareBQ*, *ShareBQ* is up to three orders of magnitude faster for TB-scale datasets. As the number of TSC objects increases, the response time of *FullScan* increases exponentially. While *ShareBQ* increases linearly compared to the base solution. This is for the same reasons mentioned above for the *Single Query Strategy*. Finally, comparing *ShareBQ* to the repeated execution of *SQ-Strategy* with each query, *ShareBQ* is two times faster than *SQ-Strategy*.

6.2.2 ***Varying Dataset Sizes By Changing TSC Cardinalities.*** *Single Query Strategies.* In Fig. 8 col.(b) row(1), comparing the base solution *(FullScan)* to our proposed solution *(SQ-Strategy)*, we found that the exact solution *FullScan* suffers from a very slow response time and from not being scalable. As the

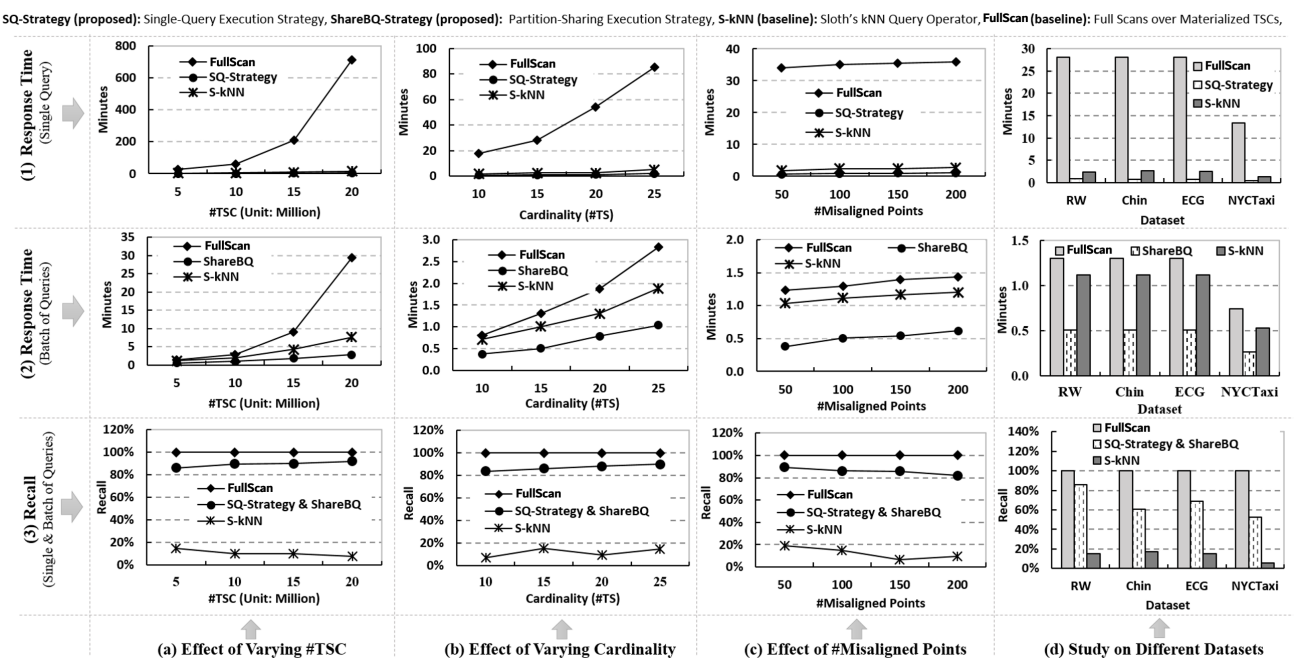

Figure 8: Evaluation of SQ-Strategy & ShareBQ: Query Response Time (1^{st} row: single query strategies, 2^{nd} row: batch of queries strategies) & Recall (3^{rd} row: single & batch of queries strategies).

cardinality of the TSC increases, the response time increases exponentially. This is expected, whether the datasets sizes increase cardinality-wise or because of number of TSCs increase, the amount of data loaded into memory is ample and thus the solution is not scalable. *SQ-Strategy* is up to four orders-of-magnitude faster compared to the baseline for the largest dataset of TSC objects with average *car(TSC)=25* (approximately 500GB). Comparing *SQ-Strategy* to *S-kNN*, it is consistently up to two times faster than *S-kNN*. Moverover, the recall of *SQ-Strategy* (Fig. 8 col.(b) row(3)) is consistently above 80% and approaching 90% as the cardinality increases. This is expected because the MK-trend test captures the convergence trend more efficiently for longer TSCs. *S-kNN's* recall is bad, < 20% for different dataset size.

Batch of Queries Strategies. In Fig. 8 col.(b) row(2), comparing *FullScan* to *ShareBQ*, the *FullScan*'s response time increases exponentially as the cardinality of the TSC increases. *ShareBQ* is up to three times faster than *FullScan*, and expected to be much faster as the trend suggests. Moreover, according to the trend in the figure, we expect that *BQ-Strategy* would be even faster compared to the baseline for larger cardinalities. Comparing *ShareBQ* to *SQ-Strategy*, *ShareBQ* is two times faster than *SQ-Strategy* while achieving the same recall (Fig. 8 col.(b) row(3)).

6.2.3 ***Varying Misaligned Data Points at the TS-level.*** This has almost the same effect on query response time of *single processing strategies, i.e., FullScan, S-kNN & SQ-strategy* (see Fig. 8 col.(c) row(1)), and *batch processing strategies, i.e., FullScan, & ShareBQ* (see Fig. 8 col.(c) row(2)). The query response time increases as the number of misaligned points increases. This is because this increase of misaligned points increases the required distance calculations at the TS-level. Despite that, the ratios between the query response time of these strategies remain unaffected. Regarding the recall (Fig. 8 col.(c) row(3)), it slightly decreases for our proposed strategies, i.e., *SQ-Strategy* and *ShareBQ*, as the number of misaligned data points increases. This is because it affects the quality of the representation-based

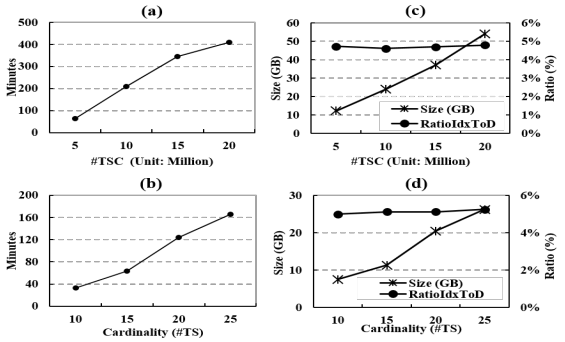

Figure 9: Index Construction Analysis (Time & Size).

distances. Despite that, it is still of an excellent quality usable in practice staying at a larger than 80% recall.

