Eurographics Conference on Visualization (EuroVis) 2018
J. Heer, H. Leitte, and T. Ropinski
(Guest Editors)

Volume 37 (2018), Number 3

Time Lattice: A Data Structure for the Interactive Visual Analysis of Large Time Series

Fabio Miranda[1], Marcos Lage[2], Harish Doraiswamy[1], Charlie Mydlarz[1], Justin Salamon[1], Yitzchak Lockerman[1], Juliana Freire[1], Claudio T. Silva[1]

[1] New York University, New York, United States [2] Universidade Federal Fluminense, Niteroi, Brazil

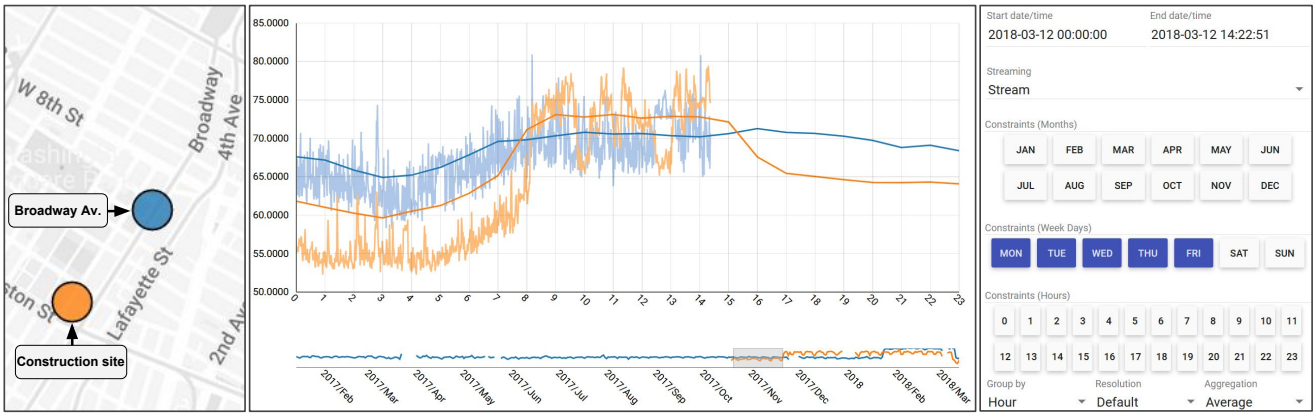

Figure 1: *Using Noise Profiler to analyze OLAP queries over acoustic data from sensors deployed in New York City. A group-by hour is used as a baseline for ambient noise (smooth line), highlighting the difference between the noise profile of two locations during weekdays. One sensor (blue) is close to a main road (Broadway Av.) and has a constant dB_A level throughout the hours of the day; the other sensor (orange) is close to a major construction site and has a distinctly higher dB_A level during construction hours between 7 a.m. and 5 p.m. The live streaming data (fluctuating line) can be used to get instantaneous information about the noise level captured by the sensors, and inform city agency noise enforcement teams about possible noise code violations such as construction sites operating outside of their allotted construction hours.*

Abstract

Advances in technology coupled with the availability of low-cost sensors have resulted in the continuous generation of large time series from several sources. In order to visually explore and compare these time series at different scales, analysts need to execute online analytical processing (OLAP) queries that include constraints and group-by's at multiple temporal hierarchies. Effective visual analysis requires these queries to be interactive. However, while existing OLAP cube-based structures can support interactive query rates, the exponential memory requirement to materialize the data cube is often unsuitable for large data sets. Moreover, none of the recent space-efficient cube data structures allow for updates. Thus, the cube must be re-computed whenever there is new data, making them impractical in a streaming scenario. We propose Time Lattice, a memory-efficient data structure that makes use of the implicit temporal hierarchy to enable interactive OLAP queries over large time series. Time Lattice is a subset of a fully materialized cube and is designed to handle fast updates and streaming data. We perform an experimental evaluation which shows that the space efficiency of the data structure does not hamper its performance when compared to the state of the art. In collaboration with signal processing and acoustics research scientists, we use the Time Lattice data structure to design the Noise Profiler, a web-based visualization framework that supports the analysis of noise from cities. We demonstrate the utility of Noise Profiler through a set of case studies.

1. Introduction

With the massive adoption of the *Internet of Things* (IoT) in various scenarios ranging from smart home devices and smart cities to medical and healthcare applications, interactive visualization frameworks are becoming paramount in the exploration and analysis of the data generated by these systems. Any such IoT setup continuously transmits data as a time series from tens and hundreds up to thousands of objects (or sensors). The exploration of these data typically requires complex online analytical processing (OLAP) queries that involve slicing and dicing the time series over different temporal

resolutions together with multiple constraints and custom aggregations. The use of a visual interface adds an additional constraint: the queries must be *interactive*, since high latency queries can break the flow of thought, making it difficult for the user to effectively make observations and generate hypotheses [LH14].

In this paper, we are specifically interested in the analysis of time series from acoustic sensors deployed to help map and understand the noisescape in cities. Noise is an ever present issue in urban environments. Besides being an annoyance, noise can have a negative effect on education and overall health [BH10]. To combat these problems, cities have developed noise codes to regulate activities that tend to produce sounds (see e.g. [NYC05, Cit]). To help monitor noise levels in New York City (NYC), as well as to aid government agencies in regulating noise throughout the city, researchers part of the *Sounds of New York City* (SONYC) project have developed and deployed low cost sensors that have the ability to measure and stream accurate sound pressure level (SPL) decibel data at high temporal resolutions (typically every second) [MSB17a, MSB17b, SON]. Thirty six such sensors have been collecting data for over a year, in addition to another twelve new sensors that have been deployed since. As the size of this network continues to grow, the amount of data produced by the sensors becomes virtually unbounded.

This necessitates the ability to handle analysis queries efficiently on such large time series data, in particular, the more complex OLAP queries that require aggregations of the data across multiple temporal resolutions. For example, noise enforcement agencies can assess a breach if the noise level is greater than the ambient background noise. However, the ambient background noise patterns are spatially localized and vary depending on the time (e.g., peak hours, night time, weekdays, weekends, etc.). So, to identify these patterns over weekdays, as shown in Figure 1, the following query is issued using a visual interface over data from sensors present in the different regions of interest:

select *time series* ***during*** weekdays ***groupby*** *hour*

Furthermore, not only can the user restrict the time range over which to perform the above query (e.g., in Figure 1, the time range is from October 2017 to December 2017), but depending on the location and its conditions (e.g., tourist spots), more constraints might also be *interactively* added to this query. Since users can continuously alter the constraints through the visual interface, it is crucial that these queries have low latency to enable seamless interaction.

Problem Statement and Challenges. The goal of this work is to design a time series data structure that supports OLAP queries and has the following important properties:

1. Interactive queries;
2. Interactive updates from new data; and
3. Low memory overhead.

Two common approaches to support OLAP queries are to use either database systems catered for time series, or data cube-based solutions. However, neither of the approaches satisfy all of the above requirements that are crucial for real-time visual analysis of the data.

Traditional time series databases [PFT*15, BKF, Inf, Kai], by supporting the powerful SQL-like syntax, can execute a wide range of queries including the OLAP queries with temporal constraints that are of interest in this work. They are often memory efficient, and support updates over new data. To execute a given query, these systems typically use an index to first retrieve intermediate results based on the constraints. The query results are then computed by explicitly aggregating the intermediate results. Unfortunately, such strategy fails to be interactive when handling data at the scale that is now available (see Section 4).

Data cube-based structures [LKS13, PSSC17, MLKS18, SMD07], on the other hand, have extremely low latency to OLAP queries. However, the size of these data structures increases exponentially with the number of dimensions. In case of a time series, the dimensions correspond to the discrete temporal resolutions for a time series. Moreover, to support temporal constraints in these queries, the time resolution of these constraints should also be a dimension of the cube. For example, specifying the time period of interest with an accuracy up to a minute requires *minute* to be a dimension of the data cube. This further increases the space overhead. While this might be admissible when working with a single time series, it becomes impractical when working with several tens to hundreds of time series that is now commonplace with IoT systems. Additionally, the more practical memory-optimized data cube structures [LKS13, PSSC17] do not support updates with new (or streaming) data, thus requiring the re-computation of the entire structure every time. Given that the cube creation time can take minutes even for reasonably small data sizes, this approach becomes impractical for handling multiple large streaming time series data.

Contributions. In this paper, we present a new data structure, *Time Lattice*, that can perform OLAP queries over time series at interactive rates. The key idea in its design is to make use of the implicit hierarchy present in temporal resolutions to materialize a sub-lattice of the data cube. This helps avoid the curse of dimensionality common with other cube-based structures and results in a *linear memory overhead*, while still being able to conceptually represent the entire cube. This drastic reduction in memory also allows us to augment our data structure with additional summaries, thus supporting the computation of measures that are otherwise not easily supported. More importantly, unlike existing approaches, our data structure allows *constant amortized time updates*.

To demonstrate the effectiveness of Time Lattice, we develop *Noise Profiler*, a proof of concept web-based visualization system, that is being used in the SONYC project to analyze acoustic data from NYC.

To summarize, our contributions are as follows:

- We introduce *Time Lattice*, a data structure that supports multi-resolution OLAP queries on time series at interactive rates. It has a *linear memory overhead*, and supports *constant amortized time updates* with new data.
- We show experimental results demonstrating both the time as well as space efficiency of Time Lattice.
- We develop Noise Profiler, a web-based visualization system to *simultaneously analyze* multiple streams of data generated from the SONYC sensors. Note that, without the underlying efficient data structure, it would not be possible to visually analyze such multiple streams in real time.
- We demonstrate the utility of Time Lattice through a set of case studies performed by subject matter experts, and which are of interest to the end users of the SONYC project.

2. Related Work

Time Series Databases. Several databases have been proposed to facilitate data acquisition and data querying of time stamped data. Their architecture and design vary greatly depending on their goal. One class of database systems such as tsdb [DMF12], Respawn [BWSR13] and Gorilla [PFT*15] are primarily concerned with providing the user with monitoring capabilities, and lack support for complex analytical queries. Respawn [BWSR13] proposes a multi-resolution time series data store to efficiently execute range queries. While it efficiently speedup range queries, it does not support aggregations (such as group-by's) over any temporal resolution.

One of the most popular database to support analytical queries on time series is InfluxDB [Inf], which offers a SQL-like language for queries, including *rollups* and *drilldowns*. KairosDB [Kai] is another popular time series database that uses Apache Cassandra for data storage, and provides much of the same features as InfluxDB. Timescale [Tim], on the other hand, builds on top of the popular Postgres to offer a database solution tailored for time series. As we show later in Section 4, a major drawback of these solutions is that they cannot drive interactive visualization, with complex OLAP queries requiring several seconds to execute. For a more detailed survey on existing time series data management systems, we refer the reader to the following surveys by Jensen et al. [JPT17] and Bader et al. [BKF].

Data cube. Data cube [GCB*97] is a popular method designed specifically to handle OLAP analytical queries. It pre-computes aggregations over every possible combination of dimensions of a data set in order to support low-latency queries. It has been extended to support data sets from different domains, such as graphs [CYZ*08] and text [LDH*08]. The main drawback of a data cube is the exponential growth of the cube with increasing dimensions making them impractical when working with large data sets. A common approach to reduce the size of a data cube is to materialize only a subset of all possible dimension combinations. One such approach, called iceberg cube [BR99], only stores aggregations that satisfy a given condition (specified as a threshold), and discards any values not above this threshold. While this approach is suitable for the analysis of historical data, updates become unfeasible since new data dynamically changes the aggregation requiring access to previously discarded values.

More recently, with the focus on spatio-temporal data, several approaches have been proposed to deal with the curse of dimensionality. Nanocube [LKS13] uses shared links to avoid unnecessary data replication along the data cube. However, the above memory reduction scheme is not sufficient to reduce the structure size when considering high resolution, dense time series typically available from IoT devices (see Section 4). Hashedcube [PSSC17], on the other hand, uses pivots to efficiently compute a subset of the aggregations on the fly from the raw data, rather than pre-computing all of them, thus achieving a considerably lower memory footprint. To do this, it requires the data to be sorted according to its dimensions. While both nanocube and hashedcube support low latency queries capable of driving interactive visualizations, they cannot handle data updates. Han et al. [HCD*05] tackle the memory explosion by restricting the analysis to a temporal window. This is accomplished by a data cube that, while updating new data points, discards old points (and the corresponding aggregations) based on a user defined retention policy. A similar retention approach is also used by Duan et al. [DWW*11]. While this approach is suitable for monitoring applications requiring analysis on recent history, it relies on approximate queries and cannot be used for historical analysis.

Our goal is have a data structure that supports real-time queries for both historical analysis as well as monitoring applications, while still being memory efficient. To accomplish this, we choose a materialization of the data cube based on the intrinsic temporal hierarchy that enables constant amortized time updates, as well as real-time query execution. However, note that the proposed data structure is not a replacement for general data cubes, which are structures applicable to any data set. Rather, it provides an efficient alternative when working with large time series and OLAP queries that slice and dice the time series over the temporal resolutions.

Time series visualization. Time stamped data has long been studied and visualized in multiple domains. Several studies propose different metaphors and interactions when dealing with time series, such as applying lenses [ZCPB11], clustering values into calendar-based bins [VWVS99] or re-ordering of the series at different aggregations to allow for an easier exploration [MMKN08]. The perception impact on the visualization of multiple time series has been studied by Javed et al. [JME10]. A full survey of different techniques was presented by Silva and Catarci [SC00], Müller and Schumann [MS03] and Aigner et al. [AMM*07]. Note that all of these approaches are orthogonal to this work. While their goal is to provide new visual metaphors, ours is to support real-time execution of queries that are used to generate the required visualizations. The visualization of time series in multiple resolutions has also been a topic of study. Berry and Munzner [BM04] aggregate the data into bins prior to the visualization. Hao et al. [HDKS07] proposed a distortion technique that generates visualizations where more visual space is allocated to data according to a measurement of interest. Jugel et al. [JJHM14] proposed M4, a technique to aggregate and reduce time series considering screen space properties. All of these approaches, however, do not focus on OLAP-type queries, limiting their techniques to essentially a range query at a coarser resolution.

Another popular area of research associated with time series is the querying of similar patterns in a time series [MVCJ16, CG16, HS04]. Time Lattice can augment these approaches by speeding up subqueries that are commonly used by them.

3. Time Lattice

The primary goal of this work is to efficiently execute queries of the following type over an input time series:

select *time series* ***between*** t_1 *and* t_2
where *constraints* C
groupby *resolutions* G

where, t_1 and t_2 specify the time period of the data to consider. The constraints $C = \bigcup_r \{C_r\}$ defines the constraints over each temporal resolution r. Here, C_r specifies a set of values in resolution r that have to be satisfied. The resolutions $g \in G$ specify the resolutions on which to perform the *group-by*. For example, if the query in Section 1 has to be executed only for data from the last 6 months of 2017, we set t_1 = 2017-06-01T00:00; t_2 = 2017-11-30T:23:59; $C = \{C_{day_{week}} = \{Monday, ..., Friday\}\}$; and $G = \{hour\}$.

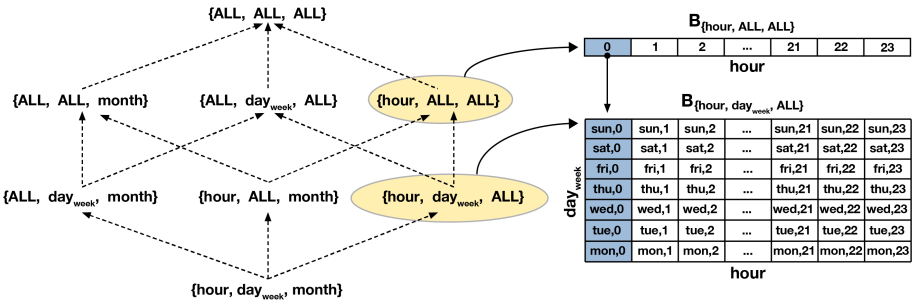

Figure 2: *Data cube with $D = \{hour, day_{week}, month\}$ has a total of $2^{|D|}$ cuboids, where each cuboid stores the aggregations for all possible values of its dimensions.*

T	Discrete space representing time.
$f : T \to \mathbb{R}$	Time series.
D	Dimensions of the data cube. It corresponds to the temporal resolutions in case of a time series.
$\mathscr{P}(D)$	Power set of D.
$\prec$	Partial order defined on the temporal resolutions.
H	Hasse diagram of the poset $(D, \prec)$.
B_r	Cuboid corresponding to resolution r.
$\alpha_r(t)$	Association function mapping time step t to an offset in B_r.
$\pi_{r \to r'}(i)$	Containment function mapping an element in B_r to an element in $B_{r'}$, where $r \to r' \in H$.

Table 1: *List of symbols.*

In this section, we describe the main data structure, Time Lattice, and discuss its properties. We also explain the query execution strategy using Time Lattice and describe extensions to the data structure that enable additional features such as support for join queries and multiple aggregations.

3.1. Data Structure

A *data cube* [GCB*97] is a method that was designed to efficiently answer aggregate queries such as the one shown above. Here, the resolutions of time are modeled as the *dimensions* D of a data cube. However, unlike general data sets, the dimensions of time corresponding to the different temporal resolutions are hierarchically dependent. We make use of this property to design a data structure that is both memory efficient and supports interactive aggregate queries. To avoid the exponential memory overhead of a data cube, we compute only a subset of the data cube. We then make use of the inter-dependency between the temporal resolutions to efficiently compute on-the-fly query results. In this section, we first provide a brief overview of data cubes followed by describing in detail the proposed data structure. We use the terms resolution and dimensions interchangeably in the remainder of the text.

Preliminaries: Data Cubes. Consider a time series $f : T \to \mathbb{R}$, which maps each time step of a discrete temporal space T to a real value. Without loss of generality, let the resolution of T be seconds and be represented using epoch time (i.e., seconds since January 1, 1970, Midnight UTC). Let f be defined for every second within a time interval $[t_1, t_2)$, $t_1, t_2 \in T$. For ease of exposition, assume that there are no gaps in the time series, that is, the function f is defined for all $t_1 \leq t < t_2$. Since we are working with time, f can also be analyzed in resolutions coarser than a second, such as *minute*, *hour*, *day of week* (day_{week}), etc.