6.2.4 ***Comparative Study Across Different Datasets.*** We observe that varying the actual datasets does not affect the query response time of *single query strategies* (Fig. 8 col.(d) row(1)) nor does it affect the query response time of *batch query strategies* (Fig. 8 col.(d) row(2)). Rather, as shown in Section. 6.2.1 & 6.2.2, the query response time is dominated and affected by the dataset size. Despite that, the recall (Fig. 8 col.(d) row(3)) differs for different dataset types for our index-based strategies (i.e., SQ-strategy & ShareBQ). This is because we calculate the TS-level distances between the representations. Thus, different shapes of the TS objects affect the similarity scores, the trend analysis, and accordingly affect the recall.

6.3 Evaluating PGPP Partition Loading

We evaluate the impact of the Predicate-Guided Partition Prioritization (PGPP) technique by integrating it into the *SQ-Strategy*. PGPP adjusts the order and amount of partitions loaded based on user-defined query constraints over the similarity progression

sequence (`simSeq`). We measure the impact of two types of convergence predicates on query time, number of loaded partitions, and recall.

Case 1: `min(simSeq[0, -1]) < 0.1` *(Tail Filtering).* This scenario targets cases when relevant matches are expected near the end of the similarity progression sequence. PGPP therefore prioritizes loading partitions corresponding to the latest segments of TSCs to focus on filtering in that region. (Table 1).

Table 1: Performance under Tail-Filtering Constraint

Strategy	Partitions Loaded	Recall	Avg. Time
FullScan	4759 / 4759	1,880 (100.00%)	32,972 sec
SQ-Strategy	352 / 4759	1,662 (88.40%‡)	466.6 sec
PGPP – Tail 60%	199 / 4759	1,627 (97.89%*)	252 sec (54.0%*)
PGPP – Tail 50%	142 / 4759	1,489 (89.58%*)	215 sec (46.1%*)
PGPP – Tail 40%	114 / 4759	1,421 (85.49%*)	198.3 sec (42.5%*)
PGPP – Tail 30%	87 / 4759	1,421 (85.49%*)	179.6 sec (38.5%*)
PGPP – Tail 20%	39 / 4759	972 (58.50%*)	151 sec (32.4%*)

‡ Relative to FullScan * Relative to SQ-Strategy

Case 2: `max(simSeq[0, -1]) > 0.9` *(Head-Filterning).* This targets cases where relevant matches are expected near the beginning of the similarity progression sequence. PGPP prioritizes loading earlier segments of TSCs to focus filtering accordingly (Table: 2).

Table 2: Performance under Head-Filtering Constraint

Strategy	Partitions Loaded	Recall	Avg. Time
FullScan	4759 / 4759	5,662 (100.00%)	34,620 sec
SQ-Strategy	352 / 4759	5,011 (88.52%‡)	464.3 sec
PGPP – Head 60%	265 / 4759	4,913 (98.05%*)	263.6 sec (56.8%*)
PGPP – Head 50%	210 / 4759	4,853 (96.84%*)	235.3 sec (50.7%*)
PGPP – Head 40%	182 / 4759	4,388 (87.56%*)	209.3 sec (45.1%*)
PGPP – Head 30%	153 / 4759	4,249 (84.76%*)	191.3 sec (41.2%*)
PGPP – Head 20%	94 / 4759	3,139 (62.63%*)	165 sec (35.5%*)

‡ Relative to FullScan * Relative to SQ-Strategy

These experiments show that PGPP significantly reduces query latency—by up to 3×—while maintaining high recall. The trade-off between performance and accuracy can be tuned by controlling the extent of partition loading based on query semantics.

6.4 Evaluation of Index Construction

Index Construction Time. This includes data shuffling, converting TSC objects into their *TSC-aware feature representation*, and organizing them into their local structure. In Fig. 9 (a) and (b), we vary the number of *TSCs* and number of *TSs* in each *TSC*, respectively. In both cases, the index construction time is linear in the dataset size. Index construction time will be quickly amortized after executing even just a few queries. For example, TSC-index construction plus executiong of a single query using *SQ-Strategy* is 30 seconds faster than executing a single query using *FullScan*.

Index Size. In Fig. 9 (c) and (d), we study the impact on index size when varying the number of TSC objects and cardinality, respectively. The index size consistently represents 5% of the dataset size. This is because the index size is dominated by the size of the feature representation of the TS objects, one for each. Other index components not only are relatively small, in KBs to few MBs, but also independent of the dataset size.

7 Conclusion

Convergence queries on time series data, particularly on the Time Series Compound (TSC) model, represent an unexplored class of analytics relevant to domains from diagnostics, maintenance, to security. Our work is the first to address convergence queries at scale, introducing a distributed processing framework that leverages the parallelism of modern computing infrastructures to evaluate convergence over terabyte-scale TSC datasets efficiently. We design a TSC-aware similarity progression measure and define the semantics of convergence queries. We develop scalable query strategies that achieve orders-of-magnitude speedup over baseline methods, yet with consistently high accuracy. Our predicate-guided partition prioritization optimization reduces data loading costs, achieving up to 3× speedup with minimal loss in recall.

Artifacts

All our source code and implementation artifacts are publicly available at GitHub: https://github.com/kaluaim/tsc-convergence-queries. This repository includes the full implementation of the convergence query engine, including the PGPP optimization, SQ-Strategy, and batch execution strategies.

Moreover, to demonstrate the integration with an existing time series management system, we built a plugin for Apache IoTDB. This plugin enables invoking convergence queries from within IoTDB and showcases how our system can interoperate with existing DB systems. The plugin source is available at: https://github.com/kaluaim/tsc-cq-iotdb-plugin.

References

[1] 1989. *Real Analysis*. Pearson Education. https://books.google.com/books?id=J_1CCoeDXsgC

[2] 2020. MrOS Sleep Study. https://sleepdata.org/datasets/mros. Accessed: 2025-10-11.

[3] 2022. Sloth Infrastructure. https://github.com/kaluaim/sloth. Accessed: 2025-10-11.

[4] Amaia Abanda, Usue Mori, and Jose A Lozano. 2019. A review on distance based time series classification. *Data Mining and Knowledge Discovery* 33, 2 (2019), 378–412.

[5] Noura Alghamdi, Liang Zhang, Huayi Zhang, Elke A Rundensteiner, and Mohamed Y Eltabakh. 2020. ChainLink: Indexing Big Time Series Data For Long Subsequence Matching. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 529–540.

[6] Noura S. Alghamdi, Liang Zhang, Elke A. Rundensteiner, and Mohamed Y. Eltabakh. 2022. Scalable Time Series Compound Infrastructure *(SIGMOD/PODS '22)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3514221.3517888

[7] James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (Nov. 1983), 832–843. doi:10.1145/182.358434

[8] Tomohiro Ando and Jushan Bai. 2017. Clustering huge number of financial time series: A panel data approach with high-dimensional predictors and factor structures. *J. Amer. Statist. Assoc.* 112, 519 (2017), 1182–1198.

[9] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-index: pushing the scalability-accuracy boundary for approximate kNN search in high-dimensional spaces. *Proceedings of the VLDB Endowment* 11, 8 (2018), 906–919.

[10] Andreas Bader, Oliver Kopp, and Michael Falkenthal. 2017. *Survey and Comparison of Open Source Time Series Databases*.

[11] Francisco J Baldán and José M Benítez. 2019. Distributed fastshapelet transform: a big data time series classification algorithm. *Information Sciences* 496 (2019), 451–463.

[12] Donald J Berndt and James Clifford. 1994. Using dynamic time warping to find patterns in time series.. In *KDD workshop*, Vol. 10. Seattle, WA, USA:, 359–370.

[13] Terri Blackwell, Kristine Yaffe, Sonia Ancoli-Israel, Susan Redline, Kristine E Ensrud, Marcia L Stefanick, Alison Laffan, Katie L Stone, and Osteoporotic Fractures in Men Study Group. 2011. Associations between sleep architecture and sleep-disordered breathing and cognition in older community-dwelling men: the osteoporotic fractures in men sleep study. *Journal of the American Geriatrics Society* 59, 12 (2011), 2217–2225.

[14] Timothy M Brown and Jorgen Christensen-Dalsgaard. 1990. A technique for estimating complicated power spectra from time series with gaps. *The Astrophysical Journal* 349 (1990), 667–674.

[15] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with isax2+. *Knowledge and information systems* 39, 1 (2014), 123–151.

[16] Yanchuan Chang, Egemen Tanin, Gao Cong, Christian S. Jensen, and Jianzhong Qi. 2024. Trajectory Similarity Measurement: An Efficiency Perspective. *Proc. VLDB Endow.* 17, 9 (May 2024), 2293–2306.

[17] Lu Chen, Yonggang Chi, Yingying Guan, and Jialin Fan. 2019. A hybrid attention-based EMD-LSTM model for financial time series prediction. In *2019 2nd International Conference on Artificial Intelligence and Big Data (ICAIBD)*. IEEE, 113–118.

[18] Mayukh Roy Chowdhury, Sharda Tripathi, and Swades De. 2020. Adaptive Multivariate Data Compression in Smart Metering Internet of Things. *IEEE Transactions on Industrial Informatics* (2020).

[19] National Research Council et al. 1996. *New materials for next-generation commercial transports*. Vol. 476. National Academies Press.

[20] Philippe Cudre-Mauroux, Eugene Wu, and Samuel Madden. 2010. TrajStore: An adaptive storage system for very large trajectory data sets. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 109–120. doi:10.1109/ICDE.2010.5447829

[21] Michele Dallachiesa, Themis Palpanas, and Ihab F Ilyas. 2014. Top-k nearest neighbor search in uncertain data series. *Proceedings of the VLDB Endowment* 8, 1 (2014), 13–24.

[22] John P d'angelo and Douglas B West. 2000. *Mathematical Thinking: Problem-Solving and Proofs*. Upper Saddle River, NJ: Prentice-Hall.

[23] Time Series Predict DB. 2017. KV-Match: An efficient subsequence matching approach for large scale time series. (2017).

[24] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1542–1552.

[25] Lingzi Duan, Fusheng Yu, Witold Pedrycz, Xiao Wang, and Xiyang Yang. 2018. Time-series clustering based on linear fuzzy information granules. *Applied Soft Computing* 73 (2018), 1053–1067.

[26] Emadeldeen Eldele, Mohamed Ragab, Zhenghua Chen, Min Wu, Chee Keong Kwoh, Xiaoli Li, and Cuntai Guan. 2021. Time-Series Representation Learning via Temporal and Contextual Contrasting. arXiv:2106.14112 [cs.LG] https://arxiv.org/abs/2106.14112

[27] Christos Faloutsos, Jan Gasthaus, Tim Januschowski, and Yuyang Wang. 2019. Classical and contemporary approaches to big time series forecasting. In *Proceedings of the 2019 International Conference on Management of Data*. 2042–2047.

[28] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. *Acm Sigmod Record* 23, 2 (1994), 419–429.

[29] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery* 33, 4 (2019), 917–963.

[30] Gajanan Gawde and Jyoti Pawar. 2018. Similarity search of time series trajectories based on shape. In *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data* (Goa, India) *(CODS-COMAD '18)*. Association for Computing Machinery, New York, NY, USA, 340–343. doi:10.1145/3152494.3167986

[31] P. Gyau-Boakye and GA Schultz. 1994. Filling gaps in runoff time series in West Africa. *Hydrological sciences journal* 39, 6 (1994), 621–636.

[32] Khaled H Hamed and A Ramachandra Rao. 1998. A modified Mann-Kendall trend test for autocorrelated data. *Journal of hydrology* 204, 1-4 (1998), 182–196.

[33] M. Manjurul Hussain and Ishtiak Mahmud. 2019. pyMannKendall: a python package for non parametric Mann Kendall family of trend tests. *Journal of Open Source Software* 4, 39 (2019), 1556.

[34] Maurice G Kendall and JD Gibbons. 1970. Rank correlation methods 4th ed. *Griffin, London* (1970).

[35] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems* 3, 3 (2001), 263–286.

[36] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. *Knowledge and information systems* 7, 3 (2005), 358–386.