A data cube represents all possible aggregations over the dimensions in D. Formally, a data cube represents the 2^d *cuboids* corresponding to the elements of the power set $\mathscr{P}(D)$, where $d = |D|$. For example, given dimensions $D = \{hour, day_{week}, month\}$:

$$\mathscr{P}(D) = \Big\{\emptyset, \{hour\}, \{day_{week}\}, \{month\}, \{hour, day_{week}\}, \{day_{week}, month\}, \{hour, month\}, \{hour, day_{week}, month\}\Big\}$$

The set of cuboids for the data cube in the above example is shown in Figure 2. The *dimension* of a cuboid B_P, $P \in \mathscr{P}(D)$, is equal to $|P|$. For example, the element $\{day_{week}\}$ forms a 1-dimensional cuboid while $\{hour, day_{week}\}$ forms a 2-dimensional cuboid. A k-dimensional cuboid (or k-cuboid) stores all possible aggregations corresponding to its k dimensions. For example, the 2-cuboid $\{hour, day_{week}, ALL\}$, corresponding to the element $\{hour, day_{week}\} \in \mathscr{P}(D)$, stores the aggregations for all possible ($hour$, day_{week}) values. Here, the aggregation is performed over the other $d - k$ dimensions represented by *ALL*, which in the above example is *month*. Thus, this cuboid has size 24×7 (there are 24 possible hours and 7 possible days). In general, the size of a k-cuboid, i.e., the number of aggregations stored by the cuboid, is equal to the product of the cardinality of each of its dimensions.

A fully materialized data cube pre-computes and stores all 2^d cuboids corresponding to $\mathscr{P}(D)$. As a rule of thumb, the number of dimensions to use to create a cube depends on the resolution of the constraints used in the query. In the above example, to support queries that group by or filter over arbitrary time ranges specified in the resolution of minutes, the dimension *minutes* should be added to D. When a new dimension is added, not only does the number of cuboids increases by a factor of 2 (2^3 to 2^4 in the example), but the total number of aggregations stored (corresponding to all the cuboids) increases by a factor equal to the number of categories in that dimension (60 in case the dimension minute is added to D, since there are 60 possible values denoting a minute). Clearly, the size of the data cube increases exponentially with new dimensions, and can quickly become intractable when working with resolutions commonly used in time series analyses.

The Time Lattice structure. Instead of materializing the entire data cube, we use the intrinsic hierarchy present in time to materialize only a subset of this cube. Formally, let $D = \{r_1, r_2, \ldots, r_d\}$ denote the different temporal resolutions. Let $\prec$ denote a partial order

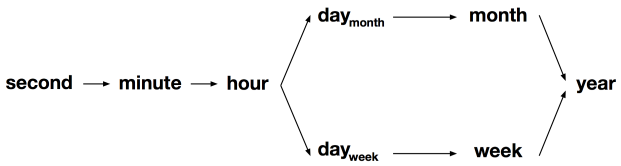

Figure 3: *Hasse diagram denoting the poset defined on the temporal resolutions used in this work.*

defined on D, such that $r_i \prec r_j$ if the time stamps in resolution r_i can be partitioned based on the time stamps in resolution r_j. For example, $minute \prec hour$ and $hour \prec day_{week}$, since the time stamps specified in minutes can be partitioned based on the hour of that time stamp, and similarly hours can be partitioned by days. Note that the above partial order is different from the partial order that defines the data cube itself (defined by the inclusion function [GCB*97]). Let H denote the *Hasse diagram* of the partially ordered set, or *poset*, $(D, \prec)$. The nodes of H correspond to the dimensions in D, and an edge exists from r_i to r_j if r_j *covers* r_i, i.e., $r_i \prec r_j$ and $\nexists r_k | r_i \prec r_k \prec r_j$. In the above example, even though $minute \prec day_{week}$, this edge does not exist in H since day_{week} does not cover $minute$ ($\exists hour$ s.t. $minute \prec hour \prec day_{week}$). Figure 3 shows the Hasse diagram for the poset covering common temporal resolutions used in this work.

Time Lattice materializes cuboids using H as follows. Consider a maximal path $(r_{i_1}, r_{i_2}, \ldots, r, \ldots, r_{i_n})$ in H such that $r_{i_1} \prec r_{i_2} \prec \ldots \prec r \prec \ldots \prec r_{i_n}$. The cuboid $(ALL, ALL, \ldots, ALL, r, \ldots, r_{i_n})$ is materialized corresponding to the node r in this path. For example, consider the node day_{month} in the poset defined in Figure 3. This results in materializing the cuboid $(ALL, ALL, ALL, day_{month}, Month, Year)$. Next, consider a resolution r which is not part of this path. A maximal path in H that includes r is next chosen to be materialized. This process is repeated until there is at least one cuboid corresponding all resolutions in H. The Time Lattice is the union of all the cuboids resulting from the above materialization. Note that, since each cuboid B_P that is materialized corresponds to a resolution $r \in H$, we refer to this cuboid using r as B_r.

Such a materialization has several advantages:

- Each materialized cuboid B_r can be represented by a contiguous array such that the aggregate values stored in B_r follow a chronological order representing a continuous time series in resolution r. Thus, the Time Lattice can have a simple array-based implementation.
- Consider the resolutions day_{month} and day_{week}. Even though they are conceptually different (the categories have different range: $\{1,2,3,\ldots\}$ vs. $\{Mon, Tue, \ldots\}$), the individual array elements of $B_{day_{month}}$ and $B_{day_{week}}$ correspond to the same days. Thus, the same array can be shared by both these cuboids.
- Because of the chronological ordering, the different cuboids can be *implicitly indexed* based on the resolution r. This implicit index is formally defined using an *association function*, $\alpha_r(t)$, corresponding to each $r \in H$, which maps a time stamp t to an offset i of the array B_r. That is, $B_r[i]$ stores the aggregated value corresponding to time step t in the cuboid B_r. Table 2 lists the association functions α_r used for the resolutions in Figure 3.
- Enables efficient updates to the data structure (see details below).
- The temporal hierarchy also allows for an implicit mapping between array elements across resolutions, enabling efficient "rollups" and "drilldown" operations that are performed on a cube (see Section 3.2). This mapping is formally defined by the *containment function* $\pi_{r \to r'}$ which maps an array offset i in resolution r to an offset j in resolution r', whenever there is an edge from r to r' in H. Essentially, $\pi_{r \to r'}(i) = j$ if and only if there exists t such that $\alpha_r(t) = i$ and $\alpha_{r'}(t) = j$. This function can also be parametrically computed similar to the association function. Since π is a many-one function, the inverse mapping $\pi^{-1}_{r \to r'}$ maps an offset in the coarser resolution r' to a sub-array in B_r. This mapping to a sub-array is only possible because of the above mentioned ordering of B_r.
- Helps efficiently execute queries with range constraints as well—only the sub-array(s) within the offsets corresponding to the query range has to be considered.

$\alpha_{second}(t)$	$B_{second}[t - t_1]$
$\alpha_{minute}(t)$	$B_{minute}[\lfloor \frac{t}{60} \rfloor - \lfloor \frac{t_1}{60} \rfloor]$
$\alpha_{hour}(t)$	$B_{hour}[\lfloor \frac{t}{60*60} \rfloor - \lfloor \frac{t_1}{60*60} \rfloor]$
$\alpha_{day}(t)$	$B_{day}[\lfloor \frac{t}{24*60*60} \rfloor - \lfloor \frac{t_1}{24*60*60} \rfloor]$
$\alpha_{week}(t)$	$B_{week}[weeksbetween(t, t_1)]$
$\alpha_{month}(t)$	$B_{month}[12 * (y(t) - y(t_1)) + m(t) - m(t_1)]$
$\alpha_{year}(t)$	$B_{year}[y(t) - y(t_1)]$

Table 2: *Association functions. Here $y()$, $m()$ returns the year and month respectively for a given time stamp, and $weeksbetween()$ returns then number of weeks between two time stamps.*

The elements of the cuboid B_r (i.e., $B_r[i]$) store one or more measurements $\mu_r(i)$. Here, μ can be any distributive and algebraic operation. In our implementation, we store the following distributive aggregates—*minimum*, *maximum*, *sum*, and *count*. This can in turn be used to compute other algebraic aggregates such as *average* (see Section 3.3 for more details). Note that if the dimension is the same as the resolution of the underlying time series, then μ simply corresponds to the time series itself.

Space requirements. Let the size of the time series be n. For analysis purposes, first consider a maximal path $r_1, r_2, \ldots, r_k$ in H s.t. $r_1 \prec r_2 \prec \ldots r_k$. Without loss of generality, let r_1 be the original resolution of the time series. Thus, the B_{r_1} simply corresponds to the underlying data itself. Let the space required for materializing at resolution r_i (size of the array B_{r_i}) be s_i. Therefore, $s_1 = n$. Then, the space required for materializing all arrays (i.e., not counting the base array, which is the underlying time series) is $s = \sum_{i=2}^{k} s_i$. The size s_{i+1} is a fraction of s_i defined by $s_{i+1} = \lceil s_i / a_{i+1} \rceil$, where $a_{i+1} = |\pi^{-1}_{r_i \to r_{i+1}}|$. For example, $a_{minute} = 60$ (60 seconds make a minute), and $a_{day} = 24$ (24 hours make a day). Therefore,

$$
\begin{aligned}
s &= \sum_{i=2}^{k} s_i \\
&= \frac{s_1}{a_2} + \frac{s_2}{a_3} + \frac{s_3}{a_4} + \ldots + \frac{s_{k-1}}{a_k} \\
&\leq s_1 \times \left(\frac{1}{a_2} + \frac{1}{a_2 \cdot a_3} + \frac{1}{a_2 \cdot a_3 \cdot a_4} + \ldots + \frac{1}{\prod_{i=2}^{k} a_d} \right) + k \\
&\leq s_1 + k \\
&\approx n \quad \{\text{assuming } k \ll n\}
\end{aligned}
$$

Let the total number of maximal paths used to materialize the

```
1:  function DrillDown(B', R, r, C, G, t1, t2)
2:    result ← []
3:    B ← B' ∩ B_{R[r]}[α_r(t1), α_r(t2)]
4:    C_r ← Constraints at resolution R[r]
5:    if |G| = 0 and |C| = 0 then
6:      result ← B
7:    else if |G| > 0 or |C| > 0 then
8:      G = G \ {R[r]}
9:      C ← C \ C_r
10:     for all b ∈ B do
11:       if b satisfies C_r
12:         result ← result ∪
                {DrillDown(π^{-1}_{R[r+1]→R[r]}(b), R, r+1, C, G, t1, t2)}
13:     if r ∈ G then
14:       result ← GroupBy(result, R[r])
15:   return result
16: function Query(C, G, t1, t2)
17:   r' ← finest resolution in C ∪ G
18:   R[] ← {path in H from r' to year containing C ∪ G} s.t. R[i+1] ≺ R[i]
19:   DrillDown(B_year, R, 0, C, G, t1, t2)
```

Figure 4: *Pseudo-code for the aggregate query.*

Time Lattice be m. Then, the size of the Time Lattice data structure is bounded by $O(m \cdot n)$. Given that typically m is a very small integer—$m = 2$ for the Hasse diagram in Figure 3, the size of the data structure is *linear* in the size of the underlying data. We would like to note that this is not a tight bound. In fact, as we show later in the experiments, the space required by the structure is significantly smaller in practice ($< 2\%$ of n as shown in Section 4.2).

Updating the data structure. One of the main goals of our proposed data structure is to support updates over new (or streaming) data. Consider an existing Time Lattice structure, and an incoming value of the time series. Since this value will have a time stamp t at the finest resolution (second for the purpose of this work), it will simply be appended to B_{second}. For resolutions $r|second \prec r$, we first need to check if the corresponding array element $B_R[\alpha_r(t)]$ already exists. If it does, we need to update the value of the aggregation μ_r to take into account $f(t)$. If this element does not exist, it is first created and appended to B_r and the value of μ_r is appropriately initialized using $f(t)$.

Assuming that the data structure is updated every second, the time complexity becomes $O(d)$ per update, where d is the number of arrays maintained and is bounded by the number of resolutions in H. Oftentimes, it is not critical to have such a high update frequency. For example, instead of updating the structure every second, it would suffice in practice to update it every minute. Let this update be performed every k seconds. In this case, there will be k appends to B_{second}, $\lceil \frac{k}{a_{minute}} \rceil$ updates / appends to B_{minute}, and so on. Thus, when $k \geq d$ (e.g. for a minute-wise update, $k = 60 > d = 7$) the time complexity is $O(k+d)$ for effectively k updates, or $O(\frac{k+d}{k}) = O(1)$ *amortized time* per update.

3.2. Querying

Aggregate query. Aggregate queries (or OLAP-type queries) are primarily used for a more nuanced analysis on the time series data. The algorithm to execute such a query is presented in Figure 4. The query is executed by first drilling down starting from the 0-dimensional cuboid of the data cube. At each successive resolution

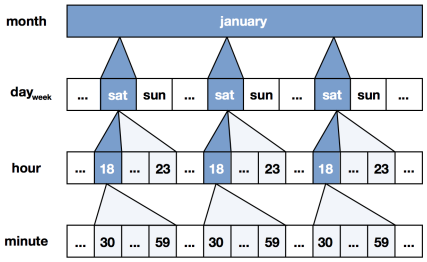

Figure 5: *Drilldown performed (w.r.t. one of the months) when a query groups-by month over all Saturdays from 18:30 to 23:59.*

r, the constraint values for that particular resolution (Line 4) are evaluated. Given a constraint in resolution r, the sub-arrays in B_r satisfying these constraints (within the given time range $[t_1, t_2)$) are first identified, and a drilldown is performed *only* with respect to these sub-arrays (Lines 10–12). Intuitively, a *drilldown* corresponds to expanding the cuboid by increasing its dimension by one. Figure 5 illustrates this procedure on a query that requires a group-by on *month* over all Saturday nights (18:30hrs–23:59hrs). For the *hour 18*, the execution drills down up to the *minute* resolution, and for the other hours in the constraints, only upto the *hour* resolution.

The drilldown is recursively repeated until the constraint in C at the finest resolution, r_c, is satisfied. Let $r_g \in G$ be the finest resolution on which a group-by is performed. If $r_g \prec r_c$, then a drilldown is further performed until r_g. On the other hand, if $r_c \prec r_g$, a rollup is performed until r_g. Intuitively, a *rollup* decreases the dimensionality of a cuboid by one by aggregating over one of its dimensions. At this stage, the group-by is performed recursively over all resolutions in G starting from r_g and rolling-up to coarser resolutions. At each resolution, the elements of the filtered (and previously grouped-by) sub-array are aggregated into the query result (Line 14).

Range query. Time Lattice also supports range queries over time series data. A range query is used to query for the time series within a given time interval at an optional user specified resolution. This query is primarily intended for the visual exploration of the time series. A resolution coarser than the original resolution of the time series returns the computed aggregates. It is common for the visualization system to control the resolution specified in the query depending on the available screen space and the time constraint. For example, when visualizing a large time series, the screen space restricts each pixel to cover a time interval larger than a single unit of time. So, the system might choose to visualize the maximum value within the time interval corresponding to each pixel in order to obtain a big picture of the time series (analogous to the level of detail rendering used for terrains, which shows only larger mountains when the camera is distant, and increases detail as the camera moves closer to the scene).

The result for a query having time constraint $[t_1, t_2]$ and resolution r is simply the sub-array of B_r from $\alpha_r(t_1)$ to $\alpha_r(t_2)$.

3.3. Extensions

Handling discontinuous time series. We have so far assumed that the given time series is continuous and without gaps. This, however, need not be true in practice. For example, a sound sensor could

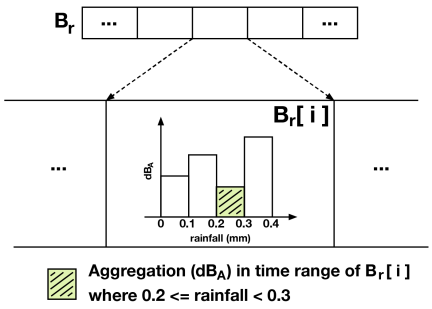

Figure 6: *An additional histogram corresponding to a second time series is stored in the cuboids to support joins in queries.*

malfunction, and hence stop transmitting data. This would result in no data from the sensor until it is corrected. Such a situation can be handled in two ways. One could simply "fill" the gaps with a default value denoting lack of data. In this case, when aggregations are performed at a coarser resolution, these values should be dealt with appropriately. The other option is to maintain separate Time Lattices for each contiguous time series. In this case, the querying approach would be modified to perform queries over all Time Lattices whose time interval intersects the query time range, and combine the multiple results into a single result.

In our current implementation, we chose the former since the time interval between failure and replacement of sensors is typically small, resulting in a small memory overhead due to the "filling" operation. However, for cases where this gap can be significant, we advise the use of multiple Time Lattices.

Supporting multiple aggregations. Time Lattice supports the use of any *aggregable* measure. Here, a measure is said to be aggregable if its sufficient statistics can be expressed as a function of commutative and associative operators [WFW*17]. Thus, it allows an aggregation at a coarser resolution to be computed purely using the immediate finer resolution (and hence not using the raw data at all). In addition, measures such as median or percentiles can also be approximated by maintaining a histogram associated with each bin. The size of this histogram can be adjusted depending on the available memory and accuracy requirements.

The low memory requirement of Time Lattice further allows the addition of more advance summaries, as long as they are aggregable. For instance, each bin can have a *tdigest* [DE14] associated with it, so that holistic measurements such as quantiles can also be computed within an error threshold. As shown later in Section 6, it is also easy to add domain specific measures to the data structure.

Supporting joins. Oftentimes, the analysis of a time series might require a join with another time series. For example, when analyzing the decibel level time series from a sound sensor, the domain expert might want to consider only time periods when there was significant rainfall (precipitation greater than a given threshold). Here the rainfall data would be represented by a second time series, say f'. To support such a join, we additionally store a histogram corresponding to f' in each element of a cuboid as follows. The bins of this histogram correspond to the range of f'. Consider one such histogram bin having range $[f'_1, f'_2)$. The value stored in this bin is equal to the aggregate of $f(t)$ where $t | f'_1 \leq f'(t) \leq f'_2$. Figure 6 illustrates once such histogram for the above rainfall example.