[37] Mourad Khayati, Alberto Lerner, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. Mind the gap: an experimental evaluation of imputation of missing values techniques in time series. *Proceedings of the VLDB Endowment* 13, 5 (2020), 768–782.

[38] Huanhuan Li, Jingxian Liu, Zaili Yang, Ryan Wen Liu, Kefeng Wu, and Yuan Wan. 2020. Adaptively constrained dynamic time warping for time series classification and clustering. *Information Sciences* 534 (2020), 97–116.

[39] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Tianfu He, Jie Bao, Junbo Zhang, Liang Hong, and Yu Zheng. 2021. TrajMesa: A Distributed NoSQL-Based Trajectory Data Management System. *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1. doi:10.1109/TKDE.2021.3079880

[40] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD '03*, 2–11. doi:10.1145/882082.882086

[41] Henry B Mann. 1945. Nonparametric tests against trend. *Econometrica: Journal of the econometric society* (1945), 245–259.

[42] Barry J. Materson, Domenic J. Reda, and David W. Williams. 1998. Comparison of Effects of Antihypertensive Drugs on Heart Rate: Changes From Baseline by Baseline Group and Over Time. *American Journal of Hypertension* 11, 5 (05 1998), 597–601. doi:10.1016/S0895-7061(97)00495-0 arXiv:https://academic.oup.com/ajh/article-pdf/11/5/597/423719/11_5_597.pdf

[43] Themis Palpanas. 2019. Evolution of a Data Series Index. In *ISIP*. Springer.

[44] John Paparrizos, Haojun Li, Fan Yang, Kaize Wu, Jens E. d'Hondt, and Odysseas Papapetrou. 2024. A Survey on Time-Series Distance Measures. arXiv:2412.20574 [cs.DB] https://arxiv.org/abs/2412.20574

[45] R. Tyrrell Rockafellar and Roger J-B Wets. 2005. *Variational Analysis* (3rd printing ed.). Grundlehren der mathematischen Wissenschaften, Vol. 317. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-540-31002-0

[46] Kexin Rong, Clara E Yoon, Karianne J Bergen, Hashem Elezabi, Peter Bailis, Philip Levis, and Gregory C Beroza. 2018. Locality-sensitive hashing for earthquake detection: A case study of scaling data-driven science. *arXiv preprint arXiv:1803.09835* (2018).

[47] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. 2015. Path Similarity Analysis: A Method for Quantifying Macromolecular Pathways. *PLOS Computational Biology* 11, 10 (Oct. 2015), e1004568. doi:10.1371/journal.pcbi.1004568

[48] Jin Shieh and Eamonn Keogh. 2008. iSAX: indexing and mining terabyte sized time series. In *SIGKDD*. ACM, 623–631.

[49] NYC Taxi and Limousine Commission. 2021. NYC Taxi Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Accessed: 2025-10-11.

[50] Henri Theil. 1950. A rank-invariant method of linear and polynomial regression analysis. *Indagationes mathematicae* 12, 85 (1950), 173.

[51] Sana Tonekaboni, Danny Eytan, and Anna Goldenberg. 2021. Unsupervised Representation Learning for Time Series with Temporal Neighborhood Coding. arXiv:2106.00750 [cs.LG] https://arxiv.org/abs/2106.00750

[52] Kostas Triantafyllopoulos. 2007. Convergence of Discount Time Series Dynamic Linear Models. *Communications in Statistics—Theory and Methods* 36 (08 2007), 2117–2127. doi:10.1080/03610920601143535

[53] Michail Vlachos, Dimitrios Gunopoulos, and George Kollios. 2002. Discovering Similar Multidimensional Trajectories. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*. IEEE Computer Society, USA, 673.

[54] Michail Vlachos, Dimitrios Gunopulos, and Gautam Das. 2004. Indexing time-series under conditions of noise. In *Data mining in time series databases*. World Scientific, 67–100.

[55] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series Database for Internet of Things. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2901–2904. doi:10.14778/3415478.3415504

[56] Qitong Wang and Themis Palpanas. 2021. Deep Learning Embeddings for Data Series Similarity Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 1708–1716. doi:10.1145/3447548.3467317

[57] Xiaqing Wang, Zicheng Fang, Peng Wang, Ruiyuan Zhu, and Wei Wang. 2017. A Distributed Multi-level Composite Index for KNN Processing on Long Time Series. In *DASFAA*. Springer, 215–230.

[58] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn Keogh. 2013. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery* 26, 2 (2013), 275–309.

[59] Eugene Wu, Philippe Cudre-Mauroux, and Samuel Madden. 2009. Demonstration of the TrajStore System. *PVLDB* 2 (08 2009), 1554–1557. doi:10.14778/1687553.1687589

[60] Jiaye Wu, Peng Wang, Ningting Pan, Chen Wang, Wei Wang, and Jianmin Wang. 2019. KV-Match: A subsequence matching approach supporting normalization and time warping. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 866–877.

[61] Dong Xie, Feifei Li, and Jeff M Phillips. 2017. Distributed trajectory similarity search. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1478–1489.

[62] Fengli Xu, Yuyun Lin, Jiaxin Huang, Di Wu, Hongzhi Shi, Jeungeun Song, and Yong Li. 2016. Big data driven mobile traffic understanding and forecasting: A time series approach. *IEEE transactions on services computing* 9, 5 (2016), 796–805.

[63] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2017. Dpisax: Massively distributed partitioned isax. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1135–1140.

[64] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Dennis Shasha. 2017. Radiussketch: massively distributed indexing of time series. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 262–271.

[65] Zhihan Yue, Yujing Wang, Juanyong Duan, Tianmeng Yang, Congrui Huang, Yunhai Tong, and Bixiong Xu. 2022. TS2Vec: Towards Universal Representation of Time Series. arXiv:2106.10466 [cs.LG] https://arxiv.org/abs/2106.10466

[66] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[67] Guo-Qiang Zhang, Licong Cui, Remo Mueller, Shiqiang Tao, Matthew Kim, Michael Rueschman, Sara Mariani, Daniel Mobley, and Susan Redline. 2018. The National Sleep Research Resource: towards a sleep data commons. *Journal of the American Medical Informatics Association* 25, 10 (2018), 1351–1358.

[68] Hanyuan Zhang, Xingyu Zhang, Qize Jiang, Baihua Zheng, Zhenbang Sun, Weiwei Sun, and Changhu Wang. 2021. Trajectory similarity learning with auxiliary supervision and optimal matching. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*.