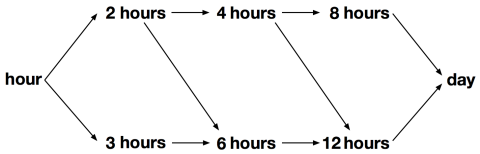

Figure 7: *Expanded materialization between the resolutions hour and day. Recall that the array corresponding to day is shared for the resolutions day_{week} and day_{month}.*

Note that the resolution of f' need not be the same as that of the time series of interest f. If the resolution of f' is finer than that of f, then f' is appropriately aggregated. If instead, it is coarser, then f' can be extrapolated to support constraints in a finer resolution. In our current implementation, we assume that the join condition based on f' is not coarser than the group-by resolution of the query. The case when the condition is coarser than the group-by resolution can be supported by storing the aggregate measure corresponding to f' as well in the Time Lattice, and drilldown performed only when an array element satisfies the condition.

Extended materialization. Depending on the size of the underlying time series, queries could still be expensive depending on the query constraints. In such cases, selectively materializing more nodes can greatly help speed up the query execution. If frequently posed queries involve a group-by different from the ones materialized, then that corresponding cuboid is materialized.

On the other hand, if frequently posed queries involve similar constraints along a single resolution, then it might be more beneficial to add a new dimension, and materialize that resolution accordingly. For example, users might frequently pose queries with constraints in hour of day to study patterns during different times of the day such as during peak hours in the morning and / or evening. One such query would be to obtain the aggregated behavior during peak morning hours (say 8 a.m. to 11 a.m.) grouped by the days of the week. To execute the queries, several sub-arrays are processed after filtering B_{hour}. As the size of the time series keeps increasing, this overhead could become significant. In such cases, a new coarser resolution can be introduced.

For a resolution r, there can be $a_r - 2$ possible resolutions that can be added corresponding to the possible time ranges. In the above example, this resolution would lie between *hour* and *day* with respect to the partial order $\prec$. There are $a_{day} - 2 = 22$ such possible resolutions ranging from 2 hours to 23 hours. If all of these resolutions are materialized, then it increases the size of the Time Lattice by a linear number of cuboids.

Materializing all such nodes for all resolutions might not be necessary for the required analysis. Instead, we allow users to specify common queries, and choose the new resolutions to be materialized. By default, one could materialize resolutions corresponding to time intervals that are factors of a_r. Figure 7 shows one such materialization between the hour and day resolutions, where the time intervals of sizes 2, 3, 4, 6, 8, and 12 hours are materialized. Queries with constraints having a different interval size are then computed by using a combination of these resolutions.

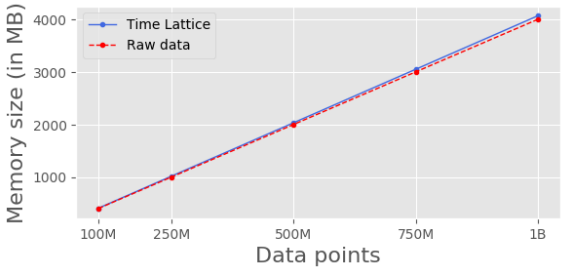

Figure 8: *Size of Time Lattice within increasing time series size. Note that the additional memory overhead used for the data structure is* ***considerably*** *smaller than the data itself (< 2%).*

4. Experimental Evaluation

In this section, we discuss results from our experiments evaluating the efficiency of the Time Lattice data structure.

4.1. Experimental Setup

Hardware Configuration. All experiments were performed on a workstation with a Intel Xeon E5-2650 CPU clocked at 2.00 GHz, 64 GB RAM, and running a Linux operating system.

Data Sets. We generate synthetic time series data sets for our evaluation. The time series is itself at the *second* resolution, and for each second, a random number from a uniform distribution is used as the value for each time step (second). We generate time series of different sizes depending on the experiment that is performed.

Q1	**select** *time series* **between** December 14, 1970 5:20 and February 3, 1972 9:20 **aggregated by** *hour*
Q2	**select** *time series* **group by** *hour*
Q3	**select** *time series* **where** time **between** 09:30 and 17:30 **group by** *day*
Q4	**select** *time series* **where** time **between** 09:30 and 17:30, month **in** [January, February, March] **group by** *hour, minute*

Table 3: *Queries used in the experiments.*

Queries. The four queries used in our evaluation are shown in Table 3. We chose these queries to cover the different scenarios that arise during the visual analysis of time series data. Query Q1 is a range query typically used in the exploration of time series, and queries for data within the given range to be visualized as an hourly time series. Q2 is a group-by query used to visualize the hourly patterns in the data. Q3 queries for the day time patterns in the data for every day of the week. The complexity of the group-by query in this case is increased by adding a constraint on time (i.e. day time range). The above two queries are typically used to study ambient noise patterns (see Section 6.1). Finally, Q4 further increases the complexity of Q2 and Q3 by adding an additional constraint, as well as another group-by dimension. This query provides detailed minute-wise day time patterns over winter months.

State-of-the-art Approaches. For a comparison of Time Lattice with the state of the art in Section 4.3, we use a combination of both data cube-based techniques as well as libraries and databases that are catered for time series data analysis.

In particular, for the data cube-based baseline, we use nanocubes [LKS13] which is also available as open-source software. We did not choose hashedcubes [PSSC17] since the available implementation supports only "count" queries and cannot perform aggregation over attributes. Also, nanocubes has better query performance than hashedcubes [PSSC17], and hence provides a better baseline. To be fair, we only chose resolutions that are used by the test queries as dimensions while constructing the nanocubes data structure. This also allows the data structure to be more memory efficient. The resolutions included were: *year*, *month*, day_{week}, *hour*, and *minute*, and *hour-minute*. The last category gives the minute of the day having a value between 0 and 1440. This was required to efficiently support queries Q3 and Q4 that have constraints on the time of day.

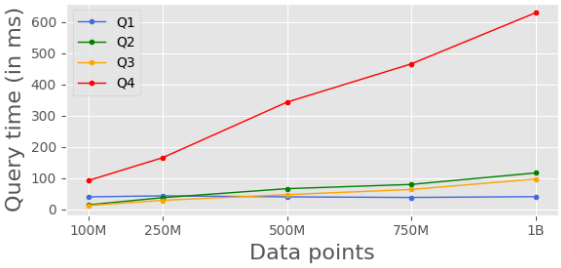

Figure 9: *Query execution time for the four test queries as the size of the data increases.*

With respect to time series databases, we chose those that support OLAP queries: PostgreSQL with the timescale [Tim] extension, InfluxDB [Inf] and KairosDB [Kai]. We created a hypertable and an index on the time dimension when using the timescale extension in PostgreSQL. For both InfluxDB and KairosDB, we created *tag columns* corresponding to the time dimensions used for querying (same as the ones used for nanocubes). In addition to the above, we also compare our data structure with the in-memory python library Pandas [McK13] that is commonly used by data scientists in the analysis of time series data. To enable efficient querying, we created a DataFrame with an index on the time dimension.

Software Configuration. The Time Lattice data structure was implemented using C++. For all the experiments, the Hasse diagram in Figure 3 was used to create the Time Lattice on the input data.

Queries were executed 5 times, and the median timings are reported.

4.2. Scalability

We first study the scalability of Time Lattice with increasing data sizes with respect to both query evaluation time as well as data structure update time.

Data Structure Size. Figure 8 shows the size of the Time Lattice data structure for different time series sizes. Note that the size of the structure includes that of the raw data, and the upper bound of additional memory overhead for the data structure is linear in the size of the data itself (see Section 3.1). In practice, as illustrated in the figure, this memory overhead is just a small fraction ($\approx 1.6\%$) of the underlying raw data.

Query Evaluation. Figure 9 shows the query evaluation time for the 4 test queries with increasing data sizes. Note that, except for Q1, the rest of queries cover the entire time series. As expected, one can see a linear scaling with data size. This is primarily due to the data structure size and query time trade-off in the design on Time Lattice. Since there is no cuboid materialized with respect to

	Size		Q1		Q2		Q3		Q4	
	(MB)	Increase	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup
Time Lattice	397	—	40.5	—	15.0	—	12.8	—	92.4	—
Nanocube	41799	105X	116.0	2.9X	4.6	0.3X	2491.8	194X	40083.9	433X
Pandas	1600	4X	1670.1	41.2X	9355.1	623.6X	10399.3	812X	11070.6	119X
InfluxDB	412	1.03X	10574.6	261X	42913.5	2860X	35259.5	2754X	29058.0	314X
TimescaleDB	7867	19X	20385.1	503X	60206.4	4013X	130594.5	10202X	101036.1	1093X
KairosDB	1301	3X	229110.9	5657X	629886.4	41992X	240168.2	18763X	75267.1	814X

Table 4: *Comparing query response times of Time Lattice with existing approaches on a time series with 100M points.*

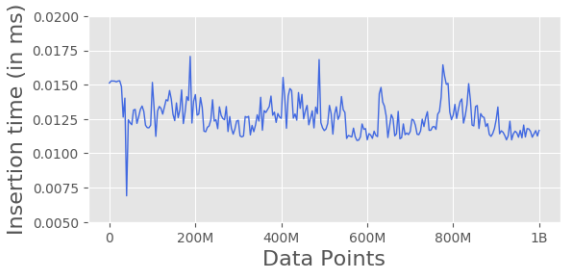

Figure 10: *Average time per update. Note that the update time remains consistent (≈ 0.012 ms) even when adding new data to a Time Lattice built on a time series of size close to a billion points.*

the group-by dimensions used in the queries, the query execution drills down to the finest resolution required, and the processing time is linear in the size of this dimension.