[69] Liang Zhang, Noura Alghamdi, Mohamed Y Eltabakh, and Elke A Rundensteiner. 2019. TARDIS: Distributed indexing framework for big time series data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1202–1213.

A Additional Details on Batch-Queries Strategies

Batch-Queries execution strategies allows multiple, say K, queries in a batch to be processed concurrently, i.e., the input is *batch-QueryTSC* in Fig. 2, line 1. The results obtained by these strategies are identical to those obtained if we had executed *SQ-Strategy* for each of the K queries individually. Better yet, they complete the processing in a much shorter time and with a better resource utilization. In the following, we describe two batch strategies in-depth.

A.0.1 Partition-Sharing Execution Strategy (ShareBQ-Strategy). Given that different queries might identify potentially several common partitions, this provides opportunities for shared processing. In principle, sharing similar partitions reduces the query performance time due to having to load potentially less additional partitions, while utilizing the processing infrastructure. Thus, given a batch of *k Q-TSC* queries, this strategy starts by identifying the union of candidate partitions of all queries in the batch, as described below.

Phase 1: Candidate Set Retrieval. First, the master node extracts the *start time t* of all *QTS* objects of every query *Q-TSC* in the batch and takes the union of these start times to determine partitions to load. Then, it computes all overlaps for the *QTS objects* (refer to Eq. 10). Then, the master node queries the time structure in TSC-index's top level with these time ranges to get their candidate partitions IDs. Thereafter, the master node loads these partitions into workers' memory. The decomposed *QTS* are transformed into their *TSC-aware feature representation [6]* and then grouped by their start time *t* which results in pairs: (t, List <oid(Q-TSC), QTS>). These pairs are broadcasted to the workers' memory.

Phase 2: Generating Similarity Progression Sequences. *In the TS-level structure-based filtering*, each worker does the following for each pair <oid(Q-TSC), QTS> in the list *(t, List <oid(Q-TSC), QTS>)*: 1) the worker fetches the *TS* objects that overlapped with QTS objects based on time *t* from the local structure, i.e., HashMap. 2) Then, each fetched *TS* object is touched once. Meaning, the candidate *TS* is compared with each single *QTS* in the list of values (i.e., *List <oid(Q-TSC), QTS>*). 3) The result would be represented as key-value pair (Key:(oid(Q-TSC), oid(TSC)), value: (Qt, dist)). *Second*, the master node groups these partial results by key which results in a collection of pairs in the form: *(Key:(oid(Q-TSC), oid(TSC)), Value: List<(Qt, dist)>)*. Then, for each pair, we sort the values based on time, followed by regularizing the sorted list of distances.

Phase 3: Analyzing the Trend. Given the resulting collection from phase 2 in the form: *(Key:(oid(Q-TSC)), Value: List<(oid(TSC), regSimSeq)>)*. Each *regSimSeq* candidates is passed to MK-test. Lastly, we note that the ranking would be similar to that of *SQ-strategy*, where we perform ranking for each *Q-TSC* independently.

A.0.2 Partition-Ranking Execution Strategy (RankBQ-Strategy). We now extend *ShareBQ* by providing additional speedup - however at the cost of decrease in the approximation accuracy. That is, *RankBQ-Strategy* features an elegant speed–accuracy trade-off capability. As foundation, we formalize this as an optimization problem. Thereafter, we design a solution that derives the optimal selection of candidate partitions to process.

The goal of *RankBQ* strategy is to give each partition a score based on its impact on query response time and accuracy. This way, we would prioritize the partitions based on these scores. This allows us to ignore some of the partitions with the lower scores to further expedite the query response time, with a small accuracy loss. Our scoring strategy for a candidate partition *(p)* depends on three important scores explained in depth as follows.

AccessScore. It classifies the candidate partitions from being "*heavy-utilization*" partitions to "*light-utilization*" partitions. A partition is called *heavy-utilized* if it contains candidates for many *QTS* in the batch, and *light-utilized* if has candidates for very few *QTS* in the batch. The *accessScore* $\in (0, 1]$, i.e., the closer it is to 1 the heavier the partition.It is calculated as *accessScore = numOfAccess/size(BoQ)*, where numOfAccess=number of QTS that access the partition and *size(BoQ)* denotes the number of query Q-TSC objects in the batch. Meaning, the *light-utilizated* partitions are less critical and thus can be ignored because the number of Q-TSC affected by this omission would be relatively small.

QuantityScore. It is also important to make sure that the Q-TSC queries affected by the omission are not critically impacted. Thus, we incorporate this measure, which we call *quantityScore*. Given a *QTS* in the query batch, this measure computes the ratio of this *QTS* to the cardinality of the single Q-TSC it is member of, i.e., if *car(Q-TSC)=x*, then ratio $quan(QTS) = 1/x$. Clearly, the larger the cardinality, the smaller the value of *quantityScore*. Now, for each partition *(p)*, we set *quantityScore quantityScore*(p) = $\max(\{\text{quan}_0, \text{quan}_1, \ldots, \text{quan}_{size(BoQ)}\})$. Notice that *quantityScore* $\in (0, 1]$.

PositionScore. It captures the relative position of *QTS* in its *Q-TSC*, along with how critical this position is relative to capturing the trend. We studied the importance of masking QTS from head, middle and tail experimentally and found that masking from head or tail has almost the same effect. Plus, this is more critical than masking from the middle. We thus adopt the following optimization procedure. First, we divide each single Q-TSC in the batch into three equal time windows: *head*, *middle*, and *tail*. Thus, each QTS would be annotated with its position computed based on which time window of the single Q-TSC it is member of. This position information for every QTS in the batch are used to score the partition with what we call the *positionScore*. Namely, for a given partition p, we calculate the summation of the ratio of heads, middles, tails of QTS objects that access *p* relative to their total number *numOfAccess* (See line 10, Algorithm 2). Notice that these ratios are multiplied by a weight, i.e., α, β, and γ, respectively, to reflect their relative importance.

Combining the three scores. Once we have the aforementioned scores assigned to each candidate partition, we compute the final *score* by multiplying the three. We give the user the ability to ignore, reduce or magnify any of these scores by assigning values to the user-provided parameters weights, i.e., *A*, *B*, *and* Γ (line 12, Algorithm 2). The overall strategy is provided in Algorithm 1 and Algorithm 2. Next, we explain both in more detail below.

Collecting statistics about the queries to score candidate partitions (Algorithm 1). In order to compute the aforementioned scores for each candidate partition, we need to collect statistics about the queries (i.e., *Q-TSC* objects) in the batch as in Algorithm 1.

We start by decomposing each *Q-TSC* into its *QTS* objects (Line 7-8). For each *QTS* object, we find its position in the Q-TSC it is member of. Compute the ratio of this *QTS* to the cardinality

Algorithm 1: Collect Statistics to Score Partitions.

```
Input : BoQ ▷ Batch of TSC Queries (Q-TSCs),
        TStr ▷ Time Structure (maps time to partitionID).
Output: SPIDs ▷ Scored Parition IDs.
1  Declare:
2      canPIDs: ⟨⟩ ▷ stores candidate Partition IDs
3      PIDAcc: ⟨⟩ ▷ stores mapping PIDs to # of QTSs access
4      PIDPos: ⟨⟩ ▷ stores mapping PIDs to QTSs positions
5      PIDQuan: ⟨⟩ ▷ stores mapping PIDs to QTSs quantity
6  foreach Q-TSC in BoQ do
7      foreach (Qt, QTS) in Q-TSC do
8          pos ← position(QTS) = Head, Middle, or Tail
9          quan ← quantify(QTS) = 1/car(Q-TSC)
10         cPIDs ← TStr.getPID(Eq. 10(Qt,|QTS|))
11         canPIDs ← canPIDs + (cPIDs, pos, quan)
12 foreach (cPIDs, pos, quan) in canPIDs do
13     foreach PID in cPIDs do
14         PIDAcc ← PIDAccess + (PID, 1)
15         PIDPos ← PIDPos + ((PID, pos),1)
16         PIDQuan ← PIDQuan + (PID, quan)
17 SPIDs ← scorePID(PIDAcc, PIDPos, PIDQuan) ▷ Algorithm 2
18 return SPIDs
```

of the Q-TSC it is member of (Line 10), followed by finding all candidate partition for this *QTS* (Line 12) by accessing the TSC-index's top layer. For each query *QTS*, we then save the list of its candidate partitions IDS annotated with this *QTS* position and quantification (Line 13). Each candidate partition could be a candidate for different *QTS* objects in the batch. Thus, after the processing for all *QTS* has been completed, we are ready to aggregate this partial intermediate scores for each candidate partition. Mainly, for each partition ID in the candidate partition IDs list (Algorithm 1, line 15), we have three variables to update: *PIDAcc*, *PIDPos*, and *PIDQuan*.

These variables store partial intermediate scores. For instance, *PIDAcc* stores the pair (PID, 1) indicating a single count for each unique QTS that touches partition p (Line 16). *PIDPos* stores ((PID, position value), 1) indicating a single count for the position of each QTS that touches partition p (Line 17). *PIDQuan* stores (PID, quan) indicating the cardinality quantification of each single QTS that touches p (Line 18). Thereafter, this partial intermediate scores are passed to Algorithm 2 to aggregate them to result in final full scores.

Scoring candidate partitions (Algorithm 2). Algorithm 2 is comprised of two main phases. 1) Aggregating the partial intermediate scores collected for each candidate partition, and 2) scoring these partitions. First, we aggregate the access count and the positions counts for each partition by triggering a *reduce by key* action where the partition ID is the key (Line 3-4). Besides, we collect *QTS* quantification for each partition by performing the *group by key* action (Line 5). We then join the results of these aggregated scores by the key, which is the partition ID (Line 6). Now, we have the three scores ready for each partition ID (Line 6). Namely, access score (Line 9), position score (Line 10), quantity score (Line 11), and the final score that aggregate them all (Line 12). We then add the partition ID with its final score into the list (Line 13). Finally, after assigning scores to all candidate partitions, we sort them based on the scores.

Ignoring partitions with lower scores. Given the set of candidate partitions IDs along with their scores, we determine strategically which partitions to ignore to most effectively utilize the resources and thus reduce I/O and computational costs. We tackle this by calculating the number of execution cycles that it takes the cluster to perform the job.

$$\#OfCycles = \frac{number\ of\ candidate\ partitions}{number\ of\ worker\ machines} \tag{12}$$

Now the user can decide how many cycles to skip by setting the user-parameter ignorance percentage *(ignorPerc)*, and thus:

$$numOfCycToIgnore = ignorPerc * numOfCycles \tag{13}$$

Thus, we find the number of partitions to ignore as:

$$\#OfPartToIgnore = \#OfCycToIgnore * \#OfWorkers \tag{14}$$

Once we determine the number of partitions to ignore, we simply get the sorted partition IDs by their scores and ignore *numOfPartToIgnore* partitions with the lowest score. Then, we pass this updated list of partition IDs to *phase 2* of the *ShareBQ-Strategy* (Section A.0.1). Thereafter, the query strategy will follow the same steps that *ShareBQ-Strategy* takes.

Algorithm 2: *scorePID:* Score Candidate Partitions.

```
Input : PIDAcc, PIDPos, PIDQuan ▷ Algorithm 1 (Line 3-5).
1  Initialize:
2  unSortedSPIDs ← to store scored partitiones
3  RDD1[(PID, sumOfAccess)] ← PIDAcc.reduceByKey(_+_)
4  RDD2[(PID, List<sumOfPos>)] ←
     PIDPos.reduceByKey(_+_)
5  RDD3[(PID, List<quan>)] ← PIDQuan.groupByKey()
6  RDD[(PID, sumOfAccess, List<sumOfPos>, List<quan>)]
     ← RDD1.join( RDD2.join(RDD3))
7  foreach PID in PIDsMeasures do
8      numAcc ← PID.sumOfAccess
9      accessScore ← numOfAccess/size(BoQ)
10     positionScore ← (α * Heads/numAcc, β * Middles/numAcc,
         γ * Tails/numAcc)
11     quantityScore ← Max(PID.List<quan>)
12     score ← ((A*accessScore) * (B * positionScore) * (Γ*
         quantityScore))
13     unSortedSPIDs ← unSortedSPIDs + (PID, score)
14 SPIDs ← unSortedSPIDs.sortBy(score)
15 return SPIDs
```

B Additional Details On PGPP

Predicate-Guided Partition Prioritization (PGPP) directs the order of partition loading based on user-defined query predicates such as `min(simSeq)`, `max(simSeq)`, or `median(simSeq)`. This appendix presents the detailed pseudocode showing how PGPP selects a constrained subset of partitions according to the predicate type and loading threshold (e.g., 20%, 40%).