Q4, in particular, is an example of a pathological case query for our data structure due to the following reasons: 1) the time range selected does not align with the dimensions used thus requiring a drill down to a finer resolution during query evaluation (as a rule of thumb, query evaluation requiring only coarser resolutions are faster than those requiring finer resolutions); and 2) the *group-by* is on two dimensions–the corresponding cuboid is not precomputed. Thus, this aggregation has to be evaluated on the fly. Note that even for such complex group-by with multiple constraints over an entire time series having as large as *one billion* time steps, the queries take *less than 650 ms*.

Performance of Updates. Figure 10 shows the time to update the data structure with streaming data. For this experiment, we start with an empty Time Lattice, and insert data one time step at a time. The plot shows the average insertion time for an update with incoming data, and thus increasing data structure size as well. As can be seen from the figure, the time to update the data structure with new data is roughly constant, and around *0.012 ms*. This ensures that even if new data arrives at a frequency of every *millisecond*, our data structure will be *updated without any lag*.

4.3. Comparison with State of the Art

Table 4 compares the performance of Time Lattice with the current state-of-the-art solutions. A time series of size 100 million seconds was used for this experiment. For all the approaches, except for Time Lattice, we had to add additional columns to the data corresponding to the resolutions before loading it. Time Lattice has the lowest space requirement, while nanocubes consumes the most memory. InfluxDB, which compresses the data comes a close second.

The table also shows the query execution times for the four test queries. The performance of Pandas and the three databases is significantly slower than that of Time Lattice. While nanocubes has good performance for Q1, it has the best performance for Q2. This is because Q2 is a straightforward group-by without any constraint or filtering on time. Since nanocubes is essentially a memory optimized data cube, this query is simply a lookup from the corresponding bin. On the other hand, when more complex constraints are imposed, the performance significantly degrades. To improve performance of nanocubes for the constraints involving time of day, one could additionally add a dimension to the data corresponding to it. However, when we tried to create the nanocubes structure with this additional dimension, it ran out of the available 64 GB memory.

5. Noise Profiler

Working with researchers from the SONYC project, we developed a prototype web-based visualization tool, Noise Profiler, that uses Time Lattice to help in the visual analysis of the SPL data obtained from the different sensors deployed in NYC. In this section, we first describe the SPL data followed by discussing the design of the Noise Profiler interface. We finally describe how Time Lattice was used to support the different features of Noise Profiler.

5.1. Sound Measurement Data

For the remainder of this paper, we use sound pressure level decibel (SPL dB_A) data obtained from the different acoustic sensors. Here, the A denotes a frequency weighting that approximates the response of the human auditory system. This data is sampled continuously at *1 second intervals* from each sensor. As mentioned in Section 1, the sensor network used for this work consists of 48 deployed nodes spread across NYC. Thus, each sensor generates a time series having approximately 31.5 million points per year. As part of the analysis, the researchers are also interested in computing a metric called *equivalent continuous A-weighted sound pressure level* (L_Aeq). L_Aeq is the sound pressure level in decibels equivalent to the total A-weighted sound energy measured over a given time period. This metric is used when exploring / analyzing acoustic data over coarser time resolutions.

5.2. Desiderata

The two main tasks that the researchers in the SONYC project are interested in are: 1) specify, execute, and visualize OLAP analytical queries over the SPL data from across the city; and 2) compare live data with the summaries obtained from these queries. To accomplish this, we develop a web-based prototype system built on top of Time Lattice to satisfy the following requirements: 1) visually

specify queries— this includes the ability to select the time period of the data to analyze, apply constraints over different time resolutions, and specify dimensions on which to perform group-by's; 2) ability to select and compare data from one or more sensors based on the location; 3) support for the L_Aeq metric as the aggregate in the queries; and 4) visualize live data together with the results of the queries We now briefly detail the interface followed by describing the backend query processing that is handled separately by a server.

5.3. Visual Interface

The Urban Noise Profiler interface consists of two main components—a query panel and a time series widget (see Figure 1 and the accompanying video).

Query Panel. The query panel (Figure 1 (right)) allows the user to visually frame the different analysis queries, and choose the measure of interest to be visualized. While framing a query, the user can set constraints at various resolutions (e.g. analysis over weekends would require a constraint on the day_{week} resolution, and night time would require a constraint on the hour of day resolution). Users can also specify the group-by dimension. The time range of interest for a query is specified by brushing on the summary view from the time series widget (described next). The query panel also has a *map widget* (Figure 1 (left)) that displays the location of the deployed acoustic sensors from which the user can choose the sensors of interest. In addition, the user can also choose between analysis mode and streaming mode. The analysis mode is primarily used for analysis of historical data, while the latter allows users to visualize streaming data together with analysis queries.

Time Series Widget. The user can create one or more time series widgets, called *time series cards*. Each card is composed of a summary view providing an overview of the entire time series, and a detailed view, visualizing the result from a query. Users can select the time range of interest by brushing over the summary view. When no constraints / group-by's are specified, the query simply corresponds to a range query, and is visualized in the detailed view. We support the level-of-detail rendering by default (see Section 3.2). The resolution at which it is visualized is determined by the screen space (number of horizontal pixels) available. Thus, by zooming in (selecting a smaller time range) the users can see more details of the time series. When group-by's are present, the result of this query over the selected time range is visualized in the detailed view.

Users can select several sensors to be visualized on a single card, and the chosen query is executed on all time series corresponding to these sensors. The color of a time series indicates the sensor source on the map. When there are multiple time series cards, queries are specified separately for each of them, thus allowing the user to use multiple cards for comparing different scenarios (e.g., day time vs night time, or different clusters of sensors).

When working in streaming mode, the live data from the selected sensors is shown together with the plots resulting from the specified query. Here, the live data is visualized using a lighter hue of the sensor color (see Figure 1).

5.4. Query Backend

We implement a server-based backend so as to allow users easy access to the Noise Profiler through a web browser. For each deployed sensor, we maintain one Time Lattice data structure. Given a query and collection of sensors (that are selected by the user), the query is executed once for each of the sensors. Due to the low latency of the Time Lattice data structure, it is possible to perform such analysis interactively. Note that this would not have been possible using existing techniques given their performance. The information about each of the sensor (e.g., location, deployed time) is stored separately, together with a reference to the Time Lattice corresponding to it. Missing and / or invalid data (e.g., when a sensor goes down) is filled with a default value.

When creating the data structure, in addition to the default minimum, maximum, sum, and average measures, we also store information to compute the L_Aeq metric that was required for the analysis tasks. We maintain one background thread per sensor which listens for new data and updates the Time Lattice data structure accordingly.

6. Case Studies

In this section, we illustrate how Noise Profiler can be used in the visual analysis of multiple large time series data with a focus on understanding the acoustic noise patterns in NYC. In particular, we discuss three case studies performed by the researchers in the SONYC project.

6.1. Exploring Noise Patterns via Grouping

To better understand the acoustic conditions of the urban environment, long-term monitoring is required to capture the variations in SPL over different periods: minutes, hours, days, weeks, months and seasons. For example, noise enforcement agencies in cities typically assess a breach of the noise code by a given rise in SPL above the ambient background SPL. In cites, this ambient background SPL varies at many different temporal resolutions, thus it is important to understand these trends in order to better enforce local noise codes.

Case Study 1: Location-wise noise patterns. In this case study, we were interested in exploring the data for global trends, in particular how the noise pattern changes throughout the course of a single day at different sensor locations. This question essentially corresponds to the following group-by query on the different sensors.

select *time series* ***between*** t_1 *and* t_2 ***groupby*** *hour*

Here, $[t_1, t_2)$ corresponding to the time period of interest. To do this, we first select sensors of interest into the same time series card and configure the above query in the query panel.

Figure 11 shows the results from 2 sensors on main traffic thoroughfares and 2 on quieter back streets. It thus required executing 4 group-by queries, each of which took 100 ms to execute. The morning rush-hour ramp up in dB_A level begins at the same time for each group of sensor locations, however, the main-street locations maintain a raised dB_A level until around 7 p.m., when the evening rush hour begins to trail off. The reduction in dB_A level after 1 p.m. for the back street sensors could suggest that these streets are typically less used for evening rush hour travel. The difference in dB_A level between the early morning (12 a.m.–5 a.m.) and peak daytime dB_A levels from 8 a.m.–7 p.m. is far more pronounced at $\approx$7dB for the main-street locations compared to $\approx$2dB for the back-street locations. This highlights the impact of traffic noise on the main-street locations.