PGPP operates over a list of ranked segment timestamps (`rsts`) and selects which partitions to load using the predicate type (`min`, `max`, or `median`), the loading threshold, and a system-defined

Algorithm 3: PGPP Partition Selection Strategy

```
Input  : rsts: Ranked time segment indices
         measurement: Predicate type (min, max, median)
         threshold: Fraction of segments to load (e.g., 0.2)
         cutPoints: Array of index cut points
         minPartitions: Minimum number of partitions
         lenOfTS: Length of each time series segment
         overlapDataPoints: Overlap length between segments
Output : results: Set of selected partition IDs
results ← empty set
partitionsCount ← ceil(length(rsts) × threshold)
if partitionsCount < minPartitions then
    partitionsCount ← minPartitions
switch (measurement) do
    max: offset ← 0
    min: offset ← length(rsts) − partitionsCount
    median: offset ← ceil(length(rsts) / 2) − ceil(partitionsCount / 2)
for i ← 0 to partitionsCount − 1 do
    rst ← rsts[i + offset]
    l ← rst − lenOfTS + overlapDataPoints
    if l < 0 then
        l ← 0
    r ← rst + lenOfTS − overlapDataPoints
    leftP ← index of first cut point > l
    rightP ← index of first cut point > r
    if leftP < 0 then
        leftP ← length(cutPoints)
    if rightP < 0 then
        rightP ← length(cutPoints)
    Add all partitions in [leftP, rightP] to results
return results
```

minimum number of partitions to ensure reliability. These selections are then mapped to partition identifiers using the cut-point boundaries of the system index.

Reconstructing simSeq After PGPP Filtering

PGPP aggressively reduces the number of initially loaded partitions to lower I/O cost. Consequently, similarity progression sequences (`simSeq`) for candidate TSCs are initially constructed using only a subset of the full data.

To avoid any loss of accuracy due to partial data, PGPP performs a second-stage refinement step for each candidate that passes the initial filter:

(1) The system fetches the full TSC record either from a local cache or a backend database.
(2) It then reconstructs the complete `simSeq` by comparing all segments of the full candidate TSC against the query TSC.
(3) The full sequence is re-evaluated against the user-specified predicate (e.g., `min(simSeq) < 0.1`).

This deferred full-resolution validation ensures that PGPP maintains high recall while significantly reducing overall compute and I/O costs. Furthermore, it preserves compatibility with both strict and statistical convergence modes, making PGPP robust to practical deployment settings.

This two-phase design enables PGPP to deliver a balanced strategy that achieves fast filtering via partial loads, and accurate results via selective recomputation of similarity progression on qualified candidates.

C Additional Ablation Studies

• **Study on the data balancing across the index partitions.** In this experiment, we examine the distribution of time series (TS) objects across the index partitions. To determine the partitioning boundaries at the first level of the index (see Section 5.1), we employ a sampling technique that estimates the distribution of the TS objects' start times. The objective is to produce partitions of approximately equal size, which is critical for efficient parallel processing.

Our evaluation is conducted on two dataset scales–1M and 10M TSCs–from the RandomWalk dataset. Table 3 reports the statistics of partition sizes, measured by the number of TS objects. The results show that the partitions are well balanced, exhibiting both low standard deviation (the **Std** column) and a small coefficient of variation, which represents the standard deviation as a percentage of the mean.

Table 3: Statistics on the Index Partition Sizes

Dataset	Min	Max	Avg	Std	Coeff. of Variation
1M TSC (#TS: 14,498,825)	29,560	33,463	31,411.48	623.6	2.0%
10M TSC (#TS: 145,006,966)	27,260	33,442	30,981.88	1125.88	3.6%

• **Study on the effectiveness of the index's second level.** In this experiment, we evaluate the performance benefits of employing the second-level index (the content-based index in Figure 5) relative to the first-level index. In the latter configuration, the TS objects within each partition are maintained without indexing and are scanned sequentially at query time.

The evaluation is conducted on four dataset sizes derived from the RandomWalk dataset, as summarized in Table 4. All experimental parameters are set according to the default configuration described in Section 6.1.3. The table reports the execution time (in minutes) for a single query under both indexing strategies. The results clearly demonstrate the importance of the second-level index, which yields a query execution speedup of approximately 50% to 60%.

Table 4: Single Query Execution Time (in mins) under Two Index Configurations.

Dataset	Level 1 Only	Level 1 + Level 2	Speedup %
5M TSC	11	5.1	54%
10M TSC	13.3	5.8	56%
15M TSC	17.3	6.9	60%
20M TSC	22.7	9.4	59%

• **Study on the impact of batch size on batch processing.** In batch processing, the size of the batch (the number of query objects) is a critical parameter. In Fig. 10 (a), we study the effect of varying the batch size on the query's response time. As illustrated, the larger the batch size, the smaller the average query response time (the y axis) for both *ShareBQ* and *FullScan*. Whereas, on the recall side, varying the batch size has no impact on the accuracy as expected (Fig. 10 (b)). The main reason the average query response time goes down is that the execution time is dominated by the I/O operations (i.e., loading the partitions into the workers' memory). This anticipated observation served

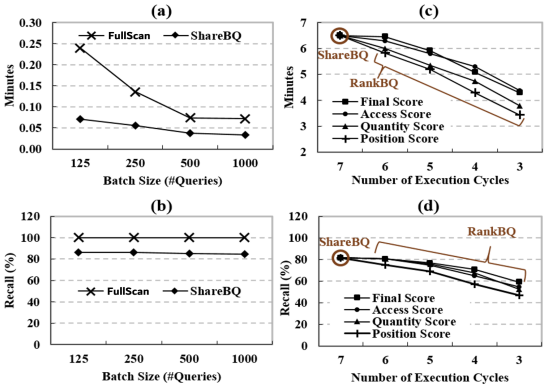

Figure 10: Evaluating Batch Processing Strategies.

as the primary motivation for the design of the *RankBQ Strategy* presented in Section 5.3 and Appendix A. In the following, we present experimental evaluation on the *RankBQ Strategy*.