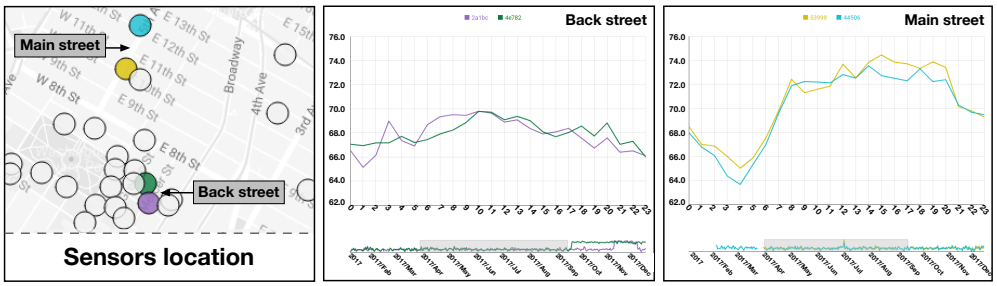

Figure 11: *Comparing daily patterns of noise in back streets with noise in main streets.*

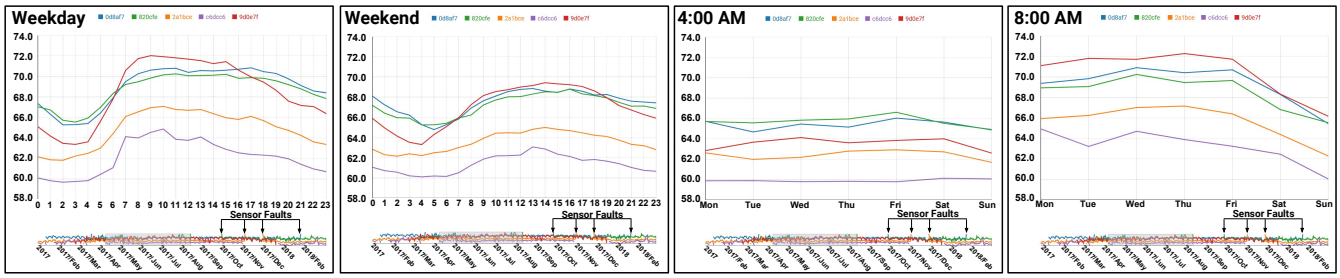

Figure 12: *The two plots on the left show noise patterns on weekdays vs. weekends on diverse locations around Washington Square Park and Central Park. The two plots on the right show the weekly noise patterns for 4am and 8am.*

Case Study 2: Weekday vs weekend patterns. On weekends the daily dB_A levels throughout the day would intuitively exhibit a different trend to those on weekdays. Knowing these differences allow city agencies to better understand the evolution of ambient background levels at different periods of the day and week. Figure 12 shows separately the weekday and weekend daily dB_A level evolutions, aggregated by hour for 5 sensors across varying locations. This shows an ≈1dB difference between weekday and weekend peak dB_A levels, highlighting the raised weekday levels. Of note is the increased gradient on the ramp-up period from early morning to peak rush hour on the weekday plot compared to that of the weekend plot. That is, during weekdays, the noise levels increase sharply between 4 a.m. and 7 a.m. On the other hand, during the weekend, the noise levels start increasing later, at 5 a.m., and take until 2 p.m. to reach peak levels. A key point that is apparent from these plots is the ≈ 1 hour later shift in this ramp up at weekends suggesting that noise making activities begin later and take longer to increase over time.

By visualizing these hourly noise patterns, the above analysis provides the hours of interest to investigate more closely. In particular, while the ramp-up patterns are clear, it is still not straightforward to make out how these hours vary over the different days of the week. This can be visualized using the following query template:

select *time series* ***between*** t_1 *and* t_2 ***where*** *hour=4am*
groupby day_{week}

Figure 12 also shows the weekly noise patterns at 4 a.m. and 8 a.m. respectively, allowing us to explore a different perspective of this data. Note how the noise level at 8 a.m. is relatively constant on weekdays, but is lower on Sundays as compared to Saturdays. On the other hand, it remains consistent throughout the week at 4 a.m. Each of the queries posed to obtain the above visualizations took on average 80 ms per sensor.

These findings can provide valuable information to city agencies

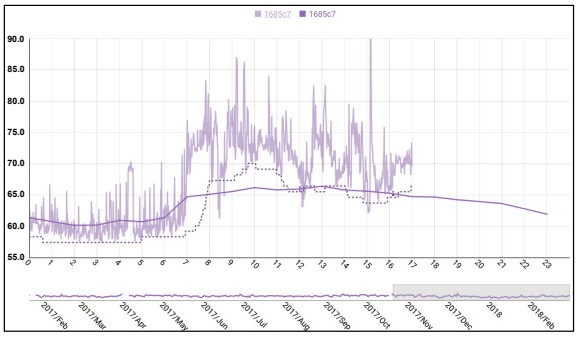

Figure 13: *Comparing live data with two different ambient noise baselines for a given sensor.*

looking to understand the temporal characteristics of dB_A levels at different days of the week. For example, construction permits are generally not issued for work over weekends to reduce the impact on city inhabitants. However, special out–of–hours permits can be requested for weekend work. With knowledge on the temporal evolution of dB_A levels on weekends for a particular location, these permits can be time limited to periods of high ambient dB_A levels, reducing the impact of construction noise on local residents.

Finally, an unexpected outcome of the visual analysis process described above was the identification of erroneous sensor data due to sensor faults as seen in the excessive and continuous raised dB_A levels that can be seen in the summary view in Figure 12. The visual interface allowed us to quickly and easily exclude this erroneous data from the analysis. This kind of sensor data anomaly identification is crucial when maintaining a sensor network of this scale.

Case Study 3: Ambient noise baselines. In NYC, the indication of a noise code violation is given when a noise source exceeds the ambient dB_A level by 10dB. This prompts city agency inspectors to investigate further into the offending noise source to determine

the extent of its breach of the noise code. This ambient level measurement is typically carried out using an instantaneous "eye–ball" measurement using a sound level meter while the offending noise source is not operating. Agency inspectors can also request that these noise sources be switched off to gain a more representative ambient measurement. The issues here are: (1) that this instantaneous ambient measurement may not be that representative of the area, as experienced by its inhabitants / noise complainants over extended periods of time on that day of the week, and (2) that a passive acoustic monitoring network does not have the ability to request the temporary shutdown of noise sources.

Given (1), it is important to consider an ambient background level computed over a more representative period of time, in order to decrease the impact of short lived noise sources and day of week influences. This is naturally captured by a group-by query. Figure 13 shows such a case, with the ambient level computed as the hourly average over the last 11 months, considering only weekdays. Note that the result of this query gets continuously updated with new incoming data. Prior to using the Noise Profiler interface with the Time Lattice data structure, we computed the ambient noise as the 90th percentile over a much shorter and "temporally naive" period of 2 hours, as in, it does not consider the holistic ambient level of this location, during the same period of time over multiple past instances of this period. This is illustrated using a dashed line in Figure 13. Note that that the ambient noise computed using this approach follows the same trend as the actual instantaneous dB level, resulting in a less representative ambient background level measurement. Thus, the use of select historical data for ambient level calculation, therefore addresses issue (2), providing a representative ambient background level measure for effective real–world noise code enforcement. Using Time Lattice, computing both the group-by queries as well as the 90th percentile measure took only 150ms even as the the data structure is simultaneously updated with incoming data.

Figure 1 presents another example showing the weekday hourly noise patterns of two different sensors. One sensor (blue) is located close to a main road (Broadway Av.), and presents a relatively constant dB_A level throughout the hours of the day. The other sensor (orange) is close to a major construction site, with a higher dB_A level during regular construction hours of 7 a.m. to 5 p.m. Also notice a temporary dip in the live noise level around lunch time.

As this use case demonstrates, the combination of OLAP queries over long time-periods and live streaming data can be used to better guide city agents when issuing noise code violations (e.g., construction sites operating outside of their allotted construction hours), as well as to better understand the noise profile of certain regions.

6.2. Feedback

As researchers using the Noise Profiler, we found several advantages in using the proposed system. The primary among them was the ability to seamlessly deal with high resolution SPL data covering large time periods. The high temporal resolution of the acoustic data streamed from our noise sensor network results in vast amounts of data. The frequently short-lived nature of urban noise events mean that all of this data needs to be considered when determining the effects of this noise on city inhabitants. We were typically limited to interacting with small subsets of the data, especially when dealing with a duration of more than a few days due to the limitations of our current tools (e.g., Pandas). The addition of the ability to interactively explore historical data simultaneously from multiple sensors helps tremendously, as now we can make more informed decisions based on the acoustic conditions at multiple locations As shown in the last case study, OLAP queries also allow the computation of a more meaningful baseline for ambient noise level measurements, a clear improvement over our previous "temporally naive" baseline. This, in particular would be of great benefit to city agencies tasked with urban noise enforcement to better understand sources noise levels with respect to a representative ambient baseline. In addition to this, the Noise Profiler would allow a noise enforcement officer to query the periods at the very start and end of the allowed construction times of 7 a.m. and 6 p.m. Construction sites that begin early or end late can be scheduled a visit, optimizing agency resource allocation to the places that matter.

We also recently demonstrated the Noise Profiler prototype to experts from NYC's Department of Environmental Protection (DEP). While they were impressed with the analysis capabilities, especially the responsiveness in querying and handling data from multiple sensors, they found the general query interface a little overwhelming. In particular, they want to simplify the query interface by making it more focused on the typical queries that they repeatedly perform. We are currently in the process of making our system live for them to use.

7. Conclusion

In this paper we presented Time Lattice, a memory efficient data structure to efficiently handle complex OLAP queries over time series data. By selectively materializing a subset of the data cube based on the intrinsic hierarchy of the time resolutions, it allows for a linear memory overhead and also supports constant amortized time updates to the data structure. We also developed Noise Profiler, a web-based visualization framework that uses Time Lattice to allow the interactive analysis of data captured from acoustic sensors deployed around New York City.

While our current implementation can easily handle time series having a billion points interactively, as the data size keeps increasing, interactivity might not always be possible. However, many steps in the query execution process can be parallelized. In future, we intend to explore both CPU as well as GPU-based parallelization strategies, which can enable sub-second response times even with time series having several billions of points.

Acknowledgements

This work was supported in part by: the Moore-Sloan Data Science Environment at NYU; NASA; DOE; The Sounds Of New York City (SONYC) project (NSF award CNS-1544753); NSF awards CNS-1229185, CCF-1533564, CNS-1730396, OAC 1640864; CNPq; and FAPERJ. J. Freire and C. T. Silva are partially supported by the DARPA MEMEX and D3M programs. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[AMM*07] AIGNER W., MIKSCH S., MÜLLER W., SCHUMANN H., TOMINSKI C.: Visualizing time-oriented data-a systematic view. *Comput. Graph. 31*, 3 (2007), 401–409. 3

[BH10] BRONZAFT A. L., HAGLER L.: *Noise: The Invisible Pollutant that Cannot Be Ignored*. Springer Netherlands, 2010, pp. 75–96. 2

[BKF] BADER A., KOPP O., FALKENTHAL M.: Survey and comparison of open source time series databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*, 266. 2, 3

[BM04] BERRY L., MUNZNER T.: Binx: Dynamic exploration of time series datasets across aggregation levels. In *Proc. of the IEEE Symposium on Information Visualization* (2004), IEEE, pp. 215.2–. 3

[BR99] BEYER K., RAMAKRISHNAN R.: Bottom-up computation of sparse and iceberg cube. In *ACM Sigmod Record* (1999), vol. 28, ACM, pp. 359–370. 3

[BWSR13] BUEVICH M., WRIGHT A., SARGENT R., ROWE A.: Respawn: A distributed multi-resolution time-series datastore. In *IEEE 34th Real-Time Systems Symposium (RTSS)* (2013), IEEE, pp. 288–297. 3

[CG16] CORRELL M., GLEICHER M.: The semantics of sketch: A visual query system for time series data. In *Proc. of the IEEE Conference on Visual Analytics Science and Technology (VAST)* (2016), IEEE. 3

[Cit] CITY OF PORTLAND: City Code, Title 18: Noise Control. URL: https://www.portlandoregon.gov/citycode/?c=28182. 2

[CYZ*08] CHEN C., YAN X., ZHU F., HAN J., PHILIP S. Y.: Graph OLAP: Towards online analytical processing on graphs. In *IEEE Int. Conf. on Data Mining* (2008), IEEE, pp. 103–112. 3

[DE14] DUNNING T., ERTL O.: Computing extremely accurate quantiles using t-digests, 2014. URL: https://github.com/tdunning/t-digest/. 7

[DMF12] DERI L., MAINARDI S., FUSCO F.: tsdb: A compressed database for time series. *Traffic Monitoring and Analysis* (2012), 143–156. 3

[DWW*11] DUAN Q., WANG P., WU M., WANG W., HUANG S.: Approximate query on historical stream data. In *Database and Expert Systems Applications* (2011), Springer, pp. 128–135. 3

[GCB*97] GRAY J., CHAUDHURI S., BOSWORTH A., LAYMAN A., REICHART D., VENKATRAO M., PELLOW F., PIRAHESH H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1*, 1 (1997), 29–53. 3, 4, 5

[HCD*05] HAN J., CHEN Y., DONG G., PEI J., WAH B. W., WANG J., CAI Y. D.: Stream cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases 18*, 2 (2005), 173–197. 3

[HDKS07] HAO M. C., DAYAL U., KEIM D. A., SCHRECK T.: Multi-resolution techniques for visual exploration of large time-series data. In *Proc. of the 9th Joint Eurographics / IEEE VGTC Conference on Visualization* (2007), pp. 27–34. 3

[HS04] HOCHHEISER H., SHNEIDERMAN B.: Dynamic query tools for time series data sets: timebox widgets for interactive exploration. *Information Visualization 3*, 1 (2004), 1–18. 3

[Inf] InfluxDB. https://github.com/influxdata/influxdb. 2, 3, 8

[JJHM14] JUGEL U., JERZAK Z., HACKENBROICH G., MARKL V.: M4: a visualization-oriented time series data aggregation. *Proc. of the VLDB Endowment 7*, 10 (2014), 797–808. 3

[JME10] JAVED W., MCDONNEL B., ELMQVIST N.: Graphical perception of multiple time series. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 927–934. 3

[JPT17] JENSEN S. K., PEDERSEN T. B., THOMSEN C.: Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering 29*, 11 (2017), 2581–2600. 3

[Kai] KairosDB. URL: https://kairosdb.github.io/. 2, 3, 8

[LDH*08] LIN C. X., DING B., HAN J., ZHU F., ZHAO B.: Text cube: Computing ir measures for multidimensional text database analysis. In *IEEE Int. Conf. on Data Mining* (2008), IEEE, pp. 905–910. 3

[LH14] LIU Z., HEER J.: The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2122–2131. 2

[LKS13] LINS L., KLOSOWSKI J. T., SCHEIDEGGER C.: Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2456–2465. 2, 3, 8

[McK13] MCKINNEY W.: *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, first ed. O'Reilly, 2013. 8

[MLKS18] MIRANDA F., LINS L., KLOSOWSKI J. T., SILVA C. T.: Topkube: A rank-aware data cube for real-time exploration of spatiotemporal data. *IEEE Transactions on Visualization and Computer Graphics 24*, 3 (2018), 1394–1407. 2

[MMKN08] MCLACHLAN P., MUNZNER T., KOUTSOFIOS E., NORTH S.: Liverac: Interactive visual exploration of system management time-series data. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems* (2008), ACM, pp. 1483–1492. 3

[MS03] MÜLLER W., SCHUMANN H.: Visualization for modeling and simulation: visualization methods for time-dependent data-an overview. In *Proc. of the 35th Conf. on Winter Simulation: Driving Innovation* (2003), Winter Simulation Conference, pp. 737–745. 3

[MSB17a] MYDLARZ C., SALAMON J., BELLO J. P.: The implementation of low-cost urban acoustic monitoring devices. *Applied Acoustics 117* (2017), 207–218. 2

[MSB17b] MYDLARZ C., SHAMOON C., BELLO J. P.: Noise monitoring and enforcement in new york city using a remote acoustic sensor network. *Proc. of INTER-NOISE and NOISE-CON 255*, 2 (2017), 5509–5520. 2

[MVCJ16] MUTHUMANICKAM P. K., VROTSOU K., COOPER M., JOHANSSON J.: Shape grammar extraction for efficient query-by-sketch pattern matching in long time series. In *Proc. of the IEEE Conference on Visual Analytics Science and Technology (VAST)* (2016), IEEE. 3

[NYC05] New York City Local Law No. 113, 2005. URL: http://www.nyc.gov/html/dep/pdf/law05113.pdf. 2

[PFT*15] PELKONEN T., FRANKLIN S., TELLER J., CAVALLARO P., HUANG Q., MEZA J., VEERARAGHAVAN K.: Gorilla: A fast, scalable, in-memory time series database. *Proc. of the VLDB Endowment 8*, 12 (2015), 1816–1827. 2, 3

[PSSC17] PAHINS C. A., STEPHENS S. A., SCHEIDEGGER C., COMBA J. L.: Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 671–680. 2, 3, 8

[SC00] SILVA S. F., CATARCI T.: Visualization of linear time-oriented data: a survey. In *Proc. of the First Int. Conf. on Web Information Systems Engineering* (2000), vol. 1, IEEE, pp. 310–319. 3

[SMD07] SABHNANI M., MOORE A. W., DUBRAWSKI A. W.: *T-Cube: A data structure for fast extraction of time series from large datasets*. Tech. rep., DTIC Document, 2007. 2

[SON] SONYC: Sounds of New York City. URL: https://wp.nyu.edu/sonyc/. 2

[Tim] TimescaleDB. URL: https://timescale.com/. 3, 8

[VWVS99] VAN WIJK J. J., VAN SELOW E. R.: Cluster and calendar based visualization of time series data. In *Proc. of the IEEE Symposium on Information Visualization* (1999), IEEE, pp. 4–. 3

[WFW*17] WANG Z., FERREIRA N., WEI Y., BHASKAR A. S., SCHEIDEGGER C.: Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 681–690. 7

[ZCPB11] ZHAO J., CHEVALIER F., PIETRIGA E., BALAKRISHNAN R.: Exploratory analysis of time-series with ChronoLenses. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2422–2431. 3