• **Study on the RankBQ Strategy on batch processing.** As presented in Section 5.3 and Appendix A, the core idea behind the *RankBQ Strategy* is to skip loading the data partitions whose contributions to the queries' answers are expected to be very limited. And thus, *RankBQ* provides a tradeoff between speeding up queries' response time, and hopefully, marginal and low drop in accuracy.

Recall that *RankBQ* relies on estimating individual scores (namely, *accessScore*, *quantityScore*, and *positionScore*) to each partition. And then, aggregate these scores to a *finalScore* as explained in Appendix A (Algorithms 1 and 2). Before generating the batch of the queries (see Section 6.1.2), we start by setting the distribution of the number of TS and the gap lengths to exponential. This guarantees that there will be definitely some shared candidate partitions between the queries. It thus guarantees that there is a room for optimization so we could test our algorithms.

We generate a batch of 1000 queries and set executors to 56. By applying Algorithm 1, we find that the number of partitions is 392. Thus, we have 7 execution cycles (Eq. 12). In Fig. 10 (c) & (d), we show the effect of each score on the query response time and recall when performing all cycles without ignoring any cycle as in *ShareBQ* (i.e., 7 cycles) versus ignoring up to 60% of cycles (i.e., 4 cycles). As expected, the less number of execution cycles, the faster the query response time and the lower the recall for all scoring options. The aggregated *finalScore* achieves the best recall among the other alternatives (Fig. 10 (d)). Also, *accessScore* is the second-best in recall, however, the slowest because the ignored partitions have the lowest computation costs as they contain the fewest number of candidate matches. The *position score* metric is the fastest, as it mitigates the costs of the MK-test. The users can select which score to use and/or to weigh them to balance between the gain in time and the loss in recall of these scores as illustrated in Algorithm 2 Line 12.

D Index Maintenance

Maintaining the index structure incrementally under data appends— which is the dominant operation in time series applications compared to deletions— is relatively straightforward. Algorithm 4 outlines the key steps required to preserve balanced partitions following an update. Specifically, after inserting the new TSC dataset D into the current index to obtain INX_{new} (Line 1), the index statistics are updated (Line 2). Next, any partition P that exceeds the computed average size by more than $\beta\%$ is split into two adjacent partitions (Lines 4–8). Conversely, if a partition P falls below the computed average size by more than $\beta\%$, the algorithm first attempts to merge P with a sibling partition, when possible (Lines 12–14). If merging is not feasible (as it will be over-sized), then P and its sibling are combined and then re-split into two balanced partitions (Line 16).

E Integration with Time Series Management Systems

To demonstrate the feasibility of integrating our convergence query engine with existing time series management systems, we developed a plugin for Apache IoTDB. The plugin enables direct invocation of indexing and convergence query operations through extended SQL commands in IoTDB.

Integration Workflow. The integration process consists of two main phases:

(1) **TSC Conversion and Indexing.** Raw time series data stored in IoTDB is first transformed into Time Series Compound (TSC) format. This conversion is triggered by a custom SQL command, and the resulting TSC dataset is then indexed using our Spark-based backend system.
(2) **Convergence Query Execution.** Once the index is built, users can run convergence queries directly from IoTDB using another extended SQL command. The plugin passes the query parameters to our system, which performs query evaluation. Due to limitations in IoTDB's output handling, the query results are written to HDFS instead of being returned inline.

Example Commands. We implemented two SQL-style extensions in the plugin:

- `SELECT tsc_create_index(...)` – Extracts and transforms TS data into TSCs, then builds the index.
  Example usage inside IoTDB:

```
SELECT tsc_create_index(
   *,
   'scope'='root.sg1.**',
   'hdfs_uri'='hdfs://.../',
   'spark_submit'='/path/to/spark-submit',
   'tsc_jar'='/path/to/cq.jar',
   'index_conf'='/path/to/index.conf'
) FROM root.sg1.**
```

- `SELECT tsc_convergence(...)` – Executes a convergence query using the prebuilt index.
  Example usage inside IoTDB:

```
SELECT tsc_convergence(
   sensor1,
   'scope'='root.sg1.device1.sensor1',
   'query_conf'='/path/to/query.conf',
   'dataset'='<dataset_id>'
) FROM root.sg1.device1
```

Plugin Repository. The source code for the IoTDB integration plugin, along with usage examples and documentation, is publicly available at: https://github.com/kaluaim/tsc-cq-iotdb-plugin

This prototype demonstrates that convergence queries can be seamlessly embedded within existing TSDB ecosystems, resulting

Algorithm 4: Incremental Index Update for Data Append

Input : D: New TSC dataset to append
$\texttt{INX}_{old}$: Current index structure
$\texttt{Stats}_{old}$: {TSC_{cnt}, TS_{cnt}, PartAvg, PartSizes[] }
β: Deviation threshold from avg., e.g., 60% (triggers re-structuring)
Output : $\texttt{INX}_{new}$: New index structure (after D's append)
$\texttt{Stats}_{new}$: {TSC_{cnt}, TS_{cnt}, PartAvg, PartSizes[] }

1 $\texttt{INX}_{new} \leftarrow$ insert D into INX_{old} using default mechanism
2 $\texttt{Stats}_{new} \leftarrow$ update Stats_{old} during D insertion
3 **for** *each partition P containing at least* β *above* Stats_{new}*.PartAvg* **do**
4 // ***Trigger Splitting***
5 Sample from P, learn start-time dist., and select a mid-way splitting point.
6 Update $\text{INX}_{new} \leftarrow$ split TS objects in P over new partitions P_1 & P_2
7 Update $\text{INX}_{new} \leftarrow$ re-construct Level 2 index for P_1 & P_2
8 Update Stats_{new}.PartSizes[]
9 **for** *each partition P containing less than* $1 - \beta$ *below* Stats_{new}*.PartAvg* **do**
10 // ***Trigger Merging***
11 $\text{P}^* \leftarrow$ Select the left or right sibling partition having lower capacity
12 **if** *capacity* $(\text{P} \cup \text{P}^*)$ *does not exceed* β *above* Stats_{new}*.PartAvg* **then**
13 Update $\text{INX}_{new} \leftarrow$ insert P's objects into P^* and merge intervals
14 Update Stats_{new}.PartSizes[]
15 **else**
16 Trigger the splitting steps on $(\text{P} \cup \text{P}^*)$ (Lines 4-8)
17 **return** $\texttt{INX}_{new}$ & $\texttt{Stats}_{new}$

in minimal overhead and enabling the broader applicability of our framework.